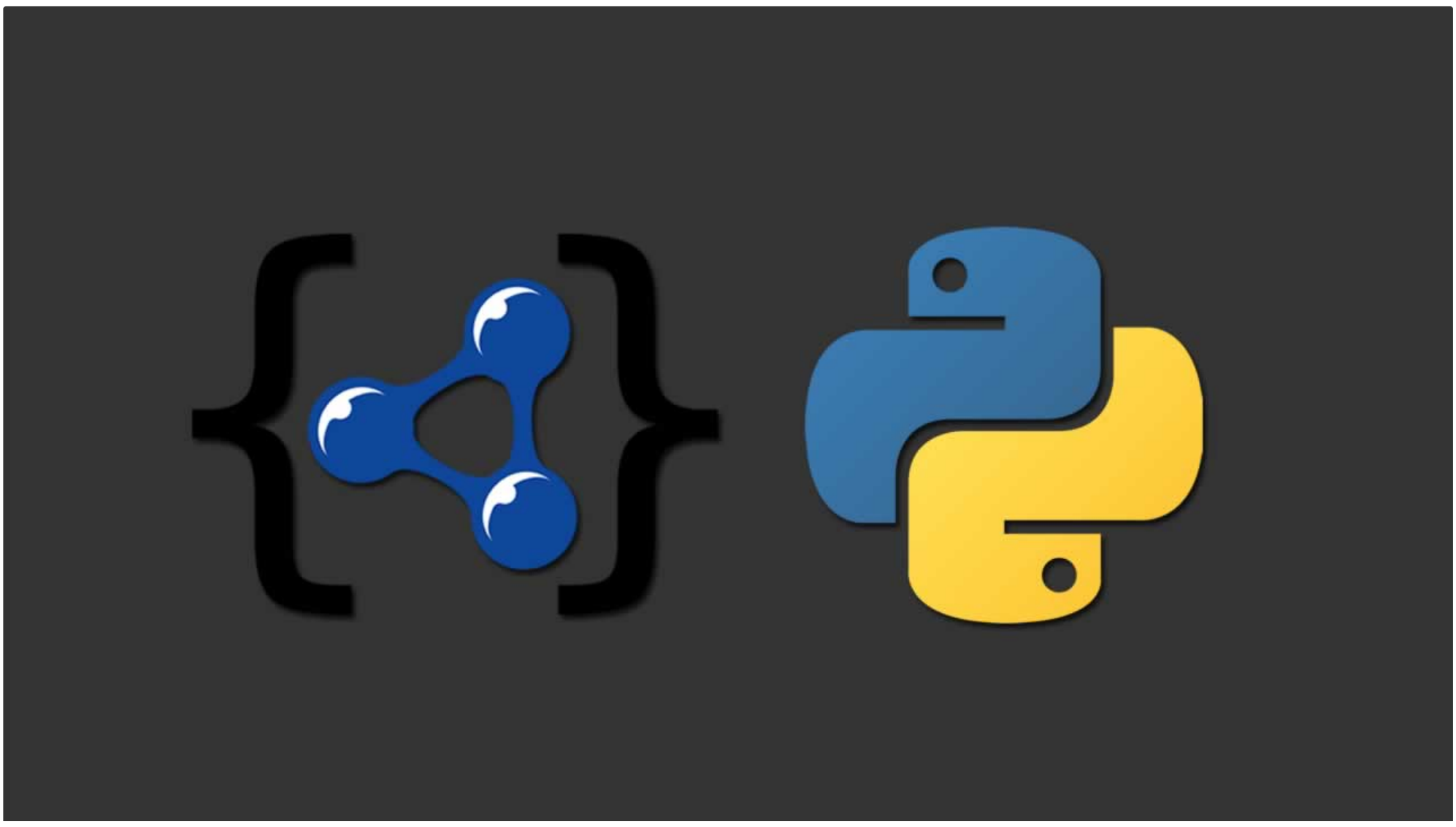

JSON—The Python Way

2019-04-17 01:04 PM 243

JSON—The Python Way JSON: The Fat-Free Alternative to XML.

What is JSON?

JavaScript Object Notation (JSON) is a lightweight data-interchange format based on the syntax of JavaScript objects. It is a text-based, human-readable, language-independent format for representing structured object data for easy transmission or saving. JSON objects can also be stored in files—typically a text file with a `.json` extension and a `application/json` MIME type. Commonly, JSON is used for two way data transmission between a web-server and a client in a REST API.

Despite the fact that it's syntax closely resembles JavaScript objects, JSON can be used independently outside JavaScript. In fact, a majority of programming languages have libraries to manipulate JSON. In this article, our focus will be on manipulating JSON data in python, using the built-in json module.

And some basic terminology …

- JSON exists as a string—a sequence (or series) of bytes. To convert a complex object (say a dictionary) in to a JSON representation, the object needs to be encoded as a "series of bytes", for easy transmission or streaming—a process known as **serialization**.
- **Deserialization** is the reverse of serialization. It involves decoding data received in JSON format as native data types, that can be manipulated further.

Why JSON?

- Compared to its predecessor in server-client communication, XML, JSON is much smaller, translating into faster data transfers, and better experiences.
- JSON exists as a "sequence of bytes" which is very useful in the case we need to transmit (stream) data over a network.
- JSON is also extremely human-friendly since it is textual, and simultaneously machine-friendly.
- JSON has expressive syntax for representing arrays, objects, numbers and booleans.

Working with Simple Built-in Datatypes

Generally, the `json` module encodes Python objects as JSON strings implemented by the `[json.JSONEncoder](https://docs.python.org/3/library/json.html#json.JSONEncoder)` class, and decodes JSON strings into Python objects using the `[json.JSONDecoder](https://docs.python.org/3/library/json.html#json.JSONDecoder)` class.

Serializing Simple Built-in Datatypes

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

- `dumps()` —to serialize an object to a JSON formatted string.
- `dump()` —to serialize an object to a JSON formatted stream (which supports writing to a file).

Lets look at an example of how to use `json.dumps()` to serialize built in data types.

```
>>> import json
>>> json.dumps({
        "name": "Foo Bar",
        "age": 78,
        "friends": ["Jane","John"],
        "balance": 345.80,
        "other_names":("Doe","Joe"),
        "active":True,
        "spouse":None
    }, sort_keys=True, indent=4)
```

And the output:

```
{
    "active": true,
    "age": 78,
    "balance": 345.8,
    "friends": [
        "Jane",
        "John"
    ],
    "name": "Foo Bar",
    "other_names": [
        "Doe",
        "Joe"
    ],
    "spouse": null
}
```

In the example above we passed a dictionary to the `json.dumps()` method, with 2 extra arguments which provide pretty printing of JSON string. `sort_keys = True` tells the encoder to return the JSON object keys in a sorted order, while the `indent` value allows the output to be formatted nicely, both for easy readability.

Similarly, lets use `json.dump()` on the same dictionary and write the output stream to a file.

```
>>> import json
>>> with open('user.json','w') as file:
        json.dump({
           "name": "Foo Bar",
           "age": 78,
           "friends": ["Jane","John"],
           "balance": 345.80,
           "other_names":("Doe","Joe"),
           "active":True,
           "spouse":None
        }, file, sort_keys=True, indent=4)
```

This example writes a `user.json` file to disk with similar content as in the previous example.

Deserializing Simple Built-in Datatypes

As in the case of serialization, the decoder converts JSON encoded data into native Python data types as in the table below:

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

The `json` module exposes two other methods for deserialization.

- `loads()` —to deserialize a JSON document to a Python object.
- `load()` —to deserialize a JSON formatted stream (which supports reading from a file) to a Python object.

```
>>> import json
>>> json.loads('{ "active": true, "age": 78, "balance": 345.8,   "friends": ["Jane","John"], "name": "Foo Bar", "other_names": ["Doe","J
```

And the output:

```
{'active': True,
 'age': 78,
 'balance': 345.8,
 'friends': ['Jane', 'John'],
 'name': 'Foo Bar',
 'other_names': ['Doe', 'Joe'],
 'spouse': None}
```

Here we passed a JSON string to the `json.loads()` method, and got a dictionary as the output.
To demonstrate how `json.load()` works, we could read from the `user.json` file that we created during serialization in the previous section.

```
>>> import json
>>> with open('user.json', 'r') as file:
        user_data = json.load(file)
>>> print(user_data)
```

From this example, we get a dictionary, again, similar to the one in `loads()` above.

Working with Custom Objects

So far we've only worked with built-in data types. However, in real world applications, we often need to deal with custom objects. We will look at how to go about serializing and deserializing custom objects.

Serializing Custom Objects

In this section, we are going to define a custom `User` class, proceed to create an instance and attempt to serialize this instance, as we did with the built in types.

```
    """
     def __init__(self,name,age,active,balance,other_names,friends,spouse):
         self.name = name
         self.age = age
         self.active = active
         self.balance = balance
         self.other_names = other_names
         self.friends = friends
         self.spouse = spouse

     def __str__(self):
         return self.name
```

json_user.py

```
>>> from json_user import User
>>> new_user = User(
        name = "Foo Bar",
        age = 78,
        friends = ["Jane","John"],
        balance = 345.80,
        other_names = ("Doe","Joe"),
        active = True,
        spouse = None)
>>> json.dumps(new_user)
```

And the output:

```
TypeError: Object of type 'User' is not JSON serializable
```

I bet this comes as no surprise to us, since earlier on we established that the `json` module only understands the built-in types, and `User` is not one of those.

We need to send our user data to a client over anetwork, so how do we get ourselves out of this error state?

A simple solution would be to convert our custom type in to a serializable type—i.e a built-in type. We can conveniently define a method `convert_to_dict()` that returns a dictionary representation of our object. `json.dumps()` takes in a optional argument, `default` , which specifies a function to be called if the object is not serializable. This function returns a JSON encodable version of the object.

```
def convert_to_dict(obj):
  """
  A function takes in a custom object and returns a dictionary representation of the object.
  This dict representation includes meta data such as the object's module and class names.
  """

  #  Populate the dictionary with object meta data
  obj_dict = {
    "__class__": obj.__class__.__name__,
    "__module__": obj.__module__
  }

  #  Populate the dictionary with object properties
  obj_dict.update(obj.__dict__)

  return obj_dict
```

json_convert_to_dict.py

Lets go through what `convert_to_dict` does:

- The function takes in an object as the only argument.
- We then create a dictionary named `obj_dict` to act as the `dict` representation of our object.
- By calling the special dunder methods `__class__.__name__` and `__module__` on the object we are able to get crucial metadata on the object i.e the class name and the module name—with which we shall reconstruct the object when decoding.
- Having added the metadata to `obj_dict` we finally add the instance attributes by accessing `obj.__dict__` . (Python stores instance attributes in a dictionary under the hood)
- The resulting `dict` is now serializable.

At this point we can comfortably call `json.dumps()` on the object and pass in `default = convert_to_dict` .

```
>>> from json_convert_to_dict import convert_to_dict
>>> data = json.dumps(new_user,default=convert_to_dict,indent=4, sort_keys=True)
>>> print(data)
```

```
    "__class__": "User",
    "__module__": "__main__",
    "active": true,
    "age": 78,
    "balance": 345.8,
    "friends": [
        "Jane",
        "John"
    ],
    "name": "Foo Bar",
    "other_names": [
        "Doe",
        "Joe"
    ],
    "spouse": null
}
```

Decoding Custom Objects

At this point, we have a JSON string with data about a custom object that `json.loads()` doesn't know about. Passing this string to `json.loads()` will give us a dictionary as output, as per the conversion table above.

```
>>> import json
>>> user_data = json.loads('{"__class__": "User", "__module__": "__main__", "name": "Foo Bar", "age": 78, "active": true, "balance": 345
>>> type(user_data)
>>> print(user_data)
```

As expected, `user_data` is of type `dict` .

```
dict
{'__class__': 'User',
 '__module__': '__main__',
 'name': 'Foo Bar',
 'age': 78,
 'active': True,
 'balance': 345.8,
 'other_names': ['Doe', 'Joe'],
 'friends': ['Jane', 'John'],
 'spouse': None}
```

However, we need `json.loads()` to reconstruct a `User` object from this dictionary. Accordingly, `json.loads()` takes in an optional argument `object_hook` which specifies a function that returns the desired custom object, given the decoded output (which in this case is a `dict`). We shall now go ahead and define a `dict_to_obj` function that returns a `User` object.

```
def dict_to_obj(our_dict):
    """
    Function that takes in a dict and returns a custom object associated with the dict.
    This function makes use of the "__module__" and "__class__" metadata in the dictionary
    to know which object type to create.
    """
    if "__class__" in our_dict:
        # Pop ensures we remove metadata from the dict to leave only the instance arguments
        class_name = our_dict.pop("__class__")

        # Get the module name from the dict and import it
        module_name = our_dict.pop("__module__")

        # We use the built in __import__ function since the module name is not yet known at runtime
        module = __import__(module_name)

        # Get the class from the module
        class_ = getattr(module,class_name)

        # Use dictionary unpacking to initialize the object
        obj = class_(**our_dict)
    else:
        obj = our_dict
    return obj
```

json_dict_to_obj.py

This is what `dict_to_obj` does:

- Extract the module name from the dictionary under the key `__module__` .
- Now we can go ahead and import the module. Notice that we use `__import__` since the module name isn't known at run time.
- From the imported module, we can get the class, which is one of the module's attributes.
- Finally instantiate a member of the class, by supplying the class constructor with instance arguments through dictionary unpacking of whatever is left of `our_dict` .

Now let's go ahead and confidently call `json.loads` with the argument `object_hook = dict_to_obj` .

```
>>> from json_dict_to_obj import dict_to_obj
>>> new_object = json.loads('{"__class__": "User", "__module__": "__main__", "name": "Foo Bar", "age": 78, "active": true, "balance": 34
>>> type(new_object)
```

Without a doubt, we can confirm that indeed `new_object` is of type `User` .

```
__main__.User
```

At this stage, we have successfully encoded a custom object to JSON and recreated the same object from our JSON data. I'd say we all deserve a pat on the back, and of course, a drink.

And since it's a free world, have you choice. 🐯 🐻 🍸Cheers!!

Conclusion

JSON is evidently a very useful standard, important for communication between different systems. If you would like to read more on the `json` module, please refer to the official docs. If you would also like to jog your memory on dictionaries, kindly refer to my previous article. Otherwise good bye, for now 😄.

Further reading:

☞ Python for Beginners: Become a Certified Python Developer

☞ Introduction to Python for Beginners

☞ The Python 3 Bible™ | Go from Beginner to Advanced in Python

☞ Complete Python Bootcamp: Go from zero to hero in Python

☞ Learn Python Through Exercises

Suggest:

☞ Complete Python Tutorial for Beginners (2019)

☞ Complete Python 3 Programming Tutorial

☞ What is Python and Why You Must Learn It in [2019]

☞ Learn Python Programming for Free Today!

☞ Python Sets and Set Theory

☞ Python programming: What can you do with Python?