Python & JSON Workshop

Filtering and Generating JSON Data  
Managing network applications, devices and services with JSON data

**Author**

Yvan Rooseleer

* <[yvan.rooseleer@biasc.be](mailto:yvan.rooseleer@biasc.be)>, Belgian IT Academy Support Center, Brussels
* <[yvan.rooseleer@odisee.be](mailto:yvan.rooseleer@odisee.be)>, Odisee University, Brussels

**Creation Date**

15 December 2020  
Last updated: 6 February 2021

**Cisco Netacad IPD Week Session Dates**

Part 1: DevNet Associate: Filtering JSON Data, 25 FEB 2021, 16:00   
Part 2: DevNet Associate: Generating JSON Data, 25 FEB 2021, 17:00

**Slide Deck**

During the workshop a **slide deck[[1]](#footnote-0)** will be used. The information in this document serves as a background document with examples, explanations and references[[2]](#footnote-1).

**Context**

This document describes a **JSON Workshop** and gives examples of how to use **Python** to manage, filter and generate JSON data with a number of system management tools.

The following Cisco Networking Academy courses provide relevant background information:

DevNet Course

DevNet Assocate 1.0

Published by: Cisco Networking Academy, 2020  
Emerging Technology Workshops  
Workshop: Experimenting with REST APIs using Webex Teams

Workshop: Model-Driven Programmability

**Topics**

Python Scripting and Programming

YANG Data Modelling (basics)

JSON, YAML and XML Data Exchange Formats

RESTCONF

Enterprise LAN - DNA Center (DNAC)

Online Collaboration - Webex

Containers & Images -- Docker Inspect

Configuration Management and Orchestration - Ansible

**Practice**

Practical examples with JSON Response Data  
Processing, filtering and generating data formats used in API calls and network configuration management

**Practice Data Files**

The data files for the practical examples and exercises are [here](https://drive.google.com/drive/folders/1Ip6nw4uxTHb5t9TxHEC3xcRPGzsTgEnH?usp=sharing).

**List of examples described in this document**

Reading, filtering and interpreting response data in JSON format, coming from Cisco DNA Center, Ansible, Docker and Webex.

Reading and processing data from Excel spreadsheets

Transforming data into a JSON tree structure

(optional) Transforming JSON data to YAML

(optional) Transforming JSON data to XML

Using JSON data generated by in API calls: e.g. spaces and members in Webex, devices in RESTCONF and devices and services in Ansible playbooks.

**Parts**

**Part 1: DevNet Associate: Filtering JSON Data**

Description: This part covers some aspects of the DevNet Associate course. Join to learn how to interpret JSON data coming from Webex Teams API, DNAC API, Docker inspect, Ansible.

**Part 2: DevNet Associate: Generating JSON Data**

Description: This part covers some aspects of the DevNet Associate course. Join to learn how to generate JSON data for Webex Teams API, IP Devices and Network Services from a source in the Excel format.

**Level**

Intermediate

**Prerequisites**

Basic understanding of:  
-- Python scripting and programming

-- Data modelling: YANG

-- Data formats: JSON, YAML, XML

-- Secure Shell/ SSH

-- HTTPS/ SSL/ TLS

-- Webex APIs

-- Restconf APIs

-- Ansible Playbooks

-- Docker

# Introduction

This Python & JSON Workshop has the purpose to introduce the reader into a number of data management and coding skills used in **DevOps** and **NetDevOps**[[3]](#footnote-2). The modern system and networking era requires professionals to be able to automate configurations and manage data accordingly.

If you are teaching the DevNet Associate Course or the Emerging Technology Courses, the following Python coding and scripting skills are relevant:

* **Constructing API Calls** 
  + Uri
  + Authentication & Authorization
* **Managing JSON Exchange Data**
  + Response data
  + Data Types
  + Data Conversion

Coding from scratch is not a requirement.

When using **API calls** in the context of **software defined networking** a large amount of data is being exchanged by applications, systems, services and network elements. This document focuses on managing the **data exchanged** in the process of deploying, configuring and monitoring IP networks, IT infrastructure, services and applications.

Practically, we will be focussing on JSON **response data** and how to manage the data using Python scripts.

## Application Programming Interfaces (APIs)

An **Application Programming Interface (API)** is a set of routines, protocols, and tools that integrate software components. A Network API provides automation and programmability which connects software to network devices. A network element is considered programmable if it exposes an API that can be used by software components to program the element[[4]](#footnote-3).

## JavaScript Object Notation (JSON)

JSON is an **encoding format** originally developed for JavaScript. JSON was intended to

provide a human and machine-readable encoding format that can be used as a replacement for XML. JSON uses a key-value representation with a ":" as a delimiter instead of

the opening and closing tags used by XML[[5]](#footnote-4). Other data exchange formats like YAML and XML are out of scope for this workshop.

## Python (programming language)

Python is a user-friendly interpreted (executed line-by-line) **high-level scripting language** with a large and comprehensive set of **libraries**. Python interpreters are available for all major operating systems[[6]](#footnote-5). The coding style used in this workshop is “as simple as possible”. In order to manage JSON data in Python scripts or programs a specific library needs to be imported[[7]](#footnote-6).

|  |
| --- |
| import json |

## Infrastructure as Code/ Network as Code

The main principles of “infrastructure as code” or “network as code” are the following[[8]](#footnote-7):

* Use machine readable formats like *JSON*, *YAML* or *XML*
* Deploy and monitor configuration using programmatic APIs and tooling
* Limit manual network configuration
* Treat the source code as single source of truth
* Store configuration information in source control systems, like *git* repositories

### Simple Example of Python Code to Manage IP Networks

Below is an simple example of Python Code relating to IP subnets, prefixes and network sizes. Key-Value pairs are often defined by Python in the form of a dictionary enclosed with {}..

If you want to **access a dictionary item**, you can do so by making a reference to its key inside a pair of square brackets (example 1) or by using the get() method (example 2):

|  |
| --- |
| **#Dictionary of network prefixes and subnet masks (partial example)**  **prefix\_subnet\_masks** = {  “/24": "255.255.255.0",  "/25": "255.255.255.128",  "/26": "255.255.255.192"  }  **# example 1**  subnet\_mask\_1 = prefix\_subnet\_masks**["/24"]**  print(subnet\_mask\_1 )  # outputs: 255.255.255.0  **# example 2**  subnet\_mask\_2 = prefix\_subnet\_masks.**get("/26")**  print(subnet\_mask\_2)  # outputs: 255.255.255.192 |

If you want to loop through a dictionary's keys and values, you can use the **items**() method.

|  |
| --- |
| for **key, value** in prefix\_subnet\_masks.items():  print("Prefix => Subnet Mask ->", **key, ":", value**) |

To check if a given key exists in a dictionary, you can use the **in** keyword.

|  |
| --- |
| if "/26" **in** prefix\_subnet\_masks:  print("Yes")  else:  print("No") |

You can use the **for loop** to loop through a dictionary.

|  |
| --- |
| for item **in** prefix\_subnet\_masks:  print(item) |

=> Experiment with the Python examples to improve your understanding

=> Write a script and execute the code above and check the output.   
How many items are shown in the output?

**\*QUOTATION MARKS**

In a Python script you can use both double “ “ and ‘ ‘ single quotes.

A Python dict structure will typically use single quotes, while JSON uses double quotes.

This can be verified in https://jsonlint.com/

### IPv4 Python Code to Select Specific Information using a Dictionary

To prepare for filtering and generating JSON we will be using with a small program that manages the conversion (or translation) of network prefixes to subnet masks, and calculates network size or nupmber of hosts per subnet. In the example a dictionary structure is used. To select a specific subnet mask or network prefix, **square brackets** are used in order to select a key. The example below will return the prefix for a given subnet mask:

**Examples**

**net\_prefix = netmask\_prefixes[“225.255.255.0”]**

**net\_mask = prefix\_netmasks[“/26”]**

**Python Code to Convert Subnet Massk and Calculate Network Size**

This Python code below shows an example of **data-driven programming**. It is not always necessary to calculate a result if the necessary data are available. In this case the necessary data are in the format of a **dictionary**. The Python program defines three datastructures of the type dictionary (similar in format to JSON).

* **netmask\_prefixes** to translate a subnet mask into a network prefix
* **prefix\_netmasks** to translate a network prefix into a subnet mask
* **netmask\_netbits** to translate a subnet mask into network bits

*In the (partial) Python code example the most relevant code fragments are highlighted[[9]](#footnote-8).*

|  |
| --- |
| **#### FIVE FUNCTIONS TO HELP DESIGN IP SUBNETS**  **## IPv4 subnet masks, prefixes, number of addressess, number of hosts**  **## Converting subnet mask into prefix notation**  **## Getting number of hosts for a specific prefix**  ***#### ONLY A SMALL COLLECTION OF PREFIXES ARE USED IN THIS EXAMPLE***  **netmask\_prefixes = {**  **'255.255.255.252': '/30'**  **,'255.255.255.248': '/29'**  **,'255.255.255.240': '/28'**  **,'255.255.255.224': '/27'**  **,'255.255.255.192': '/26'**  **,'255.255.255.128': '/25'**  **,'255.255.255.0' : '/24'**  **,'255.255.254.0' : '/23'**  **}**  **prefix\_netmasks = {**  **'/30': '255.255.255.252'**  **,'/29': '255.255.255.248'**  **,'/28': '255.255.255.240'**  **,'/27': '255.255.255.224'**  **,'/26': '255.255.255.192'**  **,'/25': '255.255.255.128'**  **,'/24': '255.255.255.0'**  **,'/23': '255.255.254.0'**  **}**  **netmask\_netbits = {**  **'255.255.255.252': '1111 1111 1111 1111 1111 1111 1111 1100',**  **'255.255.255.248': '1111 1111 1111 1111 1111 1111 1111 1000',**  **'255.255.255.240': '1111 1111 1111 1111 1111 1111 1111 0000',**  **'255.255.255.224': '1111 1111 1111 1111 1111 1111 1110 0000',**  **'255.255.255.192': '1111 1111 1111 1111 1111 1111 1100 0000',**  **'255.255.255.128': '1111 1111 1111 1111 1111 1111 1000 0000',**  **'255.255.255.0' : '1111 1111 1111 1111 1111 1111 0000 0000',**  **'255.255.254.0' : '1111 1111 1111 1111 1111 1110 0000 0000',**  **'255.255.252.0' : '1111 1111 1111 1111 1111 1100 0000 0000',**  **'255.255.248.0' : '1111 1111 1111 1111 1111 1000 0000 0000',**  **'255.255.240.0' : '1111 1111 1111 1111 1111 0000 0000 0000',**  **'255.255.224.0' : '1111 1111 1111 1111 1110 0000 0000 0000'**  **}** |

Below are **five functions** solving specific problems relating to network proefices, subnet masks and network size. In three functions it is required to select specific values within the dioctionary structures created in the table above.

|  |
| --- |
| **def get\_net\_prefix(p\_subnet\_mask):**  **try:**  **net\_prefix = netmask\_prefixes[p\_subnet\_mask]**  **return net\_prefix**  **except:**  **return "Wrong input: garbage in, garbage out"**  **def get\_number\_ip\_addresses(p\_prefix):**  **#### example /30 => 30**  **pbits = 32-int(p\_prefix[1:])**  **return 2 \*\* pbits**  **def get\_number\_ip\_hosts(p\_prefix):**  **#pbits = 32-int(p\_prefix[1:])**  **return get\_number\_ip\_addresses(p\_prefix)-2**  **#return (2 \*\* pbits)-2**  **def get\_netmask(p\_prefix):**  **try:**  **net\_mask = prefix\_netmasks[p\_prefix]**  **return net\_mask**  **except:**  **return "Wrong input: garbage in, garbage out"**  **def get\_network\_bits(p\_subnet\_mask):**  **try:**  **net\_bits = netmask\_netbits[p\_subnet\_mask]**  **return net\_bits**  **except:**  **return "Wrong input: garbage in, garbage out"** |

*\*The make sure a parameter does not interfere with the name of a variable we prepend parameters with p\_*

*This practice is common in Oracle database applications.*

Here is the **main part** of the Python program, calling the necessary functions to find out network prefixes, subnet masks and network sizes.

|  |
| --- |
| **if \_\_name\_\_ == "\_\_main\_\_":**  **### dev test function via prefix**  **subnet\_mask = ('255.255.255.192')**  **print(subnet\_mask)**  **net\_prefix = get\_net\_prefix(subnet\_mask)**  **print(net\_prefix)**  **net\_mask = get\_netmask(net\_prefix)**  **net\_bits = get\_network\_bits(net\_mask)**  **print(net\_bits)**  **net\_number\_addr = get\_number\_ip\_addresses(net\_prefix)**  **net\_number\_ip\_hosts = get\_number\_ip\_hosts(net\_prefix)**  **print(net\_number\_addr)**  **print(net\_number\_ip\_hosts)**  **#print(type(prefix\_netmasks))** |

=> Experiment with the Python examples to improve your understanding

The example above makes you familiar with a Python dictionary (dict) structure. Below we will transform a Python dictionary into a JSON string and vice versa.

## Data Types & Data Conversion

The following elements are necessary for the reader to be aware of in order to understand the examples used in this workshop:

* data type[[10]](#footnote-9)
  + str, int, boolean, list, dict, and many more ...
* dictionary
  + keys and values
* import json
  + Json = dict + list *(needs to bet converted in a Python script)*
  + json.loads
  + json.dumps

### Dictionary

Dictionaries[[11]](#footnote-10) are unordered, changeable (mutable), and indexed collections of data.   
*In Python 3.6x dictionaries have become ordered by default.*

Each dictionary is a set of **key: value** pairs.

Example 1

**ipv4** = {**'address'**: [{'ip': '192.168.56.101', 'netmask': '255.255.255.0'}]}

It is important to be able to recognize the **data type**. Very often dictionaries are a part of a list.

**Data Structures**

single\_address = {'ip': '192.168.56.101', 'netmask': '255.255.255.0'}

multiple\_addresses = [{'ip': '192.168.56.101', 'netmask': '255.255.255.0'},

{'ip': '192.0.2.1', 'netmask': '255.255.255.252'} ]

ietf\_ipv4 = {'address': [{'ip': '192.168.56.101', 'netmask': '255.255.255.0'}]}

**Recognize Data Type**

print(type(single\_address))

Output: <class 'dict'>

print(type(multiple\_addresses))

Output: <class 'list'>

print(type(ietf\_ipv4))

Output: <class 'dict'>

print(ietf\_ipv4['address'][0]['ip'])

Output: 192.168.56.101

Example 2

**netmask\_prefixes = {**

**'255.255.255.252': '/30'**

**,'255.255.255.248': '/29'**

**,'255.255.255.240': '/28'**

**,'255.255.255.224': '/27'**

**,'255.255.255.192': '/26'**

**,'255.255.255.128': '/25'**

**,'255.255.255.0' : '/24'**

**,'255.255.254.0' : '/23'**

**}**

It is important to be able to select the **data element** required.

# example 2.1 -- using square brackets to select a key

subnet\_mask\_1 = **prefix\_subnet\_masks['/24']**

print(subnet\_mask\_1 )

output: 255.255.255.0

# example 2.2 -- using get() to select a key

subnet\_mask\_2 = **prefix\_subnet\_masks.get('/26')**

print(subnet\_mask\_2)

output: 255.255.255.192

### JSON Format

The JSON format is very close to the Python dictionary type. **The key-value pairs are enclosed by quotation marks (“ “) and are separated by by a colon (:)[[12]](#footnote-11).** JSON shows hierarchy by using parentheses {}. *The Python dictionary type uses single quotes (‘ ‘), whereas the JSON structures are using double quotes(“ “)*

Example

{

"access\_token": "ZFlNzAtZGVjNjE0MGU1OGZmZEwN2ItYTU3",

"expires\_in": 1209600,

"refresh\_token": "MDEyIzNDU2Nzg5MDEyjc4OTEyMzQ1Njc4"

}

### Data Types & Data Conversion

**type() (1)**

When you interact with data structures and variables in a Python, it is necessary to be aware of the data *type*. When assigning a data value to a variable Python will define the data type.

*=>The functions, methods or operations you can use, depend on the data type*

Example

When you assign the value and other data regarding an access token to a variable in Python, the data type will be defined. In this case Python will define a dictionary data type.

atk = {

'access\_token':'ZFlNzAtZGVjNjE0MGU1OGZmZEwN2ItYTU3',

'Expires\_in': 1209600,

'Refresh\_token': 'MDEyIzNDU2Nzg5MDEyjc4OTEyMzQ1Njc4',

}

To find out the **data type** of a variable atk, you can do the following.

print (type(atk))

Output

<class 'dict'>

**import json**

In order to manage reponse data in JSON format, the data have to be tranfsormed into a **dict** structure when you are using Python. If necessary, apply the **json.loads()** method to transform a JSON string into dict format. Alternatively, in order to transmit JSON data on the network you need to transform the dict into a string using **json.dumps()**. This is also called *serialization*.

Serialization is the process of encoding data into JSON format (like converting a Python list to JSON). Deserialization is the process of decoding JSON data back into native objects you can work with (like reading JSON data into a Python list)[[13]](#footnote-12).

**json.dumps() (2) (4)**

The **json.dumps()** method transforms a **dict into a string**.

**json.loads() (5)**

The **json.loads()** method transforms a **string into a dict**.

**Example: Data Type & Data Conversion**

|  |
| --- |
| **#### SCRIPT DEVASC (BASED ON DEVASC LAB 3.6.6)[[14]](#footnote-13)**  import **json**  **atk** = {  "**access\_token**":"ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3",  "**expires\_in**":1209600,  "**refresh\_token**":"MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4",  "**refreshtokenexpires\_in**":7776000  }  print('-----1-----')  **print (type(atk))**  print('-----1B-----')  **print(atk.keys())**  print('-----2-----')  #### pretty output  **print(json.dumps(atk, indent=4))**  #### FILTERING DATA  #### filter access-token  print('-----3-----')  **print(atk["access\_token"])**  #### TRANSFORMING DATA TYPES  print('-----4-----')  ats = json.**dumps**(atk) #### SERIALIZATION  **print(type(ats))**  **print(ats["access\_token"])### TYPE ERROR**  ####  print('-----5-----')  **atj = json.loads(ats)**  **print(type(atj))** |

|  |
| --- |
| **OUTPUT OF THE SCRIPT ABOVE**  **-----1-----**  <class **'dict'**>  -----1B-----  ['access\_token', 'expires\_in', 'refresh\_token', 'refreshtokenexpires\_in']  **-----2-----**  {  "access\_token": "ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3",  "expires\_in": 1209600,  "refresh\_token": "MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4",  "refreshtokenexpires\_in": 7776000  }  **-----3-----**  ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3  **-----4-----**  <class **'str'**> (\*)  ---------------------------------------------------------------------------  TypeError Traceback (most recent call last)  ---> 24 print(ats["access\_token"])  TypeError: string indices must be integers  **-----5-----**  <class **'dict'**> |

**(\*) Remark**  
Filtering needs a data type like **dict**.

=> Experiment with the Python examples to improve your understanding

## Data Models & Structures

A **data model** is a description of how data must be encoded for **information exchange** between two entities. A data model can be created for any information exchange process and is based on Key-Value pairs[[15]](#footnote-14).

Python allows you to manage, transform and filter data structures. You can create any data structure you need in function of your use case. However, **standard data models** have been created to avoid reinventing the wheel. **YANG is a prominent example of an open standard data model.**

#### YANG Model

YANG is an IETF standard **model** for managing and configuring network devices.[[16]](#footnote-15) Unlike JSON, YAML or XML, **YANG is not a data format**, but a modeling language to structure data. Structured data ensures that key-value pairs and their hierarchy are easily identifiable.

YANG data models are an example of a specification for structured data[[17]](#footnote-16).

A common source of confusion is the relationship between YANG data models and encoding formats such as XML/JSON. **YANG is used to represent the data on a device in an abstract way**, but does not contain actual device configuration or operational data; it simply shows the structure. In other words, YANG forms the template from which XML/JSON data is generated and does not represent the actual data[[18]](#footnote-17).

Much work has been done within the IETF to create models for managing IT systems and networks. Therefore, it is relevant to understand YANG data modeling.

Apart from *IETF open standards* there are also specific models created by *vendors* such as Cisco[[19]](#footnote-18). Also Cisco has developed tools for interacting with programmable interfaces[[20]](#footnote-19).

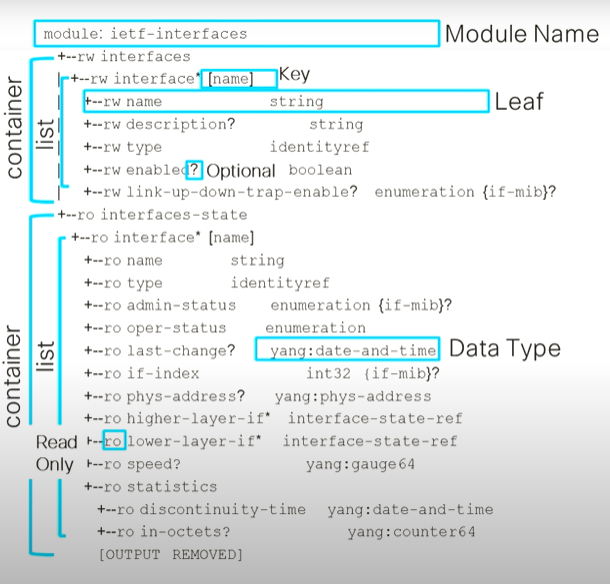
Two important groups of data models haven been defined:

* **Device data models**: devices, interfaces, vlans, ...
* **Service data models**: system management, BGP, MPLS, ...

##### Device Data Models

In this part we will focus on **device data models**.

The image below shows a YANG module describing the **interfaces** of a network device.

  
*Example of an open standard YANG model for device interfaces[[21]](#footnote-20)*

**JSON Example for the YANG Model Above (RESTCONF)**

|  |
| --- |
| {  "ietf-interfaces:**interfaces**": {  "**interface**": **[{**  "name": "GigabitEthernet1",  "description": "VBox",  "type": "iana-if-type:ethernetCsmacd",  "enabled": true,  "ietf-ip:ipv4": {  "**address**": [{  "ip": "10.1.1.1",  "netmask": "255.255.255.0"  }]  },  "ietf-ip:ipv6": {}  **}**  ]  }  } |

### YANG Structures

**Module**

A **YANG module** is a hierarchical tree structure for organizing data[[22]](#footnote-21). A module contains all the necessary data elements for a specific case. Each module is uniquely identified by a namespace URI. A module defines a single data model[[23]](#footnote-22).

Among the many structures found in the YANG data model it is important to distinguish between **containers, lists and leaf structures**. Very often dependencies are created in the form of a tree.

|  |
| --- |
| **Difference between Container, List and Leaf-List**  Container   * List   + Container * Leaf-List   + Leaf |

**Leaf**

A leaf is a single key/value pair[[24]](#footnote-23). The **value** of a data element is called a leaf node and there are no child dependencies. For simplicity, a leaf can be considered as a **value** for a specific **key**.

A leaf will often be part of a larger structure, such as a list or a container.

*A leaf node has a value, but no child nodes, in the data tree. A leaf node exists in zero or one instance in the data tree[[25]](#footnote-24).*

Examples of the type *leaf* are **hostnames** or **IP addresses** of network devices. The **token** that is used to authorize an API call can also be considered as a leaf.

Examples

|  |
| --- |
| **Hostname**  **{"hostname": "CSR1kv"}**  **IP Address**  **{"ip address": "192.0.2.1"}**  **DNA Center API: Token\***  {"Token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI1ZTlkYmI3NzdjZDQ3 6IxT1u9Vr9I\_pj8EmkC3zIUSx5Hjr\_\_TA-8VG86IGwW5-eTRRYaAcf2g8t6UkMs8Y9aGbfcDRgWfxmJOtPxx4\_20J7tQIIgzQ9Iod9xY4UYCg8g6qu1DQuEoikWFLW\_lH6aA"} |

*\*Part of the output not shown, to save space*

For devices with more than one network interface a **list** of interfaces and IP addresses needs to be defined.

**Container { }**

A container is a **grouping** of other statements (with leaves, containers or lists). Containers are a way to nest structures within within other structures.

Example (with nested containers)

|  |
| --- |
| {  "name": "cisco",  "privilege": 15,  "password": {  "encryption": "0",  "password": "cisco123"  }  } |

**List [ ]**

A list is a collection of one or more leaves or containers, possible with dependencies in the form of a tree structure.

A *leaf-list* is a list of simple structures (*leaves*), whereas a *list* is a list of containers.

*The "leaf-list" statement is used to define an array of a particular type.*

Example  
The *container* interfaces in the example contains a *list* of different types of interface.

|  |
| --- |
| "username": [  {  "name": "cisco",  "privilege": 15,  "password": {  "encryption": "0",  "password": "cisco123"  }  }  ] |

**Tree**

Configuration data and state data are often represented in a tree format consisting of containers, leaves and lists. Tree structures are complex data structures. Managing or generating tree structures may involve *recursive* algorithms.

|  |
| --- |
| "interface": {  "GigabitEthernet": [  {  "name": "1",  "description": "VBox",  "ip": {  "address": {  "dhcp": {}  }  },  "mop": {  "enabled": false,  "sysid": false  },  "Cisco-IOS-XE-ethernet:negotiation": {  "auto": true  }  }  ]  } |

# 

# Part 1: Filtering JSON Data

The first part of this Python & JSON Workshop focuses on selecting, filtering and transforming JSON data. Many examples are concerned with **response data** received from RESTCONF, DNA Center, Ansible, Docker and Webex. The second part is about generating JSON data structures.

## APIC-EM Response Data

***Remark****: APIC-EM has been replaced by DNA Center and is no longer supported.  
=> This part about APIC-EM will be removed*

In this part we will start with a number of simple examples from APIC-EM, and continue with RESTCONF and DNA Center APIs.

### APIC-EM

Cisco APIC-EM provides policy-based automation of the network infrastructure, simplifying deployment and network operations. APCI-EM has been replaced by Cisoc DNA Center.

The examples below are based on a DevNet Learning Lab **APIC-EM APIs with Python: Part 1 - The Basics***[[26]](#footnote-25).*

### Example 1 - APIC-EM Service Ticket

**Authentication tokens**

Script name on DevNet: lab1-1-post-ticket.py

Get an authentication to make a specific API call

Cisco APIC-EM uses a **Role-Based Access Control (RBAC)** mechanism that assigns a security role to every user account. RBAC-governed APIs use a **service ticket** to make access-control decisions. Almost every API call you send to Cisco APIC-EM REST must provide a service ticket.

In this example we focus on managing the JSON output of the API call using Python script **lab1-1-post-ticket.py** available on the DevNet website[[27]](#footnote-26). This script retrieves an authentication token from APIC-EM and prints out its value.

**Service ticket**

The service ticket needed for access is the value of **serviceTicket** attribute in the example below.

|  |
| --- |
| {  "**response**": {  "**serviceTicket**": "**ST-415-HpemSXpzSnTAYFQIqTNQ-cas**",  "idleTimeout": 1800,  "sessionTimeout": 21600  },  "version": "1.0"  } |

Source: <https://developer.cisco.com/learning/lab/apic-em-basic/step/1>

Below (line 2) is the Python code to filter out the Service Ticket:

|  |
| --- |
| response\_json = resp.json()  print ("ticket: ",**response\_json["response"]["serviceTicket"]**) |

Source: <https://developer.cisco.com/learning/lab/apic-em-basic/step/1>

In order to select a specific element in a JSON (or **dict**) structure it is necessary to use the **name of the key** between **square brackets: ["response"]** or to use the **get()** method. To select the level below a parent item, you add the second key between square brackets: **["response"]["serviceTicket"]**

### Example 2 - APIC-EM Response Data

**Users and roles**

Script name on DevNet: **lab1-2-get-user.py**

Get all APIC-EM users with their roles

**Retrieving information about users and their roles in APIC-EM**

In the example below is the output from a Python script that retrieves information of users and their roles. The response use **square brackets** to indicate that **more than one** data element about users can be retrieved. *In this case only one user was found in the database.*

|  |
| --- |
| {  "**response**": **[ .**  {  "**authorization**": [  {  "**role**": "ROLE\_POLICY\_ADMIN",  "**scope**": "ALL"  }  **]** ,  "**username**": "devnetuser",  "**authSource**": "internal"  }  ],  "**version**": "1.0"  } |

Source: <https://developer.cisco.com/learning/lab/apic-em-basic/step/2>

|  |
| --- |
| **# Parsing raw response to list out all users and their role**  for **item** in response\_json["**response**"]:  for item1 in **item**["**authorization**"]:  print**("User \'%s\', role is the %s."%(item["username"],(item1["role"])[5:]))** |

Source: <https://developer.cisco.com/learning/lab/apic-em-basic/step/2>

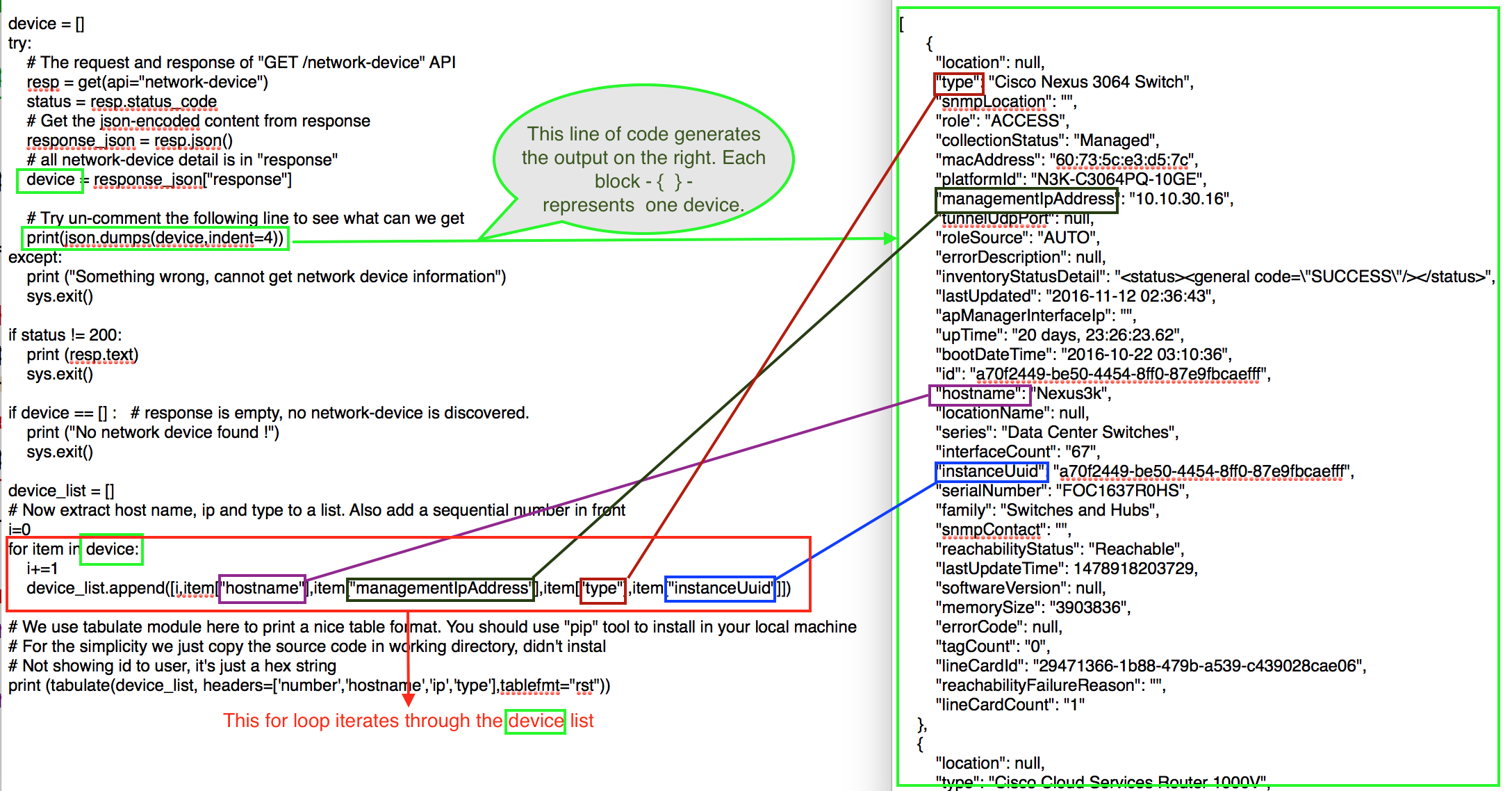
**Remark**: Item["authorization"] is the same as ["response"][0]["authorization"] in case of the first element in a list.

### Example 3 - APIC-EM Response Data for Device List

**Network device-related APIs**

Script name on DevNet: **lab2-1-get-network-device-list.py**

In this sample code, the **GET /network-device API** retrieves detailed information about all network devices known to the controller. To view the content click on the source URL below the image.

Source: <https://developer.cisco.com/learning/lab/apic-em-basic/step/3>

The Python code extracts the **Hostname**, **Management IP Address** and **Type** of the list of data about network devices that was retrieved.

|  |
| --- |
| i=0  for **item** in device:  i+=1  **device\_list.append([i,item["hostname"],item["managementIpAddress"],item["type"],item["instanceUuid"]])** |

=> Experiment with the Python examples to improve your understanding

## RESTCONF Response Data

RESTCONF is an open standard *transport* protocol developed within IETF[[28]](#footnote-27). RFC 8040 describes an HTTP-based protocol that provides a programmatic interface for accessing data defined in YANG[[29]](#footnote-28). In this part we will practice JSON response data structures that are available within RESTCONF.

In the example below we will manage a virtual router of the type CSR1000v. The virtual router is freely available for Networking Academy students and instructors.

### Example - Sending RESTCONF request (URL) to CSR1000v

**Python Code** <https://drive.google.com/file/d/1QHcVElVI7FBBxlG9VmaWxA7L3q1lngjc/view?usp=sharing>

**URL to be used**  
[https://192.168.56.101/restconf/data/ietf-interfaces:interfaces](https://192.168.56.107/restconf/data/ietf-interfaces:interfaces)  
*IP address to be adapted to the address of your own router, also: add necessary auth header*

**RESTCONF Response Data from CSR1000v**

|  |
| --- |
| **Raw Response** *(machine readable)*  {"ietf-interfaces:interfaces": {**"interface"**: [{"name": "GigabitEthernet1", "description": "VBox", "type": "iana-if-type:ethernetCsmacd", "enabled": true, "ietf-ip:ipv4": {**"address"**: [{"ip": "192.168.56.101", "netmask": "255.255.255.0"}]}, "ietf-ip:ipv6": {}}, {"name": "Loopback9", "description": "999", "type": "iana-if-type:softwareLoopback", "enabled": true, "ietf-ip:ipv4": {"address": [{"ip": "10.9.9.9", "netmask": "255.255.255.0"},{"ip": "172.29.0.9", "netmask": "255.255.255.0"}]}, "ietf-ip:ipv6": {}}]}} |

|  |
| --- |
| **JSON Pretty Response** *(human readable[[30]](#footnote-29))*  {  "ietf-interfaces:**interfaces**": {  "**interface**": **[{**  "name": "GigabitEthernet1",  "description": "VBox",  "type": "iana-if-type:ethernetCsmacd",  "enabled": true,  "ietf-ip:ipv4": {  "**address**": [{  "ip": "192.168.56.101",  "netmask": "255.255.255.0"  }]  },  "ietf-ip:ipv6": {}  },  {  "name": "Loopback9",  "description": "Lo9",  "type": "iana-if-type:softwareLoopback",  "enabled": true,  "ietf-ip:ipv4": {  "**address**": [{  "ip": "10.9.9.9",  "netmask": "255.255.255.0"  }]  },  "ietf-ip:ipv6": {}  **}**  ]  }  } |

|  |
| --- |
| **PYTHON CODE - FILTER JSON RESPONSE DATA FROM CSR1000v** (no loop, only index used)  **print("=> Printing filtered response")**  **print("Interface Name: ")**  print(response\_json**["ietf-interfaces:interfaces"]["interface"][0]["name"]**)  **print("IP Address + Subnet: " )**  ip\_subnet = response\_json**["ietf-interfaces:interfaces"]["interface"][0]["ietf-ip:ipv4"]["address"]**  print(ip\_subnet)  **print("IP Address: " )**  ip = response\_json**["ietf-interfaces:interfaces"]["interface"][0]["ietf-ip:ipv4"]["address"][0]["ip"]**  print(ip) |

|  |
| --- |
| **PYTHON RESPONSE - FILTERED OUTPUT**  **=> Printing filtered response**  **Interface Name:**  GigabitEthernet1  **IP Address + Subnet:**  [{'ip': '192.168.56.101', 'netmask': '255.255.255.0'}]  **IP Address:**  192.168.56.101 |

=> Experiment with the Python examples to improve your understanding

## Cisco DNA Center APIs

DNA-Center is an automation and assurance platform for **enterprise networks** developed by Cisco. The examples in this document are based on the video course by Hank Preston available on DevNet[[31]](#footnote-30). The video course is part of the Network Controllers module of Programming Basics available on the DevNet website.

### Example 1 - DNA Center Token

In order to get access to DNA Center API’s you have to **request a token** (and create an environment variable for the token).

**Example Code**: DNA Center 1 Northbound API Hello Network Simple  
URL: <https://drive.google.com/file/d/1N7izcl2OqXkmiH848fsD7Mxb_EyKrBwW/view?usp=sharing>

|  |
| --- |
| **PYTHON CODE**  url = “<https://sandboxdnac.cisco.com/dna/system/api/v1/auth/token>”  resp = requests.**post**( url, auth=auth, verify=False )  **RESPONSE**  {"**Token**":"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI1ZTlkYmI3NzdjZDQ3ZTAwNGM2N2RkMGUiLCJhdXRoU291cmNlIjoiaW50ZXJuYWwiLCJ0ZW5hbnROYW1lIjoiVE5UMCI6IxT1u9Vr9I\_pj8EmkC3zIUSx5Hjr\_\_TA-8VG86IGwW5-eTRRYaAcf2g8t6UkMs8Y9aGbfcDRgWfxmJOtPxx4\_20J7tQIIgzQ9Iod9xY4UYCg8g6qu1DQuEoikWFLW\_lH6aA"}  **FILTER to isolate token**  **token = resp['Token']**  **print(token)**  eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiI1ZTlkYmI3NzdjZDQ3ZTAwNGM2N2RkMGUiLCJhdXRoU291cmNlIjoiaW50ZXJuYWwiLCJ0ZW5hbnROYW1lIjoiVE5UMCI6IxT1u9Vr9I\_pj8EmkC3zIUSx5Hjr\_\_TA-8VG86IGwW5-eTRRYaAcf2g8t6UkMs8Y9aGbfcDRgWfxmJOtPxx4\_20J7tQIIgzQ9Iod9xY4UYCg8g6qu1DQuEoikWFLW\_lH6aA |

*\*Part of the output not shown to save space*

### Example 2 - DNA Center Response Data - Network Device List

The response of the DNA Center Network Device API call returns a complete **inventory** with a lot of details about networking devices that make up the network. Typically, the JSON response data is very long for human consumption. Filtering and selecting is necessary.

|  |
| --- |
| **DNA Center Request - Network Devices**  **https://sandboxdnac.cisco.com:443/dna/intent/api/v1/network-device**  **Response (json):**  **{**  **"response": [**  **{**  **"memorySize": "NA",**  **"family": "Switches and Hubs",**  **"hostname": "cat\_9k\_1",**  **"macAddress": "f8:7b:20:67:62:80",**  **"serialNumber": "FCW2136L0AK",**  **-- partly not shown --**  **"deviceSupportLevel": "Supported",**  **"collectionStatus": "Managed",**  **"upTime": "25 days, 19:42:18.12",**  **"lastUpdateTime": 1611060834555,**  **"softwareType": "IOS-XE",**  **"softwareVersion": "17.3.1",**  **"bootDateTime": "2020-12-24 17:11:54",**  **"lastUpdated": "2021-01-19 12:53:54",**  **"managementIpAddress": "10.10.22.66",**  **"platformId": "C9300-24UX",**  **"reachabilityStatus": "Reachable",**  **"series": "Cisco Catalyst 9300 Series Switches",**  **"type": "Cisco Catalyst 9300 Switch",**  **"location": null,**  **"role": "ACCESS",**  **"instanceUuid": "21335daf-f5a1-4e97-970f-ce4eaec339f6",**  **"instanceTenantId": "5dc444d31485c5004c0fb20b",**  **"id": "21335daf-f5a1-4e97-970f-ce4eaec339f6"**  **},**  **{**  **"memorySize": "NA",**  **"family": "Switches and Hubs",**  **"hostname": "cat\_9k\_2",**  **"macAddress": "f8:7b:20:71:4d:80",**  **"serialNumber": "FCW2140L039",**  **-- partly not shown --**  **"managementIpAddress": "10.10.22.70",**  **--- not shown ---**  **"type": "Cisco Catalyst 9300 Switch",**  **"location": null,**  **"role": "ACCESS",**  **"instanceUuid": "3e48558a-237a-4bca-8823-0580b88c6acf",**  **"instanceTenantId": "5dc444d31485c5004c0fb20b",**  **"id": "3e48558a-237a-4bca-8823-0580b88c6acf"**  **}**  **],**  **"version": "1.0"**  **}** |

*\*Part of the output not shown, to save space*

**Managing Response Data**

Managing response data involves **selecting, filtering and looping** through the result. Often it is useful to create a simpler data structure.

|  |
| --- |
| **dev\_list = []** #create empty list  **# loop through results and filter needed information**  **# create new JSON structure**  for **device** in resp\_devices\_json[**'response'**]:  if device[**'type'**] != None:  dev\_dict = {} #create empty dict  dev\_dict['hostname'] = device[**'hostname'**]  dev\_dict['type'] = device[**'type'**]  dev\_dict['macAddress'] = device[**'macAddress'**]  dev\_dict['managementIpAddress'] = device[**'managementIpAddress'**]  dev\_dict['serialNumber'] = device[**'serialNumber'**]  dev\_dict['softwareType'] = device[**'softwareType'**]  dev\_dict['softwareVersion'] = device[**'softwareVersion'**]  dev\_dict['reachabilityStatus'] = device[**'reachabilityStatus'**]  dev\_list.**append**(dev\_dict) |

=> Experiment with the Python examples to improve your understanding

### Example 3 - DNA Center Client Health

You can use DNA Center Assurance to **monitor and troubleshoot** the overall health of your enterprise network. An enterprise network consists of network devices and clients. End devices (computer, phone, and so on) are examples of clients. In a network cliensts connect to a network device (access point or switch). Cisco DNA Center supports both wired and wireless clients[[32]](#footnote-31).

**Example Code**: DNA Center Intent API Request Client Health

URL: <https://drive.google.com/file/d/1_FW7-r2H8j7W5PPCAMAxzheYDufy6AhG/view?usp=sharing>

The client health API (client-health) allows to monitor the state of devices in the network. Typically, the JSON response data is very long for human consumption. Filtering and selecting is a very common task.

|  |
| --- |
| **REQUEST**  # **f-string** is used for string formatting  get\_resp = requests.**get**(  **f**"https://{host}/dna/intent/api/v1/client-health",  headers=headers,  params=params,  )  get\_resp\_json = get\_resp.json() |

|  |
| --- |
| **RAW OUTPUT**  {**'response'**: [{'siteId': 'global', **'scoreDetail'**: [{'scoreCategory': {'scoreCategory': 'CLIENT\_TYPE', 'value': 'ALL'}, 'scoreValue': 29, **'clientCount'**: 82, 'clientUniqueCount': 82, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'CLIENT\_TYPE', 'value': 'WIRED'}, 'scoreValue': 100, 'clientCount': 2, 'clientUniqueCount': 2, 'starttime': 1611071700000, 'endtime': 1611072000000, 'scoreList': [{'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'POOR'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'FAIR'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'GOOD'}, 'scoreValue': -1, 'clientCount': 2, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000, 'scoreList': [{'scoreCategory': {'scoreCategory': 'deviceType', 'value': 'ALL'}, 'scoreValue': -1, 'clientCount': 2, 'clientUniqueCount': None, 'starttime': 1611071700000, 'endtime': 1611072000000}]}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'IDLE'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'NODATA'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'NEW'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}]}, {'scoreCategory': {'scoreCategory': 'CLIENT\_TYPE', 'value': 'WIRELESS'}, 'scoreValue': 28, 'clientCount': 80, 'clientUniqueCount': 80, 'starttime': 1611071700000, 'endtime': 1611072000000, 'scoreList': [{'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'POOR'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'FAIR'}, 'scoreValue': -1, 'clientCount': 58, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000, 'scoreList': [{'scoreCategory': {'scoreCategory': 'deviceType', 'value': 'ALL'}, 'scoreValue': -1, 'clientCount': 58, 'clientUniqueCount': None, 'starttime': 1611071700000, 'endtime': 1611072000000}]}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'GOOD'}, 'scoreValue': -1, 'clientCount': 22, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000, 'scoreList': [{'scoreCategory': {'scoreCategory': 'deviceType', 'value': 'ALL'}, 'scoreValue': -1, 'clientCount': 22, 'clientUniqueCount': None, 'starttime': 1611071700000, 'endtime': 1611072000000}]}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'IDLE'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'NODATA'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}, {'scoreCategory': {'scoreCategory': 'SCORE\_TYPE', 'value': 'NEW'}, 'scoreValue': -1, 'clientCount': 0, 'clientUniqueCount': 0, 'starttime': 1611071700000, 'endtime': 1611072000000}]}]}]} |

|  |
| --- |
| **FILTERING DATA IN THE RESPONSE**  print("----------1----------")  print(type(get\_resp\_json))  print("----------2----------")  print("Number of clients in first group")  print(get\_resp\_json["**response**"][0]["**scoreDetail**"][0]["**clientCount**"])  **RESULT**  ----------1----------  <class 'dict'>  ----------2----------  Number of clients in first group  **82** |

=> Experiment with the Python examples to improve your understanding

## Webex API Response Data

### Webex

Webex is a flexible cloud application tool for online collaboration delivered by Cisco. Multiple APIs exist for managing Webex users, spaces, messages and more[[33]](#footnote-32). The example below shows that the Webex API generates a lot of JSON output. It is useful to filter the response.

### Webex API Example: User Accounts Response Data

URL for API Call: <https://api.ciscospark.com/v1/people/me>

* *Bearer Authorization required*

|  |
| --- |
| **Webex Data Returned from API Call**  {"id":"Y2lzY29zcGFya...NlYzQ5NjI5MGY","emails":["Yvan.rooseleer@biasc.be"],"sipAddresses":[{"type":"personal-room","value":"838744612@biasc.webex.com","primary":false},{"type":"personal-room","value":"yvan.rooseleer@biasc.webex.com","primary":false},{"type":"cloud-calling","value":"Yvan.rooseleer@biasc.calls.webex.com","primary":true}],"displayName":"Yvan Rooseleer","nickName":"Yvan","firstName":"Yvan","lastName":"Rooseleer","avatar":"https://avatar-prod-us-east-2.webexcontent.com/Avtr~V1~e4d4112d-2548-4a47-810e-04fe64-a79b-49c5-823a-92cec494bb9c1d1de6~1600","orgId":"Y2lzY29zcGFyazovL3VzL09SR0FOSctODEwZS0wNGZlNDVlYTE4MWY","roles":["Y2lzY29zcGFya9hZG1pbg"],"licenses":["Y2lTQtNDExMS1hYThkLTA1MDI3N2Y3ZjdlOQ","Y2lzY29zcGFyazovL3VzL0xJQ0VOy53ZWJleC5jb20","Y2lzY29zcGFyazovL3VzL0xJQ0VOUxYmJlNTU2LWQwZmItNGFiNy1hMTYyLTlmNjQ2OGIyYmU5ZA","Y2lzYVOU0UvZTRkNDExMmQtMjU0OC00MWBiNDI4MmQ5NmY5NA","Y2lzY29wOTJhNTNfYmlhc2Mud2ViZXguY29t","Y2lzYZWExODFmOkZTU18xYjcyOGZmOS03ZGU4LTRjYjctOTU0MC0yOTMyMGI1YTQyY2I","Y2lzY29zUtMDRmZTQ1ZWExODFmOkZUTV9mNWZkZTM1Zi00NzA0LTQ2MGEtODEwZi00YzVkMzUyNDFlNjk","Y2lzY29zcExODFmOkNHXzVkYjcwNjYyLWNmYTItNGFjZC04MTRlLTgwYjNiNWVkZjNlZA","Y2lzY29zcGFyazovL3jQ3MzgyOC1hOTgwLTQ3MmYtODE5ZC02YjljY2UwOGU5MmI","Y2lzY29zcGFyazovLmOkZNU185ZWNhNzgxNC0zMzEzLTQ2NGYtOTY0Mi0wMjM5ODc1YmM5Zjg","Y2lzY29zcGFyazovL3VzL0xJQ0VOU0UvZTRkNDExMmQtMjU0OGE3NS1hNDNmLTBkYmJhMjIyNzg3Zl9iaWFzYy53ZWJleC5jb20","Y2lMwMjFkLTgwYjctNDFiYi1iZThhLWM0YjFiZjcyNTE4YV9iaWFzYy53ZWJleC5jb20","Y2lzY29zcGFyazovL3V0YTQ3LTgxMGUtMDRmZTQ1ZWExODFmOk1kYmFiMDcxYmY0NA","Y2lzY29zcGFyazovL3VzL0xJQ0VOUXzNkMDU3N2RiLTFjOGItNDQ4My1hMTBjLzYy53ZWJleC5jb20"],"created":"2016-12-23T08:38:22.877Z","lastModified":"2021-01-26T17:55:07.662Z","lastActivity":"2021-01-26T17:54:24.481Z","status":"active","invitePending":false,"loginEnabled":true,"type":"person","trainSiteNames":["biasc.webex.com"]} |

*Response data changed and incomplete*

|  |
| --- |
| **Selecting Webex Username from above JSON Data**  **#DISPLAY FILTERED RESULTS**  print("Displaying partial information")  #print(type(res))  print("Name: " + resp[**'displayName'**])  print("Created: " + resp[**'created'**])  print("User Type: " + resp[**'type'**])  print("User Status: " + resp**['status']**)  **Output**  Displaying partial information  Name: Yvan Rooseleer  Created: 2016-12-23T08:38:22.877Z  User Type: person  User Status: active |

### 

## Ansible Response Data

### Ansible

Ansible is a configuration management tool[[34]](#footnote-33). It is a collection of **netdevops** Python modules and libraries. Typically, a network manager will manage systems and networks using **declarative** YAML configuration files[[35]](#footnote-34) used by Ansible. If Ansible is used in a professional context, it is required to deal with data in JSON format.

### Example Ansible System Status and System Data Response Data

Ansible returns system status in JSON format. Typically, the JSON response data generated by Ansible is very long for human consumption. Filtering and selecting is necessary.

|  |
| --- |
| **Ansible Example 1**  **{"id":"20161024140306","version":"5.6.1","status":"UP"}** |

|  |
| --- |
| **Ansible Example 2**  In the example below the JSON return data are parsed by Ansible. This is not always the case, often you have to write your own parsing or filtering code.  - **name**: Check that you can connect (GET) to a page and it returns a status 200  **uri**:  url: http://www.example.com  - **name**: Check that page returns status 200 and fail if AWESOME is not in the page contents  **uri**:  url: http://www.example.com  return\_content: yes  **register**: this  **failed\_when**: "'AWESOME' not in this.content" |

Source: <https://docs.ansible.com/ansible/latest/collections/ansible/builtin/uri_module.html>

**JSON Response Data returned by Ansible Gather Facts**

Ansible consists of many different libraries and tools. A module like **gather\_facts** assists you to create an overview or inventory of device configuration and services. The example below generates a long response in JSON format[[36]](#footnote-35).

|  |
| --- |
| **Ansible Example 3**  **Ansible command to gather facts from the webservers inventory**  **$ ansible webservers -m gather\_facts --tree ./tmp\_facts**  Response from Ansible (very long json file, needs filtering)  URL for the complete file: <https://docs.google.com/document/d/1Z7wO5r8XaBirxXo06CmC8a6r5cCMezQQqUOcHxXhCyk/edit?usp=sharing>  **Partial Output (with some response data highlighted)**  {"ansible\_facts": {"ansible\_all\_ipv4\_addresses": ["192.0.2.1", "192.0.2.2", "192.0.2.3", "192.0.2.4", "192.0.2.5", "10.0.2.15", "172.17.0.1"], "ansible\_all\_ipv6\_addresses": ["fe80::9002:c8ff:fee8:bb09", "fe80::3c67:a5ff:fe17:e4cf", "fe80::a00:27ff:fee9:3de6", "fe80::42:3ff:fef6:9477"], "ansible\_apparmor": {"status": "enabled"}, "ansible\_architecture": "x86\_64", "ansible\_bios\_date": "12/01/2006", "ansible\_bios\_version": "VirtualBox", "ansible\_cmdline": {"BOOT\_IMAGE": "/boot/vmlinuz-5.4.0-37-generic", "quiet": true, "ro": true, "root": "UUID=fb261367-cf98-4bce-b682-42b3de0a8ab9", "vga": "792", "zswap.enabled": "1"}, "ansible\_date\_time": {"date": "2021-01-20", "day": "20", "epoch": "1611160850", … , "year": "2021"}, "ansible\_default\_ipv4": {"address": "10.0.2.15", "alias": "enp0s3", "broadcast": "10.0.2.255", "gateway": "10.0.2.2", "interface": "enp0s3", "macaddress": "08:00:27:e9:3d:e6", "mtu": 1500, "netmask": "255.255.255.0", "network": "10.0.2.0", "type": "ether"}, "ansible\_default\_ipv6": {}, "ansible\_device\_links": {"ids": {"sda": ["ata-VBOX\_HARDDISK\_VBbdc0f9c8-459ea1fd"], "sda1": ["ata-VBOX\_HARDDISK\_VBbdc0f9c8-459ea1fd-part1"], … , "interfaces": [], "ipv4": {"address": "192.0.2.1", "broadcast": "global", "netmask": "255.255.255.255", "network": "192.0.2.1"}, … , "ipv6": [{"address": "fe80::9002:c8ff:fee8:bb09", "prefix": "64", "scope": "link"}], "macaddress": "92:02:c8:e8:bb:09", "mtu": 1500, "promisc": false, "stp": …, "ipv4": {"address": "10.0.2.15", "broadcast": "10.0.2.255", "netmask": "255.255.255.0", "network": "10.0.2.0"}, "ipv6": [{"address": "fe80::a00:27ff:fee9:3de6", "prefix": "64", "scope": "link"}], "macaddress": "08:00:27:e9:3d:e6", "module": "e1000", "mtu": 1500, …, "HOME": "/home/devasc” , … , "PATH": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:", "PWD": "/home/devasc", "SHELL": "/bin/bash", "SHLVL": "0", "SSH\_CLIENT": "192.0.2.3 55752 22", … , "ansible\_fqdn": "labvm.vm", "ansible\_hostname": "labvm", "ansible\_hostnqn": "", … , "ansible\_is\_chroot": false, … "ansible\_lsb": {"codename": "focal", "description": "Ubuntu 20.04.1 LTS", "id": "Ubuntu", "major\_release": "20", "release": "20.04"}, "ansible\_machine": "x86\_64", "ansible\_machine\_id": "c6a52afed8564edfa075a362c20348b8", "ansible\_memfree\_mb": 258, , … , "ansible\_nodename": "labvm", "ansible\_os\_family": "Debian", "ansible\_pkg\_mgr": "apt", "ansible\_proc\_cmdline": {"BOOT\_IMAGE": "/boot/vmlinuz-5.4.0-37-generic", "quiet": true, "ro": true, "root": "UUID=fb261367-cf98-4bce-b682-42b3de0a8ab9", "vga": "792", … , "ansible\_processor": ["0", "GenuineIntel", "Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz", "1", "GenuineIntel", "Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz"], "ansible\_processor\_cores": 2, "ansible\_processor\_count": 1, … , "ansible\_product\_name": "VirtualBox", … , "ansible\_product\_version": "1.2", "ansible\_python": {"executable": "/usr/bin/python3", "has\_sslcontext": true, "type": "cpython", "version": {"major": 3, "micro": 2, "minor": 8, "releaselevel": "final", "serial": 0}, "version\_info": [3, 8, 2, "final", 0]}, "ansible\_python\_version": "3.8.2", … , "ansible\_system": "Linux", "ansible\_system\_capabilities": [""], "ansible\_system\_capabilities\_enforced": "True", "ansible\_system\_vendor": "innotek GmbH", "ansible\_uptime\_seconds": 88854, "ansible\_user\_dir": "/home/devasc", "ansible\_user\_gecos": "DevNet Associate,,,", "ansible\_user\_gid": 900, "ansible\_user\_id": "devasc", "ansible\_user\_shell": "/bin/bash", "ansible\_user\_uid": 900, … , "changed": false, "deprecations": [], "warnings": []} |

|  |
| --- |
| ***Ansible Example 3 (continued)***  Python examples to retrieve specific information from the Ansible JSON response data  **# PARSING AND FILTERING ANSIBLE JSON DATA**  **print("---------1--------")**  print("Showing dictionary keys at level 1")  #ansible\_dict = **json.loads**(ansible\_json\_doc)  print(ansible\_dict.keys())  **print("---------2--------")**  print("Showing keys of ansible facts at level 2")  print(ansible\_dict['ansible\_facts'].keys())  **print("---------3--------")**  print("Showing data below ansible facts: ip address")  print("IP Address: " + ansible\_dict["ansible\_facts"]["ansible\_default\_ipv4"]["address"])  **print('---------4--------')**  print("Showing data below ansible facts: ansible distribution")  print("Ansible Distribution: " + ansible\_dict["ansible\_facts"]["ansible\_distribution"])  print("Ansible Distribution Major: " +ansible\_dict["ansible\_facts"]["ansible\_distribution\_major\_version"])  print("Ansible Distribution Release: " +ansible\_dict["ansible\_facts"]["ansible\_distribution\_release"])  print("Ansible Distribution Version: " +ansible\_dict["ansible\_facts"]["ansible\_distribution\_version"])  **print('---------5--------')**  print("Showing data below ansible facts: kernel, nodename, os")  print("Ansible Kernel: " +ansible\_dict["ansible\_facts"]["ansible\_kernel"])  print("Ansible Nodename: " + ansible\_dict["ansible\_facts"]["ansible\_nodename"])  print("Ansible OS Family: " + ansible\_dict["ansible\_facts"]["ansible\_os\_family"])  print("Ansible PKG Manager: " + ansible\_dict["ansible\_facts"]["ansible\_pkg\_mgr"])  print("Ansible Python Version: " + ansible\_dict["ansible\_facts"]["ansible\_python\_version"])  **print('---------6--------')**  print("Showing data below ansible facts: ansible environment")  print("Ansible Home: " + ansible\_dict["ansible\_facts"]["ansible\_env"]["HOME"])  print("Ansible User: " + ansible\_dict["ansible\_facts"]["ansible\_env"]["USER"]) |

|  |
| --- |
| **Output**  ---------1--------  **Showing dictionary keys at level 1**  dict\_keys(['ansible\_facts', 'changed', 'deprecations', 'warnings'])  ---------2--------  **Showing keys of ansible facts at level 2 *(partial output)***  ['ansible\_all\_ipv4\_addresses', 'ansible\_all\_ipv6\_addresses', 'ansible\_default\_ipv4', 'ansible\_default\_ipv6''ansible\_distribution', 'ansible\_distribution\_file\_variety', 'ansible\_distribution\_major\_version', 'ansible\_distribution\_release', 'ansible\_distribution\_version', 'ansible\_env', 'ansible\_kernel', 'ansible\_kernel\_version', 'ansible\_python\_version']  ---------3--------  **Showing data below ansible facts: ip address**  IP Address: 10.0.2.15  ---------4--------  **Showing data below ansible facts: ansible distribution**  Ansible Distribution: Ubuntu  Ansible Distribution Major: 20  Ansible Distribution Release: focal  Ansible Distribution Version: 20.04  ---------5--------  **Showing data below ansible facts: kernel, nodename, os**  Ansible Kernel: 5.4.0-37-generic  Ansible Nodename: labvm  Ansible OS Family: Debian  Ansible PKG Manager: apt  Ansible Python Version: 3.8.2  ---------6--------  **Showing data below ansible facts: ansible environment**  Ansible Home: /home/devasc  Ansible User: devasc |

=> Experiment with the Python examples to improve your understanding

## Docker Response Data

### Docker

Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers[[37]](#footnote-36). Managing Docker images, containers and networks can be a daunting task. First of all, Docker supplies system managers with low-level information using inspect[[38]](#footnote-37). Also, it may be required to deal with data in JSON format output by Docker.

### Example Docker Inspect Response Data

When inspecting Docker images, containers or networks JSON response data are returned. Often the amount of data is very long for human consumption. Filtering and selecting is necessary. Two examples of Docker output in JSON format are presented below.

#### Docker Example 1: Docker Images Response Data

In the table below is shown the output from the following Docker command[[39]](#footnote-38):

**$ docker image inspect ubuntu**

|  |
| --- |
| **Example of Docker Output -- 1 (partial)**  [  {  "Id": "sha256:9140108b62dc87d9b278bb0d4fd6a3e44c2959646eb966b86531306faa81b09b",  "RepoTags": [  "ubuntu:latest"  ],  "RepoDigests": [  "ubuntu@sha256:bc2f7250f69267c9c6b66d7b6a81a54d3878bb85f1ebb5f951c896d13e6ba537"  ],  "Created": "2020-09-25T22:34:30.295807036Z",  "Container": "1046a5d685aef5c37d1829040ca8083b94e4c069ca4963f4b16a6ade2e077b06",  "ContainerConfig": {  "Hostname": "1046a5d685ae",  "Domainname": "",  "User": "",  "Env": [  "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"  ],  "Cmd": [  "/bin/sh",  "-c",  "#(nop) ",  "CMD [\"/bin/bash\"]"  ],  "Image": "sha256:4ff2090064e7e38688bce713d50f3202d227b3c89fecea1434271c912ccd47e0",  "Entrypoint": null,  },  "DockerVersion": "18.09.7",  "Config": {  "Hostname": "",  "Domainname": "",  "User": "",  "Env": [  "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"  ],  "Cmd": [  "/bin/bash"  ],  "Image": "sha256:4ff2090064e7e38688bce713d50f3202d227b3c89fecea1434271c912ccd47e0",  "Entrypoint": null,  },  "Architecture": "amd64",  "Os": "linux",  "Size": 72875723,  "VirtualSize": 72875723,  "GraphDriver": {  "Data": {  "LowerDir": "/var/lib/docker/overlay2/5d66f43.../diff",  "MergedDir": "/var/lib/docker/overlay2/c3bab84.../merged",  "UpperDir": "/var/lib/docker/overlay2/c3bab84.../diff",  "WorkDir": "/var/lib/docker/overlay2/c3bab84.../work"  },  "Name": "overlay2"  },  "RootFS": {  "Type": "layers",  "Layers": [  "sha256:d42a4fdf4b2ae8662ff2ca1b695eae571c652a62973c1beb81a296a4f4263d92",  "sha256:90ac32a0d9ab11e7745283f3051e990054616d631812ac63e324c1a36d2677f5",  "sha256:782f5f011ddaf2a0bfd38cc2ccabd634095d6e35c8034302d788423f486bb177"  ]  },  "Metadata": {  "LastTagTime": "0001-01-01T00:00:00Z"  }  }  ] |

***Remark****: Docker returns a list. In this example the list contains one item. It is possible to select the first item with the index [0]*

The code below select and filters the data returned shown in the output above.

|  |
| --- |
| **Python script to select response data from example 1**  **import json**  **print("---------1--------")**  print("Converting json string to dict, and showing keys at level 1")  docker\_dict = json.**loads**(docker\_json\_file)  print(docker\_dict[0].**keys()**)  **print("---------2--------")**  print("Converting dict to raw json")  docker\_json = json.**dumps**(docker\_dict)  **print("---------3--------")**  print("Filtering from dict")  print(docker\_dict[0]**["Created"]**)  print(docker\_dict[0]["**Architecture**"])  print(docker\_dict[0]["**Os**"]) |

Below is the output provided by the Python code.

|  |
| --- |
| **Output from the above Python script (partial)**  **---------1--------**  Converting json string to dict, and showing keys at level 1  dict\_keys(['Id', 'RepoTags', **'Created'**, 'Container', 'ContainerConfig', 'DockerVersion', 'Author', 'Config', **'Architecture'**, **'Os'**, 'Size', 'VirtualSize', 'GraphDriver', 'RootFS', 'Metadata'])  **---------2--------**  Converting dict to raw json  **---------3--------**  Filtering from dict  **2020-09-25T22:34:30.295807036Z**  **amd64**  **linux** |

=> Experiment with the Python examples to improve your understanding

#### Docker Example 2: Docker Network Response Data

In the table below is shown the output from the following Docker command[[40]](#footnote-39):

**$ docker network inspect bridge**

|  |
| --- |
| **Example of Docker Output -- 2 (partial)**  [  {  "Name": "bridge",  "Id": "566a72fc961157e2e71cc257fc2132beebc491a712967ef42bddcab70cbdbb23",  "**Created**": "2020-12-09T17:51:15.816558163Z",  "Scope": "local",  "Driver": "bridge",  "EnableIPv6": false,  "IPAM": {  "Driver": "default",  "Options": null,  "Config": [  {  "Subnet": "172.17.0.0/16",  "Gateway": "172.17.0.1"  }  ]  },  "Internal": false,  "Attachable": false,  "Ingress": false,  "ConfigFrom": {  "Network": ""  },  "ConfigOnly": false,  "**Containers**": {  "4e99a64e10dfcf6608a1d47f4349676c745bf234cebd52826d786db9a3be2811": {  "Name": "samplerunning",  "EndpointID": "22bbd3fa7e76635c3172446813fe5104537c8f69c6c23474272b379dede44fe7",  "MacAddress": "02:42:ac:11:00:03",  "**IPv4Address**": "172.17.0.3/16",  "IPv6Address": ""  }  },  "Options": {  "com.docker.network.bridge.default\_bridge": "true",  "com.docker.network.bridge.enable\_icc": "true",  "com.docker.network.bridge.enable\_ip\_masquerade": "true",  "com.docker.network.bridge.host\_binding\_ipv4": "0.0.0.0",  "com.docker.network.bridge.name": "docker0",  "com.docker.network.driver.mtu": "1500"  },  "Labels": {}  }  ] |

The code below select and filters data in the output above.

|  |
| --- |
| **Python script to select data from example 2**  **import json**  **print("---------1--------")**  print("Converting json string to dict, and showing keys at level 1")  docker\_dict2 = json.loads(docker\_json\_file2)  print(docker\_dict2[0].**keys**())  **print("---------2--------")**  print("Converting dict to raw json")  docker\_json2 = json.**dumps**(docker\_dict2)  print("Filtering from dict")  print(docker\_dict2[0]["Name"])  print(docker\_dict2[0]["Created"])  print(docker\_dict2[0]["Containers"]["4e99a64e...bd52826d786db9a3be2811"]["**IPv4Address**"])  *\*partial id* |

Below is the output provided by the Python code.

|  |
| --- |
| **Output from the above Python script**  **---------1--------**  Converting json string to dict, and showing keys at level 1  dict\_keys([**'Name'**, 'Id', **'Created'**, 'Scope', 'Driver', 'EnableIPv6', 'IPAM', 'Internal', 'Attachable', 'Ingress', 'ConfigFrom', 'ConfigOnly', **'Containers'**, 'Options', 'Labels'])  ---------2--------  Converting dict to raw json  Filtering from dict  **bridge**  **2020-12-09T17:51:15.816558163Z**  **172.17.0.3/16** |

## Practical Exercises: Filtering and Selecting JSON Data

In [Part 2](#bw6zfr6z9wm4) of this workshop the focus will be on generating JSON data structures based on basic information collected in a spreadsheet file. To prepare the practical exercises, we will explore the JSON structures that will be used in part 2.

### Webex Groups

Tasks

The JSON data for managing Webex groups require you to be able to do the following:

* Write Python code to select specific data elements, e.g. the name of a user
* Write Python code to loop through a list of data elements
* Write Python code to transform data types

Below is an example of JSON data that can be used to create spaces and groups.

|  |
| --- |
| groups\_struc = {  "groups": [  { "group": { "group\_id": "G-A"  , "group\_name": "DEVASC\_A" ,  "members": [  {"person\_id": "P-1" , "person\_name": "Noel", "email": "noel@odisee.be"},  {"person\_id": "P-2" , "person\_name": "Mary", "email": "mary@odisee.be"},  {"person\_id": "P-3" , "person\_name": "Jens", "email": "jens@odisee.be"}  ]  }  },  { "group": { "group\_id": "G-B"  , "group\_name": "DEVASC\_B" ,  "members": [  {"person\_id": "P-4" ,"person\_name": "Ives", "email": "ives@odisee.be"},  {"person\_id": "P-5" ,"person\_name": "John", "email": "john@odisee.be"},  {"person\_id": "P-6" ,"person\_name": "Alec", "email": "alec@odisee.be"}  ]  }  },  { "group": { "group\_id": "G-C" ,  "group\_name": "DEVASC\_C" ,  "members": [  {"person\_id": "P-7" ,"person\_name": "Matt", "email": "matt@odisee.be"},  {"person\_id": "P-8" ,"person\_name": "Paul", "email": "paul@odisee.be"},  {"person\_id": "P-9" ,"person\_name": "Elvi", "email": "elvi@odisee.be"}  ]  }  }  ]  } |

**Exercises**

1. Write a Python script to select the name of the first person in the first group
2. Write a Python script to make a list of all names and email addresses
3. Write a Python script to make a list of all names and email addresses of DEVASC\_B

|  |
| --- |
| **Code Example 1**  **### First group with first member, second group with first member**  ### Select first group, first person  resp\_a1 = groups\_struc["groups"]**[0]**["group"]["group\_name"]  resp\_a2 = groups\_struc["groups"]**[0]**["group"]["members"]**[0]**["person\_name"]  ### Select second group; first person  resp\_b1 = groups\_struc["groups"]**[1]**["group"]["group\_name"]  resp\_b2 = groups\_struc["groups"]**[1]**["group"]["members"]**[0]**["person\_name"]  **Output**  First group, First person  DEVASC\_A => Noel  Second group, First person  DEVASC\_B => Ives  And so on … (more groups, more members) |

|  |
| --- |
| **Code Example 2 ### All groups, members and email addresses**  **for** g **in** groups\_struc["groups"]:  print(g**["group"]["group\_name"]**)  for p in g["group"]["members"]:  print(p**["person\_name"]** + " => " + p**["email"]**)    **### Output**  **DEVASC\_A**  Noel => noel@odisee.be  Mary => mary@odisee.be  Jens => jens@odisee.be  **DEVASC\_B**  Ives => ives@odisee.be  John => john@odisee.be  Alec => alec@odisee.be  And so on … (if more groups) |

**More experiments with the JSON structure and test data**

Experiment with the code below to filter JSON data.

|  |
| --- |
| **print('------1---------')**  print(**type**(groups\_struc))  print(groups\_struc)  print('------1A--------')  # convert dict into json string  js\_groups = json.dumps(groups\_struc)  print(type(js\_groups))  print(js\_groups)  #print(json.dumps(groups\_struc, indent=2))  **print('------2---------')**  for **grp** in groups\_struc**["groups"]**:  **print('------2A--------')**  print(type(**grp**))  print(**grp**)  **print('------2B--------')**  print(**grp["group"]["group\_name"]**)  **print('------2C--------')**  for **per** in **grp["group"]["members"]**:  print(**per**["person\_name"] + " => " + **per**["email"])    **print('------3---------')**  print(groups\_struc.keys())  **print('------3A---------')**  print(groups\_struc["groups"]**[0]**.**keys**())  **print('------3B---------')**  print(groups\_struc["groups"]**[0]**["group"].**keys**())  **print('------3C---------')**  print(groups\_struc["groups"]**[0]**["group"]["members"]**[0]**.**keys**()) |

**Transforming JSON into YAML and XML**

Optionally, you might like to practice the conversion of DICT into JSON, JSON into YAML or into XML. Here are a few examples.

|  |
| --- |
| **Transforming DICT into JSON**  import json  js\_groups = json.dumps(groups\_struc)  print(json.**dumps**(groups\_struc, indent=2))  **Transforming JSON into YAML** import yaml  yaml\_data = yaml.**dump**(groups\_struc)  print(yaml\_data)  **Transforming JSON into XML**  from dicttoxml import dicttoxml  xml\_data = **dicttoxml**(groups\_struc)  print(xml\_data) |

=> Experiment with the Python examples to improve your understanding

### Network Devices

In [Part 2](#bw6zfr6z9wm4) of this workshop we will be generating JSON data structures to manage network devices. Below is a JSON structure for which it is required to write Python code in order to select specific data elements. Also, you will need to transform certain data types.

|  |
| --- |
| rack\_struc = {  "rack": [  { "device": { "dev\_id": "D1" ,  "dev\_name": "R1" ,  "role": "router" ,  "interfaces": [  {"interface": "GigabitEhternet1" ,  "ipaddress": "10.0.1.1",  "subnet\_mask": "255.255.255.0"},  {"interface": "GigabitEhternet2" ,  "ipaddress": "10.0.3.1",  "subnet\_mask": "255.255.255.0"},  {"interface": "GigabitEhternet3" ,  "ipaddress": "10.0.4.1",  "subnet\_mask": "255.255.255.0"}  ]  }  },  { "device": { "dev\_id": "D2" , "dev\_name": "C1" , "role": "core" ,  "interfaces": [  {"interface": "VLAN1" ,  "ipaddress": "10.0.1.2" ,  "subnet\_mask": "255.255.255.0"},  {"interface": "VLAN2" ,  "ipaddress": "10.0.2.1" ,  "subnet\_mask": "255.255.255.0"},  {"interface": "VLAN20" ,  "ipaddress": "10.0.20.1",  "subnet\_mask": "255.255.255.0"}  ]  }  },  { "device": { "dev\_id": "D3" , "dev\_name": "AC" , "role": "access" ,  "interfaces": [  {"interface": "VLAN2" ,  "ipaddress": "10.0.2.2",  "subnet\_mask": "255.255.255.0"}  ]  }  }  ]  } |

**Exercises**

1. Write a Python script to select the name and IP Address of the first device
2. Write a Python script to display all network devices, interfaces and IP addresses
3. Write a Python script to display all interfaces and IP addresses of device R1

|  |
| --- |
| **Code Sample**  **### All network devices interfaces and ip addresses**  for g in rack\_struc["rack"]:  print(g[**"device"]["dev\_name"]**)  for p in g["device"]["interfaces"]:  print(p**["interface"]**+" => "+p[**"ipaddress"]**)  **### Output**  **R1**  **GigabitEhternet1** => **10.0.1.1**  GigabitEhternet2 => 10.0.3.1  GigabitEhternet3 => 10.0.4.1  **C1**  VLAN1 => 10.0.1.2  VLAN2 => 10.0.2.1  VLAN20 => 10.0.20.1  **AC**  VLAN2 => 10.0.2.2  And so on … (if more network devices) |

**More experiments with the JSON structure and test data**

Experiment with the Python scripts below to select specific data elements from a larger JSON structure. This will help you solve the exercises given above.

|  |
| --- |
| **print('------1---------')**  print(type(rack\_struc))  print(rack\_struc)  **#print('------1A--------')**  js\_struc = json.**dumps**(rack\_struc)  #print(type(js\_struc))  #print(js\_struc)  #print(json.**dumps**(rack\_struc, indent=8))  **print('------1B--------')**  g = rack\_struc**["rack"][0]**  print(type(g))  print(g["device"].**keys**())  **print('------2---------')**  for g in rack\_struc**["rack"]**:  **print('------2A--------')**  print(type(g))  print(g)  print(g**["device"]["dev\_name"]**)  for p in g**["device"]["interfaces"]**:  print(p**["ipaddress"]**)    **print('------3---------')**  print("Keys device")  print(g**["device"]**.keys())  **print('------3A---------')**  print("Keys interfaces")  print(g["device"]["interfaces"]**[0]**.keys()) |

**Transforming JSON into YAML and XML**

Optionally, you might like to practice the conversion of DICT into JSON, JSON into YAML or into XML. Here are a few examples.

|  |
| --- |
| **Transforming DICT into JSON**  js\_struc = json.**dumps**(rack\_struc)  print(type(js\_struc))  #print(js\_struc)  print(json.**dumps**(rack\_struc, indent=4))  **Transforming JSON into YAML** import yaml  yaml\_data = yaml.**dump**(rack\_struc)  print(yaml\_data)  **Transforming JSON into XML**  from dicttoxml import dicttoxml  xml\_data = **dicttoxml**(rack\_struc)  print(xml\_data) |

=> Experiment with the Python examples to improve your understanding

### 

### Network Services

Below is a JSON structure relating to specific services on the nework for which it is required to write Python code and select specific data elements. Also, you will need to transform certain data types.

|  |
| --- |
| rack\_struc = {  "rack": [  { "server": { "dev\_id": "S1" , "server\_name": "svr1" , "domain": "biasc.be", "ip-address": "10.2.3.1" ,  "os": "linux" , "server\_type": "vm" ,  "services": [  {"service": "ad" , "service\_type": "vm", "protocol": "tcp", "port": "389"},  {"service": "dns", "service\_type": "vm", "protocol": "udp", "port": "53"},  {"service": "ntp", "service\_type": "vm", "protocol": "tcp", "port": "123"}  ]  }  },  { "server": { "dev\_id": "S2" , "server\_name": "svr2" , "domain": "biasc.be", "ip-address": "10.2.3.2" ,  "os": "linux" , "server\_type": "vm" ,  "services": [  {"service": "flask", "service\_type": "vm", "protocol": "tcp", "port": "8089" },  {"service": "db" , "service\_type": "vm", "protocol": "tcp", "port": "1521" }  ]  }  },  { "server": { "dev\_id": "S3" , "server\_name": "svr3" , "domain": "biasc.be" , "ip-address": "10.2.3.3",  "os": "linux" , "server\_type": "vm" ,  "services": [  {"service": "dns" , "service\_type": "vm", "protocol": "tcp", "port": "8089" },  {"service": "ntp" , "service\_type": "vm", "protocol": "tcp", "port": "8089" },  {"service": "dhcp", "service\_type": "docker", "protocol": "udp", "port": "67" }  ]  }  }  ]  }  #  *YANG leaves are comparable to a Python dict structure; YANG lists are comparable to an array (or list) in Python.* |

**Exercises**

1. Write Python code to make a list of all servers, services, ip addresses, protocol/ports
2. Write Python code to select the name of the first server
3. Write Python code to make a list of all services and protocol/ports of server S3

|  |
| --- |
| **Code Sample**  **### all servers, services, ip addresses, protocol/ports**  for g in devices\_struc["rack"]:  print("Server: " + g["**server**"]**["server\_name"]**)  print("IP Address: " + g**["server"]["ip-address"]**)  for p in g["server"]["services"]:  print(p["service"] + " => " + p["protocol"] + "/" + p**["port"]**)    **### Output**  **Server: svr1**  IP Address: **10.2.3.1**  ad => **tcp/389**  dns => udp/53  ntp => udp/123  **Server: svr2**  And so on … |

**More experiments with the JSON structure and test data**

Experiment with the Python scripts below to select specific data elements from a larger JSON structure. This will help you solve the exercises given above.

|  |
| --- |
| print('------1---------')  #print(devices\_struc)  #print('------1A--------')  js\_groups = json.**dumps**(rack\_struc)  print(js\_groups)  print(json.**dumps**(rack\_struc, indent=2))  print('------2---------')  for g in devices\_struc**["rack"]**:  print('------2A--------')  print(type(g))  print(g)  print(g**["server"]["services"]**)  for p in g["server"]["services"]:  print(p)    print('------3---------')  print(devices\_struc.keys())  print('------3A---------')  print(devices\_struc["rack"][0].keys())  print('------3B---------')  print(devices\_struc["rack"][0]["server"].keys())  print('------3C---------')  print(devices\_struc["rack"][0]["server"]["services"][0].keys()) |

**Experimenting with the target JSON structure and test data**Optionally, you might like to practice the conversion of DICT into JSON, JSON into YAML or into XML. Here are a few examples.

|  |
| --- |
| **Transforming DICT into JSON**  js\_struc = json.dumps(devices\_struc)  print(type(js\_struc))  #print(js\_struc)  print(json.dumps(devices\_struc, indent=4))  **Transforming JSON into YAML** import yaml  yaml\_data = yaml.dump(devices\_struc)  print(yaml\_data)  **Transforming JSON into XML**  from dicttoxml import dicttoxml  xml\_data = dicttoxml(devices\_struc)  print(xml\_data) |

=> Experiment with the Python examples to improve your understanding

# Part 2: Generating JSON Data

The second part of this Python & JSON Workshop is about generating JSON data structures. Spreadsheet files will serve as input for the JSON data structures to be generated. *Generating JSON form spreadsheet data involves transforming a 2D table in a JSON Tree Structure.*

There are three tasks with increasing difficulty:

* Webex Groups
* Network Devices
* Network Services

**Task 1 -- Webex Groups**

The main subtasks of this part of this workshop are the following:

* Reading and transforming an Excel sheet with groups and users into a JSON structure
* Creating, validating and testing the resulting JSON structure
* Selecting and filtering JSON data
* (optional) Transforming JSON data into YAML and XML (optional)
* (optional) Using Webex Teams API calls to create groups (spaces) and members (users)

**Task 2 -- Network Devices**

The main subtasks of this part of the workshop are the following:

* Reading and transforming an Excel sheet with new network devices and IP addresses into a JSON structure
* Creating, validating and testing the JSON structure
* Selecting and filtering JSON data
* (optional) Transforming JSON data into YAMl and XML
* (optional) Creating restconf script to configure network devices
* (optional) Creating ansible script to configure network devices

**Task 3 -- Network Services**

The main subtasks of this part of the workshop are the following:

* Reading and transforming an Excel sheet with a list of new servers and services into a JSON tree structure
* Creating, validating and testing a JSON structure
* (optional) Creating ansible playbook to configure services on servers
* (optional) Creating dockerfile to generate specific images for services
* (optional) Using jenkins to pipeline deployment of specific services

## TASK 1 - WEBEX GROUPS

In this task 1 you will be working with a Python script to read spreadsheet data about **new webex users** and generate a valid[[41]](#footnote-40) JSON data structure. Optionally, the resulting JSON data can be used to generate a **Python script** using RESTCONF or the **webexteamssdk** to create users and spaces automatically.

The data in task 1 are made up of three columns. In tasks 2 and 3 there will be more data columns to process.

### Source File

Here is an example of a source spreadsheet [webex\_groups.xlsx](https://drive.google.com/file/d/1GBEe5ATsUeGqM2luFuXiWQZEyJVwAm7e/view?usp=sharing) with three columns (group, name, email address)

|  |  |  |
| --- | --- | --- |
| **group** | **name** | **email** |
| GROUP\_ALPHA | Vincent Cassata | vincent.cassata@student.bxl.be |
| GROUP\_ALPHA | Giovanni Di Tulio | Giovanni .ditullio@student.bxl.be |
| GROUP\_ALPHA | Milan Vandevelde | milan.vandevelde@student.bxl.be |
| GROUP\_ALPHA | Tomas Vertessen | tomas.vertessen@student.bxl.be |
| GROUP\_ALPHA | Mehdi Dahli | mehdi.dahli@student.bxl.be |
| GROUP\_KAPPA | Ur Salangpour | ur.salangpour@student.bxl.be |
| GROUP\_KAPPA | Mon Gallin | mon.gallin@student.bxl.be |
| GROUP\_KAPPA | Artur Ikiya | artur.lkiya@student.bxl.be |
| GROUP\_KAPPA | Bram Vanbever | bram.vanbever@student.bxl.be |
| GROUP\_KAPPA | JR Ibara | jr.ibara@student.bxl.be |
| GROUP\_DELTA | Jona Ferbiest | jona.ferbiest@student.bxl.be |
| GROUP\_DELTA | Bart Siperius | bart.siperius@student.bxl.be |
| GROUP\_DELTA | Joren Huysegoms | joren.huysegoms2@student.bxl.be |
| GROUP\_DELTA | Sam Bulduk | sam.bulduk@student.bxl.be |
| GROUP\_DELTA | Ferre Van Malder | ferre.vanmalder@student.bxl.be |
| GROUP\_DELTA | Mikail Defossez | mikail.defossez@student.bxl.be |

**File name:** [**webex\_groups.xlsx**](https://drive.google.com/file/d/1GBEe5ATsUeGqM2luFuXiWQZEyJVwAm7e/view?usp=sharing)*\* Names are non-existent*

### 

### 

### Target Structure

The goal is to read the excel data and generate a json tree structure according to the example below:

|  |
| --- |
| **### JSON FORMAT -- Keys with id will not be used in the processing example**  groups\_struc = {  "groups": [  { "group": { "group\_id": "G-A" , "group\_name": "DEVASC\_A" ,  "members": [  {"person\_id": "P-1" , "person\_name": "Noel", "email": "noel@odisee.be"},  {"person\_id": "P-2" , "person\_name": "Mary", "email": "mary@odisee.be"},  {"person\_id": "P-3" , "person\_name": "Jens", "email": "jens@odisee.be"}  ]  }  },  { "group": { "group\_id": "G-B" , "group\_name": "DEVASC\_B" ,  "members": [  {"person\_id": "P-4" ,"person\_name": "Ives", "email": "ives@odisee.be"},  {"person\_id": "P-5" ,"person\_name": "John", "email": "john@odisee.be"},  {"person\_id": "P-6" ,"person\_name": "Alec", "email": "alec@odisee.be"}  ]  }  },  { "group": { "group\_id": "G-C" , "group\_name": "DEVASC\_C" ,  "members": [  {"person\_id": "P-7" ,"person\_name": "Matt", "email": "matt@odisee.be"},  {"person\_id": "P-8" ,"person\_name": "Paul", "email": "paul@odisee.be"},  {"person\_id": "P-9" ,"person\_name": "Elvi", "email": "elvi@odisee.be"}  ]  }  }  ]  } |

### Python Script Generate JSON Data Structure Based on an Excel Spreadsheet

### Starting with two spreadsheet rows

|  |
| --- |
| **### Simplified code: converting Excel in to initial Python dict** import **xlrd** # library to manage excel spreadsheetsimport **json** # library to manage JSON classes and functions  **wb = xlrd**.open\_workbook**("webex\_groups.xlsx")** sheet **= wb.**sheet\_by\_index(0) **# FIRST DATA ROW (ROW 0 contains column names) member\_dict["group"] =** sheet**.cell\_value(1, 0)  member\_dict["person\_name"] =** sheet**.cell\_value(1, 1) member\_dict["email"] =** sheet**.cell\_value(1, 2) print(member\_dict)**  **# SECOND DATA ROW member\_dict["group"] =** sheet**.cell\_value(2, 0)  member\_dict["person\_name"] =** sheet**.cell\_value(2, 1) member\_dict["email"] =** sheet**.cell\_value(2, 2) print(member\_dict)**  **Result Example in Python dict format:**  {**'group'**: 'GROUP\_ALPHA', **'person\_name'**: 'Vincent Cassata', **'email'**: 'vincent.cassata@student.bxl.be'}  {**'group'**: 'GROUP\_ALPHA', **'person\_name'**: 'Giovanni Di Tulio', **'email'**: 'Giovanni .ditullio@student.bxl.be'} |

### Complete script

The Python script below is able to read data in an Excel spreadsheet and transform them into a JSON data structure. In order to make the code readable a number of functions are being defined:

* **find\_all\_persons\_and\_groups**
* **make\_list\_of\_groups**
* **attach\_members\_to\_groups**

The **main()** function is called when the Python program is run directly.

#### Code Walkthrough

Here is the code in 7 steps in order to generate and validate the JSON data structures based on the excel spreadsheet [webex\_groups.xlsx](https://drive.google.com/file/d/1GBEe5ATsUeGqM2luFuXiWQZEyJVwAm7e/view?usp=sharing).

Task 1 - Step 1: Define Python Data Rules

**### REWRITING RULES TO GENERATE THE DATA STRUCTURE**

**Most of the time you will have to manage structures of type dict and list, as a result of the JSON exhange data types**

**member\_dict**  => {"person\_name": "x", "email": "y", "group":"z"}

**member\_list**  => [member\_dict]

**group\_dict**  => {group\_name, member\_list} | {group\_name, [member\_dict]}

**group\_list**  => [group\_dict]

**groups\_struc**  => {group\_list} | {[group\_dict]}

Step 2: Read Two Excel Records (example)

**### Simplified code: converting Excel into initial Python dict  
import xlrd** # library to manage excel spreadsheets **import json** # library to manage JSON classes and functions  
**wb = xlrd**.open\_workbook**("webex\_groups.xlsx")  
sheet = wb.**sheet\_by\_index(0) # read data from the first tab  
  
**member\_dict["group"] = sheet.cell\_value(1, 0)   
member\_dict["person\_name"] = sheet.cell\_value(1, 1)  
member\_dict["email"] = sheet.cell\_value(1, 2)  
  
member\_dict["group"] = sheet.cell\_value(2, 0)   
member\_dict["person\_name"] = sheet.cell\_value(2, 1)  
member\_dict["email"] = sheet.cell\_value(2, 2)  
print(member\_dict)  
  
Result Example in Python dict format:**   
{**'group'**: 'GROUP\_ALPHA', **'person\_name'**: 'Vincent Cassata', **'email'**: 'vincent.cassata@student.bxl.be'} {**'group'**: 'GROUP\_ALPHA', **'person\_name'**: 'Giovanni Di Tulio', **'email'**: 'Giovanni.ditullio@student.bxl.be'}

Step 3: Read All Excel Records (loop)

**### Simplified code: converting all Excel data into initial Python dict  
  
import xlrd** # library to manage excel spreadsheets **import json** # library to manage JSON classes and functions  
 **def find\_all\_persons\_and\_groups(xlf):  
 ### READ EXCEL FILE AND RETURN NUMBER OF ROWS  
 wb = xlrd.open\_workbook(xlf)  
 sheet = wb.sheet\_by\_index(0)  
 number\_rows = sheet.nrows  
 all\_members = []  
 for r in range(number\_rows):  
 if r > 0: ### first row contains columns names  
 COL\_A = sheet.cell\_value(r, 0) #### column A  
 COL\_B = sheet.cell\_value(r, 1) #### column B  
 COL\_C = sheet.cell\_value(r, 2) #### column C  
 member\_dict["group"] = COL\_A  
 member\_dict["person\_name"] = COL\_B   
 member\_dict["email"] = COL\_C  
 all\_members.append(member\_dict.copy())   
 return all\_members**

**Function Call in Python Script:   
member\_list = find\_all\_persons\_and\_groups("webex\_groups.xlsx")  
  
Result Example:** [{**'group'**: 'GROUP\_ALPHA', **'person\_name'**: 'Vincent Cassata', **'email'**: 'vincent.cassata@student.bxl.be'}, {**'group'**: 'GROUP\_ALPHA', **'person\_name'**: 'Giovanni Di Tulio', **'email'**: 'Giovanni .ditullio@student.bxl.be'}, … ]

Step 4: Tree Structure Level 1

**### Simplified code: making list of groups from initial Python dict**

**def make\_list\_of\_groups(membr\_list):**

**all\_groups = []**

**mem = None**

**for rec in membr\_list:**

**g = rec["group"]**

**if mem != g:**

**all\_groups.append(g)**

**mem = g**

**return all\_groups**

**Function Call** in Python Script: **group\_list = make\_list\_of\_groups(member\_list)   
  
Result Example:** **list of groups  
['GROUP\_ALPHA', 'GROUP\_KAPPA', 'GROUP\_DELTA']**

Step 5: Tree Structure Level 1 & 2

**### Simplified code: attaching group members (L2) to their group (L1)**

**def attach\_members\_to\_groups(group\_name, membr\_list):**

membr\_dict = {}

all\_group\_members = [membr\_dict]

#print(loc\_group)

for membr in membr\_list:

if membr["group"] == group\_name:

#print(membr)

if membr["person\_name"] != None:

membr\_dict["person\_name"] = membr["person\_name"]

membr\_dict["email"] = membr["email"]

#print(loc\_m\_dict)

all\_group\_members.append(membr\_dict.copy())

return all\_group\_members

**Function Call in Python Script:   
for** group\_rec **in** group\_list:  
 all\_members = attach\_members\_to\_groups(group\_rec, member\_list)  
**Result Example**  
{**'group'**: {'group': {'group\_name': 'GROUP\_ALPHA', 'members': [{'person\_name': 'Vincent Cassata', 'email': 'vincent.cassata@student.bxl.be'}, {'person\_name': 'Giovanni Di Tulio', 'email': 'Giovanni.ditullio@student.bxl.be'}, ...

Step 6: main()

**### Simplified code: calling three functions + creating final data structure  
### It is necessary to have an Excel file  
### with the name"webex\_groups.xlsx" or change the name)**

**def main():**

**member\_list = find\_all\_persons\_and\_groups("webex\_groups.xlsx")**

**group\_list = make\_list\_of\_groups(member\_list)**

**all\_members = []**

**for group\_rec in group\_list:**

**all\_members = attach\_members\_to\_groups(group\_rec, member\_list)**

**del all\_members[0] #### delete the first element, which is a copy of the last element**

**group\_dict["group"] = { "group": {"group\_name": group\_rec , "members": all\_members }}**

**groups\_struc['groups'].append(group\_dict["group"])**

**js\_groups = json.dumps(groups\_struc)**

**#### execute main() when called directly**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

**Function Call in Python Script:**#### execute main() when called directly  **if \_\_name\_\_ == '\_\_main\_\_':  
 main()**

**Result Example**: next step

Step 7: Verify or Validate Results (RAW)

{  
 "groups": [{  
 "group": {  
 "group\_name": "GROUP\_ALPHA",  
 "members": [{  
 "**person\_name**": "Vincent Cassata",  
 "**email**": "vincent.cassata@student.bxl.be"  
 },  
 {  
 "**person\_name**": "Giovanni Di Tulio",  
 "**email**": "Giovanni.ditullio@student.bxl.be"  
 },  
 {  
 "**person\_name**": "Milan Vandevelde",  
 "**email**": "milan.vandevelde@student.bxl.be"  
 }, …  
 ]  
}

### Webex API

Here is a code snippet showing how you can use the JSON data generated above to **create spaces and groups in Webex Teams**. *While you are running the code, it is a good idea to check the results live in Webex at the same time.*

|  |
| --- |
| **### Access Token 12 hours: https://developer.webex.com/docs/api/getting-started ### (login required)**  **current\_access\_token** = "Get your own token" ### add your own bearer token  **### ADD NEW SPACES AND MEMBERS TO WEBEX TEAMS**  **### THIS CODE ONLY WORKS IF YOU ARE ABLE TO GENERATE A CORRECT GROUPS\_STRUC**  **import requests**  **import json**  from **webexteamssdk** import WebexTeamsAPI  api = WebexTeamsAPI(access\_token = **current\_access\_token**)  print("Creating spaces + members --> from Excel spreadsheet in the previous cell")  **access\_token = current\_access\_token** |

|  |
| --- |
| ### Two alternative main functions have been created  ### main2() => RESTFUL API  ### main() => WEBEXTEAMSSDK  **def main2(): # using rest api**    url = '<https://api.ciscospark.com/v1/rooms>'  headers = { 'Authorization': 'Bearer {}'.format(access\_token),  'Content-Type': 'application/json' }  for rec in **groups\_struc["groups"]**:  create\_group\_name = rec["group"]["group\_name"]  Payload\_space = {"title": create\_group\_name}  if payload\_space["title"] != None: ### avoid errors if room title unknown  res\_space = requests.**post**(url, headers=headers, json=payload\_space)    if res\_space.status\_code < 300:  NEW\_SPACE\_ID = res\_space.json()["id"]    for mbr in rec["group"]["members"]:  room\_id = NEW\_SPACE\_ID  person\_email = mbr["email"]  url2 = '<https://api.ciscospark.com/v1/memberships>'  payload\_member = {'roomId': room\_id, 'personEmail': person\_email}  res\_member = requests.**post**(url2, headers=headers, json=payload\_member)  **def main(): # using webxteamssdk**  **for rec in** groups\_struc["groups"]**:**  # Create a new demo room  demo\_room = api.rooms.create(rec["group"]["group\_name"])  # Add people to the new demo room  for email in rec["group"]["members"]:  api.memberships.create(demo\_room.id, personEmail=email["email"])  **#### execute main() when called directly**  **if \_\_name\_\_ == "\_\_main\_\_":**  **main2() ### or main()** |

=> Experiment with the Python examples to improve your understanding

## TASK 2 - IP DEVICES

In this task 2 you will be working with a Python script to read spreadsheet data about network devices and generate a valid[[42]](#footnote-41) JSON data structure. For practical purposes the number of levels will be restricted to two levels. *Generating a YANG compliant JSON structure is more difficult because there are more hierarchical levels.*

### Source File

Below is an example of a source file [ipdevices.xlsx](https://drive.google.com/file/d/11fa0q7liDMzm9ZEOsm64jSadV5Nzlt2G/view?usp=sharing) with five columns (device, role, interface, ip address, subnet mask). Based on the data in the table the task is to generate a valid[[43]](#footnote-42) JSON structure. Afterwards the JSON data can optionally be used to generate RESTCONF scripts or ANSIBLE playbooks.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **device** | **role** | **interface** | **ipaddress** | **subnetmask** |
| RTR1 | router | GigabitEthernet 0 | 192.0.2.254 | 255.255.255.0 |
| RTR1 | router | GigabitEthernet 1 | 10.0.1.1 | 255.255.255.0 |
| RTR1 | router | GigabitEthernet 2 | 10.0.2.1 | 255.255.255.0 |
| MLS1 | core switch | VLAN 1 | 10.0.1.2 | 255.255.255.0 |
| MLS1 | core switch | VLAN 2 | 10.0.2.1 | 255.255.255.0 |
| MLS2 | core switch | VLAN 1 | 10.0.1.3 | 255.255.255.0 |
| MLS2 | core switch | VLAN 2 | 10.0.2.2 | 255.255.255.0 |
| ASW2 | access switch | VLAN 1 | 10.0.1.2 | 255.255.255.0 |
| ASW3 | access switch | VLAN 1 | 10.0.1.3 | 255.255.255.0 |
| ASW4 | access switch | VLAN 1 | 10.0.1.4 | 255.255.255.0 |
| ASW5 | access switch | VLAN 1 | 10.0.1.5 | 255.255.255.0 |
| ASW6 | access switch | VLAN 2 | 10.0.2.6 | 255.255.255.0 |
| ASW7 | access switch | VLAN 2 | 10.0.2.7 | 255.255.255.0 |
| ASW8 | access switch | VLAN 2 | 10.0.2.8 | 255.255.255.0 |
| ASW9 | access switch | VLAN 2 | 10.0.2.9 | 255.255.255.0 |

**File name:** [**ipdevices.xlsx**](https://drive.google.com/file/d/11fa0q7liDMzm9ZEOsm64jSadV5Nzlt2G/view?usp=sharing)

### Target Structure

The task is to read the Excel data from [ipdevices.xlsx](https://drive.google.com/file/d/11fa0q7liDMzm9ZEOsm64jSadV5Nzlt2G/view?usp=sharing) and generate a json tree structure format according to the example below:

|  |
| --- |
| **### JSON FORMAT -- Keys with id not used in the processing example**  rack\_struc = {  "rack": [  { "device": { "dev\_id": "D1" ,  "dev\_name": "R1" ,  "role": "router" ,  "interfaces": [  {"interface": "GigabitEhternet1" ,  "ipaddress": "10.0.1.1",  "subnet\_mask": "255.255.255.0"},  {"interface": "GigabitEhternet2" ,  "ipaddress": "10.0.3.1",  "subnet\_mask": "255.255.255.0"},  {"interface": "GigabitEhternet3" ,  "ipaddress": "10.0.4.1",  "subnet\_mask": "255.255.255.0"}  ]  }  },  { "device": { "dev\_id": "D2" , "dev\_name": "C1" , "role": "core" ,  "interfaces": [  {"interface": "VLAN1" ,  "ipaddress": "10.0.1.2" ,  "subnet\_mask": "255.255.255.0"},  {"interface": "VLAN2" ,  "ipaddress": "10.0.2.1" ,  "subnet\_mask": "255.255.255.0"},  {"interface": "VLAN20" ,  "ipaddress": "10.0.20.1",  "subnet\_mask": "255.255.255.0"}  ]  }  },  { "device": { "dev\_id": "D3" , "dev\_name": "AC" , "role": "access" ,  "interfaces": [  {"interface": "VLAN2" ,  "ipaddress": "10.0.2.2",  "subnet\_mask": "255.255.255.0"}  ]  }  }  ]  } |

### Generating JSON data structures based on an Excel spreadsheet

Here is the Python code that will generate the JSON data structures based on the excel spreadsheet [ipdevices.xlsx](https://drive.google.com/file/d/11fa0q7liDMzm9ZEOsm64jSadV5Nzlt2G/view?usp=sharing).

|  |
| --- |
| ### FUNCTIONS TO CREATE (A) list of devices and (B) list of device interfaces  ### Using an excel file as input for a Python script  ### This script creates a JSON structure based on information an excel spreadsheet  ### You have to understand Python dict and Python lists  ### Contact yvan.rooseleer@biasc.be if you have questions  ### It's up to you to ### WRITE YOUR OWN CODE ### where this is indicated  ### Validate your JSON structure  ### Use print statements to understand return values and return types  ### There is a simplified solution (main2) and a more difficult solution (main)  **import xlrd # library to manage excel spreadsheets**  **import json**  ### RULES  inventory\_dict = {} #### {"interface": "gi0/1", "ip\_address": "1.2.2.1", "subnet\_mask": "255.255.255.0", ...  inventory\_list = [] #### [inventory\_dict]  interface\_dict = {} #### {"interface": "gi0/1", "ip\_address": "1.2.2.1", "subnet\_mask": "255.255.255.0"}  interface\_list = [] #### [interface\_dict]  dev\_dict = {} #### {"device": {dev\_name": "n", "role": "r", interfaces": interface\_list}}  dev\_list = [] #### [dev\_dict]  rack\_struc = {} #### {dev\_dict\_list}  rack\_struc["rack"] = []  **def find\_all\_device\_interfaces(xlf):**  **### READ EXCEL FILE AND RETURN NUMBER OF ROWS**  **wb = xlrd.open\_workbook(xlf)**  **sheet = wb.sheet\_by\_index(0)**  **number\_rows = sheet.nrows**  **dev\_interfaces = []**  **for r in range(number\_rows):**  **if r > 0: ### first row contains columns names**  **COL\_A = sheet.cell\_value(r, 0) #### column A**  **COL\_B = sheet.cell\_value(r, 1) #### column B**  **COL\_C = sheet.cell\_value(r, 2) #### column C**  **COL\_D = sheet.cell\_value(r, 3) #### column D**  **COL\_E = sheet.cell\_value(r, 4) #### column E**  **inventory\_dict["device"] = COL\_A**  **inventory\_dict["role"] = COL\_B**  **inventory\_dict["interface"] = COL\_C**  **inventory\_dict["ipaddress"] = COL\_D**  **inventory\_dict["subnetmask"] = COL\_E**  **dev\_interfaces.append(inventory\_dict.copy()) # need to use copy()**  **return dev\_interfaces**  **def make\_list\_of\_devices\_and\_roles(inventory):**  **dev\_list = []**  **dev\_dict = {}**  **mem = {}**  **for rec in inventory:**  **dev\_dict["dev\_name"] = rec["device"]**  **dev\_dict["role"] = rec["role"]**  **if mem != dev\_dict["dev\_name"]:**  **dev\_list.append(dev\_dict.copy()) # need to use copy()**  **mem = dev\_dict["dev\_name"]**  **#for rec in loc\_g:**  **#print(rec)**  **#del loc\_g[0] ### if last item copied as first item**  **return dev\_list**  **def attach\_interfaces\_to\_devices(dev\_name, inventory):**  **intf\_dict = {}**  **intf\_list = [intf\_dict]**  **for item in inventory:**  **if item["device"] == dev\_name:**  **if item["device"] != None:**  **intf\_dict["interface"] = item["interface"]**  **intf\_dict["ipaddress"] = item["ipaddress"]**  **intf\_dict["subnetmask"] = item["subnetmask"]**  **intf\_list.append(intf\_dict.copy()) # need to use copy()**  **del intf\_list[0] ### if last item copied as first item**  **return intf\_list** |

|  |
| --- |
| **def main():**  inventory\_list = find\_all\_device\_interfaces("devices\_ip.xlsx")  device\_list = make\_list\_of\_devices\_and\_roles(inventory\_list)  for device\_rec in device\_list:  intf\_list = attach\_interfaces\_to\_devices(device\_rec["dev\_name"], inventory\_list)  dev\_dict["device"] = { "device": device\_rec , "interfaces": intf\_list }  rack\_struc["rack"].append(dev\_dict["device"])  js\_rack = json.dumps(rack\_struc)  #### execute main() when called directly  if \_\_name\_\_ == '\_\_main\_\_':  main() |

*The Python code above defines two main() funtions. The main2 functions was created first and is easier if you are creating the Python program on your own.*

=> Experiment with the Python examples to improve your understanding

This concludes part 2 of the workshop documentation. It is important to practice and experiment with the code examples.

# References

Ansible, gather\_facts – Gathers facts about remote hosts, , Red Hat, 2019

URL: [https://docs.ansible.com/ansible/2.9\_ja/modules/gather\_facts\_module.html#gather-facts-gathers-facts-about-remote-host](https://docs.ansible.com/ansible/2.9_ja/modules/gather_facts_module.html#gather-facts-gathers-facts-about-remote-hosts)

Ansible User Guide, Red Hat, 2019

URL: <https://docs.ansible.com/ansible/latest/user_guide/index.html>

Bierman A. & Bjorklund M., RESTCONF Protocol, IETF RFC 8040, 2017

URL: <https://tools.ietf.org/html/rfc8040>

Bjorklund M. , Ed., YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF), RFC 6020, 2010

URL: <https://tools.ietf.org/html/rfc6020>

Bjorklund M. Ed., The YANG 1.1 Data Modeling Language, IETF RFC 7950, 2016

URL: <https://tools.ietf.org/html/rfc7950>

Cepalia A, What is JSON? Working With JSON Data in Python, RealPython, 2021  
URL: <https://realpython.com/lessons/what-is-json/>

Cisco DNA Assurance User Guide, Release 1.3, Cisco, 2020

URL: <https://www.cisco.com/c/en/us/td/docs/cloud-systems-management/network-automation-and-management/dna-center-assurance/1-3/b_cisco_dna_assurance_1_3_ug.html>

Cisco Networking Academy, DevNet Associate 1.0, 2020

URL: [www.netacad.com](http://www.netacad.com)

Cisco Networking Academy, Emerging Technology Workshop: Experimenting with REST APIs using Webex Teams, 2020

URL: [www.netacad.com](http://www.netacad.com)

Cisco Networking Academy, Emerging Technology Workshop: Network Programmability with Cisco APIC-EM, *no longer supported*

URL: [www.netacad.com](http://www.netacad.com)

Cisco Networking Academy

Emerging Technology Workshop: Model-Driven Programmability, 2020

URL: [www.netacad.com](http://www.netacad.com)

Cisco Networking Academy, PCAP - Programming Essentials in Python, Python Institute

URL: [https://netacad.com](https://lms.netacad.com/course/view.php?id=260535)

Cisco Webex for Developers, Getting Started with the Webex API, Cisco

URL: <https://developer.webex.com/docs/api/getting-started>

Cohoe J., The wait is over for Cisco YANg SUITE Tooling, BrightTALK, Feb 2, 2021

URL: <https://www.brighttalk.com/webcast/17628/455586>

DevNet Labs, APIC-EM APIs with Python: Part 1 - The Basics,   
Cisco DevNet Developer Network (no longer supported?)

URL1: <https://developer.cisco.com/learning/lab/apic-em-basic/step/1>  
URL2: <https://developer.cisco.com/learning/lab/apic-em-basic/step/2>

URL3: <https://developer.cisco.com/learning/lab/apic-em-basic/step/3>

Docker, Docker overview, 2020  
URL: <https://docs.docker.com/get-started/overview/>

Docker, Docker inspect, 2020

URL: <https://docs.docker.com/engine/reference/commandline/inspect/>

JSON encoder and decoder, json.org

URL: <https://docs.python.org/3/library/json.html>

JSONLint - The JSON Validator  
URL: <https://jsonlint.com/>

Preston H, Introducing Cisco DNA Center Platform APIs and Programmability

Cisco DevNet Developer Network, 2020

URL1: <https://developer.cisco.com/video/net-prog-basics/03-network_controllers/dnac_intro>

URL2: <https://github.com/CiscoDevNet/netprog_basics>

Preston H., Network Device APIs

Cisco DevNet Developer Network, 2020

URL1: <https://developer.cisco.com/video/net-prog-basics/02-network_device_apis>

URL2: <https://github.com/CiscoDevNet/netprog_basics>

Preston H., Your Network as Code

Cisco DevNet Developer Network, 2020

URL: <https://developer.cisco.com/video/net-prog-basics/05-netdevops/ansible_part_3>

Python, Built-in Types, Python org

URL: <https://docs.python.org/3/library/stdtypes.html>

Radford A. e.a., Cisco IOS XE Programmability, Automating Device Lifecycle Management, Cisco Book Sprint  
URL: <https://www.cisco.com/c/dam/en/us/products/collateral/enterprise-networks/nb-06-ios-xe-prog-ebook-cte-en.pdf>

Tail-f, INSTANT YANG

URL: <https://www.tail-f.com/wordpress/wp-content/uploads/2014/02/Tail-f-Instant-YANG.pdf>

**---**

**Slide Deck for This Workshop**

**URL:** <https://docs.google.com/presentation/d/1vGFxT2L7Nd7WXp8RvgApYCb3LQglFlZSWvDpFveRqqA/edit?usp=sharing>

**Data Files for the Practical Examples and Exercises of This Workshop**

**URL:** <https://drive.google.com/drive/folders/1Ip6nw4uxTHb5t9TxHEC3xcRPGzsTgEnH?usp=sharing>

**Doc Created to Prepare This Workshop**

**URL:**   
<https://docs.google.com/document/d/1jWNaW4OMkcCu1wnXZbQSMKxy23dIcq6ds4CBnEEXe0Q/edit?usp=sharing>

1. The slide deck for this workshop is available online [here](https://docs.google.com/presentation/d/1vGFxT2L7Nd7WXp8RvgApYCb3LQglFlZSWvDpFveRqqA/edit?usp=sharing) [↑](#footnote-ref-0)
2. The references are at the [back](https://n-xovwktmtjsnaxyc2mwes2xu7pohqedmdm6zjw5q-0lu-script.googleusercontent.com/userCodeAppPanel#) of this document and are useful for anyone new to the topics discussed in this workshop [↑](#footnote-ref-1)
3. A good introduction into network programmability you can find [here](#8f0nplukmsyf) [↑](#footnote-ref-2)
4. [Radford](#8f0nplukmsyf), p.30 [↑](#footnote-ref-3)
5. [Radford](#8f0nplukmsyf), p.34 [↑](#footnote-ref-4)
6. [Radford](#8f0nplukmsyf), p.31 [↑](#footnote-ref-5)
7. Working With JSON Data in Python is explained [here](#m3vynal8x4gj) [↑](#footnote-ref-6)
8. More information about Network as Code is [here](#fsb52q3zjpoi) [↑](#footnote-ref-7)
9. The complete code you can find [here](https://drive.google.com/file/d/10t-pCl3UL-ZiSUECHcIGSr3WUHIzEfEr/view?usp=sharing) [↑](#footnote-ref-8)
10. If you need information about Python built-in data type go [here](#1a1gfbuthcbe) [↑](#footnote-ref-9)
11. Information taken from the [Python Institute Course](#ro1vnnmscehh), Section 4.6.1.11 [↑](#footnote-ref-10)
12. [Radford](#8f0nplukmsyf), p.35 [↑](#footnote-ref-11)
13. JSON Data Serialization in Python is explained [here](#m3vynal8x4gj) [↑](#footnote-ref-12)
14. This is a lab in the [DevNet Associate Course](#kix.raalb1ojxefr) from Cisco Networking Academy [↑](#footnote-ref-13)
15. [Radford](#8f0nplukmsyf), p.50 [↑](#footnote-ref-14)
16. More information about YANG Modeling you can find [here](#b6dq73m777k7) [↑](#footnote-ref-15)
17. [Radford](#8f0nplukmsyf), p.33 [↑](#footnote-ref-16)
18. [Radford](#8f0nplukmsyf), p.62 [↑](#footnote-ref-17)
19. Explanation by Hank Preston in a video can be found [here](#76hgu5fescur) [↑](#footnote-ref-18)
20. Information about the YANG Suite from Cisco is available [here](#oi2smpyintma) [↑](#footnote-ref-19)
21. Explanation by Hank Preston in a video can be found [here](#76hgu5fescur) [↑](#footnote-ref-20)
22. [Radford](#8f0nplukmsyf), p.54 [↑](#footnote-ref-21)
23. Background in formation on the topic of YANG you can find [here](#b6dq73m777k7) [↑](#footnote-ref-22)
24. [Radford](#8f0nplukmsyf), p.54 [↑](#footnote-ref-23)
25. More information about Leaf nodes you can find [here](#a3tv7xon9a01). [↑](#footnote-ref-24)
26. The complete lab can be found [here](#sjwc68pxsbry) [↑](#footnote-ref-25)
27. The Python code for **lab1-1-post-ticket.py** can be found [here](#bjb965s908ax) [↑](#footnote-ref-26)
28. A complete description of **RESTCONF** is available [here](#7jdouuqa2fz0) [↑](#footnote-ref-27)
29. [Radford](#8f0nplukmsyf), p.75 [↑](#footnote-ref-28)
30. JSON Pretty Print is explained [here](#xp1xcmnni09g) [↑](#footnote-ref-29)
31. The video course is available through this [link](#b9yi4klv4khg) [↑](#footnote-ref-30)
32. More information about DNA Center Assurance you can find [here](#76qu6874eor7) [↑](#footnote-ref-31)
33. More information about Webex for Developers you can find [here](#gxhzuunifhbn) [↑](#footnote-ref-32)
34. The Ansible User Guide you can find [here](#vllsuoh3fejj) [↑](#footnote-ref-33)
35. A video course on DevNet for configuration management you can find [here](#b9yi4klv4khg) [↑](#footnote-ref-34)
36. More information about gather facts you find [here](#nh6g4br1418i) [↑](#footnote-ref-35)
37. More information to start using Docker is [here](#7e4zayj6w506) [↑](#footnote-ref-36)
38. Docker inspect is explained [here](#ydqrjc8sipqw) [↑](#footnote-ref-37)
39. Here is the complete [docker response](https://drive.google.com/file/d/1DM641DhTuJ7FxbZJEGqIR8dhYPLEGaoX/view?usp=sharing) [↑](#footnote-ref-38)
40. Here is the complete [output file](https://drive.google.com/file/d/15oHKwKaFdTXx62GjhY6cvCUuNyx-cWhu/view?usp=sharing) [↑](#footnote-ref-39)
41. An online JSON validator you can find [here](#hl0cv8vb2wlv) [↑](#footnote-ref-40)
42. An online JSON validator you can find [here](#hl0cv8vb2wlv) [↑](#footnote-ref-41)
43. idem [↑](#footnote-ref-42)