A Complete Student Class & OOP Terminology

AiCE Software Engineering Foundations — Python Programming (Advanced)

This closing lesson on classes and objects brings everything from this unit together into one complete, well-designed class — the same class that will serve as the foundation for the Student Management System capstone later in this program — and closes with a full terminology reference.

1. A Complete OOP Example — The AiCE Student Class

```
# AiCE Student Class – Complete Implementation
class Student:
    """Represents a student in the AiCE learning system."""

    institution = "AiCE"
    passing_mark = 50

    def __init__(self, name, age, country, math, english,
science):
        """Initialise a student with personal details and
subject scores."""
        self.name = name.strip().title()
        self.age = age
        self.country = country.strip().title()
        self.scores = {
            "Math"   : math,
            "English": english,
            "Science": science
        }

    def get_average(self):
        """Return the student's average score across all
subjects."""
        return round(sum(self.scores.values()) /
len(self.scores), 2)

    def get_grade(self):
```

```
        """Return the letter grade corresponding to the
average score."""
        avg = self.get_average()
        if avg >= 70: return "A"
        elif avg >= 60: return "B"
        elif avg >= 50: return "C"
        elif avg >= 40: return "D"
        else: return "F"

    def get_status(self):
        """Return PASS or FAIL based on the passing mark."""
        return "PASS" if self.get_average() >=
self.passing_mark else "FAIL"

    def is_distinction(self):
        """Return True if the student achieved a distinction
(grade A)."""
        return self.get_grade() == "A"

    def to_dict(self):
        """Convert the student object to a dictionary (for
JSON storage)."""
        return {
            "name"   : self.name,
            "age"    : self.age,
            "country": self.country,
            "scores" : self.scores
        }

    @classmethod
    def from_dict(cls, data):
        """Create a Student object from a dictionary (from
JSON storage)."""
        return cls(
            data["name"], data["age"], data["country"],
            data["scores"]["Math"], data["scores"]["English"],
data["scores"]["Science"]
        )

    def __str__(self):
```

```
        """Return a readable string representation."""
        return (f"Student({self.name}, {self.country}, "
                f"Avg: {self.get_average()}, Grade:
{self.get_grade()})")

    def __repr__(self):
        """Return an unambiguous representation for
debugging."""
        return f"Student(name={self.name!r}, age={self.age})"
```

Using the class — creating several students and generating a class-wide report:

```
students = [
    Student("Amara Diallo",    22, "Senegal", 88, 76, 91),
    Student("Kofi Mensah",     19, "Ghana",   45, 52, 38),
    Student("Ngozi Adeyemi",   23, "Nigeria", 92, 85, 88),
    Student("Fatima Al-Hassan",21, "Sudan",   61, 70, 58),
    Student("Kwame Asante",    20, "Ghana",   79, 82, 75),
]

print("=" * 52)
print("  AiCE STUDENT REPORT")
print("=" * 52)
for student in students:
    print(f"  {student.name:<22} | {student.country:<10} | "
          f"Avg: {student.get_average():>5} |
{student.get_grade()} | {student.get_status()}")

# Class-level statistics
averages = [s.get_average() for s in students]
passed = [s for s in students if s.get_status() == "PASS"]
distinction = [s for s in students if s.is_distinction()]

print(f"  Total students : {len(students)}")
print(f"  Passed         : {len(passed)}")
print(f"  Distinctions   : {len(distinction)}")
print(f"  Class average  : {round(sum(averages)/len(averages),
2)}")
```

```
# Test to_dict and from_dict
amara_dict = students[0].to_dict()
print(f"As dict : {amara_dict}")
restored = Student.from_dict(amara_dict)
print(f"Restored: {restored}")
```

This class introduces two additional concepts worth understanding:

to_dict() converts the object's data into a plain Python dictionary — suitable for JSON serialization, covered fully in an upcoming unit.

from_dict() is a class method, decorated with @classmethod — a special type of method that receives the class itself (cls) as its first parameter instead of an instance. It's used as an alternative constructor, creating a Student object from a dictionary rather than individual arguments. This pattern is standard in professional Python for loading objects from stored data.

> **__repr__ vs. __str__**
> *__repr__ is a second dunder method that returns an unambiguous representation used by the Python interpreter for debugging. The !r in the f-string applies repr() to the value, adding quotes around strings automatically. __str__ is for humans reading output; __repr__ is for developers debugging.*

2. OOP Terminology — Quick Reference

Term	Meaning	Example
Class	Blueprint/template for objects	class Student:
Object / Instance	A specific thing created from a class	amara = Student(...)
Attribute	Data stored on an object	self.name, self.age
Method	Function defined inside a class	def get_average(self):
self	Reference to the current object instance	self.name = name
__init__	Constructor — called when object is created	def __init__(self, name):

Term	Meaning	Example
__str__	Controls how object looks when printed	def __str__(self): return ...
Class attribute	Shared by all instances of the class	passing_mark = 50
Instance attribute	Unique to each object	self.name = name
@classmethod	Method that receives the class, not an instance	def from_dict(cls, data):
Instantiation	The act of creating an object from a class	amara = Student('Amara', 22)
PascalCase	Naming convention for class names	Student, BankAccount, CourseModule

Key Takeaways

The complete Student class combines everything from this unit — instance and class attributes, methods calling other methods, __str__ for readable output — with two new patterns: to_dict()/from_dict() for converting between objects and dictionaries, and @classmethod for alternative constructors that build an object from existing data rather than individual arguments. These patterns are exactly what real applications use to save and load object data. With classes and objects fully understood, the next unit introduces inheritance — the mechanism that lets one class build directly on another, extending and reusing code rather than rewriting it.