Python JSON – Complete Tutorial

By Andrei Maksimov

August 10, 2021

json, python, python-course

JSON (**J**ava**S**cript **O**bject **N**otation) is a popular data format representing structured data. It is used extensively in APIs and web applications. You can use the built-in Python JSON module that provides all the necessary methods for working with JSON data. This article covers the standard Python JSON module and explains how to parse, serialize, deserialize, encode, decode, and pretty-print its data.

Table of contents

- What is JSON?
  - JSON syntax
    - Collection of name/value pairs
    - An ordered list of values
  - JSON constraints
  - What does JSON data look like?
- Working with Python JSON module
  - Serializing Python objects to JSON format
    - Serializing Python data using dump()
    - Serializing Python data using dumps()
  - Deserializing JSON data to Python object
    - Deserializing stream using load()
    - Deserializing string using loads()
  - Reading JSON data in Python
    - Reading data from a file using load()
    - Reading data from files using loads()
  - Writing JSON data into a file in Python
    - Writing data into the file using dump()
    - Writing data into files using dumps()
  - Encoding and decoding custom JSON objects in Python
    - Example of custom object encoding in Python
    - Example of custom object decoding in Python
- How to pretty print JSON data in Python?
  - Pretty printing JSON using dumps()
  - Pretty printing JSON using json.tools module
- How to sort JSON keys in Python?
- Summary

What is JSON?

JSONis a popular lightweight data-interchange format inspired by JavaScript object syntax format specified by RFC 7159 and ECMA-404. The

primary purpose of JSON format is to store and transfer data between the browser and the server, but it is widely used by microservice APIs for data exchange.

JSON syntax

The syntax of JSON is straightforward. It is built on top of two simple structures:

- A collection of name/value pairs (in Python, they are represented as a Python dictionary).
- An ordered list of values.

Here are examples of these simple universal data structures.

Collection of name/value pairs

Name/value pairs form a **JSON object** which is described using curly braces `{ }`. You can define a JSON object in the formatted form:

```
{
    "name": "James",
    "gender": "male"
    "age": 32
}
```

Or as a single string (both objects are the same):

```
{"name":"James","gender":"male","age":32}
```

An ordered list of values

An ordered list of items is defined using the square brackets `[ ]` and holds any values separated by a comma (`,`):

```
[
    "Apple",
    "Banana",
    "Cherry"
]
```

The same ordered list of items can be defined as a single string:

```
["Apple","Banana","Cherry"]
```

JSON constraints

JSON format has several constraints:

- The `name` in name/value pairs must be defined as a string in double quotes (`"`).
- The `value` must be of the valid JSON data type:
  - **String** – several plain text characters
  - **Number** – an integer
  - **Object** – a collection of JSON key/value pairs
  - **Array** – an ordered list of values
  - **Boolean** – `true` or `false`
  - **Null** – empty object
- Valid JSON object can't contain other data types, for example, `datetime`

What does JSON data look like?

Here's an example of JSON data structure:

```
{
    "firstname": "Kamil",
    "lastname": "Abdurahim",
    "age": 43,
    "speaks_languages": ["English", "French", "Amharic"],
    "programming_languages": {
        "Python": 10,
        "C++": 5,
        "Java": 6
```

```
    },
    "good_writer": false,
    "finished_this_article": null
}
```

Working with Python JSON module

JSON is a standard data exchange format used by many programming languages, including Python. JSON (JavaScript Object Notation) represents data in a human-readable text format and is easy for computers to process. Python provides a built-in module called JSON to work with JSON data. The JSON module allows you to convert Python objects into JSON strings and back up again. It also provides methods for loading and saving JSON files.

In addition, the `json` module can also be used to convert Python dictionaries into JSON objects, it contains methods for processing JSON data, including the following operations: parsing, serializing, deserializing, encoding, decoding, and pretty-printing. Overall, the `json` module makes it easy to work with JSON data in Python programming language.

Serializing Python objects to JSON format

Serialization is translating a data structure into a format that can be stored, transmitted, and reconstructed later. Applicable to Python, serialization means that we will translate Python basic data types to JSON format. The `json` module can convert Python dictionaries or lists objects into a JSON format (string).

Here's how Python `json` module handles the serialization process:

Python class	JSON type
int, long, float	number
str	string
True	true
False	false
list, tuple	array
dict	object
None	null

Python to JSON object and data type translation

There're two methods available in the Python `json` module to handle the serialization process:

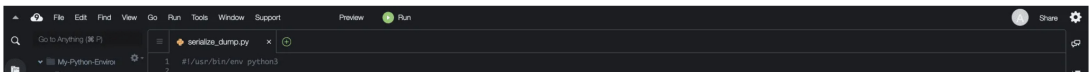
- dump() – converts Python object as a JSON formatted stream (usually used to save JSON data to the file)
- dumps() – converts Python object as a JSON formatted string (produces a Python string object which contains JSON data)

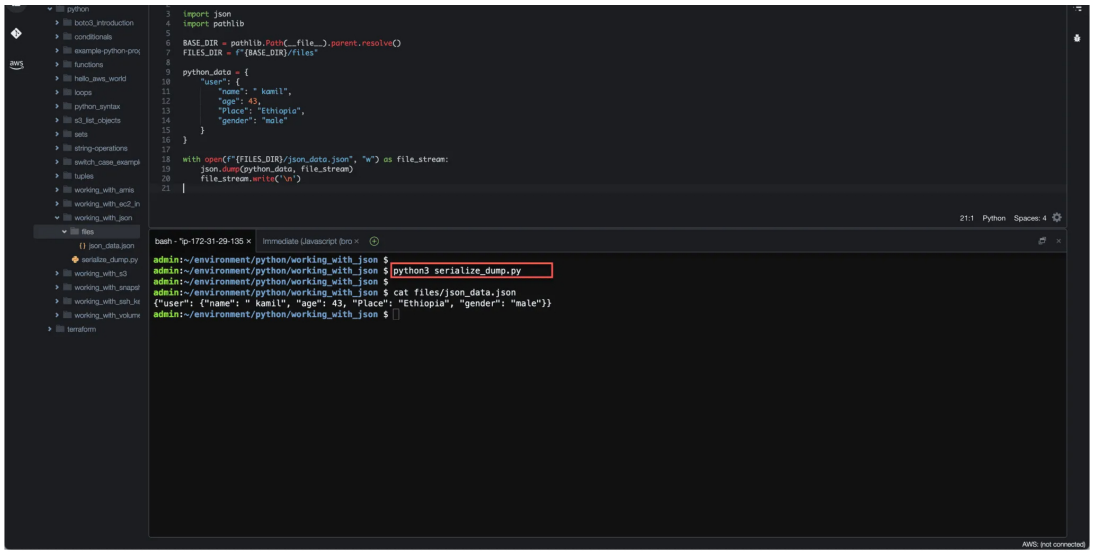
Serializing Python data using dump()

Here's an example of serializing Python data structure to the JSON stream using the `dump()` method:

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = {
    "user": {
        "name": " kamil",
        "age": 43,
        "Place": "Ethiopia",
        "gender": "male"
    }
}
# saving Python dictionary object to JSON file
with open(f"{FILES_DIR}/json_data.json", "w") as file_stream:
    json.dump(python_data, file_stream)
    file_stream.write('\n')
```

Here's an execution output:



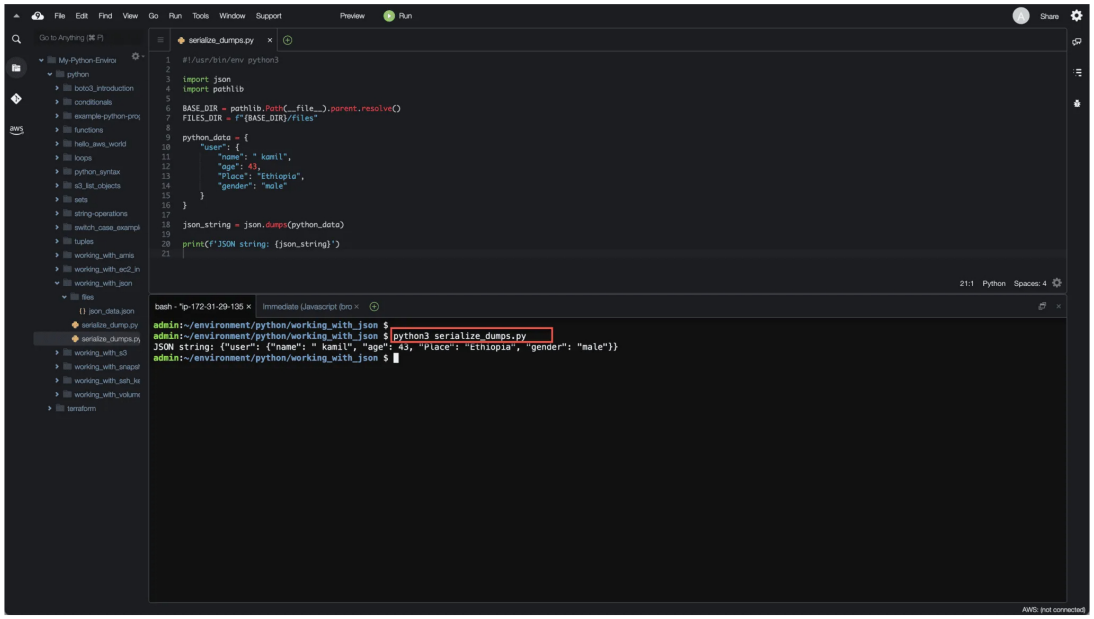


Serializing Python data using dumps()

Here's an example of serializing Python data structure to the JSON formatted Python string using the `dumps()` method:

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = {
    "user": {
        "name": " kamil",
        "age": 43,
        "Place": "Ethiopia",
        "gender": "male"
    }
}
json_string = json.dumps(python_data)
print(f'JSON string: {json_string}')
```

Here's an execution output:



Deserializing JSON data to Python object

The deserialization process is the opposite of serialization. It converts JSON data into a Python list or dictionary object.

Here's how Python `json` module handles the deserialization process:

JSON type	Python class

null	None
true	True
false	False
number (int)	int
number (real)	float
array	list
string	str
object	dict

JSON to Python object data type translation

There're two methods available in the Python `json` module to handle the deserialization process:

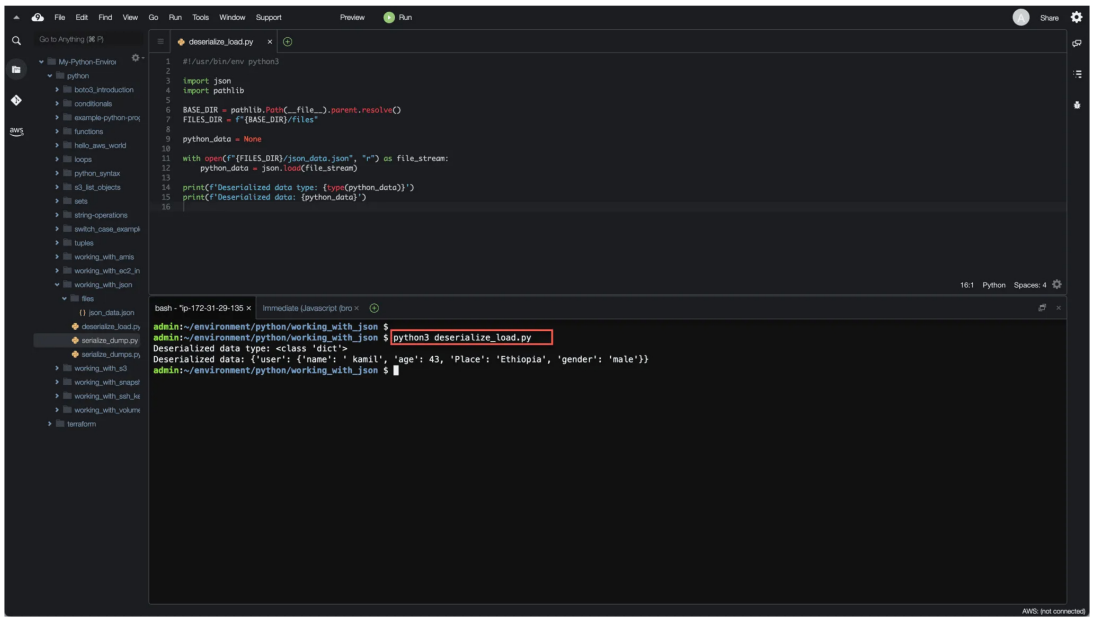
- load() – converts a **JSON** formatted stream to a Python object
- loads() – converts a **JSON** formatted string to a Python object

Deserializing stream using load()

To deserialize theJSON formatted stream to a Python object, you need to use the `load()` method:

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = None
with open(f"{FILES_DIR}/json_data.json", "r") as file_stream:
    python_data = json.load(file_stream)

print(f'Deserialized data type: {type(python_data)}')
print(f'Deserialized data: {python_data}')
```

Here's an execution output:



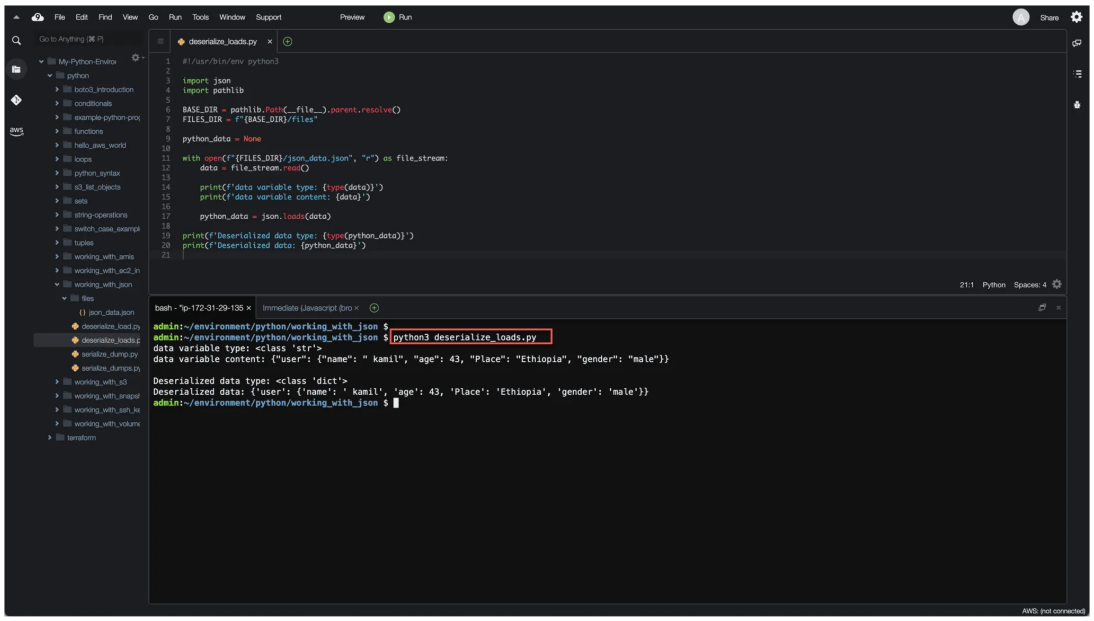
Deserializing string using loads()

Todeserialize JSON formatted string to a Python object, you need to use the `loads()` method.

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = None
# read JSON file
with open(f"{FILES_DIR}/json_data.json", "r") as file_stream:
```

```
    data = file_stream.read()

    print(f'data variable type: {type(data)}')
    print(f'data variable content: {data}')
    python_data = json.loads(data)

print(f'Deserialized data type: {type(python_data)}')
print(f'Deserialized data: {python_data}')
```

Here's an execution output:



Reading JSON data in Python

Depending on the JSON data source type (JSON formatted string orJSON formatted stream), there're two methods available in Python `json` module to handle the read operation:

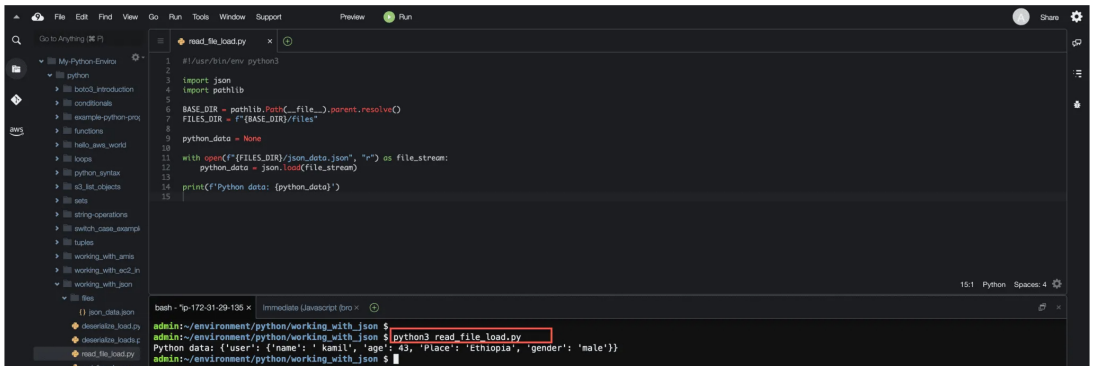
- load() – reads a **JSON** formatted stream and creates a Python object out of it
- loads() – reads a **JSON** formatted string and creates a Python object out of it

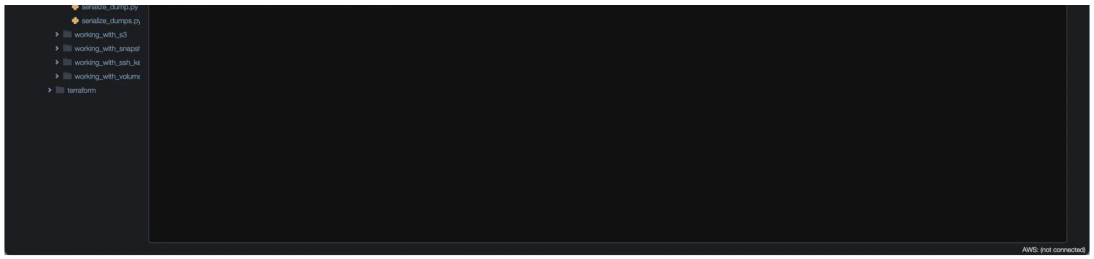
Reading data from a file using load()

You need to use the `load()` method to read theJSON formatted stream and convert it into a Python object.JSON formatted stream is returned by the Python built-in `open()` method. For more information about file operations, we recommend the Working with Files in Python article.

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = None
with open(f"{FILES_DIR}/json_data.json", "r") as file_stream:
    python_data = json.load(file_stream)

print(f'Python data: {python_data}')
```

Here's an execution output:



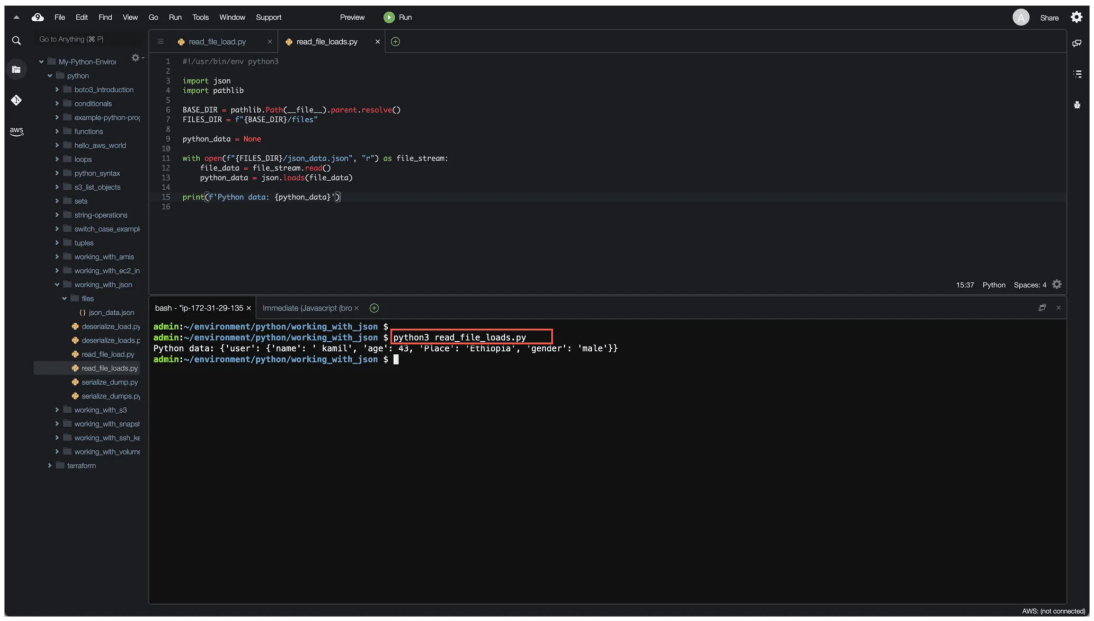


Reading data from files using loads()

You need to use the `loads()` method to read theJSON formatted string and convert it into a Python object. JSON formatted string can be obtained from the file using the Python built-in `open()` method. We recommend the Working with Files in Python article for more information about file operations.

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = None
with open(f"{FILES_DIR}/json_data.json", "r") as file_stream:
    python_data = json.load(file_stream)

print(f'Python data: {python_data}')
```

Here's an execution output:



Writing JSON data into a file in Python

Depending on the JSON data type (JSON formatted string orJSON formatted stream), there're two methods available in Python `json` module to handle the write operation:

- dump() – converts Python object as a JSON formatted stream (usually used to save data straight to the file)
- dumps() – converts Python object as a JSON formatted string (produces a Python string object which can be written to the file)

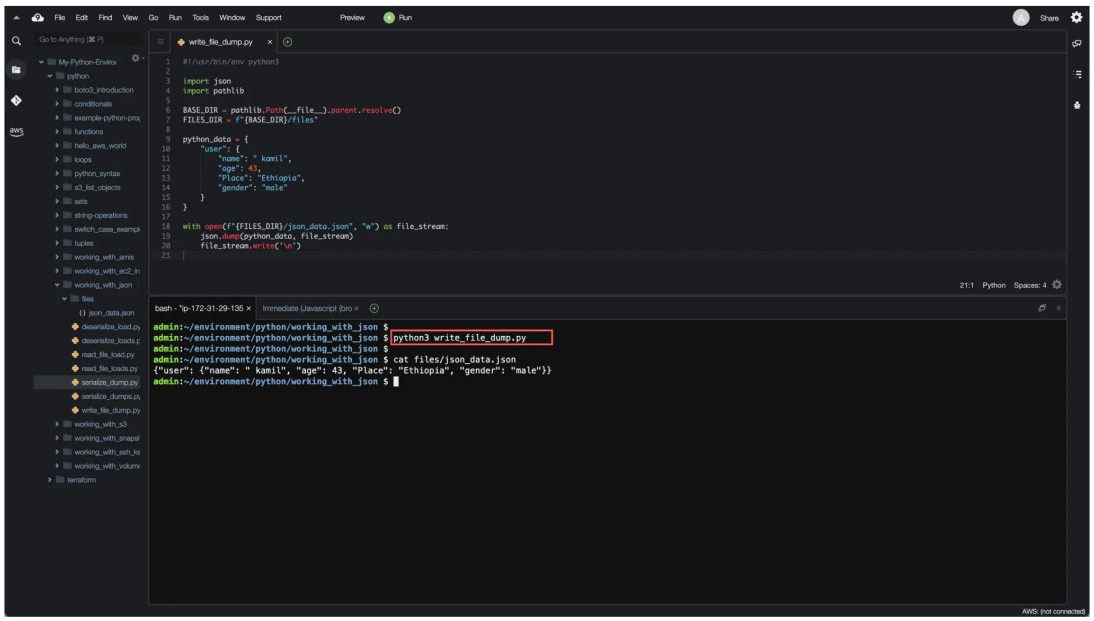
Writing data into the file using dump()

To write theJSON formatted stream to a file, you need to use the `json.dump()` method in combination with the Python built-in `open()` method.

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = {
    "user": {
```

```
        "name": " kamil",
        "age": 43,
        "Place": "Ethiopia",
        "gender": "male"
    }
}
with open(f"{FILES_DIR}/json_data.json", "w") as file_stream:
    json.dump(python_data, file_stream)
    file_stream.write('\n')
```

Here's an example output:

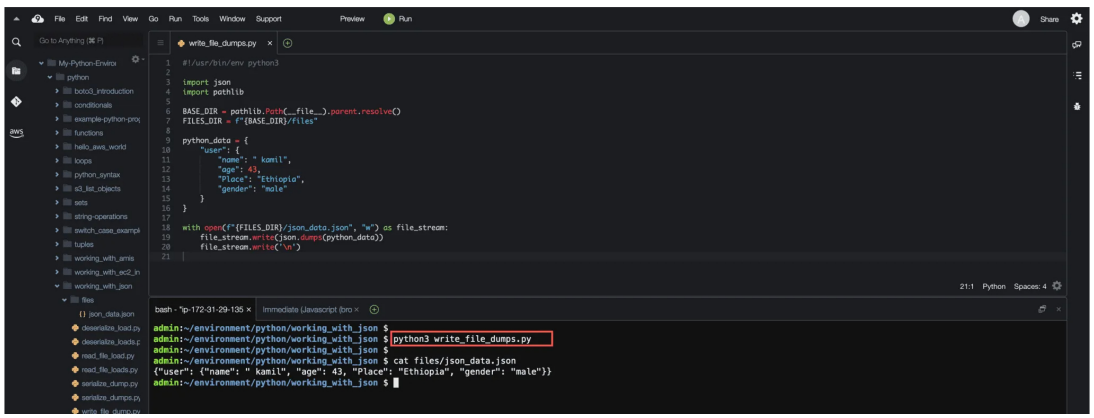


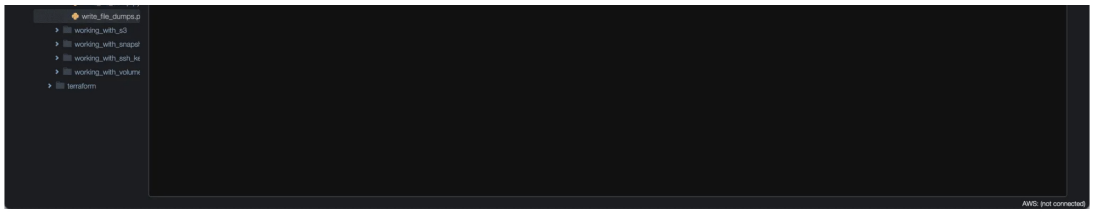
Writing data into files using dumps()

To write the JSON formatted string into a file, you need to use the `json.dumps()` method in combination with the Python built-in `open()` method.

```
#!/usr/bin/env python3
import json
import pathlib
BASE_DIR = pathlib.Path(__file__).parent.resolve()
FILES_DIR = f"{BASE_DIR}/files"
python_data = {
    "user": {
        "name": " kamil",
        "age": 43,
        "Place": "Ethiopia",
        "gender": "male"
    }
}
with open(f"{FILES_DIR}/json_data.json", "w") as file_stream:
    file_stream.write(json.dumps(python_data))
    file_stream.write('\n')
```

Here's an example output:





Encoding and decoding custom JSON objects in Python

Although the `json` module can handle most built-in Python types. It doesn't understand how to encode custom data types by default. If you need to encode a custom object, you can extend a JSONEncoder class and override its `default()` method. This method is used to JSONinfy custom objects.

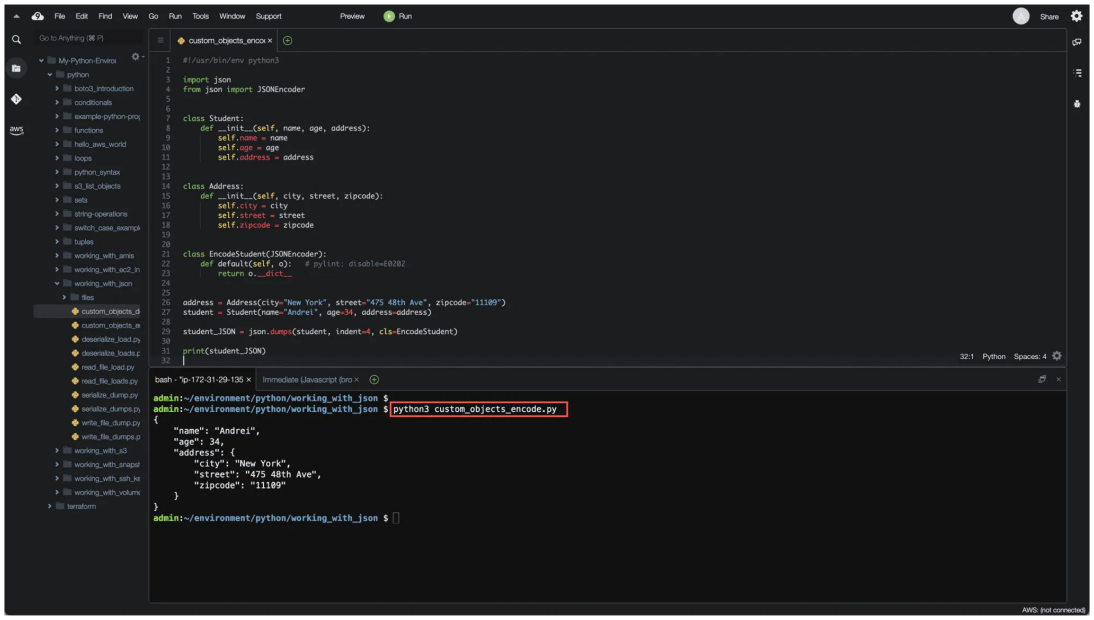
Example of custom object encoding in Python

Let's take a look at the example. Suppose you have a couple of user-defined classes: a `Student` and an `Address` And you want to serialize them to a JSON document.

That's how you can do it:

```
#!/usr/bin/env python3
import json
from json import JSONEncoder

class Student:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address

class Address:
    def __init__(self, city, street, zipcode):
        self.city = city
        self.street = street
        self.zipcode = zipcode

class EncodeStudent(JSONEncoder):
    def default(self, o):   # pylint: disable=E0202
        return o.__dict__

address = Address(city="New York", street="475 48th Ave", zipcode="11109")
student = Student(name="Andrei", age=34, address=address)
student_JSON = json.dumps(student, indent=4, cls=EncodeStudent)
print(student_JSON)
```

Here's an example output:



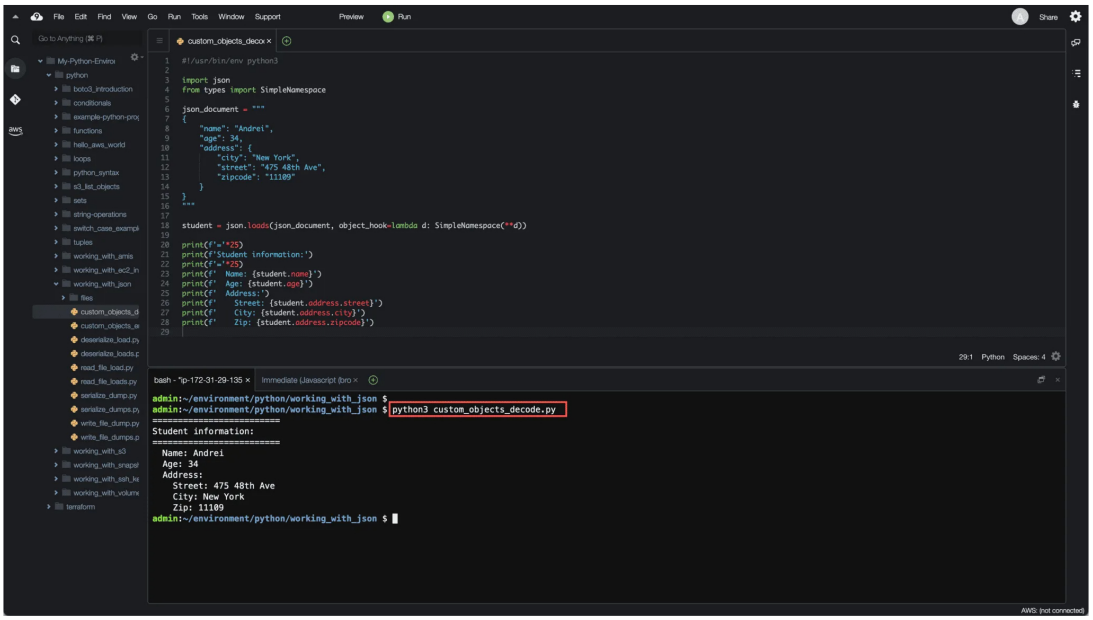
Example of custom object decoding in Python

If you need to convert the JSON document into some other Python object (i.e., not the default dictionary), the simplest way of doing that is to use the SimpleNamespace class and the object_hook argument of the `load()` or the `loads()` method.

The

```
#!/usr/bin/env python3
import json
from types import SimpleNamespace
json_document = """
{
    "name": "Andrei",
    "age": 34,
    "address": {
        "city": "New York",
        "street": "475 48th Ave",
        "zipcode": "11109"
    }
}
"""
student = json.loads(json_document, object_hook=lambda d: SimpleNamespace(**d))
print(f'='*25)
print(f'Student information:')
print(f'='*25)
print(f'  Name: {student.name}')
print(f'  Age: {student.age}')
print(f'  Address:')
print(f'    Street: {student.address.street}')
print(f'    City: {student.address.city}')
print(f'    Zip: {student.address.zipcode}')
```

Here's an example output:



How to pretty print JSON data in Python?

There are two methods available for you to print a pretty JSON message:

- Use `indent` argument of the `dumps()` method – this is an ideal option when you're printing JSON messages from your Python code
- Use the json.tool module – you can use this method when you need to format a JSON message in your shell

Pretty printing JSON using dumps()

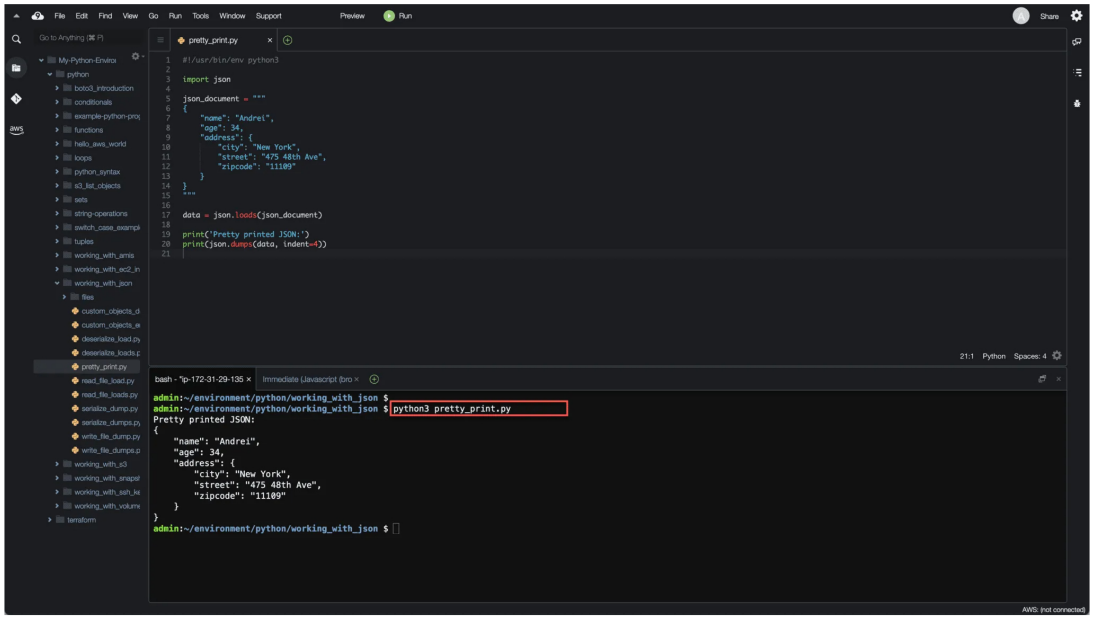
Pretty printing JSON using the `dumps()` method is straightforward:

```
#!/usr/bin/env python3
import json
json_document = """
{
    "name": "Andrei",
    "age": 34,
```

```
    "address": {
        "city": "New York",
        "street": "475 48th Ave",
        "zipcode": "11109"
    }
}
"""
data = json.loads(json_document)
print('Pretty printed JSON:')
print(json.dumps(data, indent=4))
```

The `indent` argument defines an **indent**ation (number or spaces) for JSON objects during the print operation.

Here's an execution output:

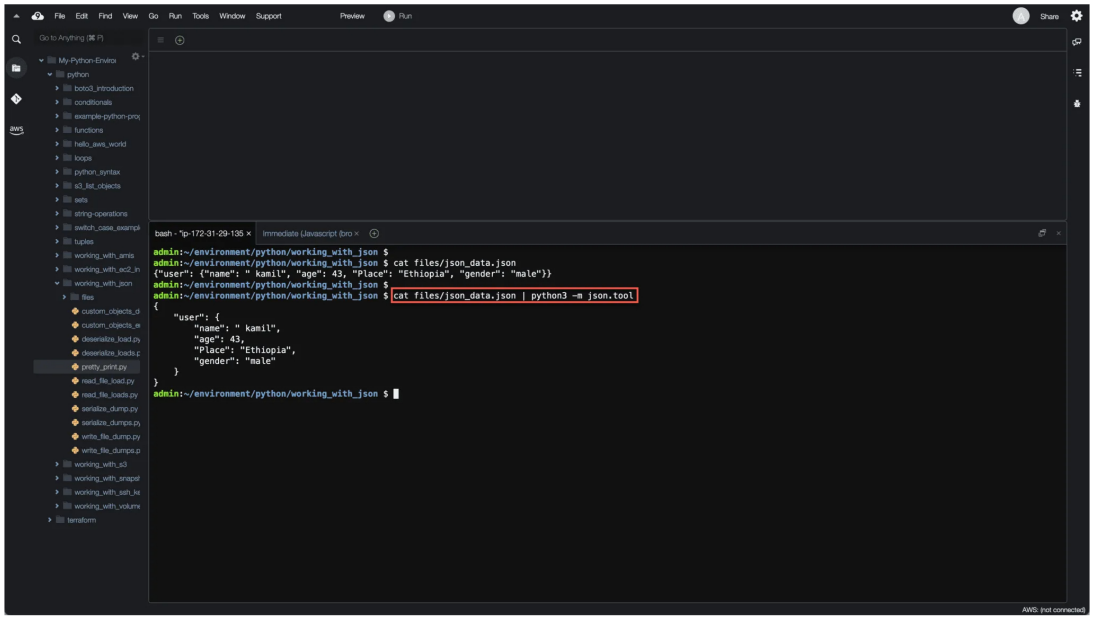


Pretty printing JSON using json.tools module

To format JSON documents in your shell without using third-party tools, you can use `json.tool` Python module:

```
cat json_document.json | python3 -m json.tool
```

Here's an example:

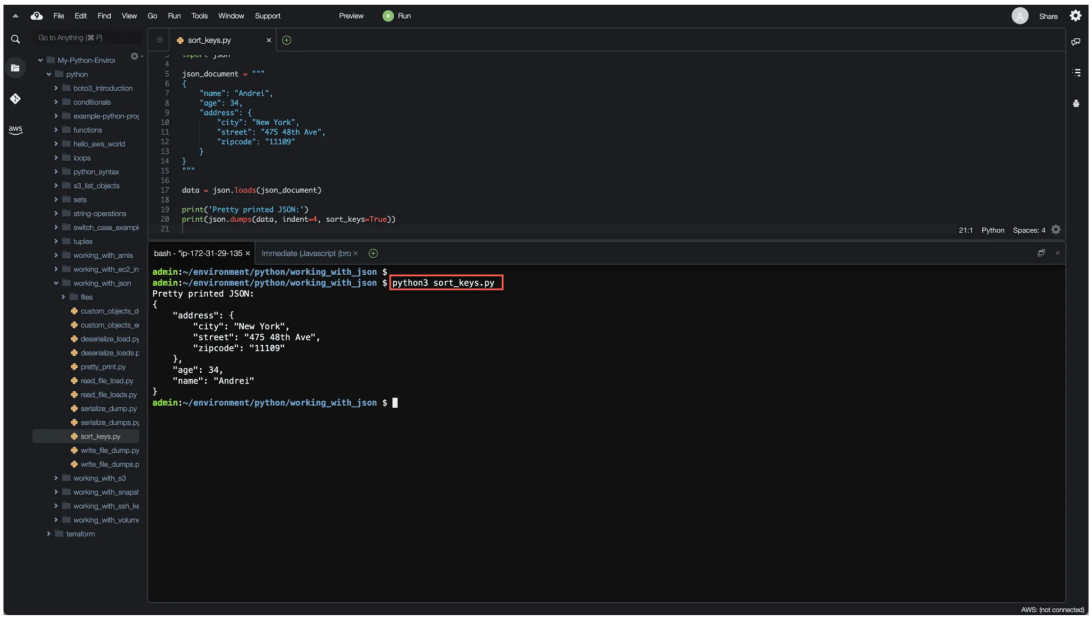


How to sort JSON keys in Python?

When you need to sort JSON keys (sort JSON objects by name), you can set the `sort_keys` argument to `True` in the `dumps()` method:

```
#!/usr/bin/env python3
import json
json_document = """
{
    "name": "Andrei",
    "age": 34,
    "address": {
        "city": "New York",
        "street": "475 48th Ave",
        "zipcode": "11109"
    }
}
"""
data = json.loads(json_document)
print('Pretty printed JSON:')
print(json.dumps(data, indent=4, sort_keys=True))
```

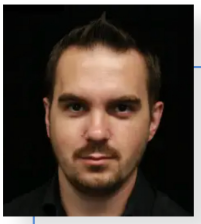
Here's an execution output:



Summary

This article covered the basics and advanced JSON processing technics in Python, includingparsing, serializing, deserializing, encoding, decoding, and pretty-printing JSON data using Python. An ability to process JSON in Python is a must-have hands-on skill for every AWS automation engineer, for example, when you need to deal with DynamoDB stream processing in your AWS Lambda function.

Previous Next

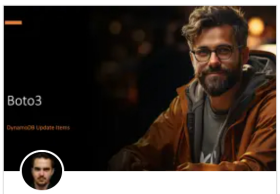


Andrei Maksimov

I'm a passionate Cloud Infrastructure Architect with more than 20 years of experience in IT. In addition to the tech, I'm covering Personal Finance topics at https://amaksimov.com.

Any of my posts represent my personal experience and opinion about the topic.

Related Posts



Andrei Maksimov | 07/14/2023

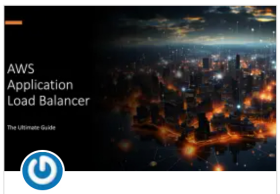
Boto3 DynamoDB Update Item – Comprehensive Guide

Andrei Maksimov | 07/13/2023

Mastering AWS EC2 Instances – Comprehensive Guide

Andrei Maksimov | 07/10/2023

Boto3 DynamoDB create_backup – Easy DynamoDB Backups



Automation User | 07/10/2023

AWS Application Load Balancer – The Ultimate Guide

Subscribe now to get the latest updates!

Name | Email | Sign Up