Describe parsing of common data format (XML, JSON, and YAML) to Python data structures

JSON:

You can easily convert JSON to lists (for a JSON array) and dictionaries (for JSON objects) with the built-in JSON module. There are four functions that you work with to perform the conversion of JSON data into Python objects and back.

- **load():** This allows you to import native JSON and convert it to a Python dictionary **from a file.**
- **loads():** This will import JSON data from a string for parsing and manipulating **within your program.**
- **dump():** This is used to write JSON data from Python objects **to a file**.
- **dumps():** This allows you to take JSON dictionary data and convert it into a serialized string for parsing and manipulating **within Python**.

The **s** at the end of dump and load refers to a string, as in dump string. To see this in action, you load the JSON file and map the file handle to a Python object.

JSON sample

```
{
    "interface": {
        "name": "GigabitEthernet1",
        "description": "Router Uplink",
        "enabled": true,
        "ipv4": {
            "address": [
                {
                    "ip": "192.168.1.1",
                    "netmask": "255.255.255.0"
                }
            ]
        }
    }
}
```

Python script: python working_with_json.py

```
import json

# open the json file as 'data'
with open("OCG_json_sample.json") as data:
    #assign the contents to a variable
    json_data = data.read()

# create a dictionary and load the json data from the previous variable
json_dict = json.loads(json_data)

# print the dictionary
print(json_dict) # {'interface': {'name': 'GigabitEthernet1', 'description': 'Router
Uplink', 'enabled': True, 'ipv4': {'address': [{'ip': '192.168.1.1', 'netmask':
```

```
'255.255.255.0'}]}}}

# Modify/update the interface description - 2 tiers down
json_dict["interface"]["description"] = "Backup Link"

# print the updated json dictionary
print(json_dict) #{'interface': {'name': 'GigabitEthernet1', 'description': 'Backup
Link', 'enabled': True, 'ipv4': {'address': [{'ip': '192.168.1.1', 'netmask':
'255.255.255.0'}]}}}


# Modify/update the interface ipv4 address - 3 tiers down

"""
Originally had they following:

json_dict["interface"]["ipv4"]["address"] = "10.1.1.1"

which output the following:

{'interface': {'name': 'GigabitEthernet1', 'description': 'Backup Link', 'enabled':
True, 'ipv4': {'address': '10.1.1.1'}}}

What happened to the netmask???

The issue arises because the original JSON structure for the address field is a list
of dictionaries,
but in your script, you are directly assigning a string to json_dict["interface"]
["ipv4"]["address"],
which changes the type of address from a list to a string. This results in the loss of
the netmask field.

Corrected line to preserve is below.
"""

json_dict["interface"]["ipv4"]["address"][0]["ip"] = "10.1.1.1"

# print the updated json dictionary
print(json_dict) # {'interface': {'name': 'GigabitEthernet1', 'description': 'Backup
Link', 'enabled': True, 'ipv4': {'address': [{'ip': '10.1.1.1', 'netmask':
'255.255.255.0'}]}}}



"""
We need the [0] because the address field is a list of dictionaries
and we are working with the first one.
"""
```

```python
# Modify/update the interface ipv4 netmask
json_dict["interface"]["ipv4"]["address"][0]["netmask"] = "255.255.255.255"

# print the updated json dictionary
print(json_dict) # {'interface': {'name': 'GigabitEthernet1', 'description': 'Backup
Link', 'enabled': True, 'ipv4': {'address': [{'ip': '10.1.1.1', 'netmask':
'255.255.255.255'}]}}}
```