The AiCE Platform Hierarchy, Composition & Checking Relationships

AiCE Software Engineering Foundations — Python Programming (Advanced)

This closing lesson on inheritance combines everything from this unit into one complete, realistic class hierarchy, shows how to check inheritance relationships directly in code, and introduces composition — the alternative to inheritance, and the judgment needed to choose correctly between them.

1. A Complete Inheritance Hierarchy — The AiCE Platform

A complete, realistic class hierarchy modeling the AiCE platform — a Person base class with Student, Instructor, and Administrator all inheriting from it:

```
# AiCE Platform — Complete Inheritance Hierarchy
class Person:
    """Base class for all people on the AiCE platform."""
    platform = "AiCE"

    def __init__(self, name, age, country, email):
        self.name = name.strip().title()
        self.age = age
        self.country = country.strip().title()
        self.email = email.strip().lower()

    def introduce(self):
        return f"I am {self.name} from {self.country}."

    def get_contact(self):
        return self.email

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

class Student(Person):
```

```
    """Student on the AiCE platform."""
    def __init__(self, name, age, country, email, student_id,
course):
        super().__init__(name, age, country, email)
        self.student_id = student_id
        self.course = course
        self.scores = {}

    def add_score(self, subject, score):
        if 0 <= score <= 100:
            self.scores[subject] = score
        else:
            raise ValueError(f"Score {score} out of range
(0-100).")

    def get_average(self):
        if not self.scores:
            return 0.0
        return round(sum(self.scores.values()) /
len(self.scores), 2)

    def get_grade(self):
        avg = self.get_average()
        return "A" if avg >= 70 else "B" if avg >= 60 else "C"
if avg >= 50 else "F"

    def introduce(self):
        return (f"I am {self.name} ({self.student_id}),
enrolled in {self.course} "
                f"at {self.platform}.")

    def display_profile(self):
        print(f"  [{self.student_id}] {self.name} |
{self.country} | {self.course}")
        print(f"  Average: {self.get_average()} | Grade:
{self.get_grade()}")

class Instructor(Person):
    """Instructor on the AiCE platform."""
```

```
    def __init__(self, name, age, country, email,
instructor_id, specialisation):
        super().__init__(name, age, country, email)
        self.instructor_id = instructor_id
        self.specialisation = specialisation
        self.courses_taught = []

    def assign_course(self, course_name):
        self.courses_taught.append(course_name)

    def introduce(self):
        return (f"I am {self.name} ({self.instructor_id}), "
                f"an instructor specialising in
{self.specialisation}.")

    def display_profile(self):
        courses = ", ".join(self.courses_taught) if
self.courses_taught else "None assigned"
        print(f"  [{self.instructor_id}] {self.name} |
{self.specialisation}")
        print(f"  Courses: {courses}")

class Administrator(Person):
    """Platform administrator."""
    def __init__(self, name, age, country, email, admin_id,
access_level):
        super().__init__(name, age, country, email)
        self.admin_id = admin_id
        self.access_level = access_level    # 1=Basic,
2=Manager, 3=SuperAdmin

    def introduce(self):
        levels = {1: "Basic", 2: "Manager", 3: "Super Admin"}
        level_name = levels.get(self.access_level, "Unknown")
        return f"I am {self.name}, {self.platform}
{level_name} Administrator."

    def can_delete_records(self):
        return self.access_level >= 2
```

Building the platform and generating a full report:

```
students = [
    Student("Amara Diallo", 22, "Senegal",
"amara@aice.africa", "AiCE-S001", "Python"),
    Student("Kofi Mensah", 19, "Ghana", "kofi@aice.africa",
"AiCE-S002", "JavaScript"),
    Student("Ngozi Adeyemi", 23, "Nigeria",
"ngozi@aice.africa", "AiCE-S003", "Python"),
]
students[0].add_score("Math", 88)
students[1].add_score("Math", 45)
students[2].add_score("Math", 92)

instructor = Instructor("Adesina Olagunju", 35, "Nigeria",
"adesina@aice.africa", "AiCE-I001", "Python")
instructor.assign_course("Python Fundamentals")

admin = Administrator("Emeka Okafor", 40, "Nigeria",
"emeka@aice.africa", "AiCE-A001", 3)

all_users = students + [instructor, admin]

print("=" * 55)
print("  AiCE PLATFORM USER DIRECTORY")
print("=" * 55)
for user in all_users:
    print(f"\n  {user.introduce()}")
    if hasattr(user, 'display_profile'):
        user.display_profile()

print(f"\n  Admin can delete records:
{admin.can_delete_records()}")
```

Each class has its own introduce() and display_profile() implementations. The all_users loop iterates over a mixed list and calls the correct method on each object through polymorphism.

self.__class__.__name__

Notice self.__class__.__name__ in Person.__str__ — this returns the actual class name of the object (Student, Instructor, or Administrator), not just 'Person'. This useful trick makes the base class's __str__ automatically adapt to any subclass, without needing to be overridden.

2. Checking Inheritance Relationships

Python provides two built-in functions for checking inheritance directly:

```
amara = Student("Amara", 22, "Senegal", "amara@aice.africa",
"S001", "Python")

# isinstance() – checks if an object is an instance of a class
(or subclass)
print(isinstance(amara, Student))       # True
print(isinstance(amara, Person))        # True – Student
inherits from Person
print(isinstance(amara, Instructor))    # False

# issubclass() – checks if a class inherits from another class
print(issubclass(Student, Person))      # True
print(issubclass(Instructor, Person))   # True
print(issubclass(Student, Instructor))  # False

# Check parent classes
print(Student.__bases__)    # (<class '__main__.Person'>,)
print(Person.__bases__)     # (<class 'object'>,) – all
classes inherit from object
```

Every class in Python ultimately inherits from the built-in object class — even if you do not write it explicitly. Person.__bases__ shows (object,); Student.__bases__ shows (Person,).

3. Inheritance vs. Composition

Inheritance is not the only way to reuse code between classes. Composition is an alternative — instead of one class inheriting from another, a class contains an instance of another class as one of its attributes.

	Inheritance	Composition
Relationship	IS-A (a Student IS a Person)	HAS-A (a Student HAS scores)
Syntax	class Child(Parent):	self.attribute = OtherClass()
Use when	Classes genuinely share identity	Classes share behaviour but different identity
Example	Student inherits from Person	Student has a ScoreRecord object
Coupling	Tightly coupled — inherits everything	Loosely coupled — uses what it needs

The rule of thumb: use inheritance when the child class truly is a specialized version of the parent (a Student IS a Person). Use composition when one class simply uses another (a Car HAS an Engine). Overusing inheritance — making classes inherit from others just to reuse code — leads to fragile, hard-to-modify designs. Let the relationship between the concepts guide the choice.

Key Takeaways

The AiCE Platform hierarchy demonstrates inheritance, super(), method overriding, and polymorphism working together — one Person base class, three distinct subclasses, each behaving correctly through the same shared interface. isinstance() and issubclass() let you check these relationships directly in code, and every class ultimately traces back to Python's built-in object class. Composition — a class containing another as an attribute rather than inheriting from it — is the right choice whenever the relationship is HAS-A rather than IS-A. With inheritance fully understood, the next unit moves outside Python's standard library entirely, teaching you to install and use external packages with pip — the tool that connects your programs to the wider Python ecosystem.