JSON in Python

Introduction

Programs often need to store and exchange structured data.

For example, a user profile might contain:

```
Name
Age
Skills
Email
Projects
```

A simple text file can store this information, and CSV works well for rows and columns.

But what if the data is more complex?

For example:

```
A user
 □□□ name
 □□□ age
 □□□ skills
 □    □□□ Python
 □    □□□ JavaScript
 □□□ projects
     □□□ Website
     □□□ Mobile App
```

CSV is not ideal for this type of nested data.

This is where **JSON** becomes useful.

JSON stands for:

JavaScript Object Notation

Despite its name, JSON is widely used with Python and many other programming languages.

JSON is commonly used for:

- APIs
- configuration files
- application data
- data exchange
- web applications
- storing structured information
- communication between frontend and backend systems

Python provides a built-in module called `json` for working with JSON data.

---

1. What Is JSON?

JSON is a text-based format for representing structured data.

Example:

```
{
   "name": "Hafsa",
   "age": 25,
```

```
    "course": "Python"
}
```

This looks similar to a Python dictionary:

```
{
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}
```

But they are not exactly the same thing.

A Python dictionary is a Python object in memory.

JSON is a text format used to represent data.

Think of it like:

```
Python object
      
JSON representation
      
File / API / Network
```

---

2. Why Is JSON Important?

Modern applications constantly exchange structured data.

For example:

```
Frontend
   ↓
JSON
   ↓
Backend
```

Or:

```
Python application
   ↓
JSON
   ↓
API
```

A weather API might return:

```
{
   "city": "Lahore",
   "temperature": 32,
   "condition": "Sunny"
}
```

A Python program can convert this JSON into Python data and work with it.

JSON therefore acts as a common language between different systems.

---

3. Python's `json` Module

Python has a built-in module called:

```
json
```

Import it:

```
import json
```

No installation is required.

Example:

```
import json

data = {
    "name": "Hafsa",
    "age": 25
}
```

Now Python can convert this data into JSON.

---

4. Python Dictionary vs JSON Object

A Python dictionary:

```
data = {
    "name": "Hafsa",
    "age": 25
}
```

A JSON object:

```
{
    "name": "Hafsa",
    "age": 25
```

}

They look almost identical.

The important difference is:

Python dictionary Python object
JSON object Text-based data format

You often convert between the two.

---

5. JSON Data Types

JSON supports several basic data types.

String

"name": "Hafsa"

Number

"age": 25

Boolean

"is_active": true

Null

"middle_name": null

Array

```
"skills": [
    "Python",
    "JavaScript"
]
```

Object

```
"address": {
    "city": "Gujranwala",
    "country": "Pakistan"
}
```

This allows JSON to represent complex structures.

---

6. JSON vs Python Data Types

Python and JSON use slightly different names.

Python	JSON
`dict`	object
`list`	array
`str`	string
`int` / `float`	number

True	true
False	false
None	null

Notice the capitalization difference:

Python:

```
True
False
None
```

JSON:

```
true
false
null
```

---

7. Converting Python Data to JSON With `json.dumps()`

Suppose we have:

```
import json

user = {
    "name": "Hafsa",
```

```
    "age": 25,
    "skills": ["Python", "JavaScript"]
}
```

We can convert it into a JSON string:

```
json_data = json.dumps(user)

print(json_data)
```

Output:

```
{"name": "Hafsa", "age": 25, "skills": ["Python", "JavaScript"]}
```

The important function is:

```
json.dumps()
```

The `s` means:

string

So:

```
dumps → Python object → JSON string
```

---

8. Pretty Printing JSON

The default JSON output can be difficult to read.

You can make it easier to read using:

`json.dumps(data, indent=4)`

Example:

```
import json

user = {
    "name": "Hafsa",
    "age": 25,
    "skills": ["Python", "JavaScript"]
}

json_data = json.dumps(user, indent=4)

print(json_data)
```

Output:

```
{
    "name": "Hafsa",
    "age": 25,
    "skills": [
        "Python",
        "JavaScript"
    ]
}
```

`indent=4` makes the structure easier to read.

---

9. Converting JSON to Python With `json.loads()`

Now suppose you have a JSON string:

```
json_data = '''
{
    "name": "Hafsa",
    "age": 25
}
'''
```

You can convert it into a Python object:

```
import json

user = json.loads(json_data)

print(user)
```

Output:

```
{'name': 'Hafsa', 'age': 25}
```

Now it is a Python dictionary.

The important function is:

```
json.loads()
```

The `s` means:

string

So:

`loads → JSON string → Python object`

---

10. dumps() vs loads()

These two functions are easy to confuse.

Remember:

Python → JSON
dumps()

JSON → Python
loads()

Example:

```
json_data = json.dumps(user)
```

and:

```
user = json.loads(json_data)
```

A simple mental model:

```
Python object
     ↓
  dumps()
     ↓
JSON string
     ↓
  loads()
     ↓
Python object
```

---

11. Writing JSON to a File

You can store JSON in a file.

Suppose:

```
user = {
    "name": "Hafsa",
    "age": 25,
    "skills": ["Python", "JavaScript"]
}
```

Use:

```
import json

with open("user.json", "w", encoding="utf-8") as file:
    json.dump(user, file, indent=4)
```

The resulting file:

```
{
    "name": "Hafsa",
    "age": 25,
    "skills": [
        "Python",
        "JavaScript"
    ]
}
```

Notice the difference:

dumps() → creates JSON string
dump() → writes JSON directly to a file

---

12. dump() vs dumps()

The extra s is important.

json.dumps()

Creates a JSON string:

```
json_data = json.dumps(user)
```

json.dump()

Writes JSON directly to a file:

```
json.dump(user, file)
```

Think:

```
dump → file
dumps → string
```

---

13. Reading JSON From a File

Suppose user.json contains:

```
{
   "name": "Hafsa",
```

```
    "age": 25,
    "skills": [
        "Python",
        "JavaScript"
    ]
}
```

Read it using:

```
import json

with open("user.json", "r", encoding="utf-8") as file:
    user = json.load(file)

print(user)
```

Output:

```
{
    'name': 'Hafsa',
    'age': 25,
    'skills': ['Python', 'JavaScript']
}
```

The function:

```
json.load()
```

reads JSON from a file and converts it into Python data.

---

14. `load()` vs `loads()`

Again, remember the s.

load → file
loads → string

Example:

```
user = json.load(file)
```

reads from a file.

While:

```
user = json.loads(json_data)
```

reads from a JSON string.

---

15. The Four Important JSON Functions

You should know these four:

Function	Purpose
json.dumps()	Python → JSON string
json.loads()	JSON string → Python

json.dump()	Python → JSON file
json.load()	JSON file → Python

Mental model:

```
              JSON
            ↑   ↓
       dumps()   dump()
         ↑         ↓
      Python      File

       ↑           ↓
      loads()    load()
         ↑         ↓
       Python ↑↑↑↑↑↑↑ File
```

More simply:

```
Python object
   → dumps()
JSON string
   → loads()
Python object


Python object
   → dump()
JSON file
   → load()
Python object
```

---

16. Accessing JSON Data

After loading JSON:

```
import json

json_data = '''
{
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}
'''

user = json.loads(json_data)
```

You can access values like a dictionary:

```
print(user["name"])
```

Output:

```
Hafsa
```

And:

```
print(user["course"])
```

Output:

```
Python
```

---

17. Working With JSON Arrays

JSON arrays become Python lists.

JSON:

```
{
  "skills": [
    "Python",
    "JavaScript",
    "Flutter"
  ]
}
```

Python:

```
data = {
  "skills": [
    "Python",
    "JavaScript",
    "Flutter"
  ]
}
```

Access them:

```
print(data["skills"][0])
```

Output:

```
Python
```

Loop through them:

```
for skill in data["skills"]:
    print(skill)
```

Output:

```
Python
JavaScript
Flutter
```

---

18. Nested JSON Objects

JSON can contain objects inside objects.

Example:

```
{
    "name": "Hafsa",
    "address": {
        "city": "Gujranwala",
        "country": "Pakistan"
    }
}
```

After loading:

```
data = json.loads(json_data)
```

You can access:

```
print(data["address"]["city"])
```

Output:

```
Gujranwala
```

The structure is:

```
data
 □□□ address
     □□□ city
     □□□ country
```

This is one reason JSON is useful for complex data.

---

19. Lists of Objects

JSON can also contain a list of objects.

```
{
   "students": [
      {
         "name": "Hafsa",
         "course": "Python"
      },
      {
         "name": "Asim",
         "course": "JavaScript"
      }
   ]
}
```

Python can process it:

```
import json

data = json.loads(json_data)

for student in data["students"]:
    print(student["name"])
```

Output:

```
Hafsa
Asim
```

This structure appears frequently in APIs.

---

20. JSON and APIs

JSON is one of the most common formats used by APIs.

Imagine requesting information from a server:

```
Python application
        
    API
        
    JSON
        
Python application
```

The API might return:

```
{
```

```
    "id": 101,
    "name": "Hafsa",
    "role": "Developer"
}
```

Python can load the response and access:

```
data["name"]
```

This is a fundamental concept in web development.

---

21. Example: API-Style Data

Suppose we receive:

```
response_data = '''
{
    "id": 101,
    "title": "Python Functions",
    "category": "Python",
    "level": "Beginner"
}
'''
```

Convert it:

```
import json

resource = json.loads(response_data)

print(resource["title"])
print(resource["category"])
```

Output:

```
Python Functions
Python
```

The same idea can be used with real API responses.

---

22. JSON Configuration Files

JSON is also useful for configuration.

For example:

```
{
   "theme": "dark",
   "language": "en",
   "notifications": true
}
```

Python can load the settings:

```
import json

with open("config.json", "r", encoding="utf-8") as file:
   config = json.load(file)

print(config["theme"])
```

Output:

```
dark
```

This separates configuration from application logic.

---

23. Updating JSON Data

Suppose we load:

```
{
   "name": "Hafsa",
   "age": 25
}
```

Python:

```
import json

with open("user.json", "r", encoding="utf-8") as file:
   user = json.load(file)
```

Change the data:

```
user["age"] = 26
```

Then save it:

```
with open("user.json", "w", encoding="utf-8") as file:
   json.dump(user, file, indent=4)
```

The workflow is:

```
Read JSON
  
```

Convert to Python

Modify data

Convert back to JSON

Save file

---

24. Adding New Data

Suppose:

```
user = {
    "name": "Hafsa",
    "skills": ["Python"]
}
```

Add a skill:

```
user["skills"].append("JavaScript")
```

Then save:

```
with open("user.json", "w", encoding="utf-8") as file:
    json.dump(user, file, indent=4)
```

The file becomes:

```
{
    "name": "Hafsa",
    "skills": [
        "Python",
```

```
      "JavaScript"
    ]
}
```

---

25. JSON and `None`, `True`, `False`

Python:

```
data = {
    "active": True,
    "verified": False,
    "nickname": None
}
```

Convert:

```
json_data = json.dumps(data)
print(json_data)
```

Output:

```
{
    "active": true,
    "verified": false,
    "nickname": null
}
```

Python automatically converts these values to their JSON equivalents.

---

26. Handling JSON Errors

JSON must follow a valid structure.

For example, this is invalid JSON:

```
{
   "name": "Hafsa",
   "age": 25,
}
```

The trailing comma can cause a JSON parsing error.

When parsing JSON, you may encounter:

```
json.JSONDecodeError
```

Example:

```
import json

try:
   data = json.loads('{"name": "Hafsa",}')
except json.JSONDecodeError:
   print("Invalid JSON data.")
```

This allows your program to handle malformed JSON safely.

---

27. Common JSON Mistakes

Mistake 1: Using single quotes in JSON

Python dictionary:

```
{
   "name": "Hafsa"
}
```

JSON should use double quotes:

```
{
   "name": "Hafsa"
}
```

This is not valid JSON:

```
{'name': 'Hafsa'}
```

It may look like a Python dictionary representation, but it is not standard JSON.

Mistake 2: Using Python booleans in JSON

Incorrect JSON:

```
{
   "active": True
}
```

Correct JSON:

```
{
   "active": true
}
```

Mistake 3: Using `None` in JSON

Python:

```
None
```

JSON:

```
null
```

Mistake 4: Trailing commas

Avoid:

```
{
   "name": "Hafsa",
   "age": 25,
}
```

Use:

```
{
   "name": "Hafsa",
   "age": 25
}
```

Mistake 5: Confusing `dump()` and `dumps()`

Remember:

dump → file
dumps → string

---

28. JSON vs CSV

Both formats are useful, but they solve different problems.

Feature	CSV	JSON
Rows and columns	Excellent	Possible but less natural
Nested data	Poor	Excellent
Human readability	Good	Good
APIs	Common	Very common
Simple tables	Excellent	Possible
Complex objects	Limited	Excellent
Python dictionaries	Less direct	Natural

Example CSV:

```
Name,Age,Course
Hafsa,25,Python
Asim,26,JavaScript
```

Example JSON:

```
{
  "students": [
    {
      "name": "Hafsa",
      "age": 25,
      "course": "Python"
    },
    {
      "name": "Asim",
      "age": 26,
      "course": "JavaScript"
    }
  ]
}
```

Think:

```
CSV  table
JSON  structured objects
```

---

29. JSON vs Python Dictionary

A Python dictionary:

```
user = {
  "name": "Hafsa",
```

```
    "age": 25
}
```

is an object Python can immediately manipulate.

JSON:

```
{
    "name": "Hafsa",
    "age": 25
}
```

is a portable text representation.

So:

```
Python dictionary
      
json.dumps()
      
JSON text
      
Store / Send
      
json.loads()
      
Python dictionary
```

---

30. JSON and haas.dev

Imagine haas.dev resource data is represented as:

```
resources = {
    "resources": [
        {
            "title": "Python Functions",
            "category": "Python",
            "level": "Beginner"
        },
        {
            "title": "Working With CSV",
            "category": "Python",
            "level": "Beginner"
        },
        {
            "title": "JSON in Python",
            "category": "Python",
            "level": "Beginner"
        }
    ]
}
```

This can be saved as:

```
import json

with open("resources.json", "w", encoding="utf-8") as file:
    json.dump(resources, file, indent=4)
```

The resulting JSON could look like:

```
{
    "resources": [
        {
            "title": "Python Functions",
            "category": "Python",
```

```
            "level": "Beginner"
        },
        {
            "title": "Working With CSV",
            "category": "Python",
            "level": "Beginner"
        },
        {
            "title": "JSON in Python",
            "category": "Python",
            "level": "Beginner"
        }
    ]
}
```

Later, Python can load it:

```
with open("resources.json", "r", encoding="utf-8") as file:
    data = json.load(file)

for resource in data["resources"]:
    print(resource["title"])
```

This is similar to how real applications work with structured resource data.

---

31. A Practical JSON Workflow

When working with JSON, think:

Step 1: Identify the data structure

Ask:

Is it an object?
A list?
Nested objects?
A list of objects?

Step 2: Convert if necessary

Python JSON:

`json.dumps()`

or:

`json.dump()`

JSON Python:

`json.loads()`

or:

`json.load()`

Step 3: Access the data

For objects:

`data["name"]`

For arrays:

`data["skills"][0]`

For nested objects:

```
data["address"]["city"]
```

Step 4: Process it

You can:

```
search
filter
modify
calculate
validate
transform
```

Step 5: Save or send it

Convert the updated Python data back into JSON when needed.

---

32. Mini Project: User Profile

Create a profile:

```
import json

user = {
    "name": "Hafsa",
    "age": 25,
    "role": "Developer",
    "skills": [
        "Python",
        "JavaScript",
        "Flutter"
    ]
```

```
}

with open("profile.json", "w", encoding="utf-8") as file:
    json.dump(user, file, indent=4)

print("Profile saved.")
```

The file contains:

```
{
    "name": "Hafsa",
    "age": 25,
    "role": "Developer",
    "skills": [
        "Python",
        "JavaScript",
        "Flutter"
    ]
}
```

---

33. Mini Project: Read and Update a Profile

First load the profile:

```
import json

with open("profile.json", "r", encoding="utf-8") as file:
    user = json.load(file)
```

Update it:

```
user["skills"].append("AI")
```

Save it:

```
with open("profile.json", "w", encoding="utf-8") as file:
    json.dump(user, file, indent=4)
```

Now the profile contains the new skill.

---

34. Mini Project: Resource JSON File

Create:

```
import json

resources = {
    "resources": [
        {
            "title": "Python Functions",
            "category": "Python"
        },
        {
            "title": "Reading Files",
            "category": "Python"
        },
        {
            "title": "Working With CSV",
            "category": "Python"
        }
    ]
}

with open("resources.json", "w", encoding="utf-8") as file:
    json.dump(resources, file, indent=4)
```

```
print("Resources exported.")
```

Then read and filter them:

```
with open("resources.json", "r", encoding="utf-8") as file:
    data = json.load(file)

for resource in data["resources"]:
    if resource["category"] == "Python":
        print(resource["title"])
```

---

35. 5 Minute Challenge

Create a JSON file for a developer profile containing:

```
Name
Role
Experience
Skills
Projects
```

Example structure:

```
{
    "name": "Hafsa",
    "role": "Developer",
    "experience": 2,
    "skills": [
        "Python",
        "JavaScript"
    ],
```

```
  "projects": [
    "haas.dev",
    "Portfolio"
  ]
}
```

Then write a Python program that:

1. reads the JSON file
2. prints the name
3. prints every skill
4. prints every project
5. adds a new skill
6. saves the updated JSON

Extra challenge

Add a nested `contact` object:

```
"contact": {
  "email": "example@example.com",
  "country": "Pakistan"
}
```

Then print the country.

---

36. Exercises

Exercise 1

Create a Python dictionary and convert it into a JSON string using `json.dumps()`.

Exercise 2

Use `json.loads()` to convert that JSON string back into a Python dictionary.

Exercise 3

Save a dictionary to `data.json` using `json.dump()`.

Exercise 4

Read `data.json` using `json.load()`.

Exercise 5

Create a JSON object containing a list of five programming languages.

Exercise 6

Create a nested JSON structure containing:

```
student
 □□□ name
 □□□ age
 □□□ address
     □□□ city
     □□□ country
```

Exercise 7

Create a list of dictionaries representing five products.

Exercise 8

Save the products to JSON.

Exercise 9

Read the JSON and print products whose price is greater than 1000.

Exercise 10

Create a haas.dev resource JSON file and write a program that prints resources from a specific category.

---

37. Mini Quiz

Question 1

What does JSON stand for?

A. Java Source Object Notation

B. JavaScript Object Notation

C. JSON Structured Object Network

D. Java Standard Object Network

Answer: B

Question 2

Which module is used for JSON in Python?

A. `data`

B. `json`

C. `object`

D. `api`

Answer: B

Question 3

What does `json.dumps()` do?

A. Reads a JSON file

B. Converts Python data into a JSON string

C. Deletes JSON

D. Writes directly to a JSON file

Answer: B

Question 4

What does `json.loads()` do?

A. Converts a JSON string into Python data

B. Writes JSON to a file

C. Deletes a JSON object

D. Creates a CSV

Answer: A

Question 5

Which function writes Python data directly to a JSON file?

A. `json.write()`

B. `json.dumps()`

C. `json.dump()`

D. `json.save()`

Answer: C

Question 6

Which function reads JSON from a file?

A. `json.read()`

B. `json.load()`

C. `json.loads()`

D. `json.open()`

Answer: B

Question 7

What does Python `None` become in JSON?

A. `None`

B. `undefined`

C. `null`

D. `nil`

Answer: C

Question 8

What does a JSON array become in Python?

A. Dictionary

B. Tuple

C. List

D. Set

Answer: C

---

38. Interview Questions

1. What is JSON?

JSON is a text-based format used to represent and exchange structured data.

2. Why is JSON widely used?

It is lightweight, readable, supported by many programming languages, and works especially well for APIs and structured application data.

3. What is the difference between `dump()` and `dumps()`?

`dump()` writes JSON directly to a file, while `dumps()` returns JSON as a string.

4. What is the difference between `load()` and `loads()`?

`load()` reads JSON from a file, while `loads()` reads JSON from a string.

5. What does JSON `true` become in Python?

`True`

6. What does JSON `null` become in Python?

`None`

7. Can JSON contain nested objects?

Yes. JSON supports objects inside objects and arrays containing objects.

8. Why is JSON commonly used with APIs?

It provides a standardized way for different applications and programming languages to exchange structured data.

9. Is JSON the same as a Python dictionary?

No. A dictionary is a Python data structure, while JSON is a text-based data interchange format.

10. When might JSON be better than CSV?

When the data contains nested objects, arrays, or more complex relationships that do not naturally fit rows and columns.

---

39. Cheat Sheet

Import

`import json`

Python JSON string

`json_data = json.dumps(data)`

JSON string Python

```
data = json.loads(json_data)
```

Python → JSON file

```
with open("data.json", "w", encoding="utf-8") as file:
    json.dump(data, file, indent=4)
```

JSON file → Python

```
with open("data.json", "r", encoding="utf-8") as file:
    data = json.load(file)
```

Access a value

```
data["name"]
```

Access a nested value

```
data["address"]["city"]
```

Loop through an array

```
for item in data["skills"]:
    print(item)
```

Pretty JSON

```
json.dumps(data, indent=4)
```

Handle invalid JSON

```
try:
    data = json.loads(json_data)
except json.JSONDecodeError:
    print("Invalid JSON")
```

---

40. Key Takeaways

- JSON stands for **JavaScript Object Notation**.
- JSON is a text-based format for structured data.
- Python provides the built-in `json` module.
- JSON is widely used in APIs and web applications.
- JSON objects are similar to Python dictionaries.
- JSON arrays are similar to Python lists.
- `json.dumps()` converts Python data into a JSON string.
- `json.loads()` converts a JSON string into Python data.
- `json.dump()` writes Python data directly to a JSON file.
- `json.load()` reads JSON from a file.
- Python `True`, `False`, and `None` become JSON `true`, `false`, and `null`.
- JSON supports nested objects and arrays.
- JSON is generally better than CSV for complex or nested data.
- Always use valid JSON syntax when exchanging JSON with other systems.
- `json.JSONDecodeError` can be handled when JSON data is invalid.

The core pattern to remember is:

`Python Data`
  ↓
`json.dumps()`
  ↓
`JSON String`

and for files:

`Python Data`
  ↓
`json.dump()`

JSON File

json.load()

Python Data

JSON is one of the most important formats to understand before working with **APIs, web applications, configuration files, and real-world data exchange**.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

haas.dev

https://dev-roast-app.vercel.app/resources