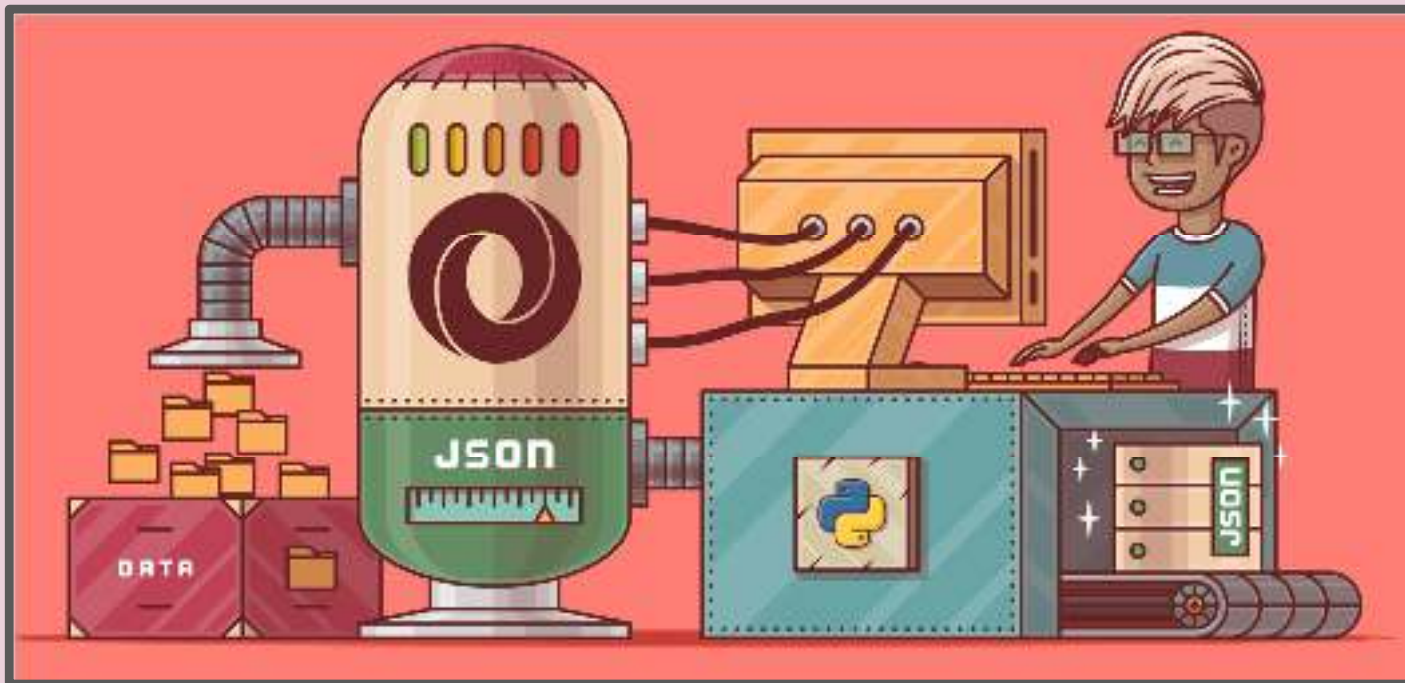


Working With JSON Data in Python



Welcome



- SEO Instructor

Mr. Kurt Seiffert

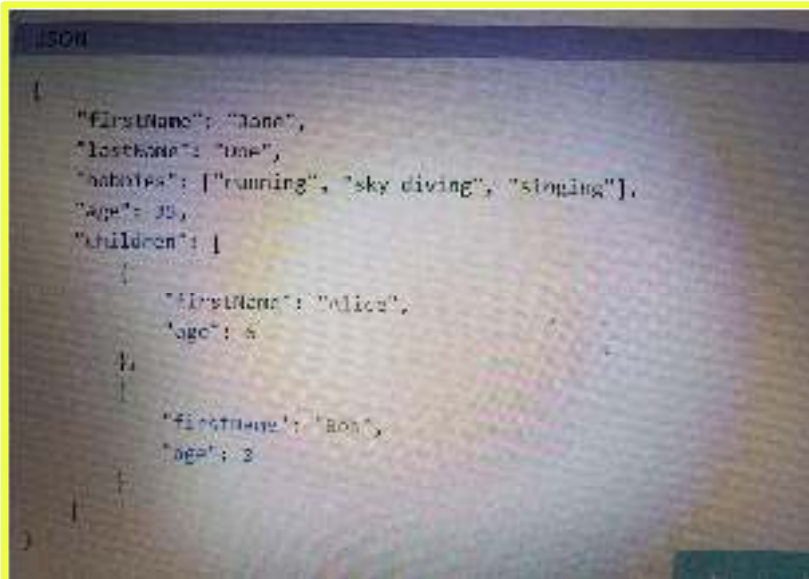
Office Hours Monday-Thursday 10a.m. - 11:00a.m.

A (Very) Brief History of JSON

- ❑ Not so surprisingly, JavaScript Object Notation was inspired by a subset of the [JavaScript programming language](#) dealing with object literal syntax. They've got a [nifty website \(json.org\)](#) 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**.



```
{
  "firstName": "Jane",
  "lastName": "Doe",
  "hobbies": ["running", "sky diving", "singing"],
  "age": 35,
  "children": [
    {
      "firstName": "Alice",
      "age": 6
    },
    {
      "firstName": "Bob",
      "age": 3
    }
  ]
}
```

As you can see, **JSON** supports primitive types, like [strings](#) and [numbers](#), as well as nested lists and objects.

Python Supports JSON Natively!

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

Add this at the top of your file:

```
import json
```

Buzz Words for Understanding

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.

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)
```

A Simple Serialization Example (Con't)

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()`.

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:

```
>>>
```

```
>>> 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.

Lesson Demo

DEMO

Thank You!

Q&A

- SEO Instructor

Mr. Kurt Seiffert

Office Hours Monday-Thursday 10a.m. - 11:00a.m.