Working With JSON Data in Python

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Working With JSON Data in Python**

Since its inception, JSON has quickly become the de facto standard for information exchange. Chances are you're here because you need to transport some data from here to there. Perhaps you're gathering information through an API or storing your data in a document database. One way or another, you're up to your neck in JSON, and you've got to Python your way out.

Luckily, this is a pretty common task, and—as with most common tasks—Python makes it almost disgustingly easy. Have no fear, fellow Pythoneers and Pythonistas. This one's gonna be a breeze!

> **So, we use JSON to store and exchange data?** Yup, you got it! It's nothing more than a standardized format the community uses to pass data around. Keep in mind, JSON isn't the only format available for this kind of work, but XML and YAML are probably the only other ones worth mentioning in the same breath.

A (Very) Brief History of JSON

Not so surprisingly, **J**ava**S**cript **O**bject **N**otation was inspired by a subset of the JavaScript programming language dealing with object literal syntax. They've got a nifty website that explains the whole thing. Don't worry though: JSON has long since become language agnostic and exists as its own standard, so we can thankfully avoid JavaScript for the sake of this discussion.

Ultimately, the community at large adopted JSON because it's easy for both humans and machines to create and understand.

Look, it's JSON!

Get ready. I'm about to show you some real life JSON—just like you'd see out there in the wild. It's okay: JSON is supposed to be readable by anyone who's used a C-style language, and Python is a C-style language…so that's you!

```
{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobbies": ["running", "sky diving", "singing"],
    "age": 35,
    "children": [
        {
            "firstName": "Alice",
            "age": 6
        },
        {
            "firstName": "Bob",
            "age": 8
        }
    ]
}
```

As you can see, JSON supports primitive types, like strings and numbers, as well as nested lists and objects.

> **Wait, that looks like a Python dictionary!** I know, right? It's pretty much universal object notation at this point, but I don't think UON rolls off the tongue quite as nicely. Feel free to discuss alternatives in the comments.

Whew! You survived your first encounter with some wild JSON. Now you just need to learn how to tame it.

Python Supports JSON Natively!

Python comes with a built-in package called `json` for encoding and decoding JSON data.

Just throw this little guy up at the top of your file:

```
import json
```

A Little Vocabulary

The process of encoding JSON is usually called **serialization**. This term refers to the transformation of data into a *series of bytes* (hence *serial*) to be stored or transmitted across a network. You may also hear the term **marshaling**, but that's a whole other discussion. Naturally, **deserialization** is the reciprocal process of decoding data that has been stored or delivered in the JSON standard.

> **Yikes! That sounds pretty technical.** Definitely. But in reality, all we're talking about here is *reading* and *writing*. Think of it like this: *encoding* is for *writing* data to disk, while *decoding* is for *reading* data into memory.

Serializing JSON

What happens after a computer processes lots of information? It needs to take a data dump. Accordingly, the `json` library exposes the `dump()` method for writing data to files. There is also a `dumps()` method (pronounced as “dump-s”) for writing to a Python string.

Simple Python objects are translated to JSON according to a fairly intuitive conversion.

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

A Simple Serialization Example

Imagine you’re working with a Python object in memory that looks a little something like this:

```
data = {
    "president": {
        "name": "Zaphod Beeblebrox",
        "species": "Betelgeusian"
    }
}
```

It is critical that you save this information to disk, so your mission is to write it to a file.

Using Python’s context manager, you can create a file called `data_file.json` and open it in write mode. (JSON files conveniently end in a `.json` extension.)

```
with open("data_file.json", "w") as write_file:
    json.dump(data, write_file)
```

Note that `dump()` takes two positional arguments: (1) the data object to be serialized, and (2) the file-like object to which the bytes will be written.

Or, if you were so inclined as to continue using this serialized JSON data in your program, you could write it to a native Python `str` object.

```
json_string = json.dumps(data)
```

Notice that the file-like object is absent since you aren’t actually writing to disk. Other than that, `dumps()` is just like `dump()`.

Hooray! You’ve birthed some baby JSON, and you’re ready to release it out into the wild to grow big and strong.

Some Useful Keyword Arguments

Remember, JSON is meant to be easily readable by humans, but readable syntax isn't enough if it's all squished together. Plus you've probably got a different programming style than me, and it might be easier for you to read code when it's formatted to your liking.

> **NOTE:** Both the `dump()` and `dumps()` methods use the same keyword arguments.

The first option most people want to change is whitespace. You can use the `indent` keyword argument to specify the indentation size for nested structures. Check out the difference for yourself by using `data`, which we defined above, and running the following commands in a console:

console:

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

Another formatting option is the `separators` keyword argument. By default, this is a 2-tuple of the separator strings `(", ", ": ")`, but a common alternative for compact JSON is `(",", ":")`. Take a look at the sample JSON again to see where these separators come into play.

There are others, like `sort_keys`, but I have no idea what that one does. You can find a whole list in the docs if you're curious.

Deserializing JSON

Great, looks like you've captured yourself some wild JSON! Now it's time to whip it into shape. In the `json` library, you'll find `load()` and `loads()` for turning JSON encoded data into Python objects.

Just like serialization, there is a simple conversion table for deserialization, though you can probably guess what it looks like already.

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

Technically, this conversion isn't a perfect inverse to the serialization table. That basically means that if you encode an object now and then decode it again later, you may not get exactly the same object back. I imagine it's a bit like teleportation: break my molecules down over here and put them back together over there. Am I still the same person?

In reality, it's probably more like getting one friend to translate something into Japanese and another friend to translate it back into English. Regardless, the simplest example would be encoding a `tuple` and getting back a `list` after decoding, like so:

```
>>> blackjack_hand = (8, "Q")
>>> encoded_hand = json.dumps(blackjack_hand)
>>> decoded_hand = json.loads(encoded_hand)

>>> blackjack_hand == decoded_hand
False
>>> type(blackjack_hand)
<class 'tuple'>
>>> type(decoded_hand)
<class 'list'>
>>> blackjack_hand == tuple(decoded_hand)
True
```

A Simple Deserialization Example

This time, imagine you've got some data stored on disk that you'd like to manipulate in memory. You'll still use the context manager, but this time you'll open up the existing `data_file.json` in read mode.

```
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)
```

Things are pretty straightforward here, but keep in mind that the result of this method could return any of the allowed data types from the conversion table. This is only important if you're loading in data you haven't seen before. In most cases, the root object will be a `dict` or a `list`.

If you've pulled JSON data in from another program or have otherwise obtained a string of JSON formatted data in Python, you can easily deserialize that with `loads()`, which naturally loads from a string:

```
json_string = """
{
    "researcher": {
        "name": "Ford Prefect",
        "species": "Betelgeusian",
        "relatives": [
            {
                "name": "Zaphod Beeblebrox",
                "species": "Betelgeusian"
            }
        ]
    }
}
"""
data = json.loads(json_string)
```

Voilà! You've tamed the wild JSON, and now it's under your control. But what you do with that power is up to you. You could feed it, nurture it, and even teach it tricks. It's not that I don't trust you…but keep it on a leash, okay?

A Real World Example (sort of)

For your introductory example, you'll use JSONPlaceholder, a great source of fake JSON data for practice purposes.

First create a script file called `scratch.py`, or whatever you want. I can't really stop you.

You'll need to make an API request to the JSONPlaceholder service, so just use the `requests` package to do the heavy lifting. Add these imports at the top of your file:

```
import json
import requests
```

Now, you're going to be working with a list of TODOs cuz like…you know, it's a rite of passage or whatever.

Go ahead and make a request to the JSONPlaceholder API for the `/todos` endpoint. If you're unfamiliar with `requests`, there's actually a handy `json()` method that will do all of the work for you, but you can practice using the `json` library to deserialize the `text` attribute of the response object. It should look something like this:

```
response = requests.get("https://jsonplaceholder.typicode.com/todos")
todos = json.loads(response.text)
```

You don't believe this works? Fine, run the file in interactive mode and test it for yourself. While you're at it, check the type of `todos`. If you're feeling adventurous, take a peek at the first 10 or so items in the list.

```
>>> todos == response.json()
True
>>> type(todos)
<class 'list'>
>>> todos[:10]
...
```

See, I wouldn't lie to you, but I'm glad you're a skeptic.

> **What's interactive mode?** Ah, I thought you'd never ask! You know how you're always jumping back and forth between the your editor and the terminal? Well, us sneaky Pythoneers use the `-i` interactive flag when we run the script. This is a great little trick for testing code because it runs the script and then opens up an interactive command prompt with access to all the data from the script!

All right, time for some action. You can see the structure of the data by visiting the endpoint in a browser, but here's a sample TODO:

```
{
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
}
```

There are multiple users, each with a unique `userId`, and each task has a Boolean `completed` property. Can you determine which users have completed the most tasks?

```
# Map of userId to number of complete TODOs for that user
todos_by_user = {}

# Increment complete TODOs count for each user.
for todo in todos:
    if todo["completed"]:
        try:
            # Increment the existing user's count.
```

```
            todos_by_user[todo["userId"]] += 1
        except KeyError:
            # This user has not been seen. Set their count to 1.
            todos_by_user[todo["userId"]] = 1

# Create a sorted list of (userId, num_complete) pairs.
top_users = sorted(todos_by_user.items(),
                   key=lambda x: x[1], reverse=True)

# Get the maximum number of complete TODOs.
max_complete = top_users[0][1]

# Create a list of all users who have completed
# the maximum number of TODOs.
users = []
for user, num_complete in top_users:
    if num_complete < max_complete:
        break
    users.append(str(user))

max_users = " and ".join(users)
```

Yeah, yeah, your implementation is better, but the point is, you can now manipulate the JSON data as a normal Python object!

I don't know about you, but when I run the script interactively again, I get the following results:

```
>>> s = "s" if len(users) > 1 else ""
>>> print(f"user{s} {max_users} completed {max_complete} TODOs")
users 5 and 10 completed 12 TODOs
```

That's cool and all, but you're here to learn about JSON. For your final task, you'll create a JSON file that contains the **completed** TODOs for each of the users who completed the maximum number of TODOs.

All you need to do is filter `todos` and write the resulting list to a file. For the sake of originality, you can call the output file `filtered_data_file.json`. There are may ways you could go about this, but here's one:

```
# Define a function to filter out completed TODOs
# of users with max completed TODOS.
def keep(todo):
    is_complete = todo["completed"]
    has_max_count = str(todo["userId"]) in users
    return is_complete and has_max_count

# Write filtered TODOs to file.
with open("filtered_data_file.json", "w") as data_file:
    filtered_todos = list(filter(keep, todos))
    json.dump(filtered_todos, data_file, indent=2)
```

Perfect, you've gotten rid of all the data you don't need and saved the good stuff to a brand new file! Run the script again and check out `filtered_data_file.json` to verify everything worked. It'll be in the same directory as `scratch.py` when you run it.

Now that you've made it this far, I bet you're feeling like some pretty hot stuff, right? Don't get cocky: humility is a virtue. I am inclined to agree with you though. So far, it's been smooth sailing, but you might want to batten down the hatches for this last leg of the journey.

Encoding and Decoding Custom Python Objects

What happens when we try to serialize the `Elf` class from that Dungeons & Dragons app you're working on?

```
class Elf:
    def __init__(self, level, ability_scores=None):
        self.level = level
        self.ability_scores = {
            "str": 11, "dex": 12, "con": 10,
            "int": 16, "wis": 14, "cha": 13
        } if ability_scores is None else ability_scores
        self.hp = 10 + self.ability_scores["con"]
```

Not so surprisingly, Python complains that `Elf` isn't *serializable* (which you'd know if you've ever tried to tell an Elf otherwise):

```
>>> elf = Elf(level=4)
>>> json.dumps(elf)
TypeError: Object of type 'Elf' is not JSON serializable
```

Although the `json` module can handle most built-in Python types, it doesn't understand how to encode customized data types by default. It's like trying to fit a square peg in a round hole—you need a buzzsaw and parental supervision.

Simplifying Data Structures

Now, the question is how to deal with more complex data structures. Well, you could try to encode and decode the JSON by hand, but there's a slightly more clever solution that'll save you some work. Instead of going straight from the custom data type to JSON, you can throw in an intermediary step.

All you need to do is represent your data in terms of the built-in types `json` already understands. Essentially, you translate the more complex object into a simpler representation, which the `json` module then translates into JSON. It's like the transitive property in mathematics: if A = B and B = C, then A = C.

To get the hang of this, you'll need a complex object to play with. You could use any custom class you like, but Python has a built-in type called `complex` for representing complex numbers, and it isn't serializable by default. So, for the sake of these examples, your complex object is going to be a `complex` object. Confused yet?

```
>>> z = 3 + 8j
>>> type(z)
<class 'complex'>
>>> json.dumps(z)
TypeError: Object of type 'complex' is not JSON serializable
```

> **Where do complex numbers come from?** You see, when a real number and an imaginary number love each other very much, they add together to produce a number which is (justifiably) called *complex*.

A good question to ask yourself when working with custom types is **What is the minimum amount of information necessary to recreate this object?** In the case of complex numbers, you

only need to know the real and imaginary parts, both of which you can access as attributes on the `complex` object:

```
>>> z.real
3.0
>>> z.imag
8.0
```

Passing the same numbers into a `complex` constructor is enough to satisfy the `__eq__` comparison operator:

```
>>> complex(3, 8) == z
True
```

Breaking custom data types down into their essential components is critical to both the serialization and deserialization processes.

Encoding Custom Types

To translate a custom object into JSON, all you need to do is provide an encoding function to the `dump()` method's `default` parameter. The `json` module will call this function on any objects that aren't natively serializable. Here's a simple decoding function you can use for practice:

```
def encode_complex(z):
    if isinstance(z, complex):
        return (z.real, z.imag)
    else:
        type_name = z.__class__.__name__
        raise TypeError(f"Object of type '{type_name}' is not JSON
serializable")
```

Notice that you're expected to raise a `TypeError` if you don't get the kind of object you were expecting. This way, you avoid accidentally serializing any Elves. Now you can try encoding complex objects for yourself!

```
>>> json.dumps(9 + 5j, default=encode_complex)
'[9.0, 5.0]'
>>> json.dumps(elf, default=encode_complex)
TypeError: Object of type 'Elf' is not JSON serializable
```

> **Why did we encode the complex number as a `tuple`?** Great question! That certainly wasn't the only choice, nor is it necessarily the best choice. In fact, this wouldn't be a very good representation if you ever wanted to decode the object later, as you'll see shortly.

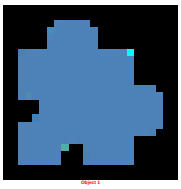
The other common approach is to subclass the standard `JSONEncoder` and override its `default()` method:

```
class ComplexEncoder(json.JSONEncoder):
    def default(self, z):
        if isinstance(z, complex):
            return (z.real, z.imag)
        else:
            return super().default(z)
```

Instead of raising the `TypeError` yourself, you can simply let the base class handle it. You can use this either directly in the `dump()` method via the `cls` parameter or by creating an instance of the encoder and calling its `encode()` method:

```
>>> json.dumps(2 + 5j, cls=ComplexEncoder)
'[2.0, 5.0]'

>>> encoder = ComplexEncoder()
>>> encoder.encode(3 + 6j)
'[3.0, 6.0]'
```



Decoding Custom Types

While the real and imaginary parts of a complex number are absolutely necessary, they are actually not quite sufficient to recreate the object. This is what happens when you try encoding a complex number with the `ComplexEncoder` and then decoding the result:

```
>>> complex_json = json.dumps(4 + 17j, cls=ComplexEncoder)
>>> json.loads(complex_json)
[4.0, 17.0]
```

All you get back is a list, and you'd have to pass the values into a `complex` constructor if you wanted that complex object again. Recall our discussion about teleportation. What's missing is *metadata*, or information about the type of data you're encoding.

I suppose the question you really ought ask yourself is **What is the minimum amount of information that is both *necessary* and *sufficient* to recreate this object?**

The `json` module expects all custom types to be expressed as `objects` in the JSON standard. For variety, you can create a JSON file this time called `complex_data.json` and add the following `object` representing a complex number:

```
{
    "__complex__": true,
    "real": 42,
    "imag": 36
}
```

See the clever bit? That `"__complex__"` key is the metadata we just talked about. It doesn't really matter what the associated value is. To get this little hack to work, all you need to do is verify that the key exists:

```
def decode_complex(dct):
    if "__complex__" in dct:
        return complex(dct["real"], dct["imag"])
    return dct
```

If `"__complex__"` isn't in the dictionary, you can just return the object and let the default decoder deal with it.

Every time the `load()` method attempts to parse an `object`, you are given the opportunity to intercede before the default decoder has its way with the data. You can do this by passing your decoding function to the `object_hook` parameter.

Now play the same kind of game as before:

```
>>> with open("complex_data.json") as complex_data:
...     data = complex_data.read()
...     z = json.loads(data, object_hook=decode_complex)
...
>>> type(z)
<class 'complex'>
```

While `object_hook` might feel like the counterpart to the `dump()` method's `default` parameter, the analogy really begins and ends there.

This doesn't just work with one object either. Try putting this list of complex numbers into `complex_data.json` and running the script again:

```
[
  {
    "__complex__":true,
    "real":42,
    "imag":36
  },
  {
    "__complex__":true,
    "real":64,
    "imag":11
  }
]
```

If all goes well, you'll get a list of `complex` objects:

```
>>> with open("complex_data.json") as complex_data:
...     data = complex_data.read()
...     numbers = json.loads(data, object_hook=decode_complex)
...
>>> numbers
[(42+36j), (64+11j)]
```

You could also try subclassing `JSONDecoder` and overriding `object_hook`, but it's better to stick with the lightweight solution whenever possible.

All done!

Congratulations, you can now wield the mighty power of JSON for any and all of your nefarious Python needs.

While the examples you've worked with here are certainly contrived and overly simplistic, they illustrate a workflow you can apply to more general tasks:

1. Import the `json` package.
2. Read the data with `load()` or `loads()`.

3. Process the data.
4. Write the altered data with `dump()` or `dumps()`.

What you do with your data once it's been loaded into memory will depend on your use case. Generally, your goal will be gathering data from a source, extracting useful information, and passing that information along or keeping a record of it.

Today you took a journey: you captured and tamed some wild JSON, and you made it back in time for supper! As an added bonus, learning the `json` package will make learning `pickle` and `marshal` a snap.

Good luck with all of your future Pythonic endeavors!