Reading & Writing JSON Files: dump/load & Classes

AiCE Software Engineering Foundations — Python Programming (Advanced)

With dumps() and loads() covering JSON strings, this lesson moves to files — json.dump() and json.load() — and brings JSON together with the classes from earlier in this program, using the exact to_dict()/from_dict() pattern that powers the Student Management System capstone ahead.

1. json.dump() — Writing JSON to a File

json.dump() writes a Python object directly to a file as JSON. This is how you persist data between program runs:

```
import json

students = [
    {"name": "Amara Diallo", "age": 22, "country": "Senegal",
     "scores": {"Math": 88, "English": 76, "Science": 91}},
    {"name": "Kofi Mensah", "age": 19, "country": "Ghana",
     "scores": {"Math": 45, "English": 52, "Science": 38}},
    {"name": "Ngozi Adeyemi", "age": 23, "country": "Nigeria",
     "scores": {"Math": 92, "English": 85, "Science": 88}},
]

# Write to a JSON file
with open("students.json", "w") as file:
    json.dump(students, file, indent=4)

print("Data saved to students.json")

# Verify by reading the file content
with open("students.json", "r") as file:
    print(file.read())
```

A file called students.json is created in your current directory. Checking it with cat students.json confirms the contents — beautifully formatted, indented JSON. This file can be read by Python again, but also by any other language, any API client, or any JSON viewer.

2. json.load() — Reading JSON from a File

json.load() reads a JSON file and returns the data as a Python object. This is how you load persisted data back into your program:

```
import json

# Load the students from the JSON file
with open("students.json", "r") as file:
    students = json.load(file)

print(f"Loaded {len(students)} students.")

# Process the loaded data — it is a regular Python list of
dicts
for student in students:
    name = student["name"]
    country = student["country"]
    scores = student["scores"]
    avg = sum(scores.values()) / len(scores)
    grade = "A" if avg >= 70 else "B" if avg >= 60 else "C" if
avg >= 50 else "F"
    print(f"  {name:<22} | {country:<10} | Avg: {avg:.1f} |
Grade: {grade}")
```

After json.load(), students is an ordinary Python list of dictionaries. You can loop over it, filter it, sort it — everything you know how to do with lists and dictionaries applies immediately. The data is completely restored from the file to an in-memory Python structure.

json.load vs. json.loads

json.load(file) reads from an open file object. json.loads(string) parses a string. The difference is one letter — l-o-a-d vs. l-o-a-d-s. Getting them mixed up is a very common mistake. If you try to pass a filename string to json.load(), Python will raise a TypeError. Always open the file first, then pass the file object to json.load().

3. Handling JSON with Classes — The Complete Pattern

In the Student Management System, student data will be stored as JSON files. The pattern is to convert objects to dictionaries for storage and reconstruct them from dictionaries on load — the to_dict()/from_dict() pattern introduced when you first covered classes.

```
import json
import os

class Student:
    """Student class with JSON serialisation support."""
    passing_mark = 50

    def __init__(self, name, age, country, math, english,
science):
        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 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 get_status(self):
```

```
        return "PASS" if self.get_average() >=
self.passing_mark else "FAIL"

    def to_dict(self):
        """Convert student object to a JSON-serialisable
dictionary."""
        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."""
        return cls(
            data["name"], data["age"], data["country"],
            data["scores"]["Math"], data["scores"]["English"],
data["scores"]["Science"]
        )

    def __str__(self):
        return f"Student({self.name}, {self.country}, Avg:
{self.get_average()})"

DATABASE_FILE = "aice_students.json"

def save_students(students):
    """Save a list of Student objects to a JSON file."""
    data = [s.to_dict() for s in students]
    with open(DATABASE_FILE, "w") as file:
        json.dump(data, file, indent=4)
    print(f"Saved {len(students)} students to
{DATABASE_FILE}")

def load_students():
    """Load students from JSON file. Returns empty list if
file missing."""
    if not os.path.exists(DATABASE_FILE):
        return []
    with open(DATABASE_FILE, "r") as file:
        data = json.load(file)
```

```
    return [Student.from_dict(d) for d in data]

# Demonstrate the full save-load cycle
cohort = [
    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),
]

print("Saving students...")
save_students(cohort)

print("\nLoading students...")
loaded = load_students()

print(f"\nLoaded {len(loaded)} students:")
for student in loaded:
    print(f"  {student.name:<22} | {student.country:<10} | "
          f"Avg: {student.get_average()} |
{student.get_grade()} | {student.get_status()}")
```

The program creates four Student objects, saves them to aice_students.json, loads them back, and verifies the data is intact. The save uses a list comprehension to convert each object to a dict. The load uses another list comprehension to reconstruct each object from its dict. This complete round-trip — object to JSON file to object — is exactly what powers the Student Management System.

Key Takeaways

json.dump(obj, file, indent=4) writes a Python object directly to a file as formatted JSON, and json.load(file) reads it back — remembering that load takes a file object, never a filename string. The to_dict()/from_dict() pattern is the bridge between your classes and JSON storage: to_dict() prepares an object for saving, and from_dict(), as a @classmethod, rebuilds it faithfully on load. This save-load round trip, built from list comprehensions on both ends, is the exact foundation the Student Management System capstone will rely on. The next lesson covers handling JSON errors gracefully, real-world

JSON scenarios like API responses and configuration files, and the load-modify-save pattern for updating stored JSON data.