The Basics of Inheritance & super()

AiCE Software Engineering Foundations — Python Programming (Advanced)

In the previous unit, you learned to define classes and create objects. Inheritance takes this further — allowing one class to build on another, automatically receiving all of its attributes and methods while adding or customizing its own. It is the mechanism that makes object-oriented code reusable at scale.

1. What Is Inheritance?

Inheritance is a relationship between classes. When one class inherits from another, it receives all of the parent class's attributes and methods automatically — without rewriting any of them. The child class can then add new attributes and methods of its own, or override the parent's methods to change their behavior.

The real-world analogy is straightforward: a child inherits traits from a parent. They share a biological foundation but are distinct individuals with their own characteristics. In programming, a child class (also called a subclass or derived class) inherits from a parent class (also called a superclass or base class).

Consider an AiCE learning platform. There are students, instructors, and administrators. All three types of person share common characteristics — they all have a name, an email, and an age. But each type also has unique properties and behaviors. Students have scores. Instructors have courses they teach. Administrators have system access levels.

Without inheritance, you would write the common attributes (name, email, age) in every class, repeating yourself three times. With inheritance, you write those attributes once in a Person base class and have Student, Instructor, and Administrator each inherit from it — getting the common attributes for free while adding their own.

> **Write It Once, Inherit It Everywhere**
> *"Inheritance is the programming equivalent of not repeating yourself at the architectural level. If two classes share behavior, that behavior belongs in a parent class. Write it once. Inherit it everywhere." — Adesina Olagunju, Founder, AiCE*

2. The Basic Inheritance Syntax

To make a class inherit from another, write the parent class name in parentheses after the child class name:

```
class ParentClass:
    # Parent attributes and methods
    pass

class ChildClass(ParentClass):
    # Child inherits everything from ParentClass
    # Plus its own additional attributes and methods
    pass
```

```
# Base class — shared by all people on the AiCE platform
class Person:
    """Represents any person on the AiCE platform."""
    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}, aged {self.age}."

    def get_contact(self):
        return f"Email: {self.email}"

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

# Child class — inherits from Person
class Student(Person):
    """Represents a student. Inherits from Person."""
    pass

# Creating a Student object
amara = Student("Amara Diallo", 22, "amara@aice.africa")
```

```
# Using inherited methods
print(amara.introduce())        # From Person
print(amara.get_contact())      # From Person
print(amara.name)               # Inherited attribute
print(amara.age)                # Inherited attribute
print(type(amara))

# Check inheritance relationship
print(isinstance(amara, Student))    # True
print(isinstance(amara, Person))     # True — a Student IS a
Person
```

Student inherits everything from Person — __init__, introduce(), get_contact(), and __str__ — without writing a single one of them. The isinstance() check confirms that a Student object is simultaneously an instance of Student AND an instance of Person. This is the is-a relationship: a Student is a Person.

3. Extending the Child Class

A child class that only inherits without adding anything is not very useful. The power of inheritance comes when the child class extends the parent — adding new attributes and methods specific to that type:

```
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}, aged {self.age}."

class Student(Person):
    """Student — extends Person with academic data."""
    def __init__(self, name, age, email, student_id, course):
        # Call parent __init__ to set up shared attributes
        super().__init__(name, age, email)
        # Add Student-specific attributes
```

```
        self.student_id = student_id
        self.course = course
        self.scores = {}

    def add_score(self, subject, score):
        """Add a subject score for this student."""
        self.scores[subject] = score

    def get_average(self):
        """Calculate and return average score."""
        if not self.scores:
            return 0
        return round(sum(self.scores.values()) /
len(self.scores), 2)

    def display_profile(self):
        """Display complete student profile."""
        print(f"  Name       : {self.name}")
        print(f"  Student ID : {self.student_id}")
        print(f"  Course     : {self.course}")
        print(f"  Average    : {self.get_average()}")

amara = Student("Amara Diallo", 22, "amara@aice.africa",
"AiCE001", "Python")
amara.add_score("Math", 88)
amara.add_score("English", 76)
amara.add_score("Science", 91)

print(amara.introduce())    # Inherited from Person
amara.display_profile()     # Defined in Student
```

The Student __init__ now accepts five parameters — three shared with Person (name, age, email) and two specific to Student (student_id, course). super().__init__() is the critical line that calls the parent class's __init__ to handle the shared setup. Without it, name, age, and email would never be assigned.

4. super() — Calling the Parent Class

super() returns a proxy object that gives you access to the parent class. Its most common use is calling the parent's __init__ from inside the child's __init__ — delegating the shared setup to the parent rather than duplicating it.

```
# Without super() – duplicates the parent logic
class Student(Person):
    def __init__(self, name, age, email, student_id):
        self.name = name.strip().title()     # Duplicated!
        self.age = age                        # Duplicated!
        self.email = email.strip().lower()    # Duplicated!
        self.student_id = student_id
```

```
# With super() – delegates shared setup to parent
class Student(Person):
    def __init__(self, name, age, email, student_id):
        super().__init__(name, age, email)    # Parent handles
shared setup
        self.student_id = student_id          # Only the new
bit here
```

The super() version is shorter, cleaner, and correct. If Person's __init__ is ever updated — say, adding validation logic — the Student class automatically benefits without any changes. The duplicate version would fall behind immediately.

> **Always Call super().__init__() in Child Classes**
> *Whenever a child class defines its own __init__, it must call super().__init__() to properly initialize the parent class's attributes. Forgetting this means the child object will be missing the parent's attributes — leading to AttributeError when you try to use them. Make this a reflex.*

Key Takeaways

Inheritance — class Child(ParentClass): — lets a child class receive every attribute and method from its parent automatically, following an is-a relationship confirmed by

isinstance(). A child class earns its usefulness by extending the parent with new attributes and methods, and super().__init__() is the reflexive first line of any child __init__, delegating shared setup to the parent instead of duplicating it. The next lesson covers method overriding, where a child class replaces a parent's method entirely, and polymorphism, the ability for different classes to implement the same method name differently while Python calls the correct version automatically.