What Is JSON & the json Module: dumps/loads

AiCE Software Engineering Foundations — Python Programming (Advanced)

Every web API, every mobile application, every configuration file, every data exchange system — they all speak the same language: JSON. It is the universal format for structured data. This lesson opens JSON itself and Python's built-in json module, starting with converting Python data to and from JSON strings.

1. What Is JSON?

JSON stands for JavaScript Object Notation. Despite the name, it has nothing to do with JavaScript in practice — it is a language-independent text format that any programming language can read and write. It was designed to be lightweight, human-readable, and easy for machines to parse and generate.

JSON looks remarkably like Python dictionaries and lists — because both JSON and Python borrowed their syntax from similar mathematical concepts:

```
{
  "name": "Amara Diallo",
  "age": 22,
  "country": "Senegal",
  "enrolled": true,
  "gpa": 3.85,
  "courses": ["Python", "SQL", "Git"],
  "scores": {
    "Math": 88,
    "English": 76,
    "Science": 91
  }
}
```

If you have worked with Python dictionaries throughout this program, this looks almost identical. The differences are subtle but important:

JSON	Python Equivalent	Note
"string"	'string' or "string"	JSON always uses double quotes for strings
true / false	True / False	JSON uses lowercase; Python uses capitalised
null	None	JSON's null maps to Python's None
number (int)	int	Same
number (float)	float	Same
array [...]	list [...]	JSON arrays map directly to Python lists
object {...}	dict {...}	JSON objects map directly to Python dicts

JSON is used everywhere because it is universal. A Python program can write data to a JSON file. A JavaScript web app can read that same file. A Java backend can process it. A mobile app on Android can consume it. JSON is the common language that connects systems built in different programming languages.

2. Python's json Module

Python's standard library includes the json module — no installation required. It provides four core functions:

Function	Direction	Input	Output
json.dumps(obj)	Python → JSON string	Python dict/list	JSON-formatted string
json.loads(str)	JSON string → Python	JSON string	Python dict/list
json.dump(obj, f)	Python → JSON file	Python obj + file	Writes to file
json.load(f)	JSON file → Python	File object	Python dict/list

The Naming Convention

dumps and loads work with strings (the s stands for string). dump and load work with files. Once you notice this pattern, the entire json module becomes easy to remember: an s at the end always means you're dealing with a string, not a file.

3. json.dumps() — Converting Python to a JSON String

json.dumps() serialises a Python object (dictionary, list, or any JSON-compatible type) into a JSON-formatted string. This is the first step toward storing or transmitting data:

```
import json

student = {
    "name"    : "Amara Diallo",
    "age"     : 22,
    "country" : "Senegal",
    "enrolled": True,
    "gpa"     : 3.85,
    "courses" : ["Python", "SQL", "Git"],
    "scores"  : {"Math": 88, "English": 76, "Science": 91}
}

# Convert to JSON string
json_string = json.dumps(student)
print(json_string)
print(type(json_string))

# Pretty-printed JSON with indentation
pretty_json = json.dumps(student, indent=4)
print(pretty_json)

# Notice the Python-to-JSON conversions: True -> true
data_with_none = {"name": "Test", "email": None, "active":
False}
print(json.dumps(data_with_none, indent=2))
```

The indent=4 parameter produces human-readable, indented JSON. Without it, the output is a single compact line — suitable for transmission but hard to read. Notice that Python's True became JSON's true, and None became null.

Controlling JSON Output

```
# Sort keys alphabetically
print(json.dumps(student, indent=2, sort_keys=True))

# Compact — no spaces (smallest possible output)
compact = json.dumps(student, separators=(",", ":"))
print(compact)
```

sort_keys=True produces alphabetically ordered keys — useful for consistent output and version control diffs. The separators parameter controls the characters between items and key-value pairs, producing the most compact possible JSON for transmission where every byte matters.

4. json.loads() — Converting a JSON String to Python

json.loads() does the reverse — it parses a JSON string and returns a Python dictionary or list. This is how you receive and process JSON data from APIs, files, or network responses:

```
import json

# A JSON string (as you might receive from an API)
json_string = """
{
    "name": "Kofi Mensah",
    "age": 19,
    "country": "Ghana",
    "enrolled": true,
    "scores": {"Math": 45, "English": 52, "Science": 38}
}
"""
```

```
# Parse the JSON string into a Python dictionary
student = json.loads(json_string)
print(type(student))
print(student["name"])
print(student["scores"]["Math"])
print(student["enrolled"])    # True — Python bool, not JSON true

# Now you can use it like any Python dictionary
avg = sum(student["scores"].values()) / len(student["scores"])
print(f"Average: {avg:.2f}")
```

After json.loads(), you have a regular Python dictionary — you can access it with square brackets, iterate over it, pass it to functions, everything. JSON true becomes Python True. JSON null becomes Python None. JSON arrays become Python lists.

Key Takeaways

JSON is a universal, language-independent text format that maps almost directly onto Python's dictionaries and lists, with a few notable differences: double-quoted strings only, lowercase true/false/null. json.dumps() converts a Python object into a JSON string, with indent and sort_keys controlling readability, while json.loads() parses a JSON string back into Python data you can use immediately. The naming convention — an s means string, no s means file — carries through the entire module. The next lesson covers writing JSON directly to files with json.dump() and reading it back with json.load(), plus the complete to_dict()/from_dict() pattern for saving and loading your own classes as JSON.