Getting Started with Network Automation
Welcome!

Please grab a white lab card from the front.
Use your own laptop.

Bonus: Bring the GitHub up ahead of time
https://github.internet2.edu/sbyrnes/2025-nanog93-tutorial-automation

internet2.edu

Getting Started with Network Automation

03-FEB-2025

Shannon Byrnes
Sr NetDevOps Engineer, Internet2

ABOUT INTERNET2

NETWORK

CLOUD

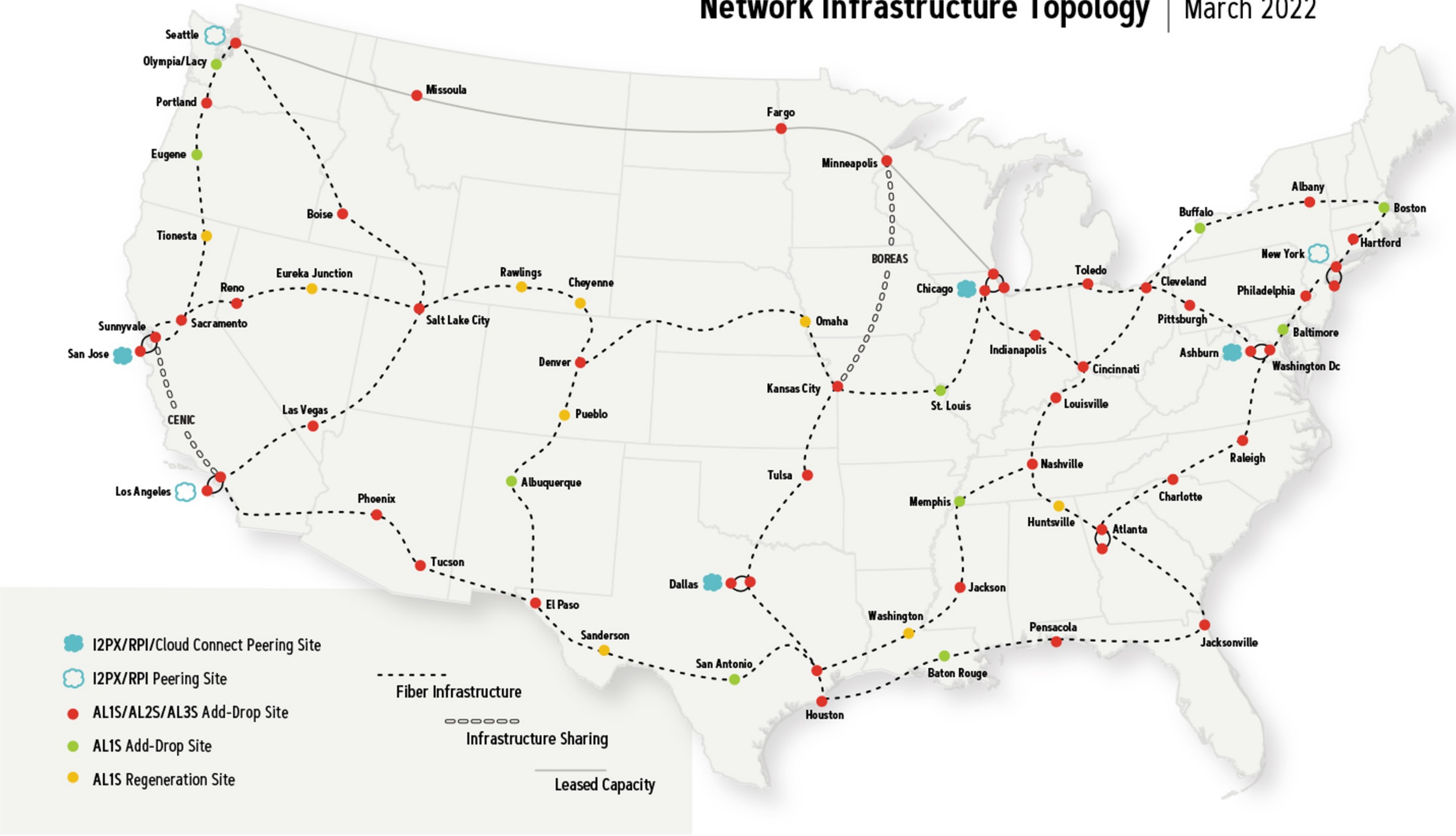
Internet2 is a non-profit, member-driven advanced technology community providing a secure high-speed national network, eduroam global Wi-Fi access service, cloud solutions, research support, and services and training tailored for research and education (R&E). Our community includes higher education, research institutions, government entities, corporations, and cultural organizations.

Through InCommon, Internet2 provides security, privacy, and identity and access management tools built for R&E.

SECURITY

COMMUNITY

Network Infrastructure Topology | March 2022



Agenda

1. What is Network Automation?
2. Common Tools
3. Lab Login and Preparation
4. Lab1: Reading Configuration and Operational Data
5. Lab2: Standardizing Configuration
6. Wrap-up

What is Network Automation?

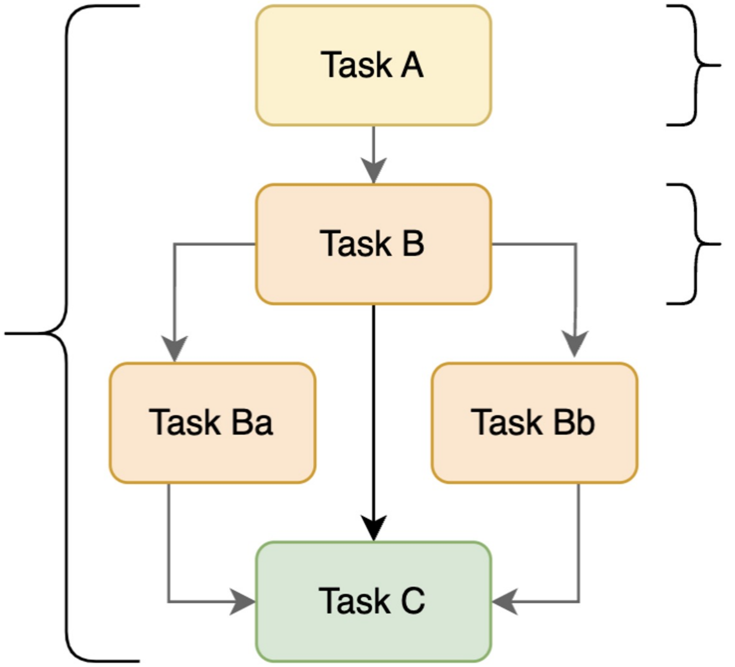
04-FEB-2025

NANOG™

For the purposes of this tutorial...

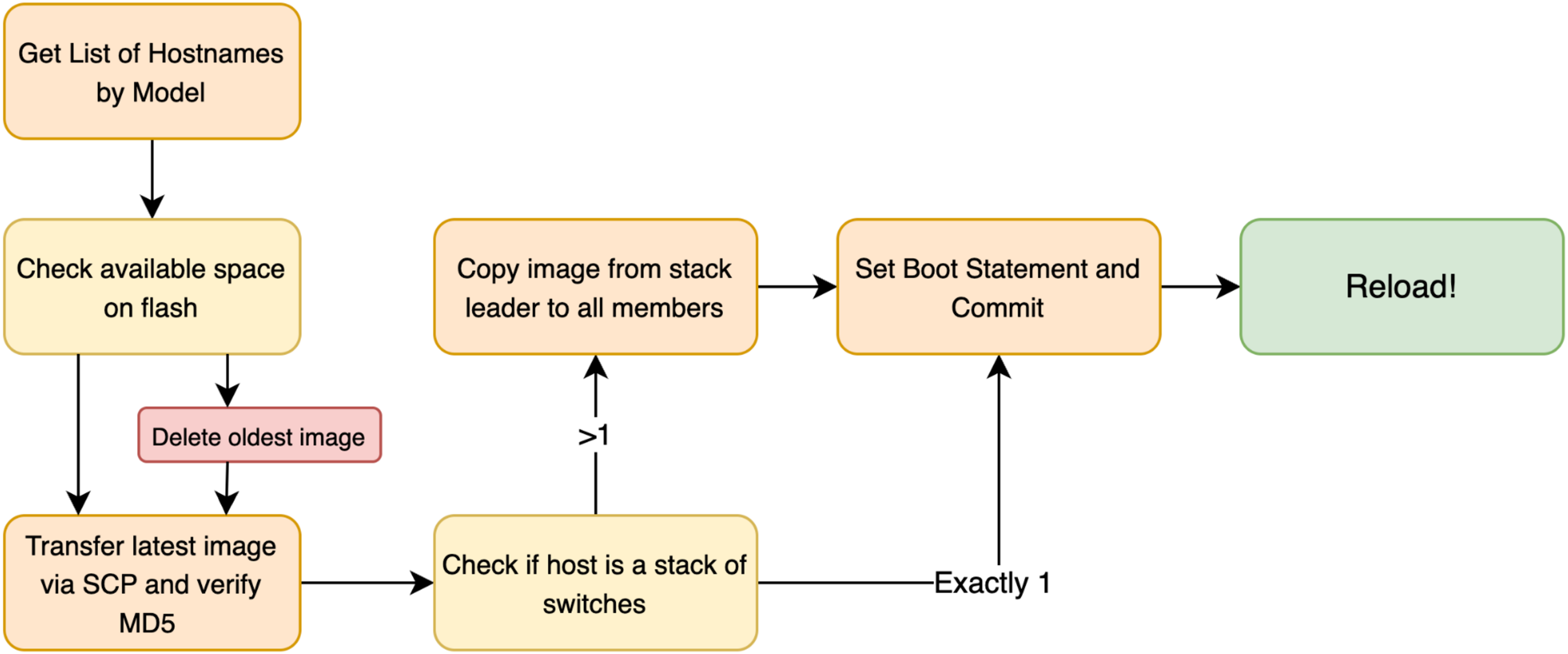
“Network automation refers to the automation of a singular task or small number of closely related tasks.”

“Orchestration refers to the transformation of automated tasks into a larger workflow.”



What is Network Automation?

Example scenario: Upgrade of 800 Cisco Catalyst 2960Xs



Get List of Hostnames by Model

Check available space on flash

Delete oldest image

Transfer latest image via SCP and verify MD5

Check if host is a stack of switches

>1

Exactly 1

Copy image from stack leader to all members

Set Boot Statement and Commit

Reload!

Challenges in Automation

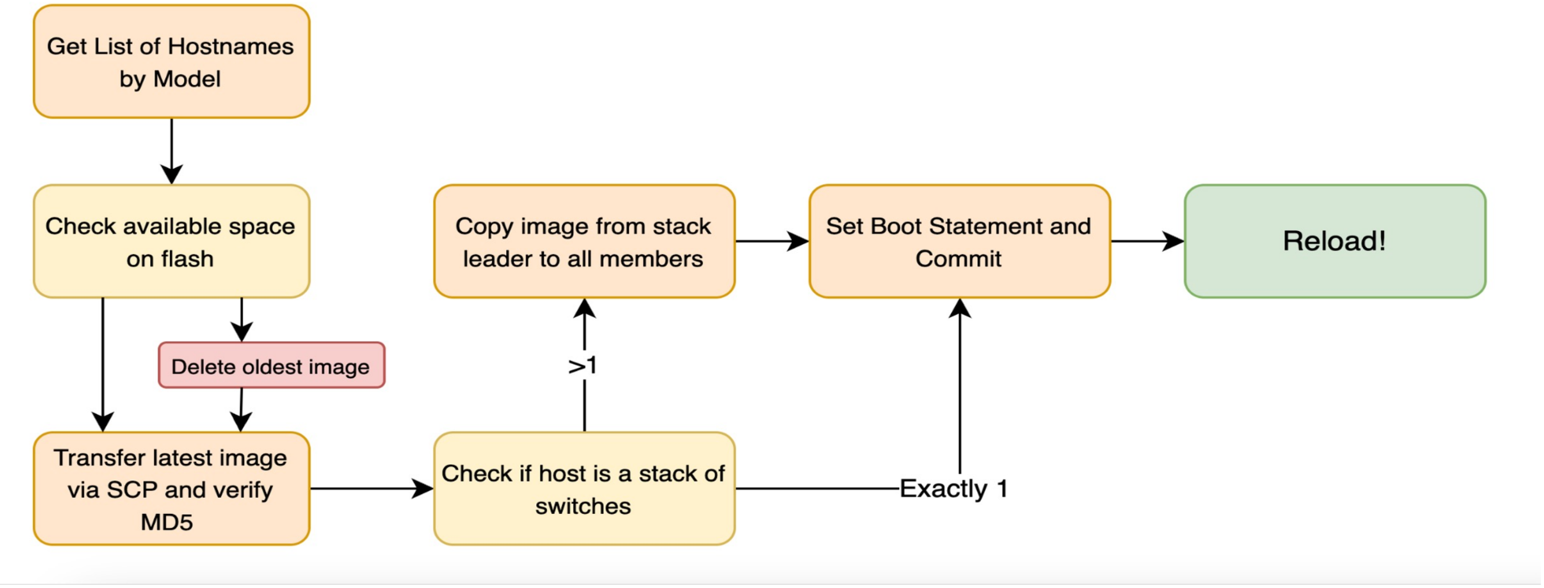
Networks are unique. There is no One Size Fits All.

- Different device models and vendors have different configuration syntax.

- Methods available to interface with devices
  - Not all devices support NETCONF or RESTCONF.
  - Your vendor’s Complete Orchestration Solution™ is not guaranteed to support all current devices in your network.

Challenges in Automation

- Device state is not guaranteed to be consistent across platforms. (ex. Stack sizes, available space on disk)

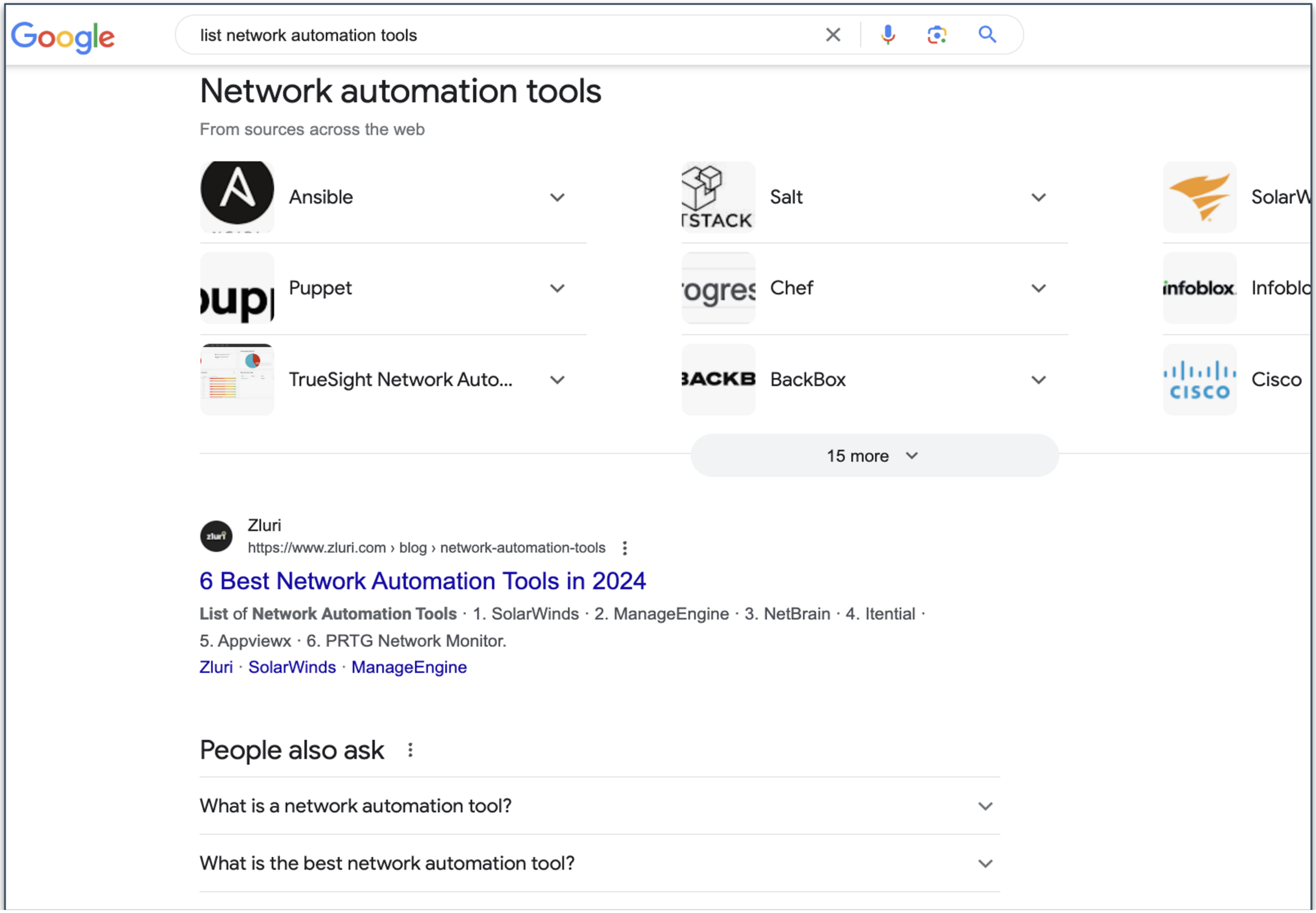

Get List of Hostnames by Model
Check available space on flash
Delete oldest image
Transfer latest image via SCP and verify MD5
Check if host is a stack of switches
>1
Exactly 1
Copy image from stack leader to all members
Set Boot Statement and Commit
Reload!

How Do I Get Started?

- Pick one task and automate it.
- Do not try to automate everything at once, or handle every use-case.
- Be ready to throw away work.

“All good ideas start out as bad ideas. That’s why it takes so long.”
- Steven Spielberg

How Do I Get Started?



list network automation tools

Network automation tools

From sources across the web

Ansible	Salt	SolarW
Puppet	Chef	Infoblo
TrueSight Network Auto...	BackBox	Cisco

15 more

Zluri
https://www.zluri.com › blog › network-automation-tools

6 Best Network Automation Tools in 2024

List of **Network Automation Tools** · 1. SolarWinds · 2. ManageEngine · 3. NetBrain · 4. Itential · 5. Appviewx · 6. PRTG Network Monitor.

Zluri · SolarWinds · ManageEngine

People also ask

What is a network automation tool?

What is the best network automation tool?

How Do I Get Started?

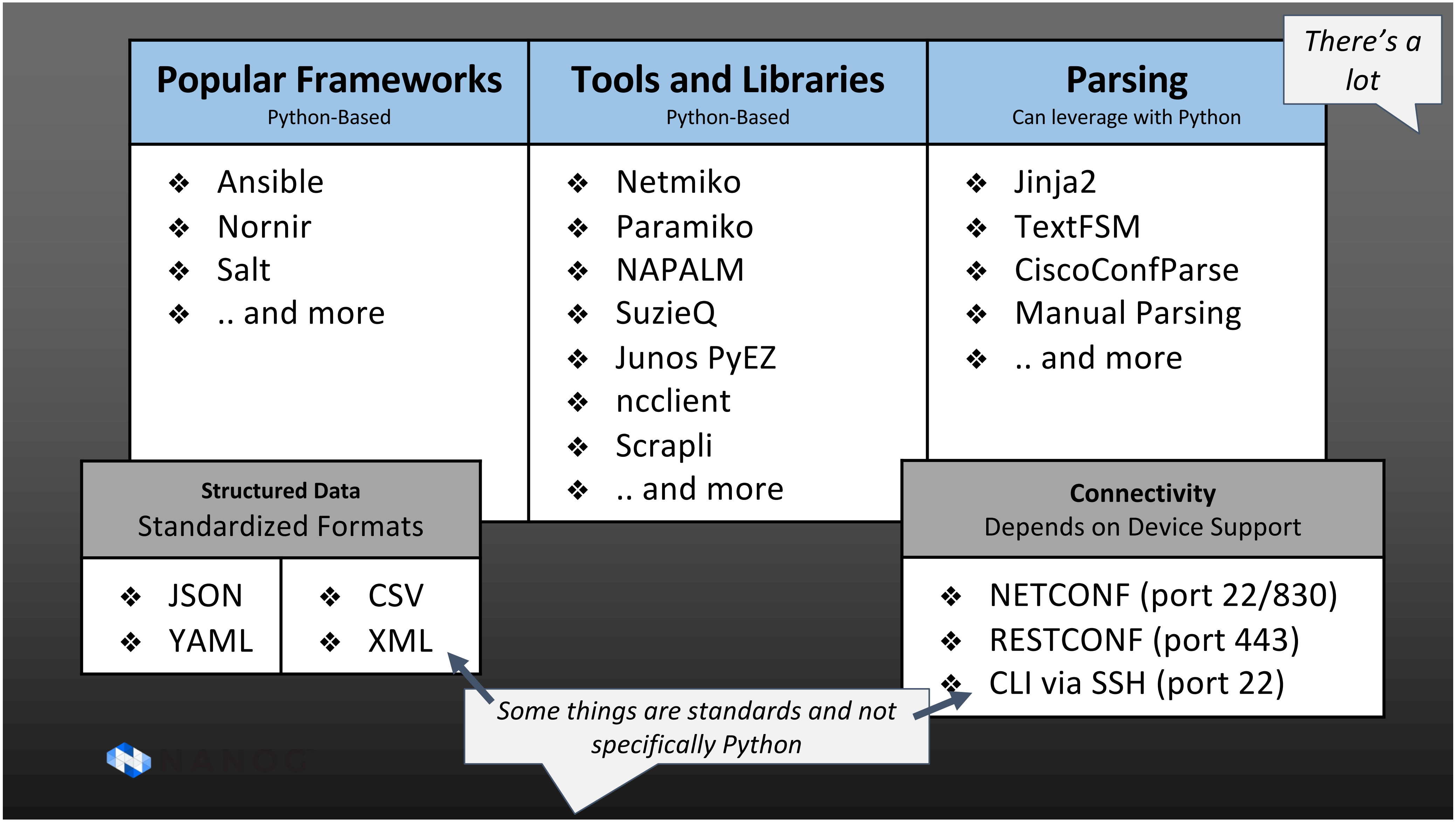
If you’ve done enough Google searching about “network automation”, you’ve seen every buzzword and tool name under the sun.

Open-source vs. closed-source, paid vs. free, libraries, tools, programming languages...

We will focus only on Python, Python-based tools, connectivity methods, and structured data often used by Python.

Common Tools

It seems like there's a bajillion?



There's a lot

Popular Frameworks
Python-Based

- Ansible
- Nornir
- Salt
- .. and more

Tools and Libraries
Python-Based

- Netmiko
- Paramiko
- NAPALM
- SuzieQ
- Junos PyEZ
- ncclient
- Scrapli
- .. and more

Parsing
Can leverage with Python

- Jinja2
- TextFSM
- CiscoConfParse
- Manual Parsing
- .. and more

Structured Data
Standardized Formats

- JSON
- YAML
- CSV
- XML

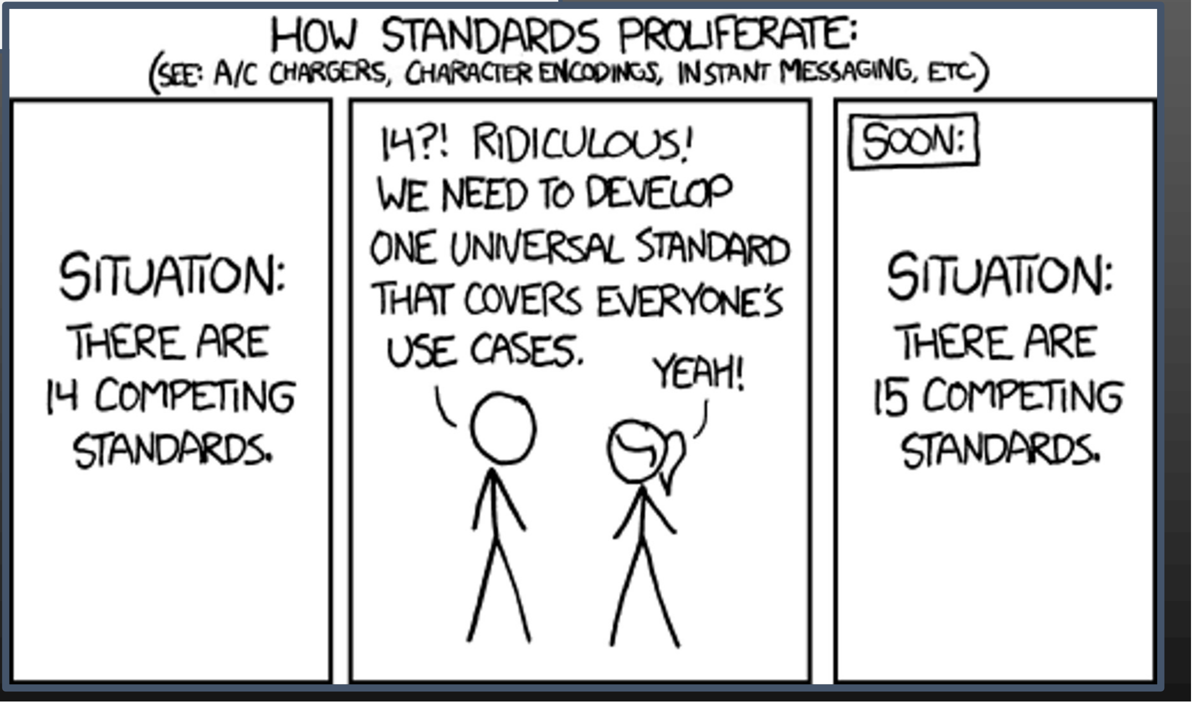
Connectivity
Depends on Device Support

- NETCONF (port 22/830)
- RESTCONF (port 443)
- CLI via SSH (port 22)

Some things are standards and not specifically Python

Stand on the shoulders of giants

There are many, many tools out there already.
We can choose the ones appropriate for our environment, and employ them.
Avoid home-grown solutions where possible.


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

NANOG

Popular Frameworks

Python-Based

- ❖ Ansible
- ❖ Nornir
- ❖ Salt
- ❖ .. and more

Tools and Libraries

Python-Based

- ❖ Netmiko
- ❖ Paramiko
- ❖ NAPALM
- ❖ SuzieQ
- ❖ Junos PyEZ
- ❖ ncclient
- ❖ Scrapli
- ❖ .. and more

Parsing

Can leverage with Python

- ❖ Jinja2
- ❖ TextFSM
- ❖ CiscoConfParse
- ❖ Manual Parsing
- ❖ .. and more

Structured Data

Standardized Formats

- ❖ JSON
- ❖ YAML
- ❖ CSV
- ❖ XML

Connectivity

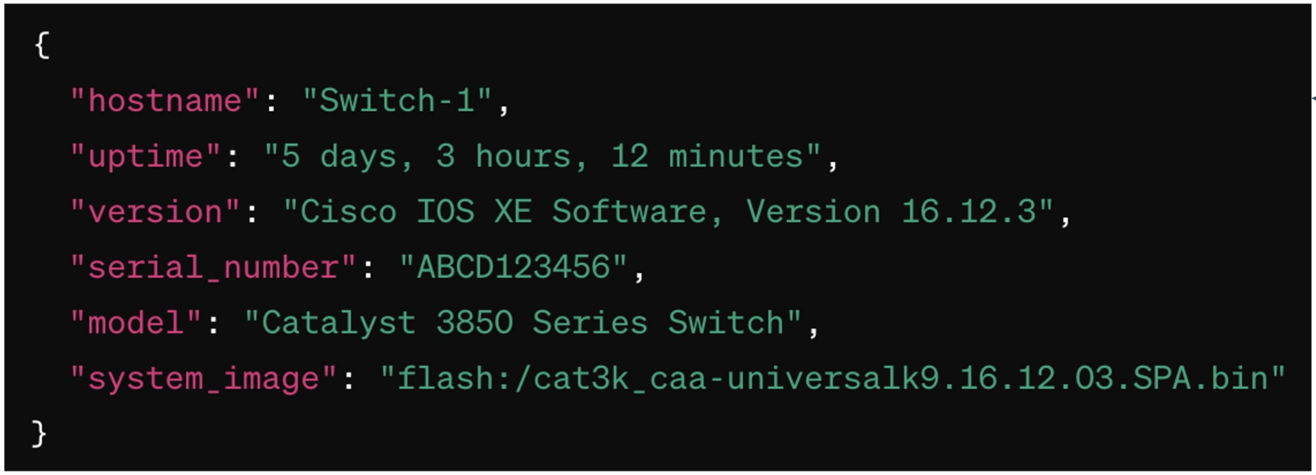
Depends on Device Support

- ❖ NETCONF (port 22/830)
- ❖ RESTCONF (port 443)
- ❖ CLI via SSH (port 22)

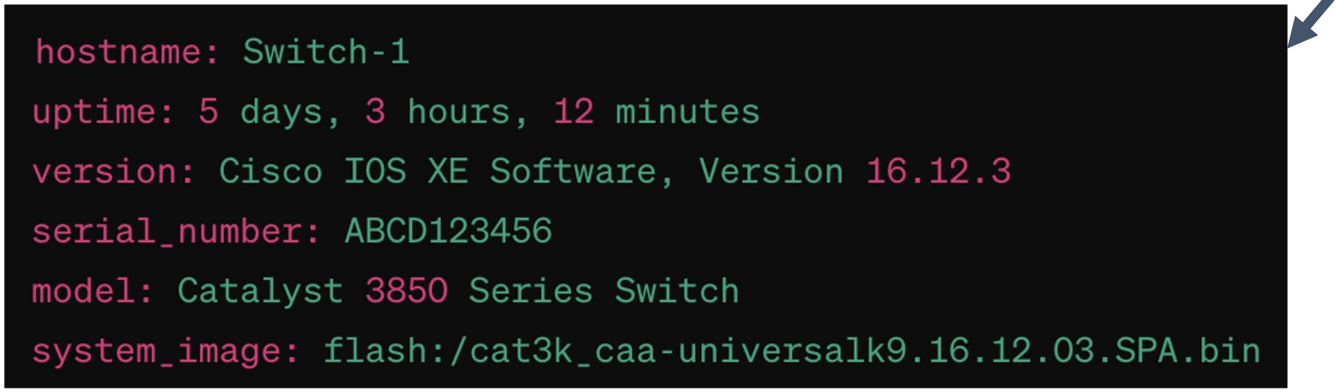
It’s just standards.

Structured Data
Standardized Formats

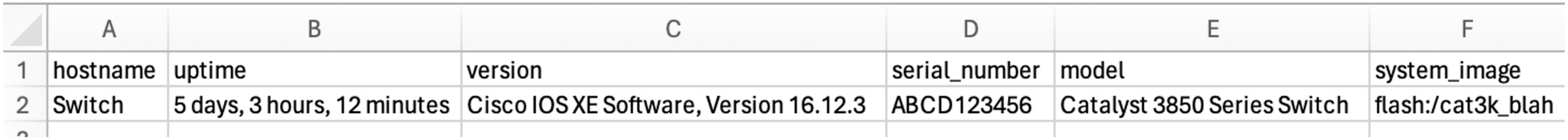
- ❖ JSON
- ❖ YAML
- ❖ CSV
- ❖ XML



```
{
  "hostname": "Switch-1",
  "uptime": "5 days, 3 hours, 12 minutes",
  "version": "Cisco IOS XE Software, Version 16.12.3",
  "serial_number": "ABCD123456",
  "model": "Catalyst 3850 Series Switch",
  "system_image": "flash:/cat3k_caa-universalk9.16.12.03.SPA.bin"
}
```



```
hostname: Switch-1
uptime: 5 days, 3 hours, 12 minutes
version: Cisco IOS XE Software, Version 16.12.3
serial_number: ABCD123456
model: Catalyst 3850 Series Switch
system_image: flash:/cat3k_caa-universalk9.16.12.03.SPA.bin
```



	A	B	C	D	E	F
1	hostname	uptime	version	serial_number	model	system_image
2	Switch	5 days, 3 hours, 12 minutes	Cisco IOS XE Software, Version 16.12.3	ABCD123456	Catalyst 3850 Series Switch	flash:/cat3k_blah

Popular Frameworks

Python-Based

- Ansible
- Nornir
- Salt
- .. and more

Tools and Libraries

Python-Based

- Netmiko
- Paramiko
- NAPALM
- SuzieQ
- Junos PyEZ
- ncclient
- Scrapli
- .. and more

Parsing

Can leverage with Python

- Jinja2
- TextFSM
- CiscoConfParse
- Manual Parsing
- .. and more

Structured Data

Standardized Formats

- JSON
- YAML
- CSV
- XML

Connectivity

Depends on Device Support

- NETCONF (port 22/830)
- RESTCONF (port 443)
- CLI via SSH (port 22)

Ways to Talk

- “**CLI Scraping**”, **NETCONF**, and **RESTCONF** are three common ways to get output of a command from a network device.

- **CLI**: Your code logs into a device as you would over SSH and sends your command.

- **NETCONF**: Your code sends a text payload, containing a command to a device via SSH over port 22 or 830.

- **RESTCONF**: Your code sends a web request to your device, such as a GET or POST request, that contains your command.

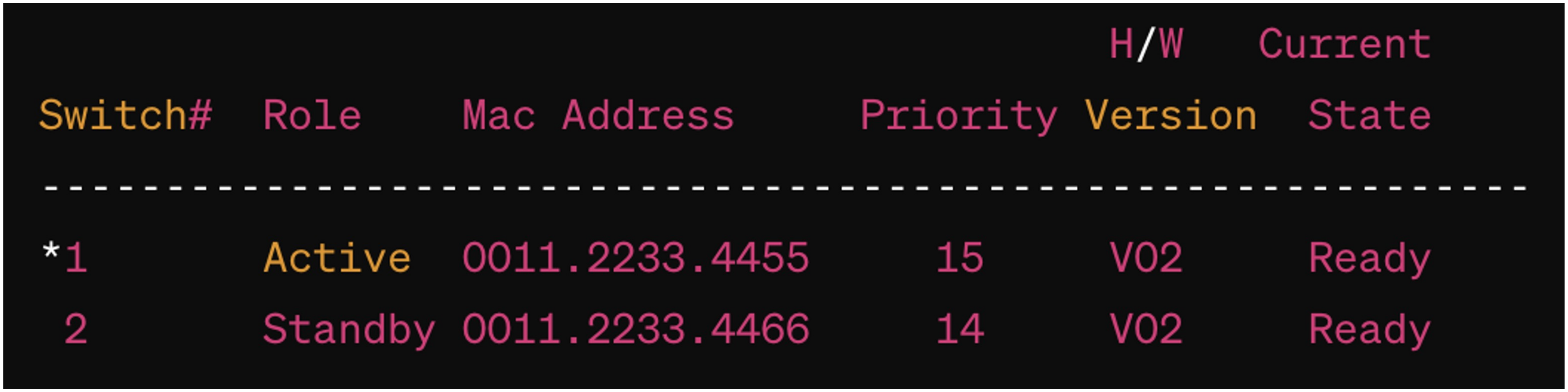
Connectivity
Depends on Device Support

❖ NETCONF (port 22/830)
❖ RESTCONF (port 443)
❖ CLI via SSH (port 22)

CLI Scraping (port 22)

```
Switch(config)# transport input ssh
```

```
output = connection.send_command("show switch")
```



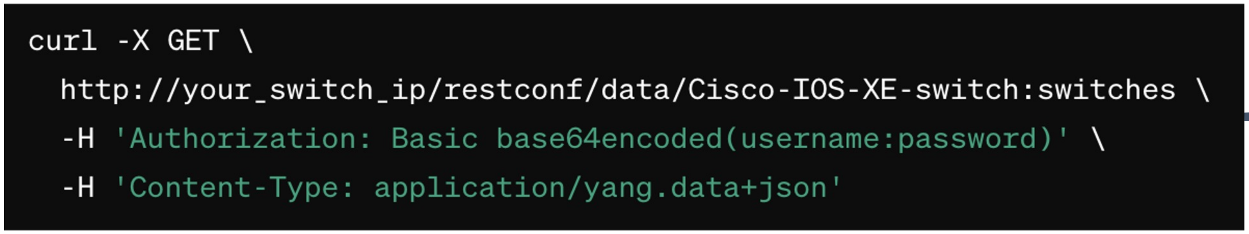
```
                                           H/W   Current
Switch#  Role    Mac Address     Priority Version  State
-----------------------------------------------------------
*1       Active  0011.2233.4455     15      V02      Ready
 2       Standby 0011.2233.4466     14      V02      Ready
```

Most familiar-looking

RESTCONF (port 443)

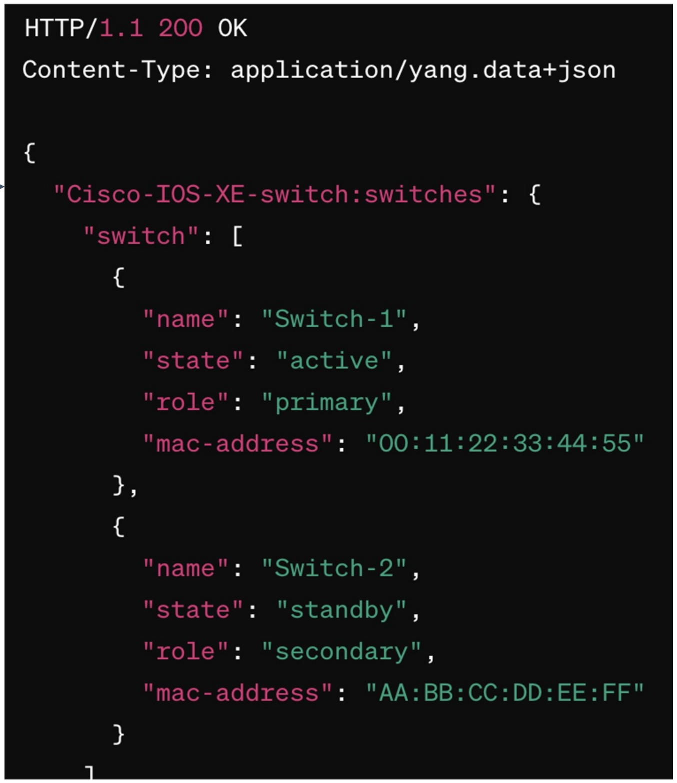
`Switch(config)# rest api` (and **`ip http secure-server,`** `etc`)

```
output =
connection.get('https://ipaddr/restconf/data/Cisco-IOS-XE-switch:System/switches')
```



```
curl -X GET \
  http://your_switch_ip/restconf/data/Cisco-IOS-XE-switch:switches \
  -H 'Authorization: Basic base64encoded(username:password)' \
  -H 'Content-Type: application/yang.data+json'
```

Request



```
HTTP/1.1 200 OK
Content-Type: application/yang.data+json

{
  "Cisco-IOS-XE-switch:switches": {
    "switch": [
      {
        "name": "Switch-1",
        "state": "active",
        "role": "primary",
        "mac-address": "00:11:22:33:44:55"
      },
      {
        "name": "Switch-2",
        "state": "standby",
        "role": "secondary",
        "mac-address": "AA:BB:CC:DD:EE:FF"
      }
    ]
```

Response

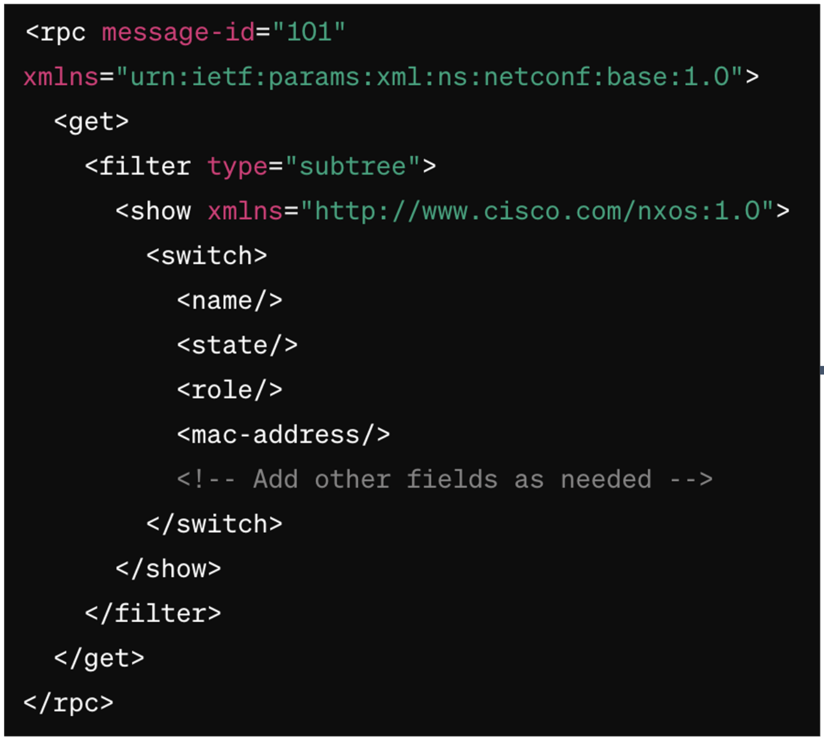
Maybe still familiar

NETCONF (port 22 or 830)

`Switch(config)# netconf-ssh` (or **`netconf-yang`**)

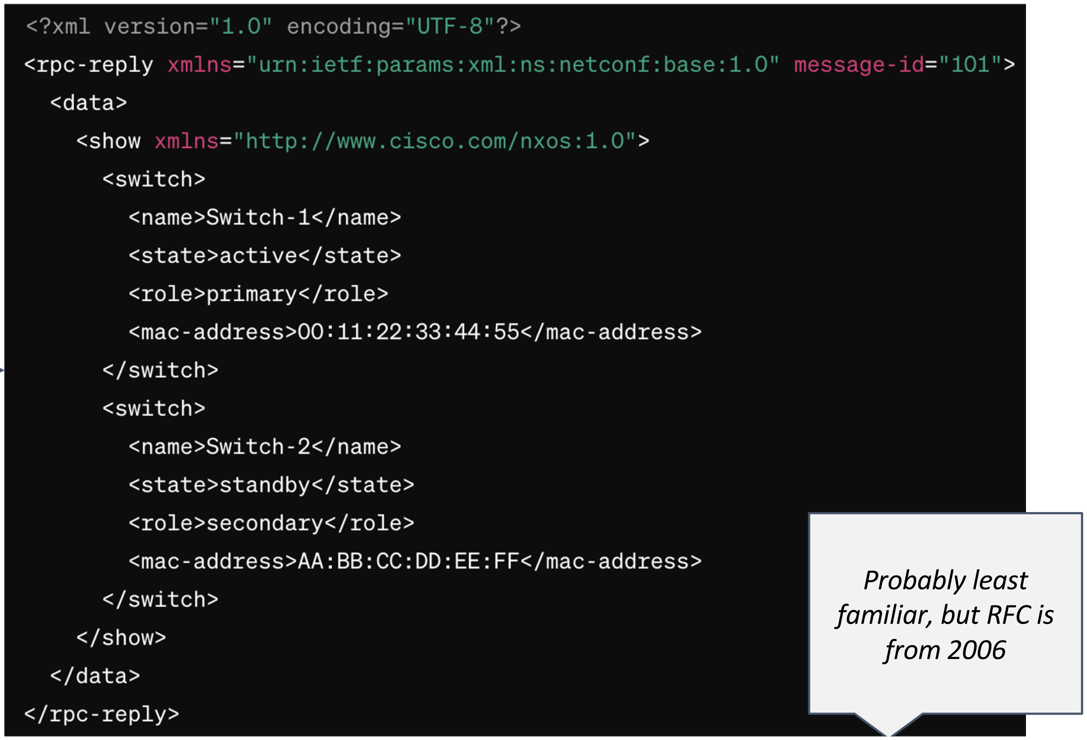
```
output = connection.get(filter=('subtree', 'something like below'))
```

Request



```
<rpc message-id="101"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <filter type="subtree">
      <show xmlns="http://www.cisco.com/nxos:1.0">
        <switch>
          <name/>
          <state/>
          <role/>
          <mac-address/>
          <!-- Add other fields as needed -->
        </switch>
      </show>
    </filter>
  </get>
</rpc>
```

Response



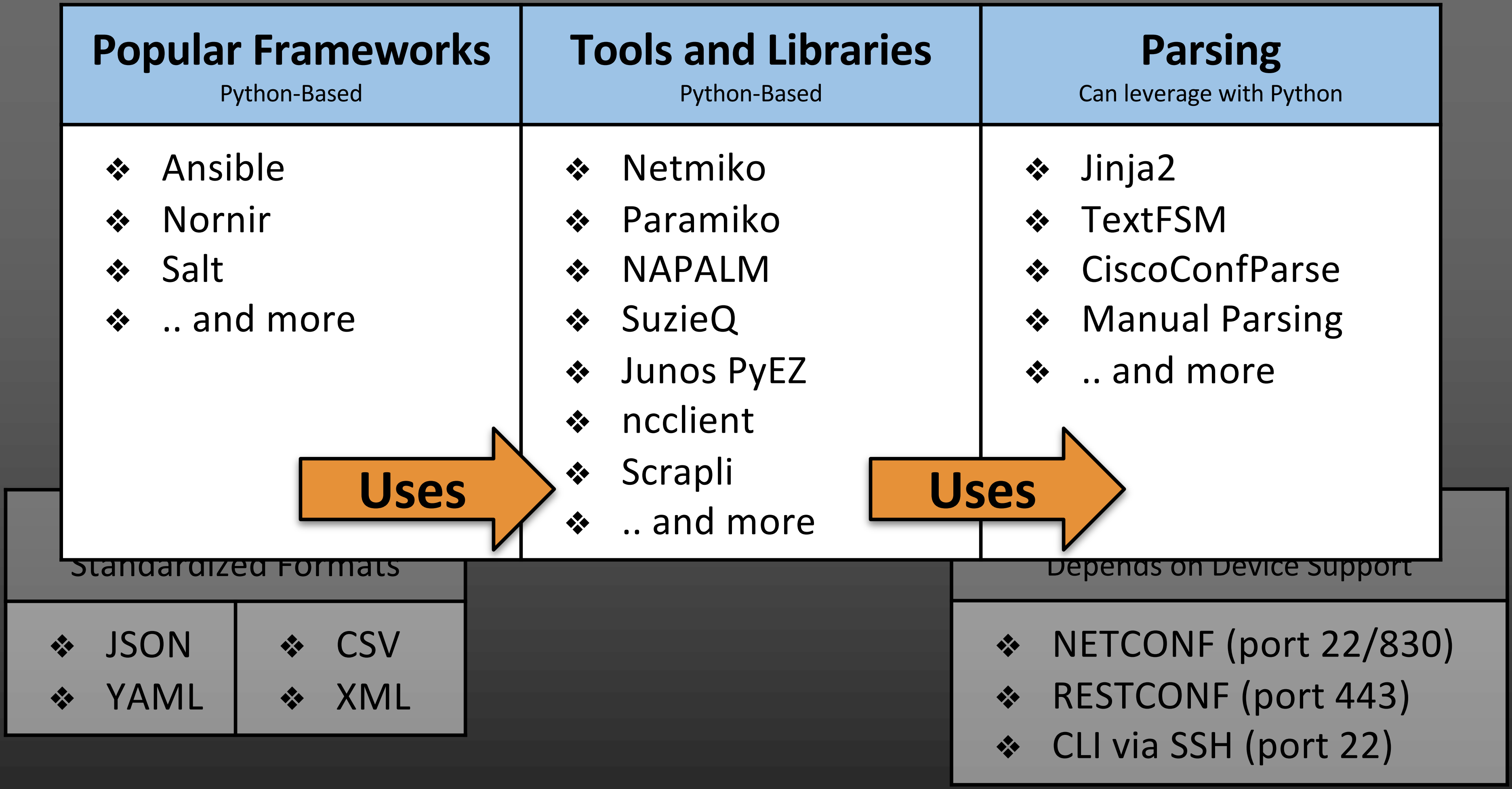
```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <data>
    <show xmlns="http://www.cisco.com/nxos:1.0">
      <switch>
        <name>Switch-1</name>
        <state>active</state>
        <role>primary</role>
        <mac-address>00:11:22:33:44:55</mac-address>
      </switch>
      <switch>
        <name>Switch-2</name>
        <state>standby</state>
        <role>secondary</role>
        <mac-address>AA:BB:CC:DD:EE:FF</mac-address>
      </switch>
    </show>
  </data>
</rpc-reply>
```

Probably least familiar, but RFC is from 2006

Common Tools: Part 2

Standards are neat, but where’s the actual Python stuff?

NANOG™



Popular Frameworks

Python-Based

- Ansible
- Nornir
- Salt
- .. and more

Uses

Tools and Libraries

Python-Based

- Netmiko
- Paramiko
- NAPALM
- SuzieQ
- Junos PyEZ
- ncclient
- Scrapli
- .. and more

Uses

Parsing

Can leverage with Python

- Jinja2
- TextFSM
- CiscoConfParse
- Manual Parsing
- .. and more

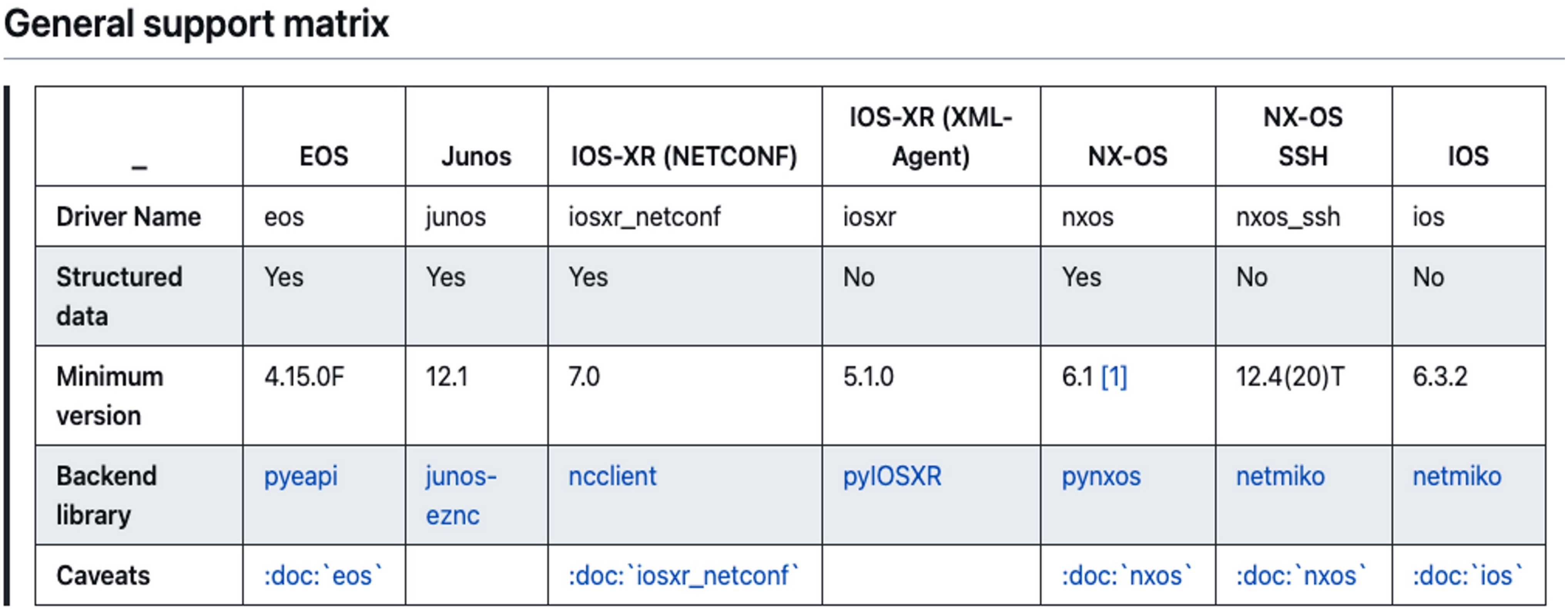
Standardized Formats

- JSON
- YAML
- CSV
- XML

Depends on Device Support

- NETCONF (port 22/830)
- RESTCONF (port 443)
- CLI via SSH (port 22)

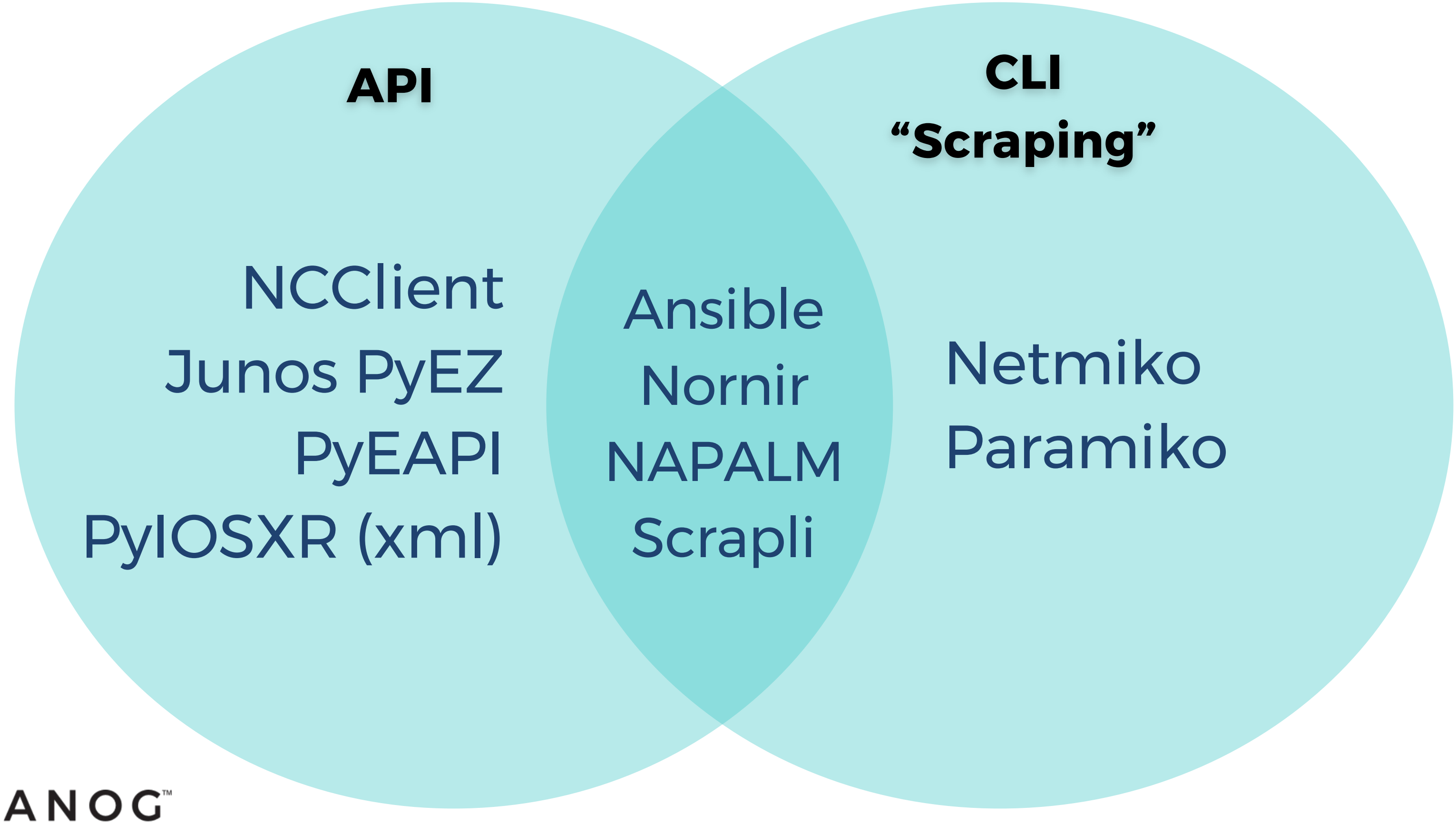
Example from NAPALM Drivers



General support matrix

_	EOS	Junos	IOS-XR (NETCONF)	IOS-XR (XML-Agent)	NX-OS	NX-OS SSH	IOS
Driver Name	eos	junos	iosxr_netconf	iosxr	nxos	nxos_ssh	ios
Structured data	Yes	Yes	Yes	No	Yes	No	No
Minimum version	4.15.0F	12.1	7.0	5.1.0	6.1 [1]	12.4(20)T	6.3.2
Backend library	pyeapi	junos-eznc	ncclient	pyIOSXR	pynxos	netmiko	netmiko
Caveats	:doc:`eos`		:doc:`iosxr_netconf`		:doc:`nxos`	:doc:`nxos`	:doc:`ios`

Python Libraries can use CLI OR an API like NETCONF or RESTCONF



API

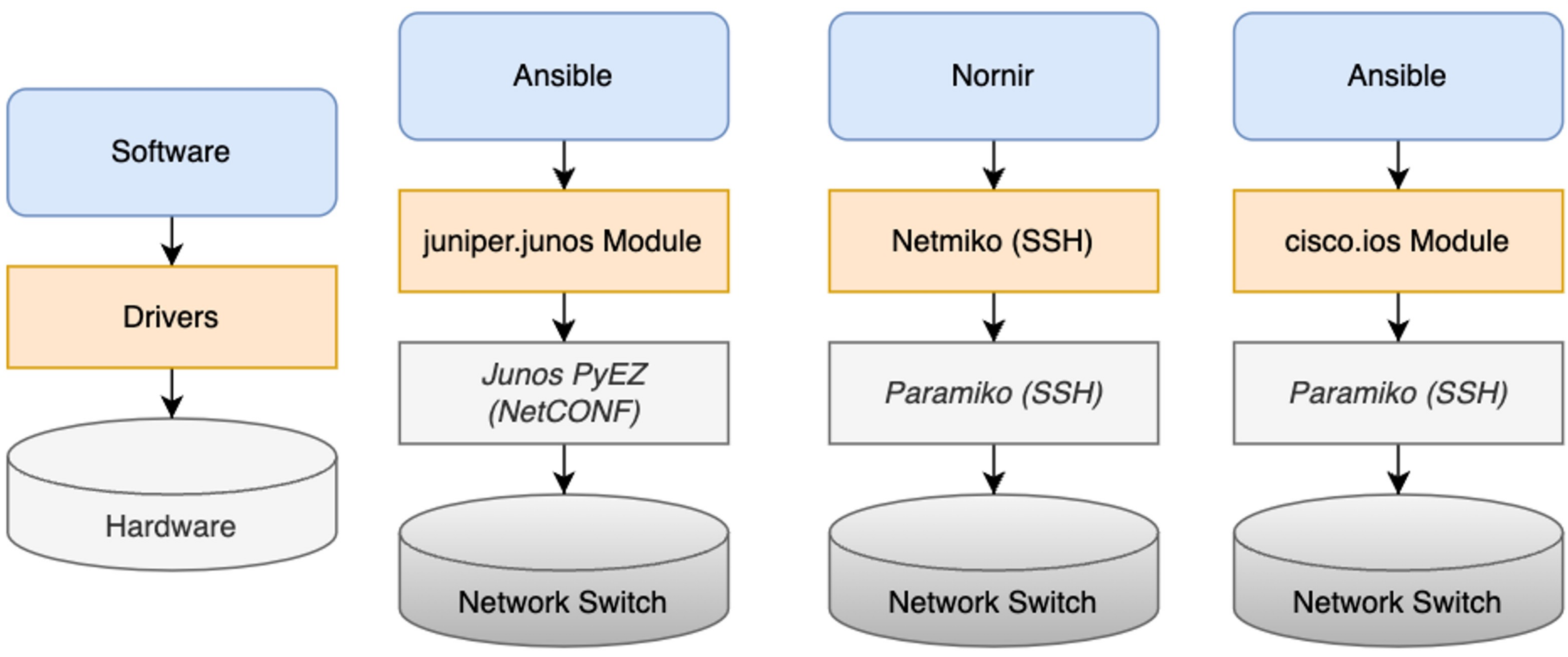
NCClient
Junos PyEZ
PyEAPI
PyIOSXR (xml)

Ansible
Nornir
NAPALM
Scrapli

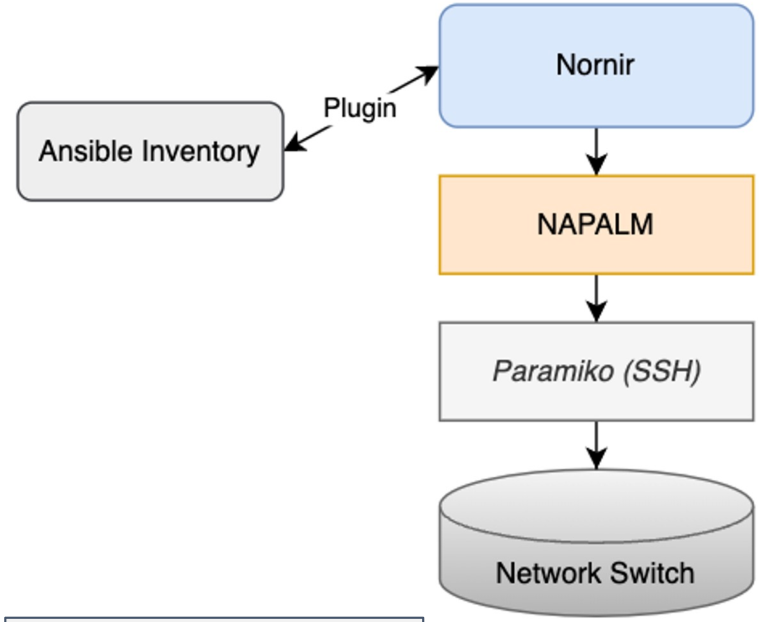
CLI
“Scraping”

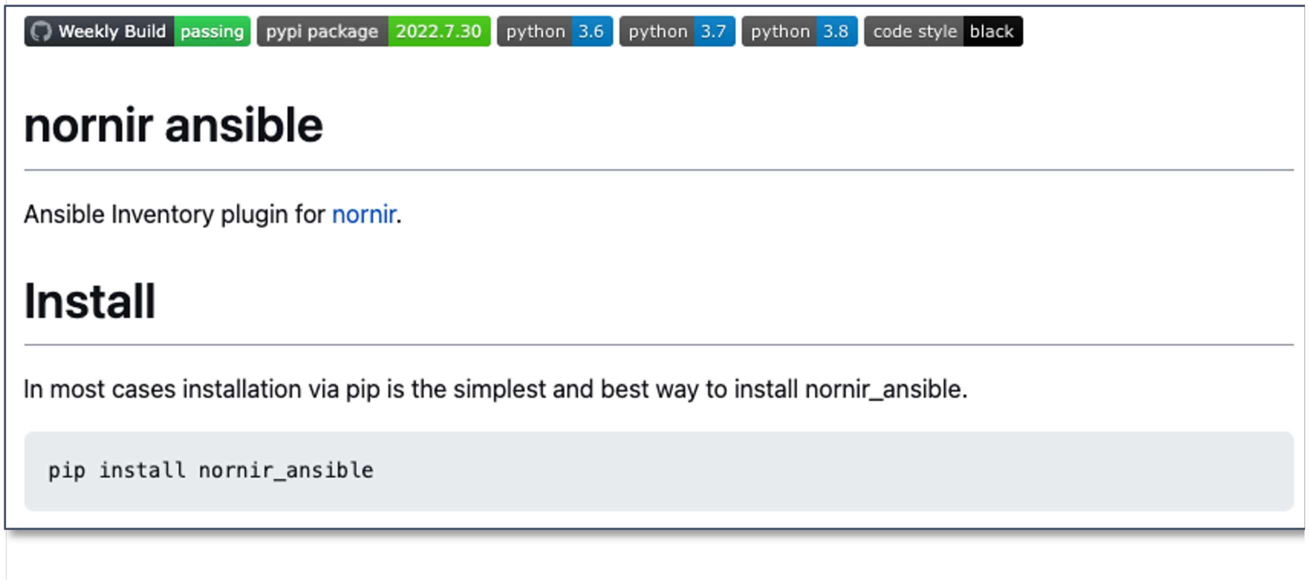
Netmiko
Paramiko

An Analogy


Software
Drivers
Hardware
Ansible
juniper.junos Module
Junos PyEZ (NetCONF)
Network Switch
Nornir
Netmiko (SSH)
Paramiko (SSH)
Network Switch
Ansible
cisco.ios Module
Paramiko (SSH)
Network Switch

Frameworks are highly pluggable





nornir ansible

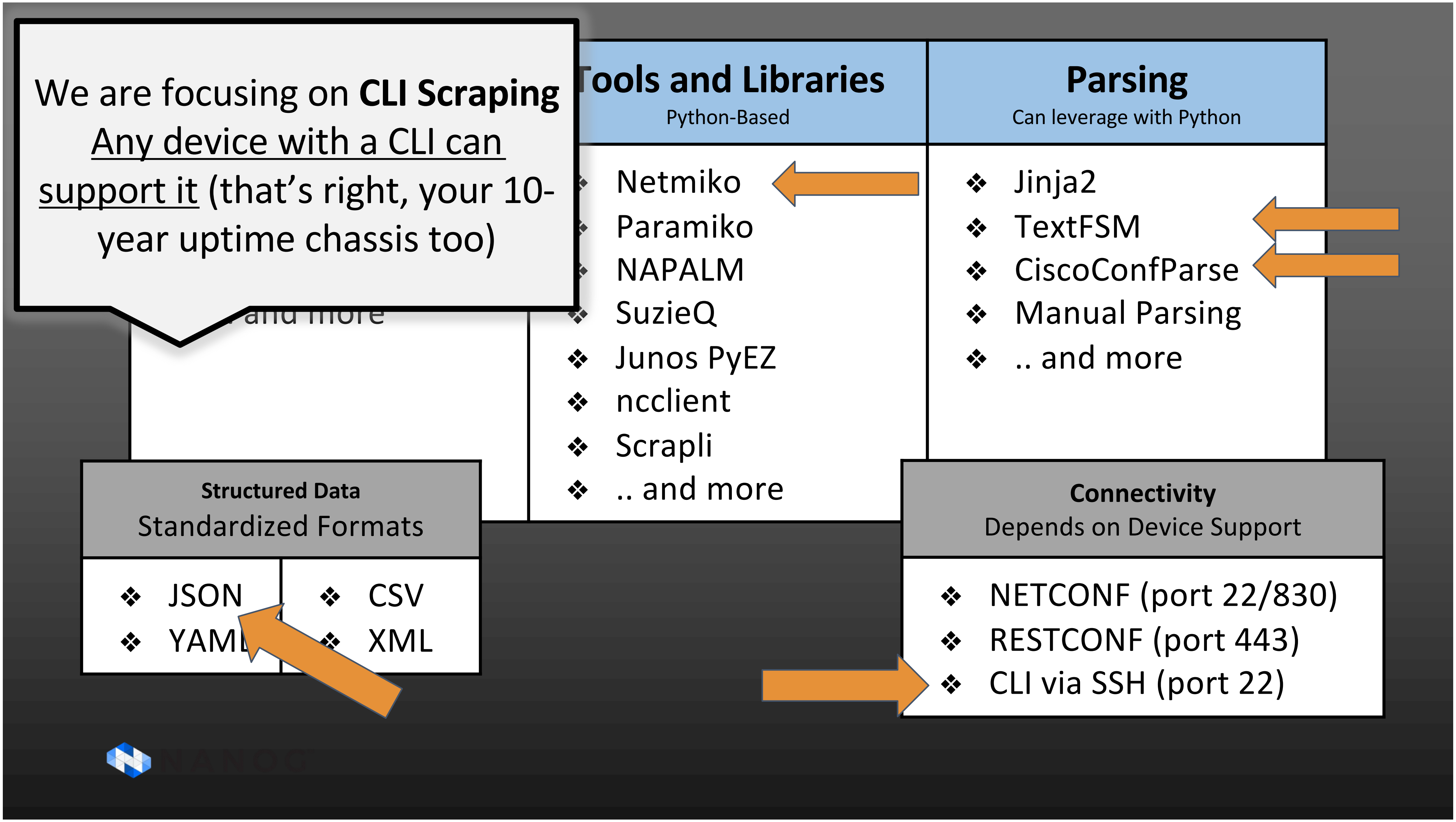
Ansible Inventory plugin for nornir.

Install

In most cases installation via pip is the simplest and best way to install nornir_ansible.

```
pip install nornir_ansible
```

NetBox and Nautobot also have Nornir plugins for inventory



We are focusing on **CLI Scraping** Any device with a CLI can support it (that’s right, your 10-year uptime chassis too)

.. and more

Tools and Libraries

Python-Based

- Netmiko
- Paramiko
- NAPALM
- SuzieQ
- Junos PyEZ
- ncclient
- Scrapli
- .. and more

Parsing

Can leverage with Python

- Jinja2
- TextFSM
- CiscoConfParse
- Manual Parsing
- .. and more

Structured Data

Standardized Formats

- JSON
- YAML
- CSV
- XML

Connectivity

Depends on Device Support

- NETCONF (port 22/830)
- RESTCONF (port 443)
- CLI via SSH (port 22)

In a brownfield environment, understanding how **CLI scraping** works is an effective way to gain foundational understanding of network automation.

It is also the **lowest common denominator** in programmatic ways to interface with your devices.

We are stripping away some magic.

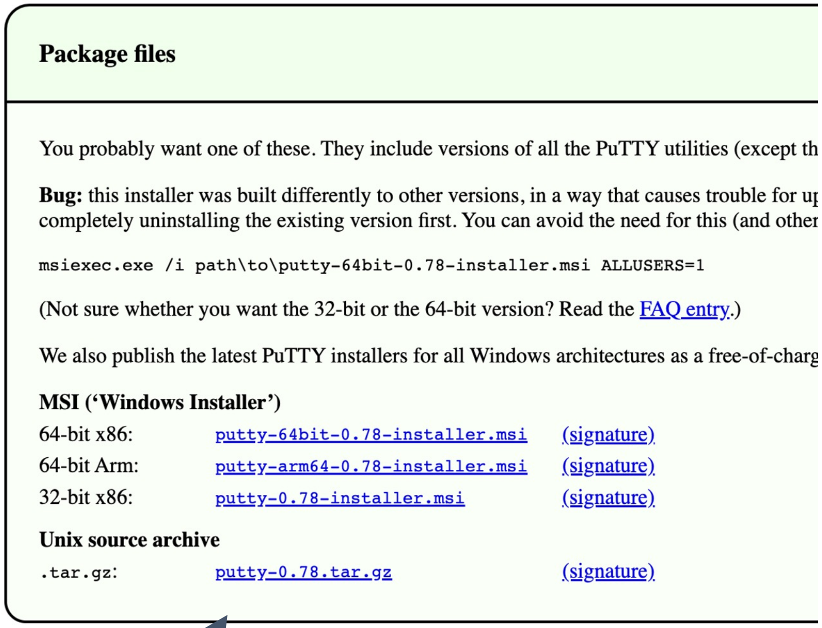
NANOG

Lab Login and Preparation

Logging in to the Lab UNIX Machine

Open your favorite terminal!

- **Mac and Linux**: Use the built-in "Terminal" or your favorite terminal software.
- **Windows**: Use PuTTY or your favorite terminal software.



Package files

You probably want one of these. They include versions of all the PuTTY utilities (except th

Bug: this installer was built differently to other versions, in a way that causes trouble for up completely uninstalling the existing version first. You can avoid the need for this (and other

```
msiexec.exe /i path\to\putty-64bit-0.78-installer.msi ALLUSERS=1
```

(Not sure whether you want the 32-bit or the 64-bit version? Read the FAQ entry.)

We also publish the latest PuTTY installers for all Windows architectures as a free-of-charg

MSI ('Windows Installer')

64-bit x86:	putty-64bit-0.78-installer.msi	(signature)
64-bit Arm:	putty-arm64-0.78-installer.msi	(signature)
32-bit x86:	putty-0.78-installer.msi	(signature)

Unix source archive

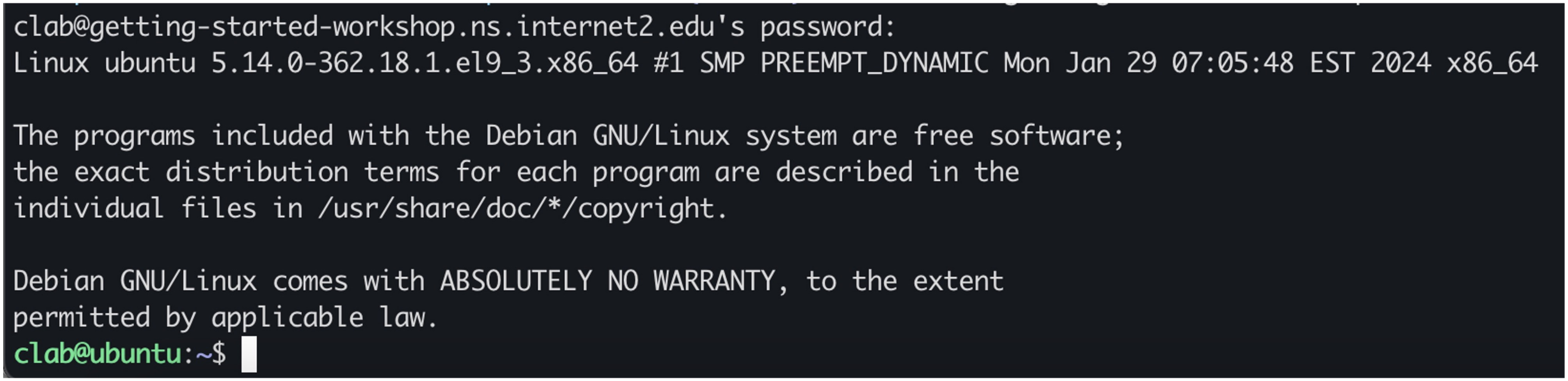
.tar.gz:	putty-0.78.tar.gz	(signature)

- If you use Windows and do not have PuTTY:
  - Go to https://www.putty.org/ and click on the "Download PuTTY" link, **OR**
  - Go directly to: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
  - Download the "64-bit x86" version and install.

Logging in to the Lab UNIX Machine

Refer to the small index card you received for port and credentials.

```
ssh clab@getting-started-workshop.ns.internet2.edu -p 2XX5
```

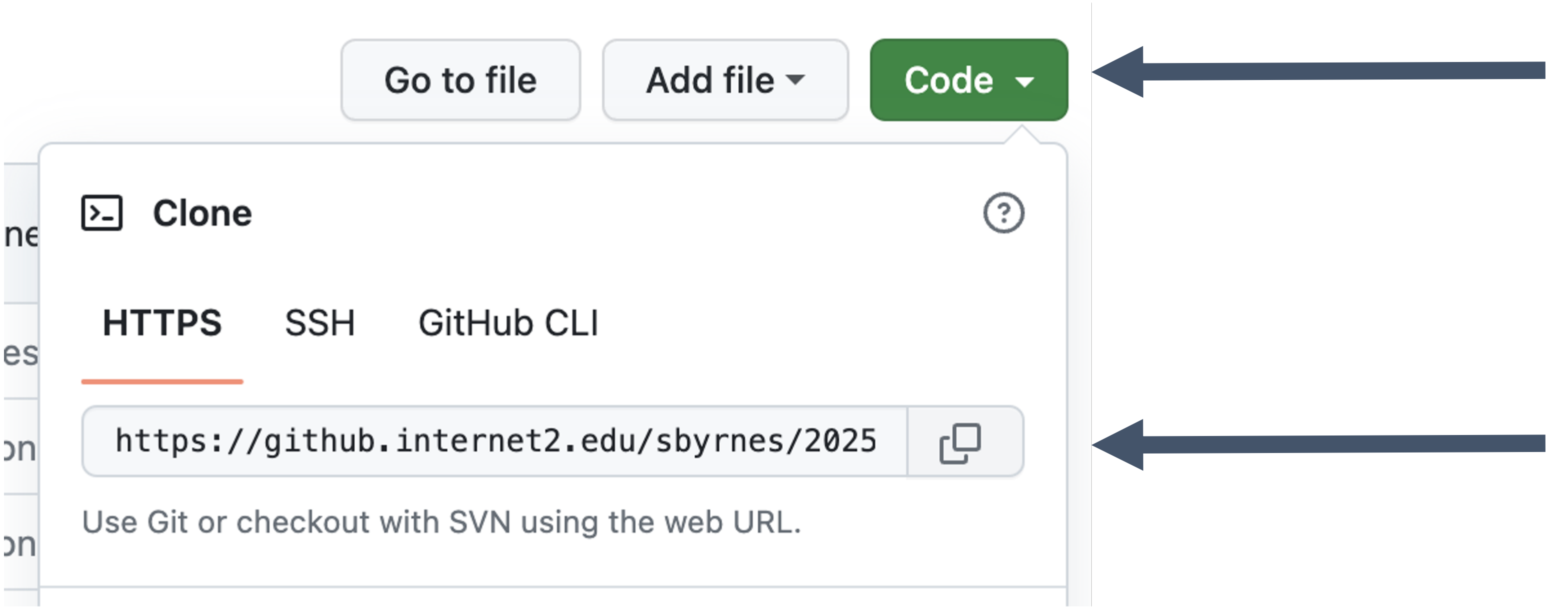


```
clab@getting-started-workshop.ns.internet2.edu's password:
Linux ubuntu 5.14.0-362.18.1.el9_3.x86_64 #1 SMP PREEMPT_DYNAMIC Mon Jan 29 07:05:48 EST 2024 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
clab@ubuntu:~$ 
```

Downloading Lab Assets into your Lab

Go to:
https://github.internet2.edu/sbyrnes/2025-nanog93-tutorial-automation

Click on “Code” and copy the HTTPS link.



https://github.internet2.edu/sbyrnes/2025

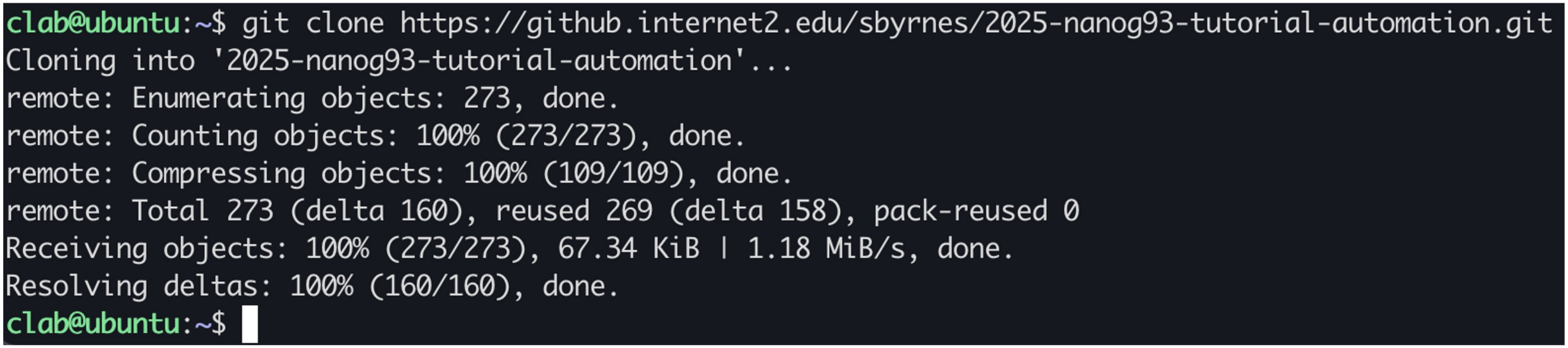
Use Git or checkout with SVN using the web URL.

Downloading Lab Assets into your Lab

In your UNIX machine, run:

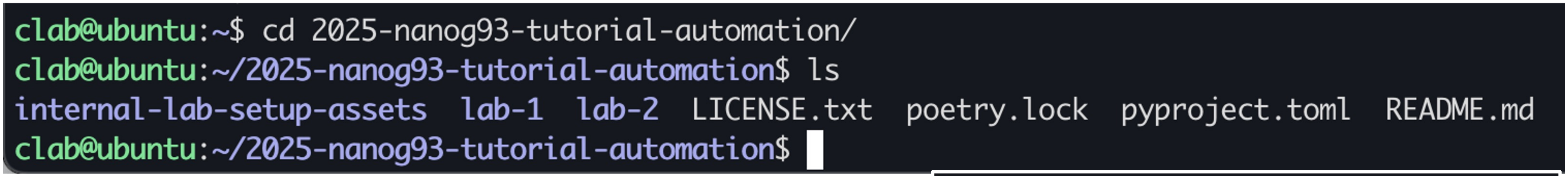
git clone https://github.internet2.edu/sbyrnes/2025-nanog93-tutorial-automation.git

You should see output like below.



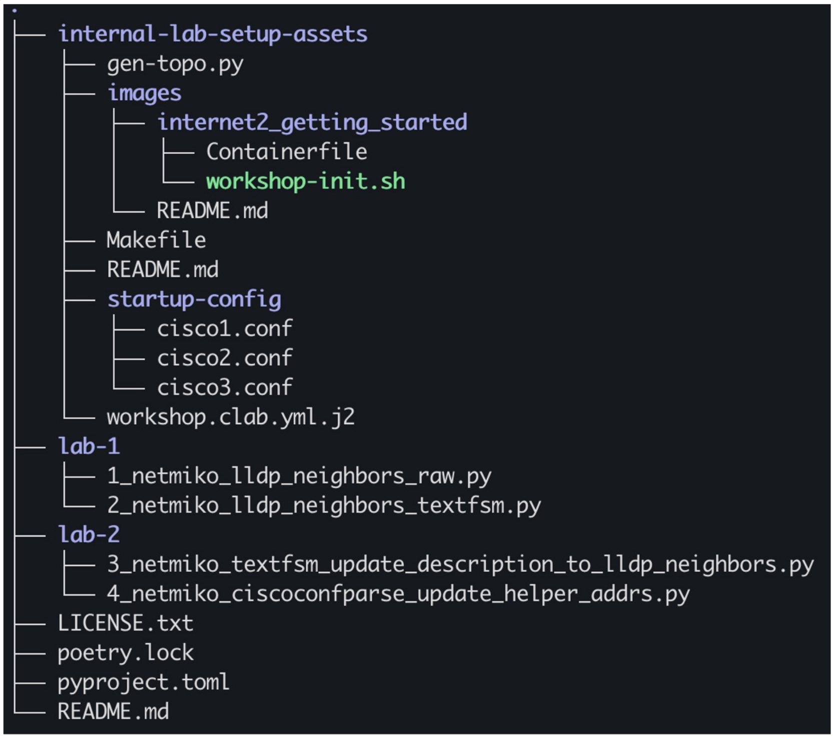
```
clab@ubuntu:~$ git clone https://github.internet2.edu/sbyrnes/2025-nanog93-tutorial-automation.git
Cloning into '2025-nanog93-tutorial-automation'...
remote: Enumerating objects: 273, done.
remote: Counting objects: 100% (273/273), done.
remote: Compressing objects: 100% (109/109), done.
remote: Total 273 (delta 160), reused 269 (delta 158), pack-reused 0
Receiving objects: 100% (273/273), 67.34 KiB | 1.18 MiB/s, done.
Resolving deltas: 100% (160/160), done.
clab@ubuntu:~$ 
```

`cd` into your newly cloned “2025-nanog93-tutorial-automation” folder and see the contents with **`ls`**. It should look like so.



```
clab@ubuntu:~$ cd 2025-nanog93-tutorial-automation/
clab@ubuntu:~/2025-nanog93-tutorial-automation$ ls
internal-lab-setup-assets  lab-1  lab-2  LICENSE.txt  poetry.lock  pyproject.toml  README.md
clab@ubuntu:~/2025-nanog93-tutorial-automation$ 
```

You can also run **`tree`** to get an idea of the lab structure.



```
.
├── internal-lab-setup-assets
│   ├── gen-topo.py
│   ├── images
│   │   ├── internet2_getting_started
│   │   │   ├── Containerfile
│   │   │   └── workshop-init.sh
│   │   └── README.md
│   ├── Makefile
│   ├── README.md
│   ├── startup-config
│   │   ├── cisco1.conf
│   │   ├── cisco2.conf
│   │   └── cisco3.conf
│   └── workshop.clab.yml.j2
├── lab-1
│   ├── 1_netmiko_lldp_neighbors_raw.py
│   └── 2_netmiko_lldp_neighbors_textfsm.py
├── lab-2
│   ├── 3_netmiko_textfsm_update_description_to_lldp_neighbors.py
│   └── 4_netmiko_ciscoconfparse_update_helper_addrs.py
├── LICENSE.txt
├── poetry.lock
├── pyproject.toml
└── README.md
```

(Disregard the `internal-lab-setup-assets` folder…)

Finally, install Poetry

Poetry is a **Python package manager**.

Out of the box, Python does not have the packages we need in order to run our scripts.

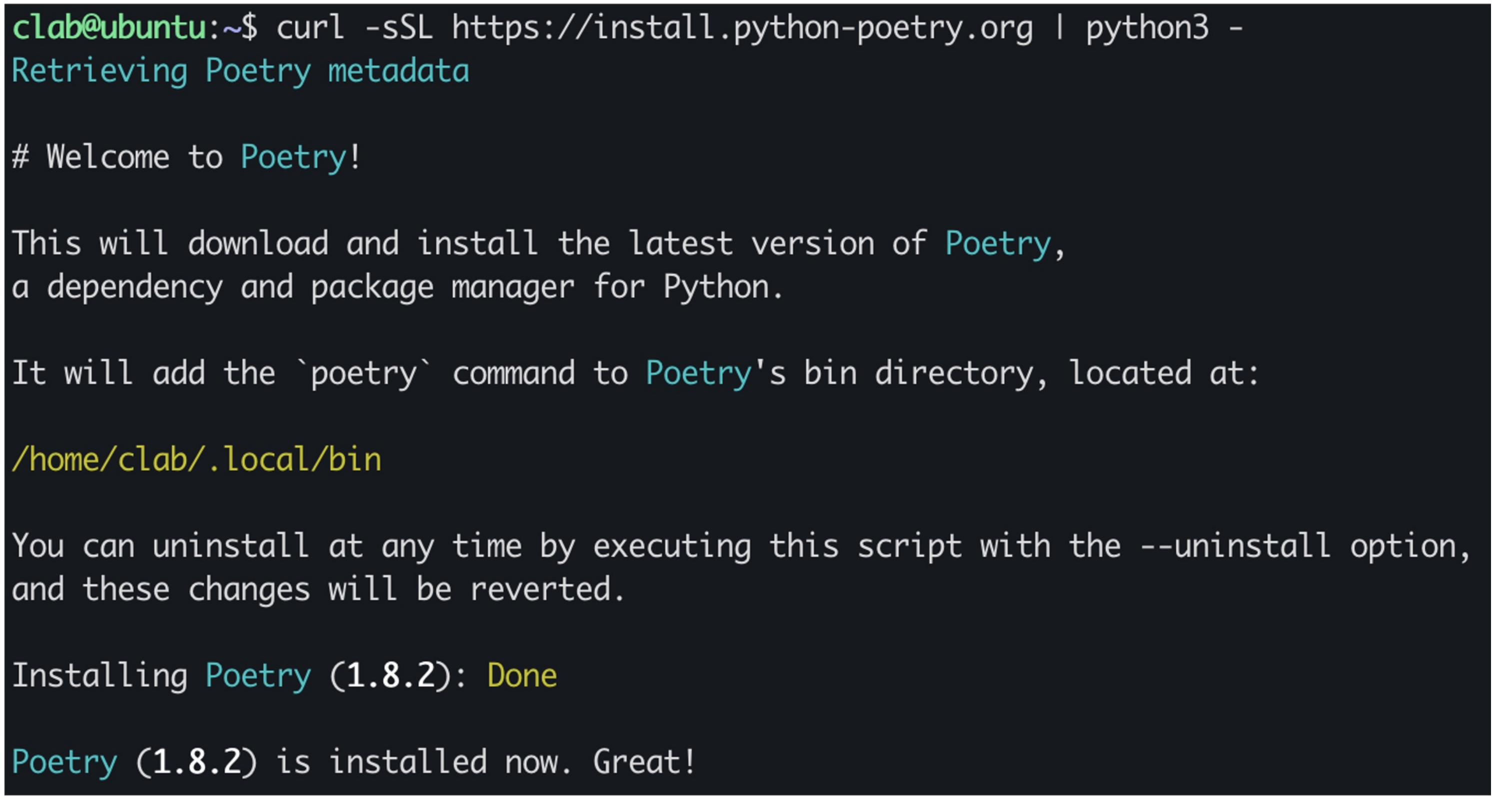
Notably: It does not have the **Netmiko** package, which we use to SSH into our devices in our Python scripts.

Install Poetry with:

`curl -sSL https://install.python-poetry.org | python3 -`

Then run:

`export PATH="/home/clab/.local/bin:$PATH"`



```
clab@ubuntu:~$ curl -sSL https://install.python-poetry.org | python3 -
Retrieving Poetry metadata

# Welcome to Poetry!

This will download and install the latest version of Poetry,
a dependency and package manager for Python.

It will add the `poetry` command to Poetry's bin directory, located at:

/home/clab/.local/bin

You can uninstall at any time by executing this script with the --uninstall option,
and these changes will be reverted.

Installing Poetry (1.8.2): Done

Poetry (1.8.2) is installed now. Great!
```

After this step, you should be able to run:

`poetry --version`

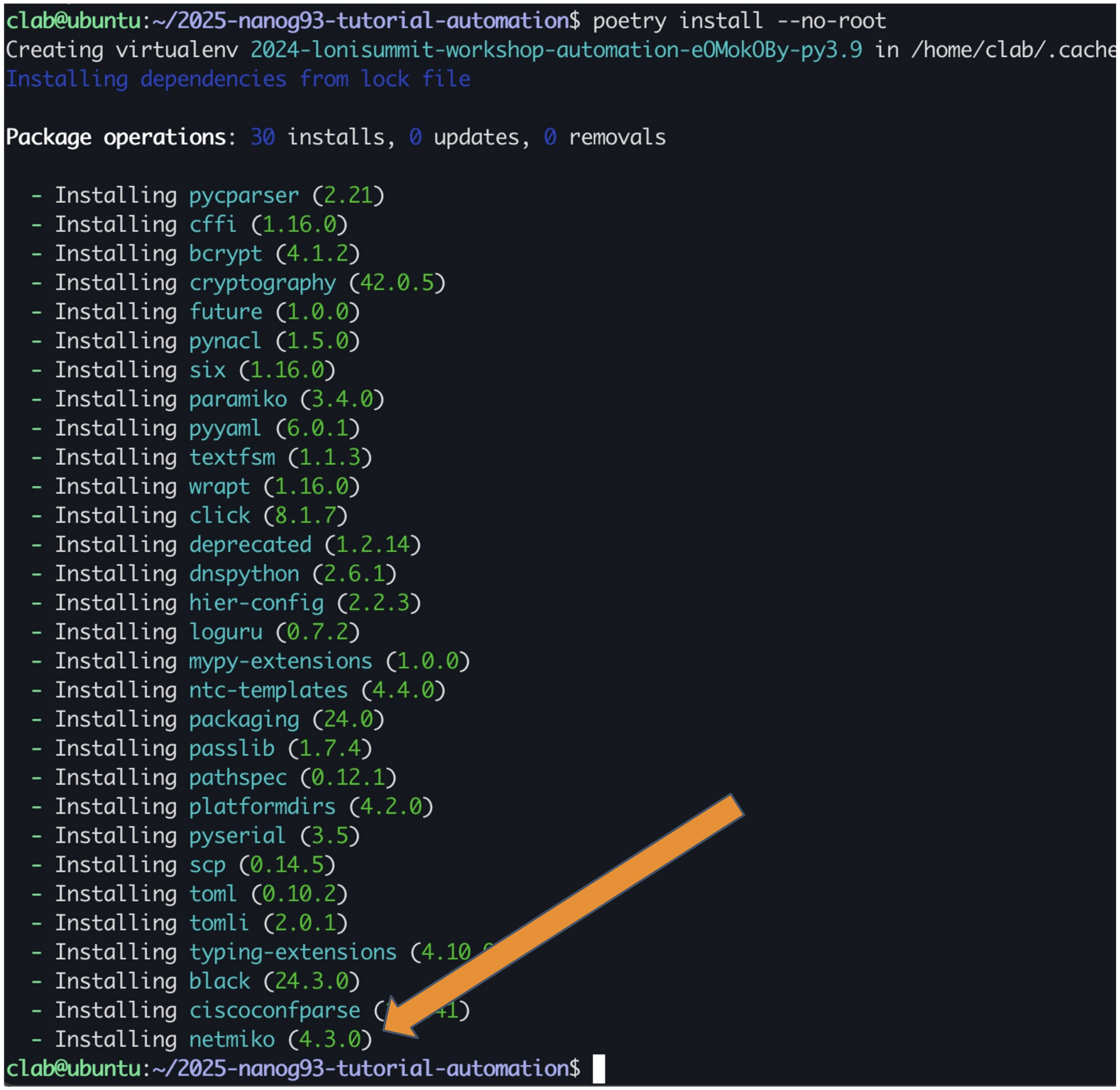
Finally, install Poetry

`**cd**` into the workshop folder, then run:

`poetry install --no-root`

This installs all the Python packages we need for this workshop, including **Netmiko**.

It knows what we need due to the “pyproject.toml” file in our folder.



```
clab@ubuntu:~/2025-nanog93-tutorial-automation$ poetry install --no-root
Creating virtualenv 2024-lonisummit-workshop-automation-eOMokOBy-py3.9 in /home/clab/.cache
Installing dependencies from lock file

Package operations: 30 installs, 0 updates, 0 removals

  - Installing pycparser (2.21)
  - Installing cffi (1.16.0)
  - Installing bcrypt (4.1.2)
  - Installing cryptography (42.0.5)
  - Installing future (1.0.0)
  - Installing pynacl (1.5.0)
  - Installing six (1.16.0)
  - Installing paramiko (3.4.0)
  - Installing pyyaml (6.0.1)
  - Installing textfsm (1.1.3)
  - Installing wrapt (1.16.0)
  - Installing click (8.1.7)
  - Installing deprecated (1.2.14)
  - Installing dnspython (2.6.1)
  - Installing hier-config (2.2.3)
  - Installing loguru (0.7.2)
  - Installing mypy-extensions (1.0.0)
  - Installing ntc-templates (4.4.0)
  - Installing packaging (24.0)
  - Installing passlib (1.7.4)
  - Installing pathspec (0.12.1)
  - Installing platformdirs (4.2.0)
  - Installing pyserial (3.5)
  - Installing scp (0.14.5)
  - Installing toml (0.10.2)
  - Installing tomli (2.0.1)
  - Installing typing-extensions (4.10
  - Installing black (24.3.0)
  - Installing ciscoconfparse (    41)
  - Installing netmiko (4.3.0)
clab@ubuntu:~/2025-nanog93-tutorial-automation$ 
```

Lab 1:

Reading Configuration and Operational Data

NANOG™

Disclaimer

We are going to look at a lot of code, and learn about network automation-related Python packages, but **you are not expected to know or learn how to code**.

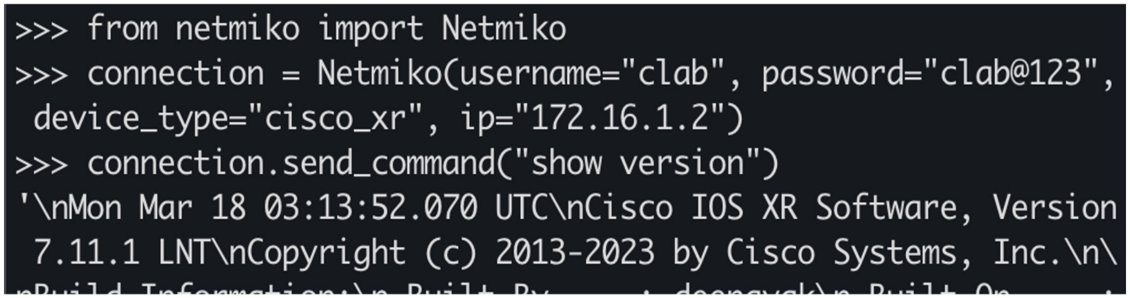
You are also heavily encouraged to open these slides on your own computer as well, so you can follow along easier, catch up, or move ahead at your own speed.

Lab 1, Part 1: Netmiko

Netmiko is an open-source **Python library** that does the heavy lifting of talking to your network devices and giving you their output.

https://github.com/ktbyers/netmiko

It is a **CLI Scraping** tool.



```
>>> from netmiko import Netmiko
>>> connection = Netmiko(username="clab", password="clab@123",
 device_type="cisco_xr", ip="172.16.1.2")
>>> connection.send_command("show version")
'\nMon Mar 18 03:13:52.070 UTC\nCisco IOS XR Software, Version
 7.11.1 LNT\nCopyright (c) 2013-2023 by Cisco Systems, Inc.\n\
nBuild Information:\n Built By     : deenayak\n Built On     :
```

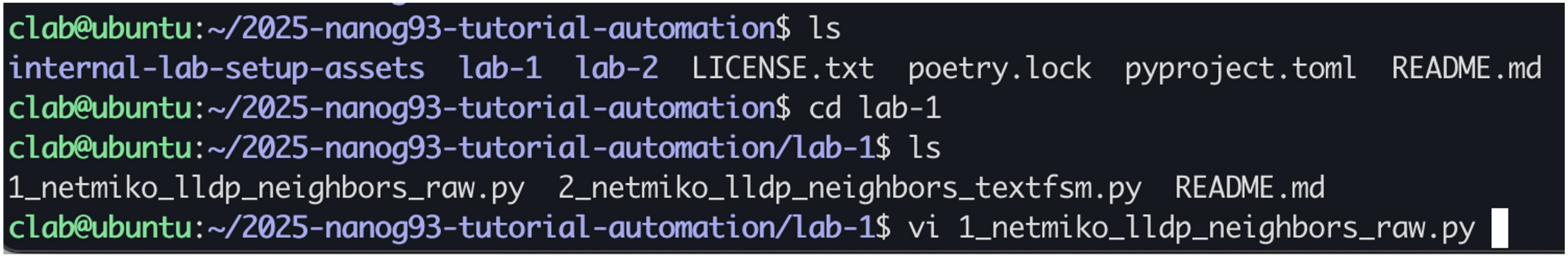
From the source:

“Network automation to screen-scraping devices is primarily concerned with gathering output from show commands and with making configuration changes.”

Our First Script

- From our folder, **`cd`** into the **`lab-1`** folder
- Open with your favorite text editor: **1_netmiko_lldp_neighbors_raw**

(vim, nano, and emacs are available)



```
clab@ubuntu:~/2025-nanog93-tutorial-automation$ ls
internal-lab-setup-assets  lab-1  lab-2  LICENSE.txt  poetry.lock  pyproject.toml  README.md
clab@ubuntu:~/2025-nanog93-tutorial-automation$ cd lab-1
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-1$ ls
1_netmiko_lldp_neighbors_raw.py  2_netmiko_lldp_neighbors_textfsm.py  README.md
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-1$ vi 1_netmiko_lldp_neighbors_raw.py
```

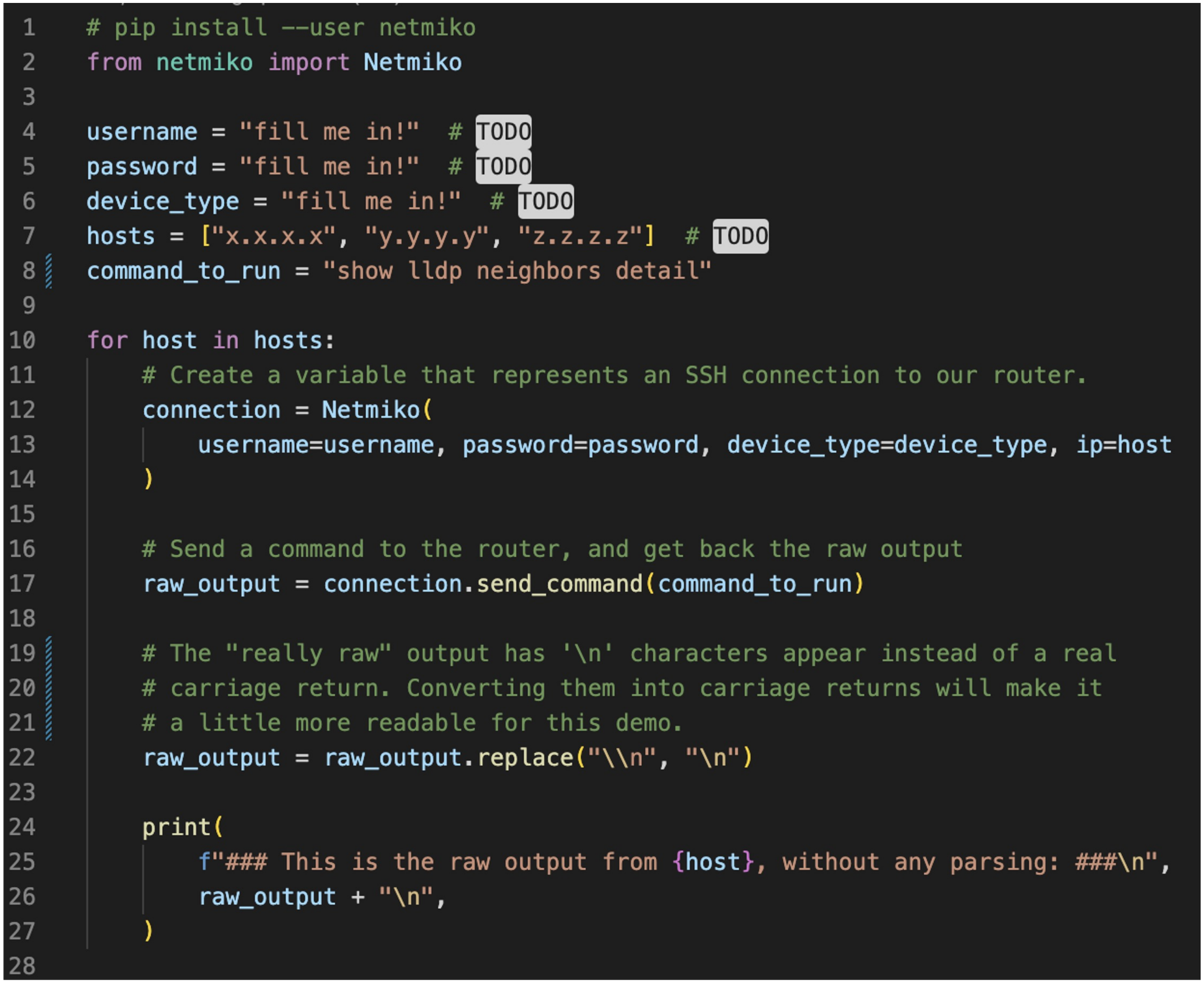
Our First Script

Input:

- Username
- Password
- Device Type (or platform)
- List of IPs
- A command to run

Output:

Prints the raw output that it sees on the router when the command is run

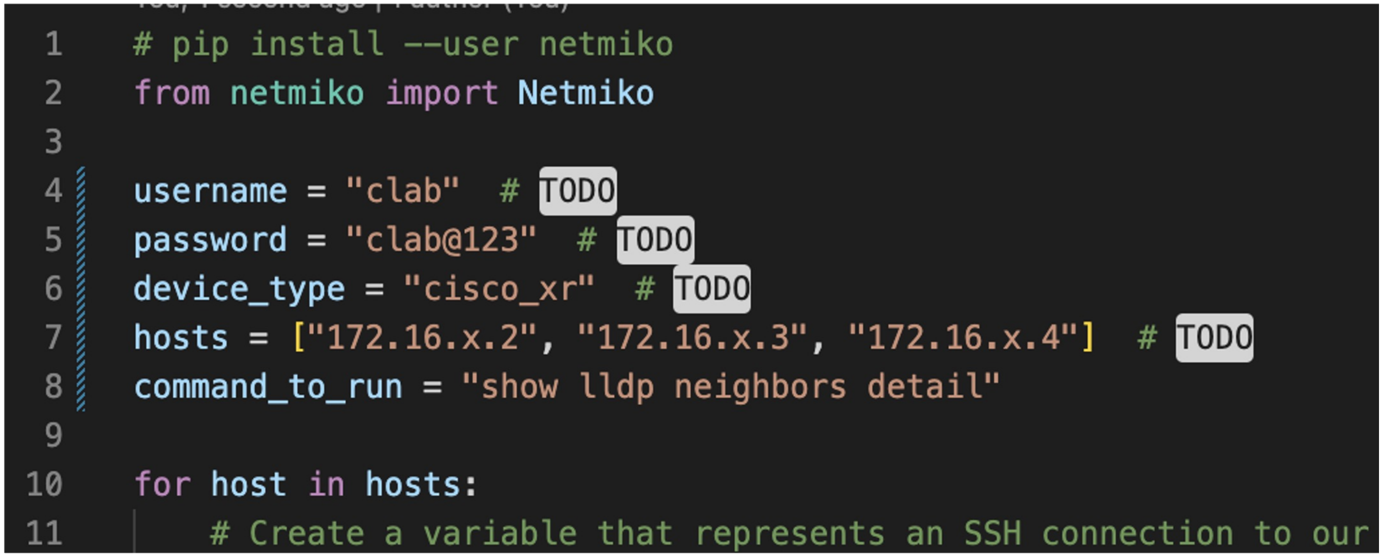


```python
# pip install --user netmiko
from netmiko import Netmiko

username = "fill me in!"  # TODO
password = "fill me in!"  # TODO
device_type = "fill me in!"  # TODO
hosts = ["x.x.x.x", "y.y.y.y", "z.z.z.z"]  # TODO
command_to_run = "show lldp neighbors detail"

for host in hosts:
    # Create a variable that represents an SSH connection to our router.
    connection = Netmiko(
        username=username, password=password, device_type=device_type, ip=host
    )

    # Send a command to the router, and get back the raw output
    raw_output = connection.send_command(command_to_run)

    # The "really raw" output has '\n' characters appear instead of a real
    # carriage return. Converting them into carriage returns will make it
    # a little more readable for this demo.
    raw_output = raw_output.replace("\\n", "\n")

    print(
        f"### This is the raw output from {host}, without any parsing: ###\n",
        raw_output + "\n",
    )

```

Our First Script

Using your lab card and your favorite terminal text editor:

1. Use **“cisco_xr”** as the device_type.



```
# pip install --user netmiko
from netmiko import Netmiko

username = "clab"  # TODO
password = "clab@123"  # TODO
device_type = "cisco_xr"  # TODO
hosts = ["172.16.x.2", "172.16.x.3", "172.16.x.4"]  # TODO
command_to_run = "show lldp neighbors detail"

for host in hosts:
    # Create a variable that represents an SSH connection to our
```

2. Replace the “x” in the IP addresses with your lab number. **The IP addresses should match what is on your card**.

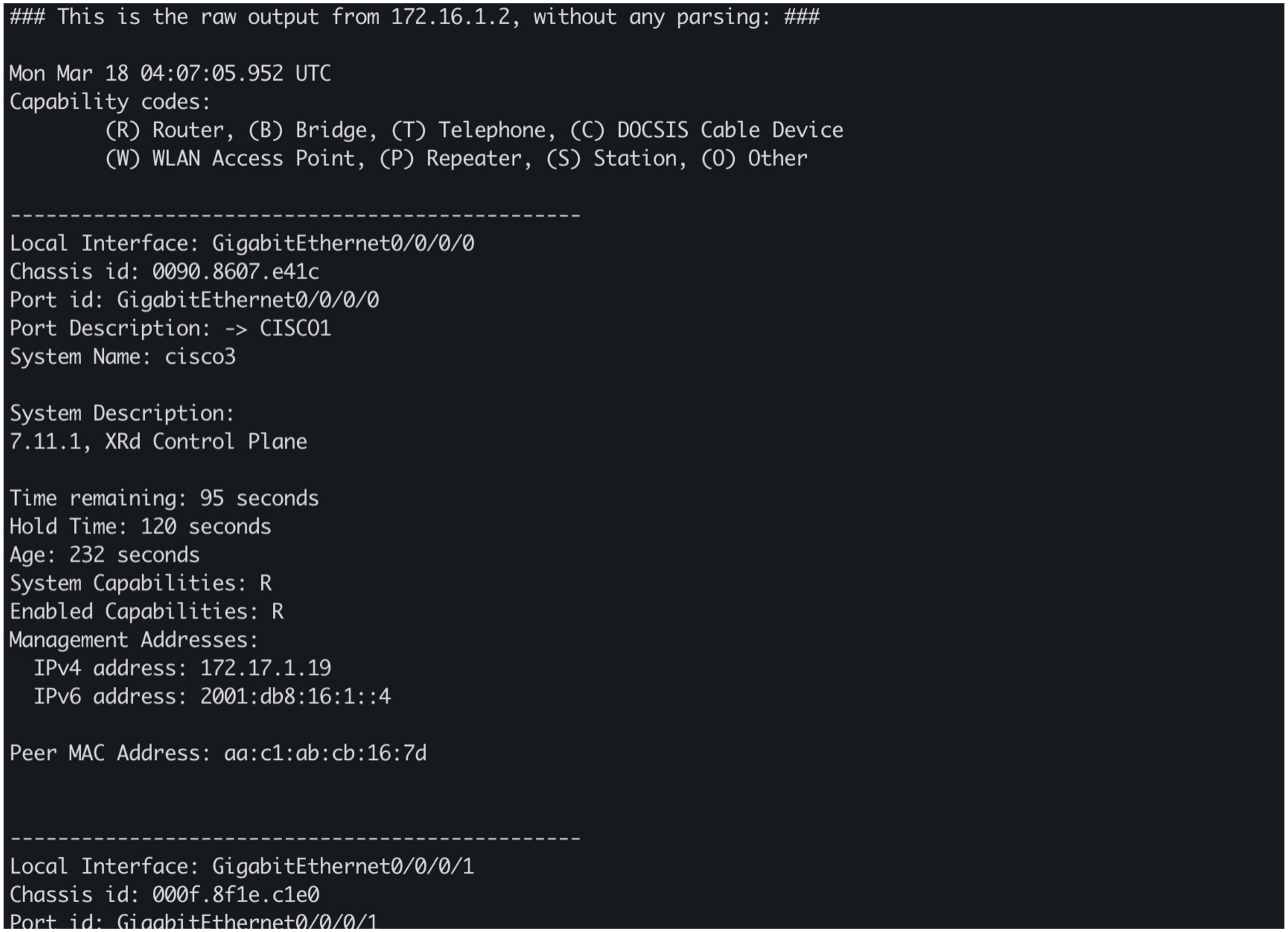
3. Save and exit the script when you are finished.

Our First Script

Finally, run the script with:

```
poetry run python 1_netmiko_lldp_neighbors_raw.py
```

poetry run python 1_netmiko_lldp_neighbors_raw.py



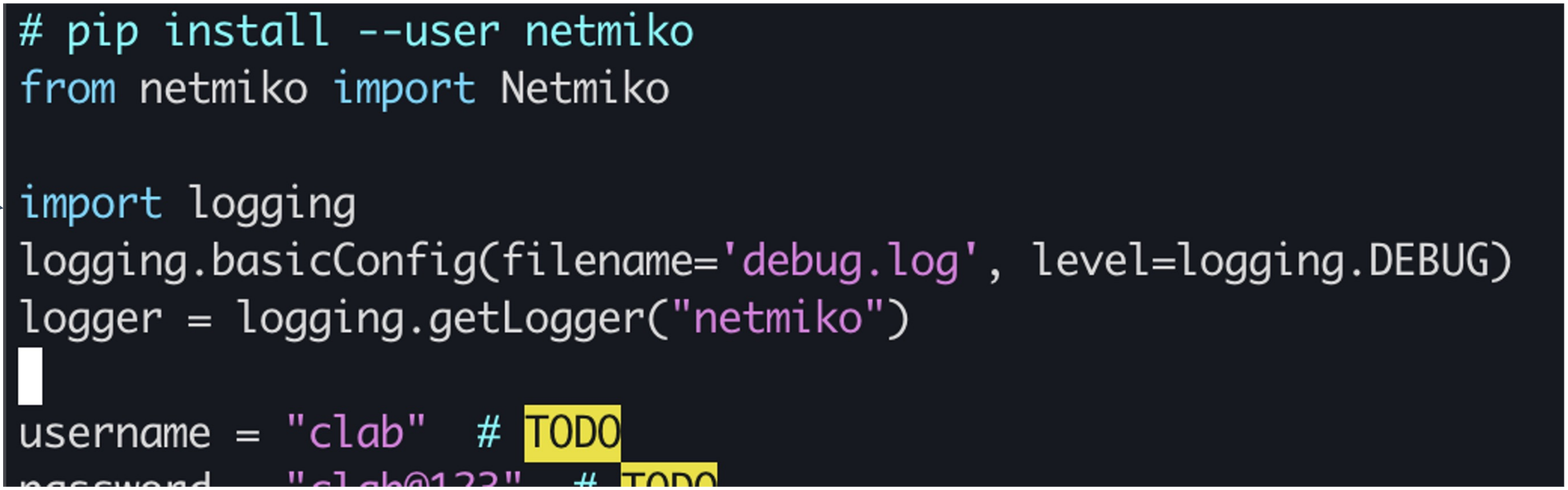
```
### This is the raw output from 172.16.1.2, without any parsing: ###

Mon Mar 18 04:07:05.952 UTC
Capability codes:
        (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
        (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

------------------------------------------------
Local Interface: GigabitEthernet0/0/0/0
Chassis id: 0090.8607.e41c
Port id: GigabitEthernet0/0/0/0
Port Description: -> CISCO1
System Name: cisco3

System Description:
7.11.1, XRd Control Plane

Time remaining: 95 seconds
Hold Time: 120 seconds
Age: 232 seconds
System Capabilities: R
Enabled Capabilities: R
Management Addresses:
  IPv4 address: 172.17.1.19
  IPv6 address: 2001:db8:16:1::4

Peer MAC Address: aa:c1:ab:cb:16:7d



------------------------------------------------
Local Interface: GigabitEthernet0/0/0/1
Chassis id: 000f.8f1e.c1e0
Port id: GigabitEthernet0/0/0/1
```

Kinda cool, right?

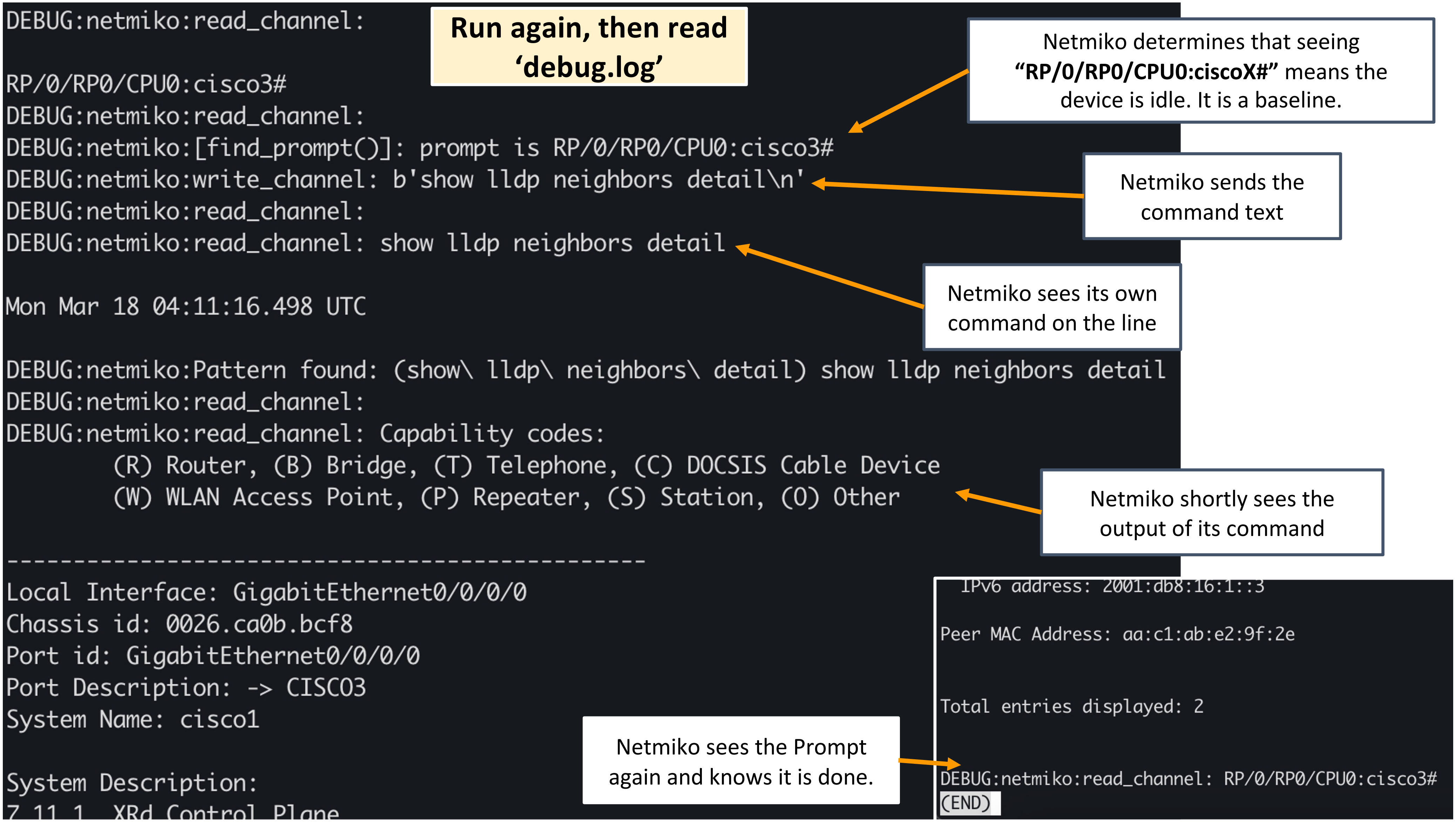
But what is really happening?

Let’s enable Netmiko’s logging functionality and look for ourselves. We can add these three logging-related lines to the top of the script, like so.

(You’re not required to actually do so, we’ll see things on the next slide)



```
# pip install --user netmiko
from netmiko import Netmiko

import logging
logging.basicConfig(filename='debug.log', level=logging.DEBUG)
logger = logging.getLogger("netmiko")

username = "clab"  # TODO
password = "clab@123"  # TODO
```



Run again, then read 'debug.log'

```
DEBUG:netmiko:read_channel:

RP/0/RP0/CPU0:cisco3#
DEBUG:netmiko:read_channel:
DEBUG:netmiko:[find_prompt()]: prompt is RP/0/RP0/CPU0:cisco3#
DEBUG:netmiko:write_channel: b'show lldp neighbors detail\n'
DEBUG:netmiko:read_channel:
DEBUG:netmiko:read_channel: show lldp neighbors detail

Mon Mar 18 04:11:16.498 UTC

DEBUG:netmiko:Pattern found: (show\ lldp\ neighbors\ detail) show lldp neighbors detail
DEBUG:netmiko:read_channel:
DEBUG:netmiko:read_channel: Capability codes:
        (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
        (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

------------------------------------------------
Local Interface: GigabitEthernet0/0/0/0
Chassis id: 0026.ca0b.bcf8
Port id: GigabitEthernet0/0/0/0
Port Description: -> CISCO3
System Name: cisco1

System Description:
7.11.1, XRd Control Plane
```

```
  IPv6 address: 2001:db8:16:1::3

Peer MAC Address: aa:c1:ab:e2:9f:2e


Total entries displayed: 2


DEBUG:netmiko:read_channel: RP/0/RP0/CPU0:cisco3#
(END)
```

Netmiko determines that seeing **"RP/0/RP0/CPU0:ciscoX#"** means the device is idle. It is a baseline.

Netmiko sends the command text

Netmiko sees its own command on the line

Netmiko shortly sees the output of its command

Netmiko sees the Prompt again and knows it is done.

Lab 1, Part 2: TextFSM

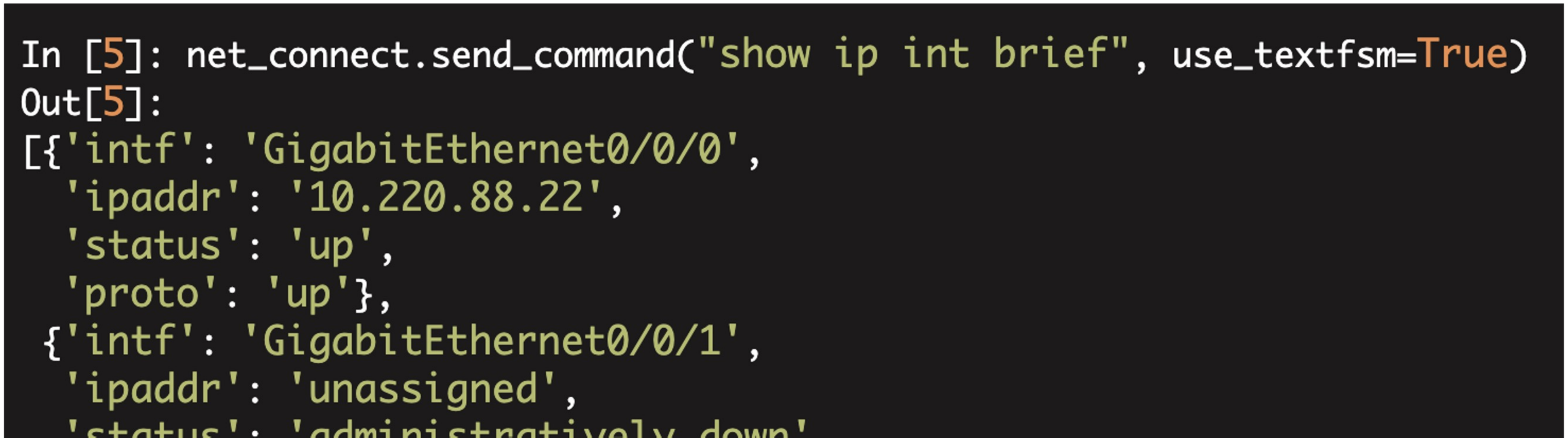
TextFSM is a templating framework for network devices, developed internally at Google.
https://github.com/google/textfsm

Their description:
“Python module which implements a template based state machine for parsing semi-formatted text. Originally developed to allow programmatic access to information returned from the command line interface (CLI) of networking devices.”

A different description, from our friends at NetworkToCode:
“TextFSM is a tool to help make parsing cli commands more manageable.”

TextFSM: Getting a little more complicated

Using **TextFSM** with **TextFSM templates** (created by us or the community, including NetworkToCode), can “auto-format” the raw output from a device into a standardized, computer-readable format.



```
In [5]: net_connect.send_command("show ip int brief", use_textfsm=True)
Out[5]:
[{'intf': 'GigabitEthernet0/0/0',
  'ipaddr': '10.220.88.22',
  'status': 'up',
  'proto': 'up'},
 {'intf': 'GigabitEthernet0/0/1',
  'ipaddr': 'unassigned',
  'status': 'administratively down'
```

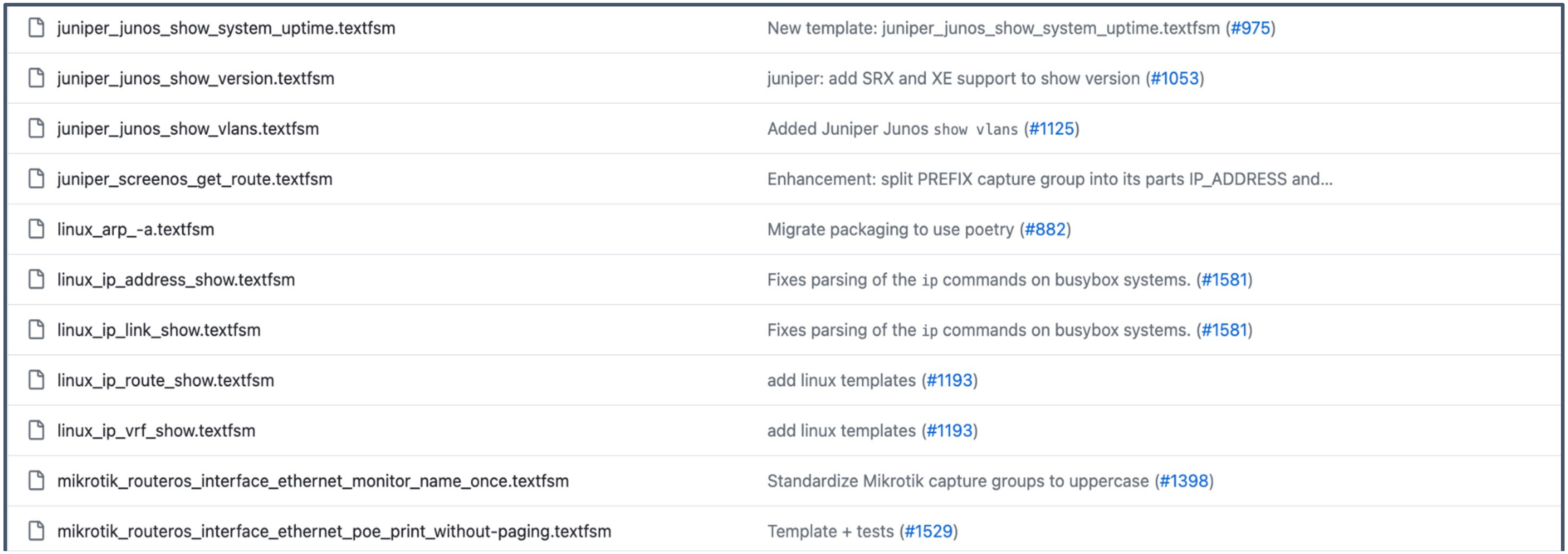
Pictured: The computer-readable format of “show ip int brief”. This is known as a Python Dictionary

Netmiko can use **TextFSM**, and makes it easy for us to leverage it! We just need to install the right packages. This has already been installed for you.

TextFSM: Someone already did the work

NetworkToCode collaborates with the open-source community to develop templates for different commands, platforms, and networking vendors.

https://github.com/networktocode/ntc-templates/tree/master/ntc_templates/templates



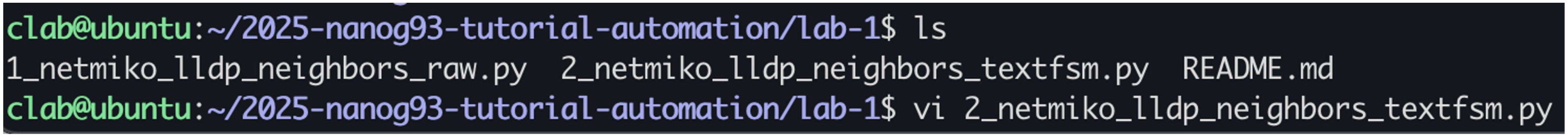
juniper_junos_show_system_uptime.textfsm	New template: juniper_junos_show_system_uptime.textfsm (#975)
juniper_junos_show_version.textfsm	juniper: add SRX and XE support to show version (#1053)
juniper_junos_show_vlans.textfsm	Added Juniper Junos `show vlans` (#1125)
juniper_screenos_get_route.textfsm	Enhancement: split PREFIX capture group into its parts IP_ADDRESS and...
linux_arp_-a.textfsm	Migrate packaging to use poetry (#882)
linux_ip_address_show.textfsm	Fixes parsing of the `ip` commands on busybox systems. (#1581)
linux_ip_link_show.textfsm	Fixes parsing of the `ip` commands on busybox systems. (#1581)
linux_ip_route_show.textfsm	add linux templates (#1193)
linux_ip_vrf_show.textfsm	add linux templates (#1193)
mikrotik_routeros_interface_ethernet_monitor_name_once.textfsm	Standardize Mikrotik capture groups to uppercase (#1398)
mikrotik_routeros_interface_ethernet_poe_print_without-paging.textfsm	Template + tests (#1529)

Our Second Script

Within the same **`lab-1`** folder…

Open with your favorite text editor:

2_netmiko_lldp_neighbors_textfsm.py



```
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-1$ ls
1_netmiko_lldp_neighbors_raw.py  2_netmiko_lldp_neighbors_textfsm.py  README.md
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-1$ vi 2_netmiko_lldp_neighbors_textfsm.py
```

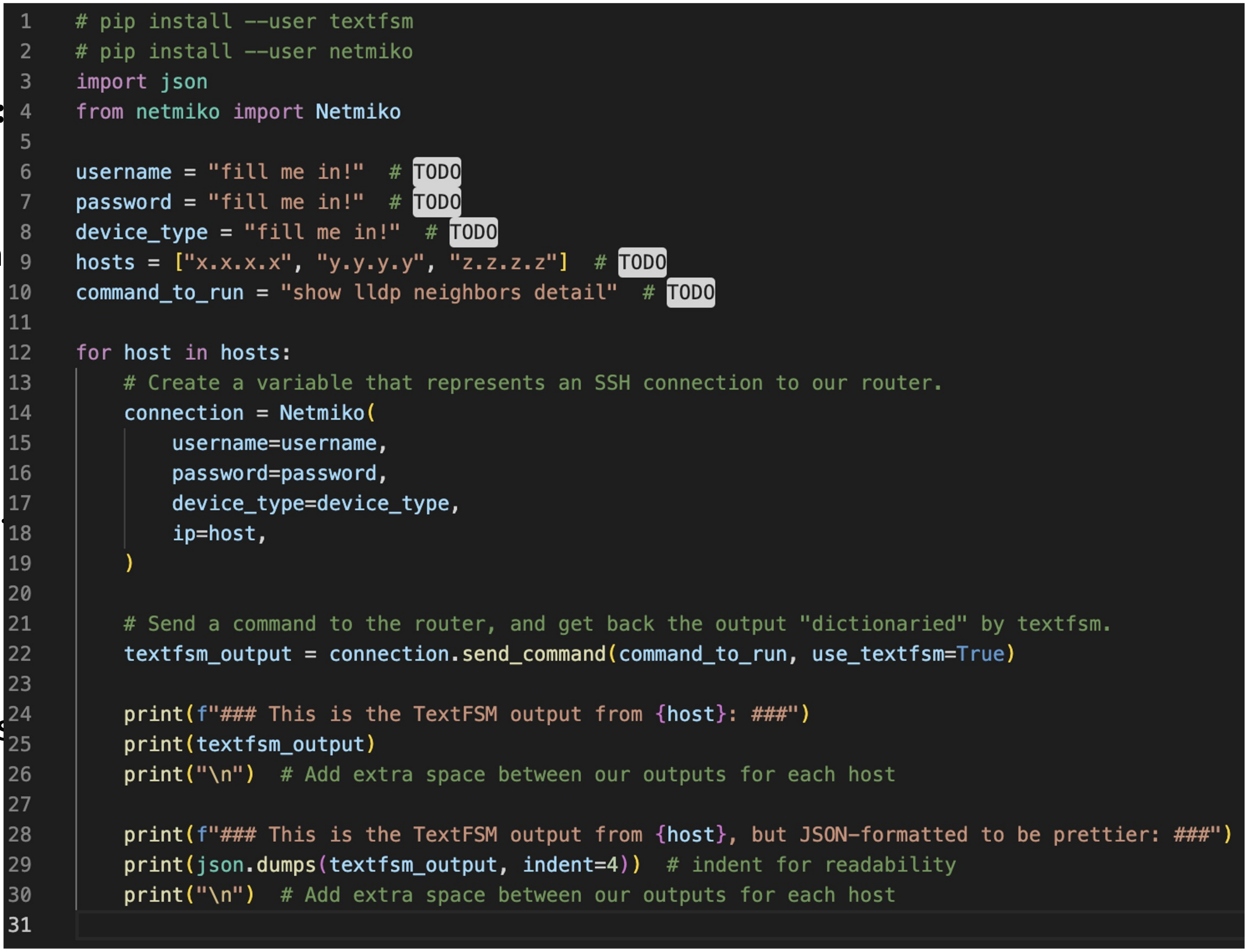
Our Second Script

Input (Same as last time):

- Username
- Password
- Device Type (or platform
- List of IPs
- A command to run

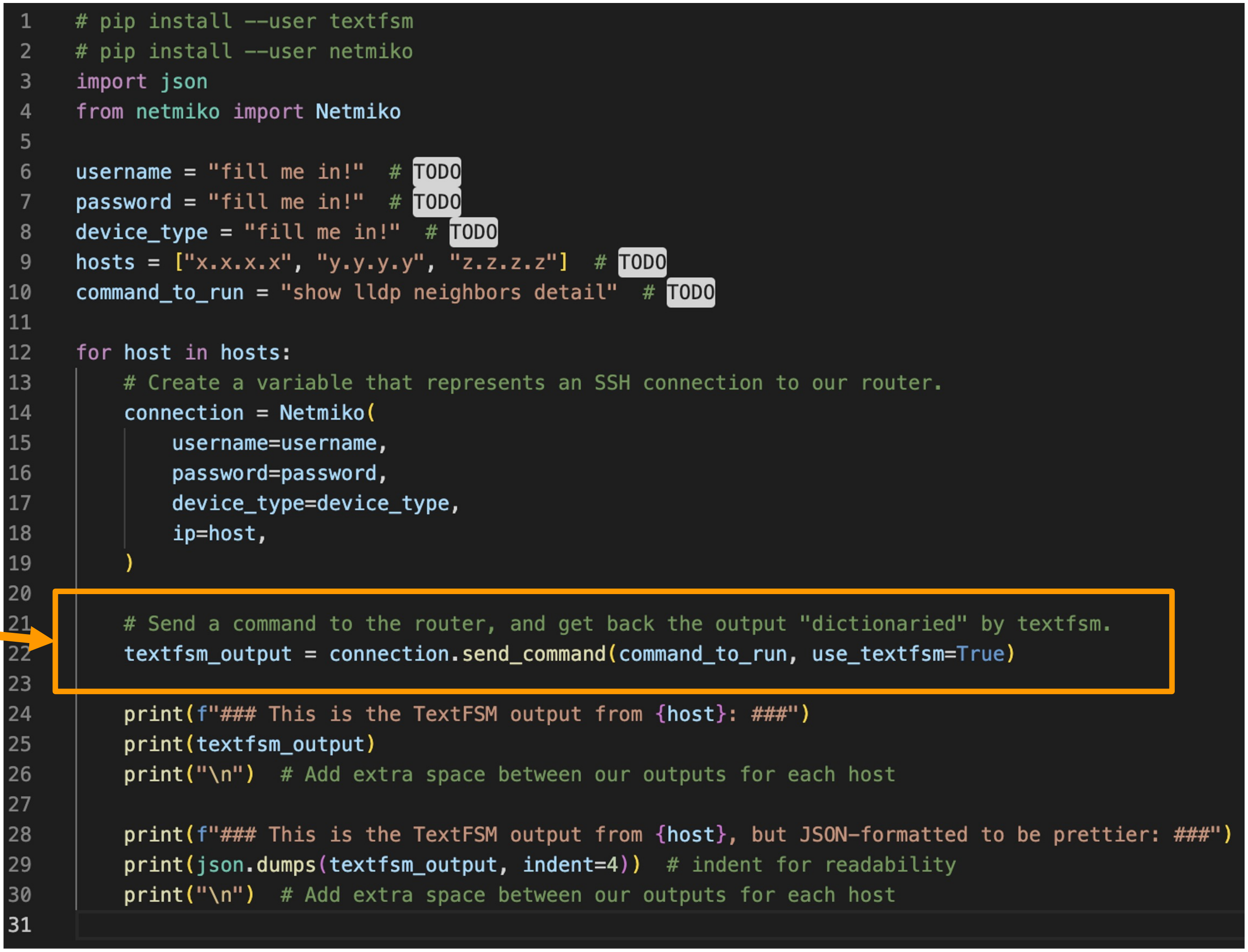
Output:

Prints **formatted** output. Netmiko retrieves the same raw router output, but formats it through TextFSM before it returns it.



```python
# pip install --user textfsm
# pip install --user netmiko
import json
from netmiko import Netmiko

username = "fill me in!"  # TODO
password = "fill me in!"  # TODO
device_type = "fill me in!"  # TODO
hosts = ["x.x.x.x", "y.y.y.y", "z.z.z.z"]  # TODO
command_to_run = "show lldp neighbors detail"  # TODO

for host in hosts:
    # Create a variable that represents an SSH connection to our router.
    connection = Netmiko(
        username=username,
        password=password,
        device_type=device_type,
        ip=host,
    )

    # Send a command to the router, and get back the output "dictionaried" by textfsm.
    textfsm_output = connection.send_command(command_to_run, use_textfsm=True)

    print(f"### This is the TextFSM output from {host}: ###")
    print(textfsm_output)
    print("\n")  # Add extra space between our outputs for each host

    print(f"### This is the TextFSM output from {host}, but JSON-formatted to be prettier: ###")
    print(json.dumps(textfsm_output, indent=4))  # indent for readability
    print("\n")  # Add extra space between our outputs for each host
```

Our Second Script



```python
# pip install --user textfsm
# pip install --user netmiko
import json
from netmiko import Netmiko

username = "fill me in!"  # TODO
password = "fill me in!"  # TODO
device_type = "fill me in!"  # TODO
hosts = ["x.x.x.x", "y.y.y.y", "z.z.z.z"]  # TODO
command_to_run = "show lldp neighbors detail"  # TODO

for host in hosts:
    # Create a variable that represents an SSH connection to our router.
    connection = Netmiko(
        username=username,
        password=password,
        device_type=device_type,
        ip=host,
    )

    # Send a command to the router, and get back the output "dictionaried" by textfsm.
    textfsm_output = connection.send_command(command_to_run, use_textfsm=True)

    print(f"### This is the TextFSM output from {host}: ###")
    print(textfsm_output)
    print("\n")  # Add extra space between our outputs for each host

    print(f"### This is the TextFSM output from {host}, but JSON-formatted to be prettier: ###")
    print(json.dumps(textfsm_output, indent=4))  # indent for readability
    print("\n")  # Add extra space between our outputs for each host
```

This is the magic

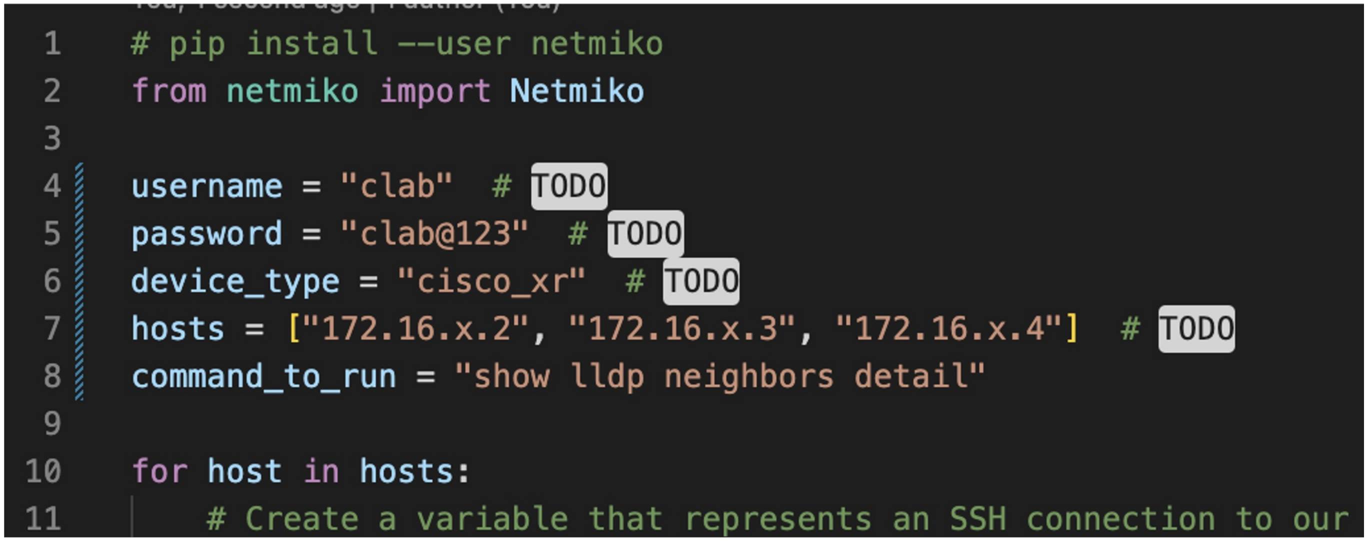
Our Second Script

“Once more, with feeling”

Using your lab card and your favorite terminal text editor, fill in the data **just like we did in the first script**.

Use **“cisco_xr”** as the device_type.

Save and exit the script when you are finished.



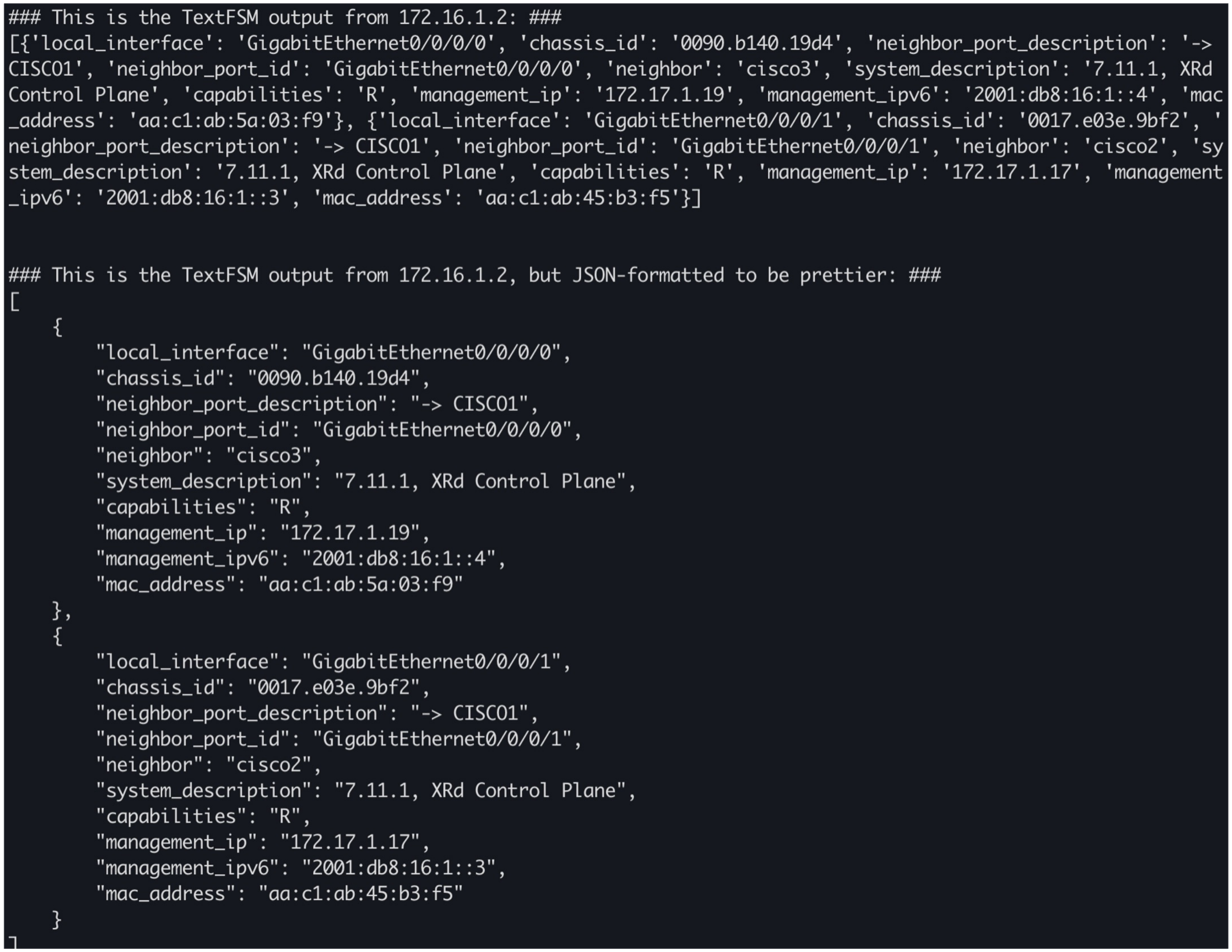
```
# pip install --user netmiko
from netmiko import Netmiko

username = "clab"  # TODO
password = "clab@123"  # TODO
device_type = "cisco_xr"  # TODO
hosts = ["172.16.x.2", "172.16.x.3", "172.16.x.4"]  # TODO
command_to_run = "show lldp neighbors detail"

for host in hosts:
    # Create a variable that represents an SSH connection to our
```

Our Second Script

Finally, run the script with:

```
poetry run python 2_netmiko_lldp_neighbors_textfsm.py
```

poetry run python 2_netmiko_lldp_neighbors_textfsm.py

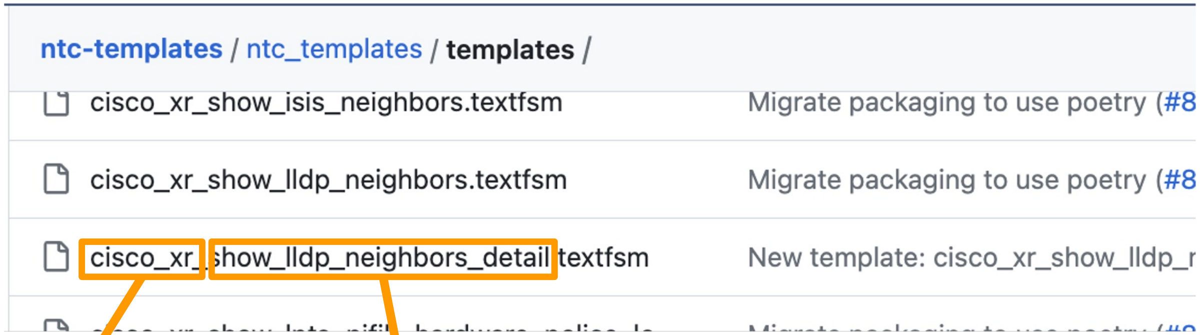


```
### This is the TextFSM output from 172.16.1.2: ###
[{'local_interface': 'GigabitEthernet0/0/0/0', 'chassis_id': '0090.b140.19d4', 'neighbor_port_description': '->
CISCO1', 'neighbor_port_id': 'GigabitEthernet0/0/0/0', 'neighbor': 'cisco3', 'system_description': '7.11.1, XRd
Control Plane', 'capabilities': 'R', 'management_ip': '172.17.1.19', 'management_ipv6': '2001:db8:16:1::4', 'mac
_address': 'aa:c1:ab:5a:03:f9'}, {'local_interface': 'GigabitEthernet0/0/0/1', 'chassis_id': '0017.e03e.9bf2', '
neighbor_port_description': '-> CISCO1', 'neighbor_port_id': 'GigabitEthernet0/0/0/1', 'neighbor': 'cisco2', 'sy
stem_description': '7.11.1, XRd Control Plane', 'capabilities': 'R', 'management_ip': '172.17.1.17', 'management
_ipv6': '2001:db8:16:1::3', 'mac_address': 'aa:c1:ab:45:b3:f5'}]


### This is the TextFSM output from 172.16.1.2, but JSON-formatted to be prettier: ###
[
    {
        "local_interface": "GigabitEthernet0/0/0/0",
        "chassis_id": "0090.b140.19d4",
        "neighbor_port_description": "-> CISCO1",
        "neighbor_port_id": "GigabitEthernet0/0/0/0",
        "neighbor": "cisco3",
        "system_description": "7.11.1, XRd Control Plane",
        "capabilities": "R",
        "management_ip": "172.17.1.19",
        "management_ipv6": "2001:db8:16:1::4",
        "mac_address": "aa:c1:ab:5a:03:f9"
    },
    {
        "local_interface": "GigabitEthernet0/0/0/1",
        "chassis_id": "0017.e03e.9bf2",
        "neighbor_port_description": "-> CISCO1",
        "neighbor_port_id": "GigabitEthernet0/0/0/1",
        "neighbor": "cisco2",
        "system_description": "7.11.1, XRd Control Plane",
        "capabilities": "R",
        "management_ip": "172.17.1.17",
        "management_ipv6": "2001:db8:16:1::3",
        "mac_address": "aa:c1:ab:45:b3:f5"
    }
]
```

TextFSM: How did it know which template to use?

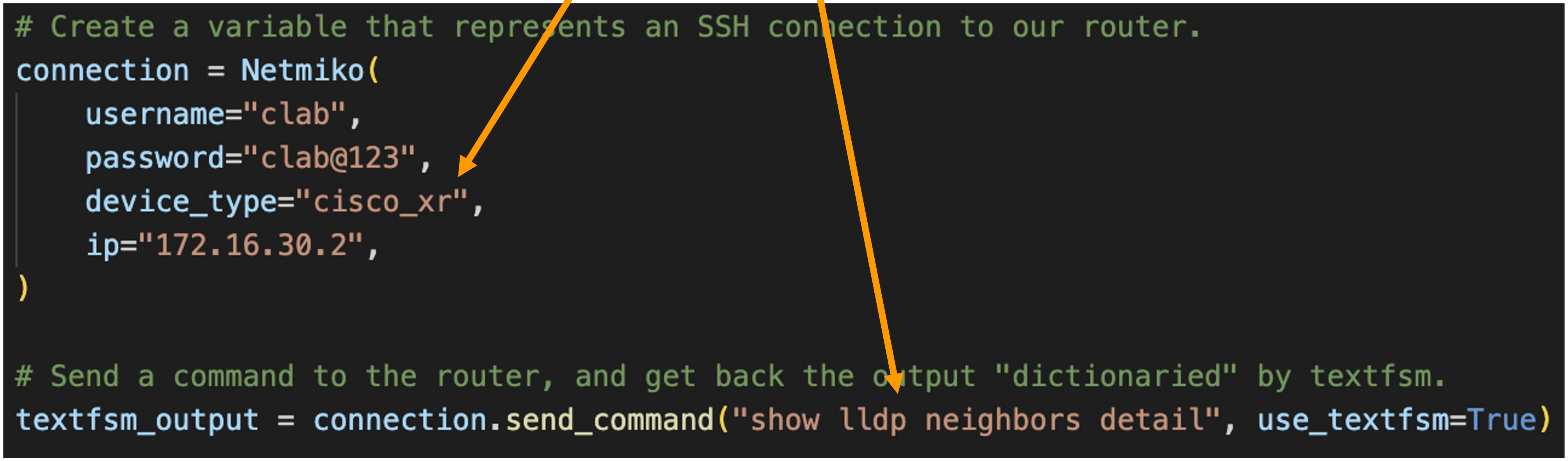
Netmiko takes the **device_type** and **command** statement and resolves it to a **filename**:

“cisco_xr_show_lldp_neighbors_detail.textfsm”



ntc-templates / ntc_templates / templates /

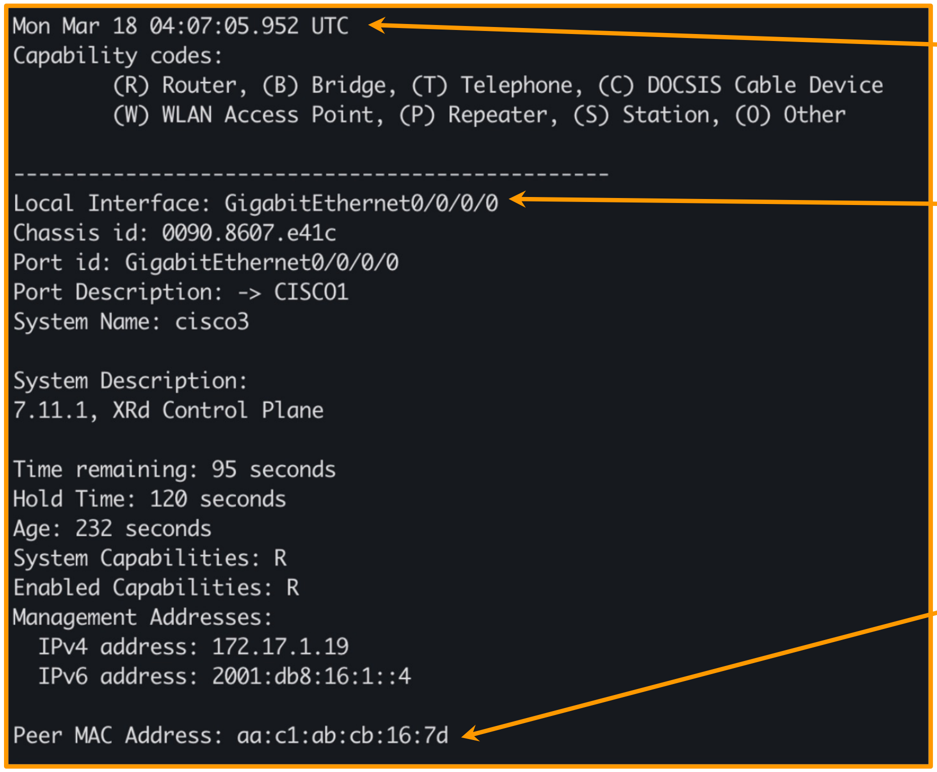
cisco_xr_show_isis_neighbors.textfsm	Migrate packaging to use poetry (#8
cisco_xr_show_lldp_neighbors.textfsm	Migrate packaging to use poetry (#8
cisco_xr_show_lldp_neighbors_detail.textfsm	New template: cisco_xr_show_lldp_r
cisco_xr_show_lpts_pifib_hardware_police_lo	Migrate packaging to use poetry (#8



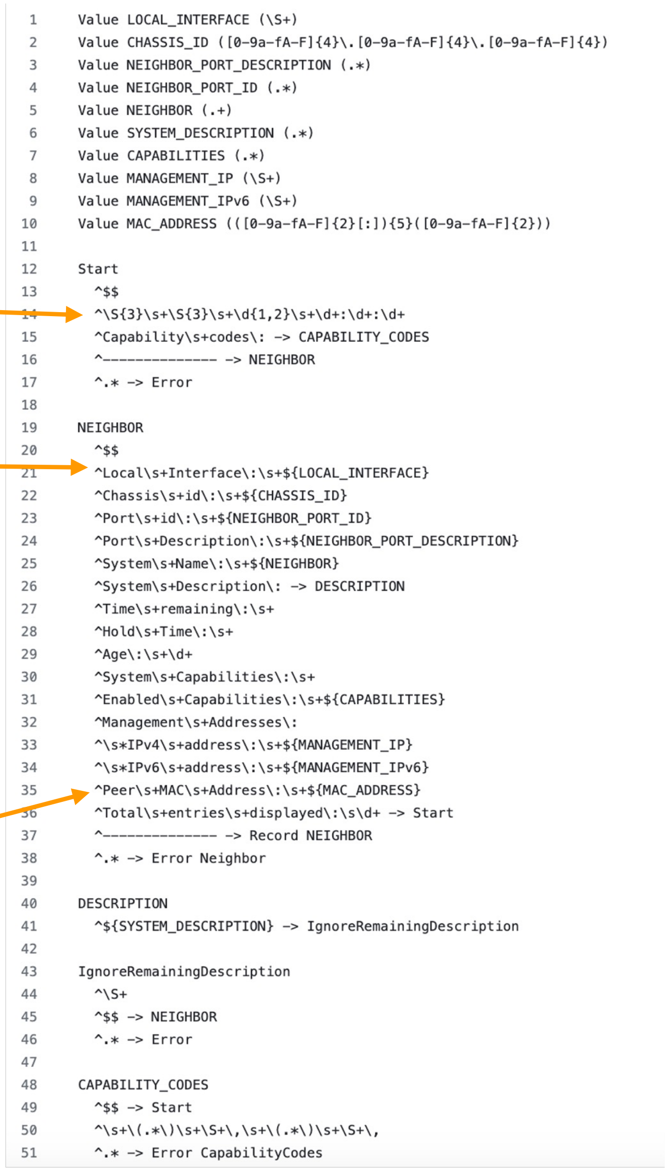
```
# Create a variable that represents an SSH connection to our router.
connection = Netmiko(
    username="clab",
    password="clab@123",
    device_type="cisco_xr",
    ip="172.16.30.2",
)

# Send a command to the router, and get back the output "dictionaried" by textfsm.
textfsm_output = connection.send_command("show lldp neighbors detail", use_textfsm=True)
```

TextFSM: What does a “template” look like anyway?

Answer: A little painful. But still familiar in some ways.



```
Mon Mar 18 04:07:05.952 UTC
Capability codes:
        (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
        (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

------------------------------------------------
Local Interface: GigabitEthernet0/0/0/0
Chassis id: 0090.8607.e41c
Port id: GigabitEthernet0/0/0/0
Port Description: -> CISCO1
System Name: cisco3

System Description:
7.11.1, XRd Control Plane

Time remaining: 95 seconds
Hold Time: 120 seconds
Age: 232 seconds
System Capabilities: R
Enabled Capabilities: R
Management Addresses:
  IPv4 address: 172.17.1.19
  IPv6 address: 2001:db8:16:1::4

Peer MAC Address: aa:c1:ab:cb:16:7d
```



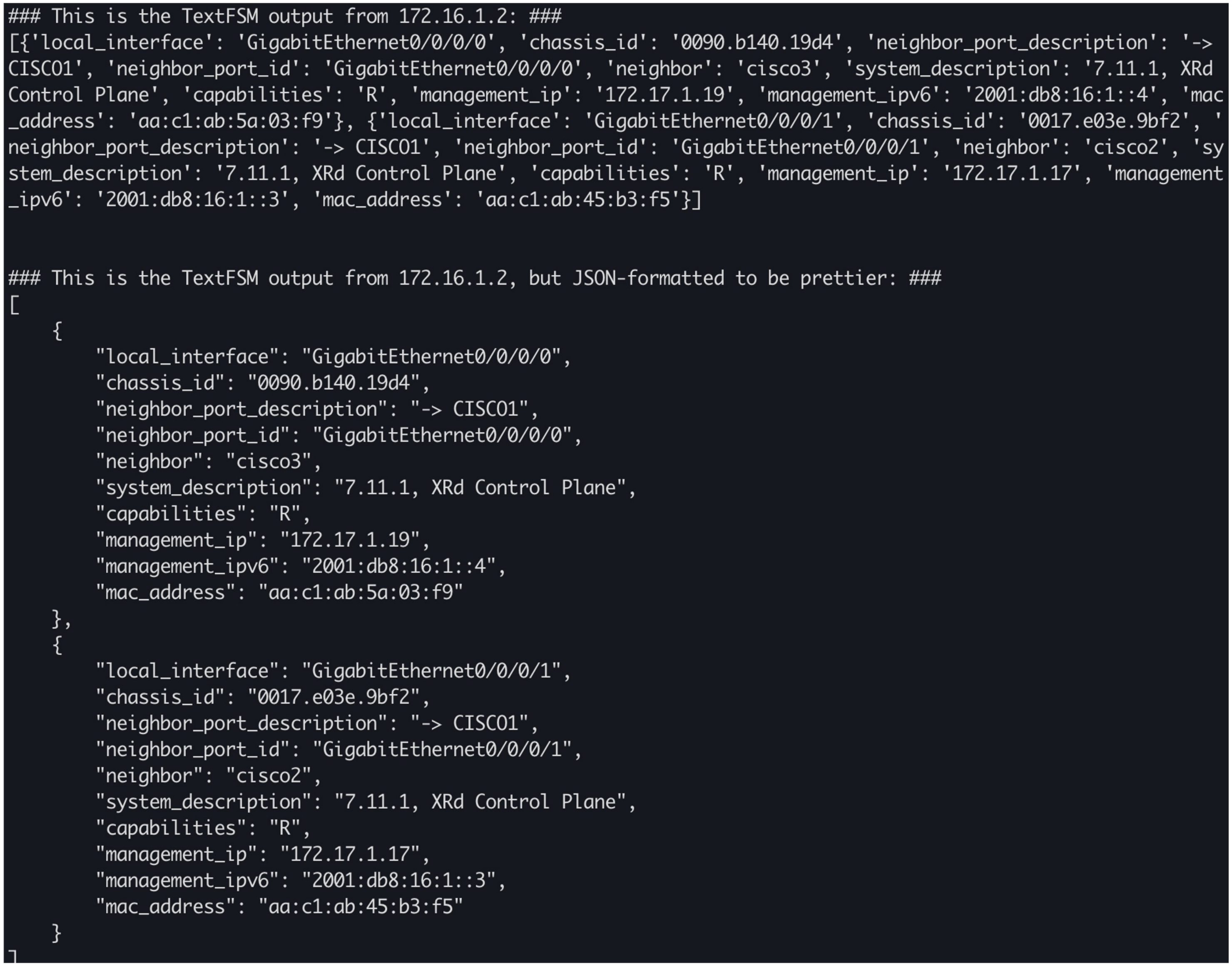
```
Value LOCAL_INTERFACE (\S+)
Value CHASSIS_ID ([0-9a-fA-F]{4}\.[0-9a-fA-F]{4}\.[0-9a-fA-F]{4})
Value NEIGHBOR_PORT_DESCRIPTION (.*)
Value NEIGHBOR_PORT_ID (.*)
Value NEIGHBOR (.+)
Value SYSTEM_DESCRIPTION (.*)
Value CAPABILITIES (.*)
Value MANAGEMENT_IP (\S+)
Value MANAGEMENT_IPv6 (\S+)
Value MAC_ADDRESS (([0-9a-fA-F]{2}[:]){5}([0-9a-fA-F]{2}))

Start
  ^$$
  ^\S{3}\s+\S{3}\s+\d{1,2}\s+\d+:\d+:\d+
  ^Capability\s+codes\: -> CAPABILITY_CODES
  ^-------------- -> NEIGHBOR
  ^.* -> Error

NEIGHBOR
  ^$$
  ^Local\s+Interface\:\s+${LOCAL_INTERFACE}
  ^Chassis\s+id\:\s+${CHASSIS_ID}
  ^Port\s+id\:\s+${NEIGHBOR_PORT_ID}
  ^Port\s+Description\:\s+${NEIGHBOR_PORT_DESCRIPTION}
  ^System\s+Name\:\s+${NEIGHBOR}
  ^System\s+Description\: -> DESCRIPTION
  ^Time\s+remaining\:\s+
  ^Hold\s+Time\:\s+
  ^Age\:\s+\d+
  ^System\s+Capabilities\:\s+
  ^Enabled\s+Capabilities\:\s+${CAPABILITIES}
  ^Management\s+Addresses\:
  ^\s*IPv4\s+address\:\s+${MANAGEMENT_IP}
  ^\s*IPv6\s+address\:\s+${MANAGEMENT_IPv6}
  ^Peer\s+MAC\s+Address\:\s+${MAC_ADDRESS}
  ^Total\s+entries\s+displayed\:\s\d+ -> Start
  ^-------------- -> Record NEIGHBOR
  ^.* -> Error Neighbor

DESCRIPTION
  ^${SYSTEM_DESCRIPTION} -> IgnoreRemainingDescription

IgnoreRemainingDescription
  ^\S+
  ^$$ -> NEIGHBOR
  ^.* -> Error

CAPABILITY_CODES
  ^$$ -> Start
  ^\s+\(.*\)\s+\S+\,\s+\(.*\)\s+\S+\,
  ^.* -> Error CapabilityCodes
```

https://github.com/networktocode/ntc-templates/blob/master/ntc_templates/templates/cisco_xr_show_lldp_neighbors_detail.textfsm

Goal met:
We extracted data from the network into a standardized format, **JSON**.

This is also considered a representation of **Infrastructure as Code (IaC)**

Let's take it a step further. **In the next lab**, we will use this LLDP information to configure interface descriptions.



```
### This is the TextFSM output from 172.16.1.2: ###
[{'local_interface': 'GigabitEthernet0/0/0/0', 'chassis_id': '0090.b140.19d4', 'neighbor_port_description': '->
CISCO1', 'neighbor_port_id': 'GigabitEthernet0/0/0/0', 'neighbor': 'cisco3', 'system_description': '7.11.1, XRd
Control Plane', 'capabilities': 'R', 'management_ip': '172.17.1.19', 'management_ipv6': '2001:db8:16:1::4', 'mac
_address': 'aa:c1:ab:5a:03:f9'}, {'local_interface': 'GigabitEthernet0/0/0/1', 'chassis_id': '0017.e03e.9bf2', '
neighbor_port_description': '-> CISCO1', 'neighbor_port_id': 'GigabitEthernet0/0/0/1', 'neighbor': 'cisco2', 'sy
stem_description': '7.11.1, XRd Control Plane', 'capabilities': 'R', 'management_ip': '172.17.1.17', 'management
_ipv6': '2001:db8:16:1::3', 'mac_address': 'aa:c1:ab:45:b3:f5'}]


### This is the TextFSM output from 172.16.1.2, but JSON-formatted to be prettier: ###
[
    {
        "local_interface": "GigabitEthernet0/0/0/0",
        "chassis_id": "0090.b140.19d4",
        "neighbor_port_description": "-> CISCO1",
        "neighbor_port_id": "GigabitEthernet0/0/0/0",
        "neighbor": "cisco3",
        "system_description": "7.11.1, XRd Control Plane",
        "capabilities": "R",
        "management_ip": "172.17.1.19",
        "management_ipv6": "2001:db8:16:1::4",
        "mac_address": "aa:c1:ab:5a:03:f9"
    },
    {
        "local_interface": "GigabitEthernet0/0/0/1",
        "chassis_id": "0017.e03e.9bf2",
        "neighbor_port_description": "-> CISCO1",
        "neighbor_port_id": "GigabitEthernet0/0/0/1",
        "neighbor": "cisco2",
        "system_description": "7.11.1, XRd Control Plane",
        "capabilities": "R",
        "management_ip": "172.17.1.17",
        "management_ipv6": "2001:db8:16:1::3",
        "mac_address": "aa:c1:ab:45:b3:f5"
    }
]
```

Lab 2:

Standardizing Configuration

04-FEB-2025

NANOG™

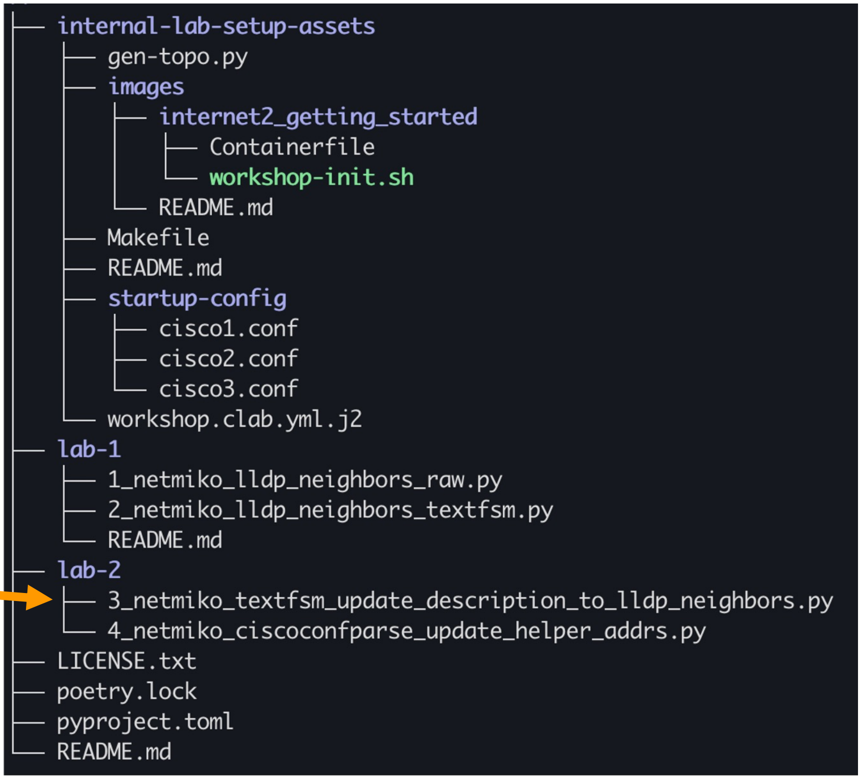
Lab 2, Part 1: Our Third Script

Committing configuration with Netmiko

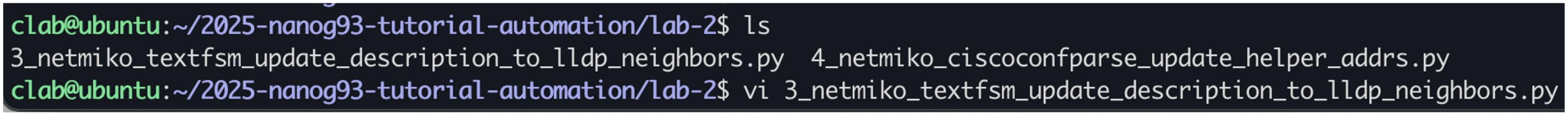
From the top folder, **`cd`** into the **`lab-2`** folder.

Open with your favorite text editor:

3_netmiko_textfsm_update_description_to_lldp_neighbors.py



```
├── internal-lab-setup-assets
│   ├── gen-topo.py
│   ├── images
│   │   ├── internet2_getting_started
│   │   │   ├── Containerfile
│   │   │   └── workshop-init.sh
│   │   └── README.md
│   ├── Makefile
│   ├── README.md
│   ├── startup-config
│   │   ├── cisco1.conf
│   │   ├── cisco2.conf
│   │   └── cisco3.conf
│   └── workshop.clab.yml.j2
├── lab-1
│   ├── 1_netmiko_lldp_neighbors_raw.py
│   ├── 2_netmiko_lldp_neighbors_textfsm.py
│   └── README.md
├── lab-2
│   ├── 3_netmiko_textfsm_update_description_to_lldp_neighbors.py
│   └── 4_netmiko_ciscoconfparse_update_helper_addrs.py
├── LICENSE.txt
├── poetry.lock
├── pyproject.toml
└── README.md
```

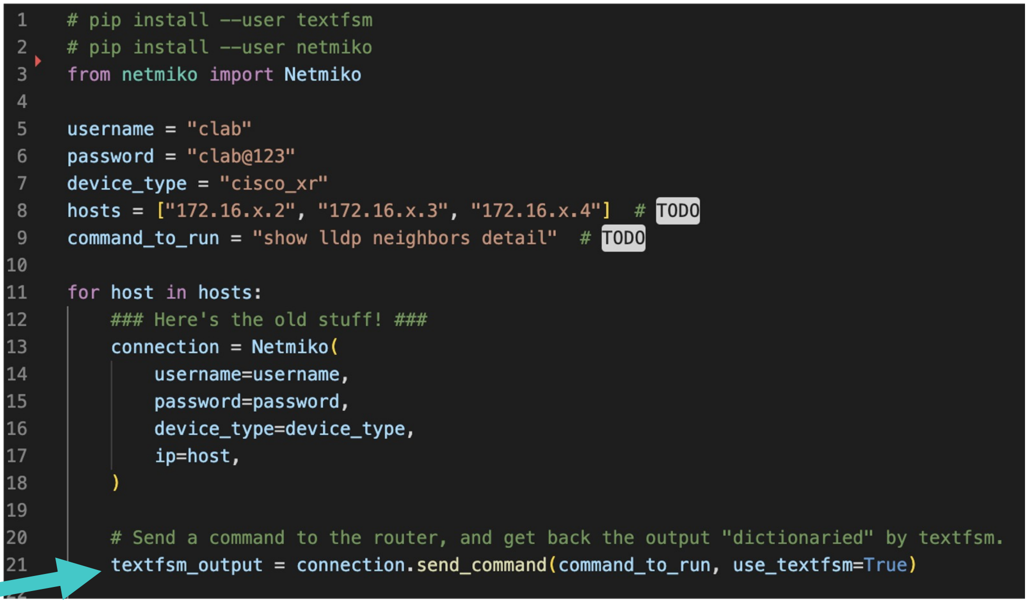


```
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-2$ ls
3_netmiko_textfsm_update_description_to_lldp_neighbors.py  4_netmiko_ciscoconfparse_update_helper_addrs.py
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-2$ vi 3_netmiko_textfsm_update_description_to_lldp_neighbors.py
```

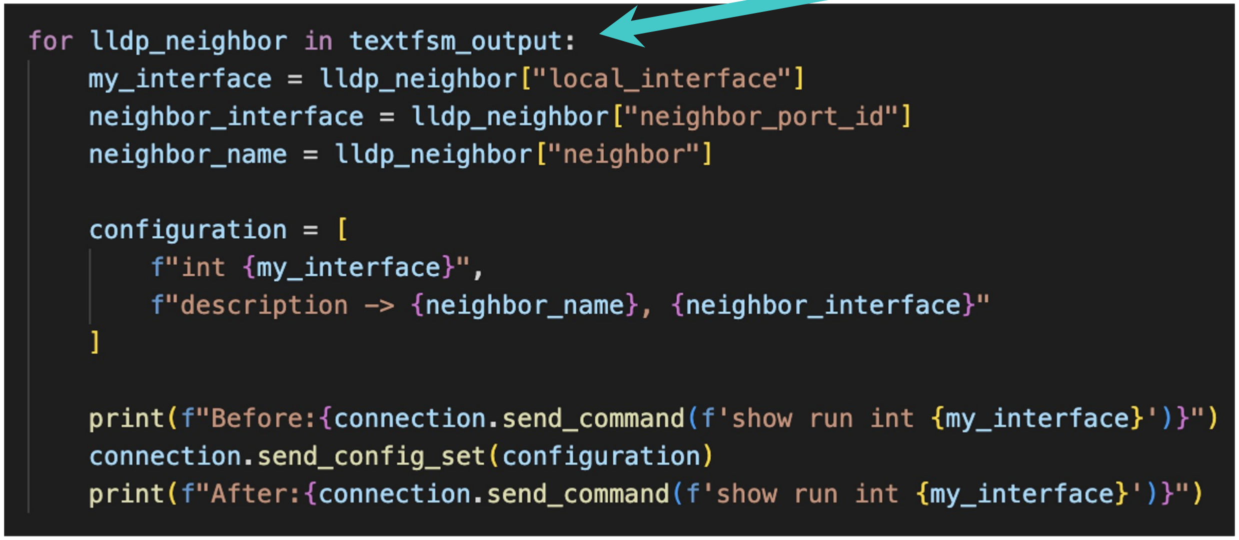
Our Third Script

This looks awfully familiar.

But if we scroll down...



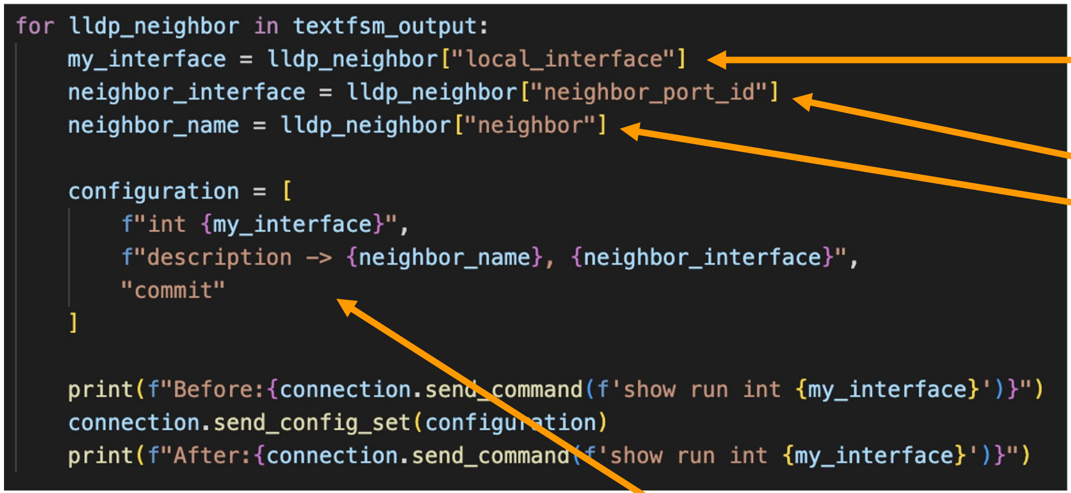
```
# pip install --user textfsm
# pip install --user netmiko
from netmiko import Netmiko

username = "clab"
password = "clab@123"
device_type = "cisco_xr"
hosts = ["172.16.x.2", "172.16.x.3", "172.16.x.4"]  # TODO
command_to_run = "show lldp neighbors detail"  # TODO

for host in hosts:
    ### Here's the old stuff! ###
    connection = Netmiko(
        username=username,
        password=password,
        device_type=device_type,
        ip=host,
    )

    # Send a command to the router, and get back the output "dictionaried" by textfsm.
    textfsm_output = connection.send_command(command_to_run, use_textfsm=True)
```



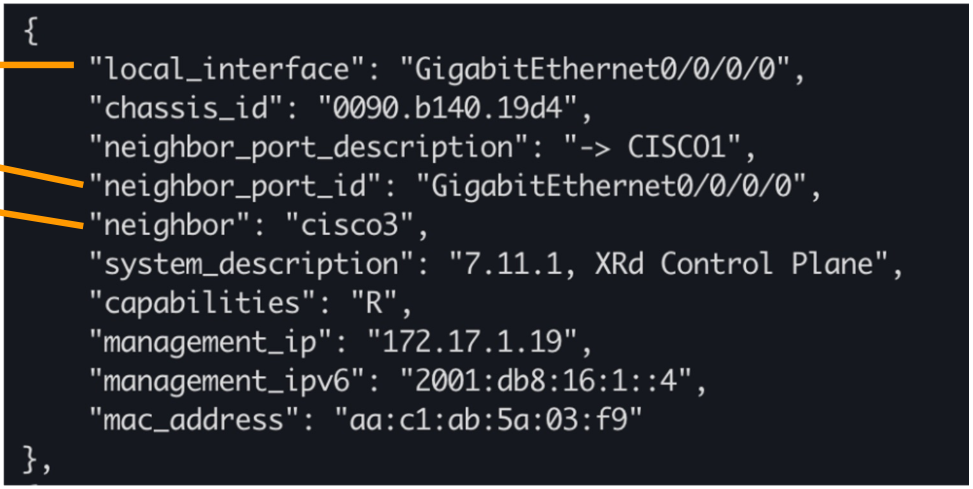
```
for lldp_neighbor in textfsm_output:
    my_interface = lldp_neighbor["local_interface"]
    neighbor_interface = lldp_neighbor["neighbor_port_id"]
    neighbor_name = lldp_neighbor["neighbor"]

    configuration = [
        f"int {my_interface}",
        f"description -> {neighbor_name}, {neighbor_interface}"
    ]

    print(f"Before:{connection.send_command(f'show run int {my_interface}')}")
    connection.send_config_set(configuration)
    print(f"After:{connection.send_command(f'show run int {my_interface}')}")
```

Our Third Script

Let’s remember what the output looks like and compare.

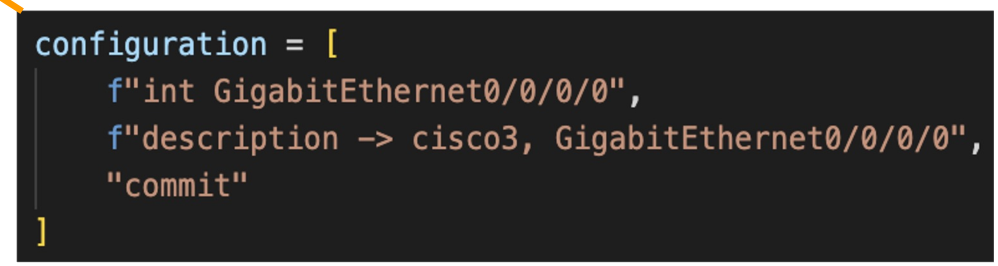


```
for lldp_neighbor in textfsm_output:
    my_interface = lldp_neighbor["local_interface"]
    neighbor_interface = lldp_neighbor["neighbor_port_id"]
    neighbor_name = lldp_neighbor["neighbor"]

    configuration = [
        f"int {my_interface}",
        f"description -> {neighbor_name}, {neighbor_interface}",
        "commit"
    ]

    print(f"Before:{connection.send_command(f'show run int {my_interface}')}")
    connection.send_config_set(configuration)
    print(f"After:{connection.send_command(f'show run int {my_interface}')}")
```



```
{
    "local_interface": "GigabitEthernet0/0/0/0",
    "chassis_id": "0090.b140.19d4",
    "neighbor_port_description": "-> CISCO1",
    "neighbor_port_id": "GigabitEthernet0/0/0/0",
    "neighbor": "cisco3",
    "system_description": "7.11.1, XRd Control Plane",
    "capabilities": "R",
    "management_ip": "172.17.1.19",
    "management_ipv6": "2001:db8:16:1::4",
    "mac_address": "aa:c1:ab:5a:03:f9"
},
```

For this particular neighbor, the **configuration** variable will ultimately look like this:



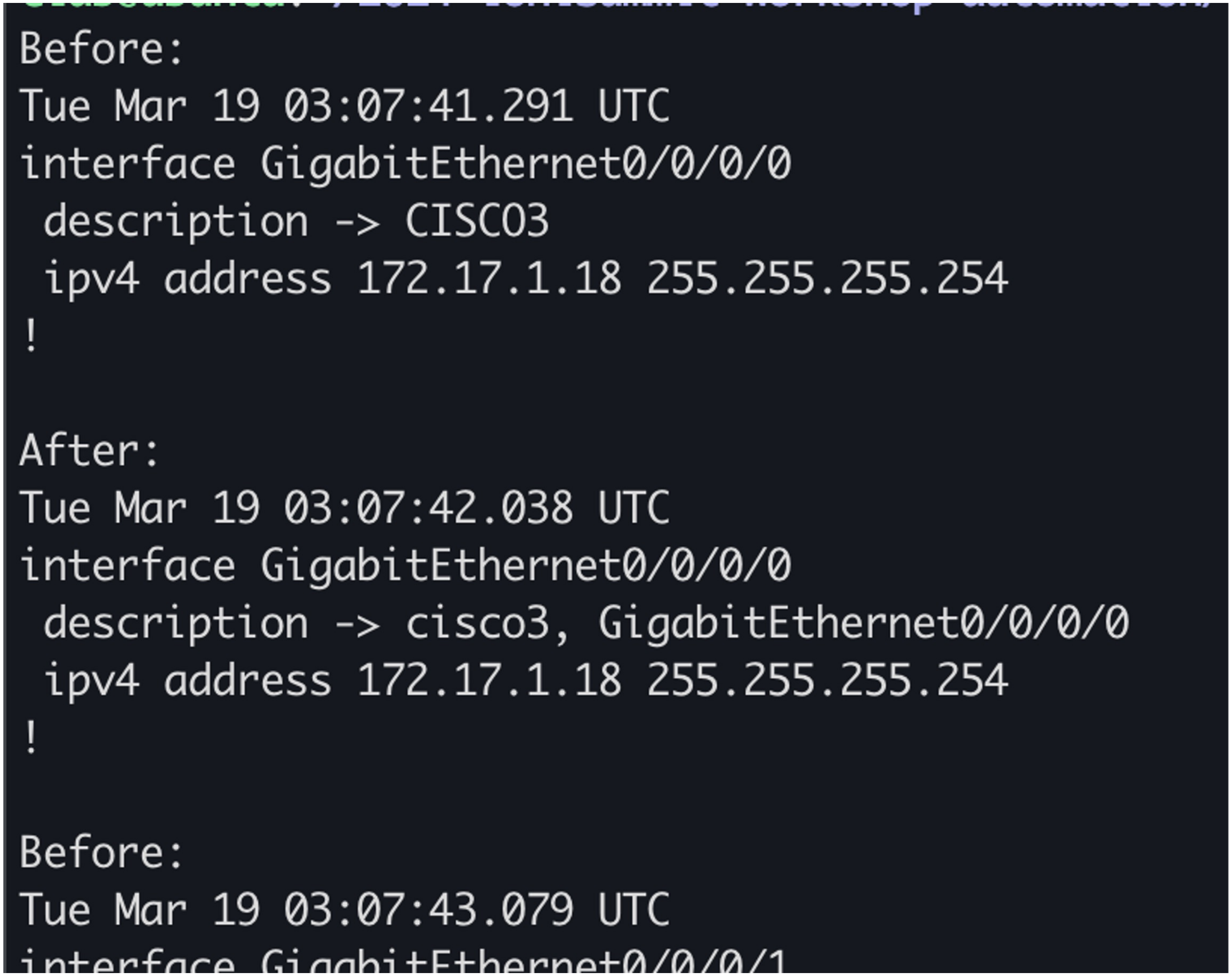
```
configuration = [
    f"int GigabitEthernet0/0/0/0",
    f"description -> cisco3, GigabitEthernet0/0/0/0",
    "commit"
]
```

Our Third Script

Same as the other two exercises, update the **`hosts`** variable at the top with your lab IPs.

Then, run the (really long named) script with:

```
poetry run python 3_netmiko_textfsm_update_description_to_lldp_neighbors.py
```



```
Before:
Tue Mar 19 03:07:41.291 UTC
interface GigabitEthernet0/0/0/0
 description -> CISCO3
 ipv4 address 172.17.1.18 255.255.255.254
!

After:
Tue Mar 19 03:07:42.038 UTC
interface GigabitEthernet0/0/0/0
 description -> cisco3, GigabitEthernet0/0/0/0
 ipv4 address 172.17.1.18 255.255.255.254
!

Before:
Tue Mar 19 03:07:43.079 UTC
interface GigabitEthernet0/0/0/1
```

So far, we've learned what it looks like to:

1. **[Lab 1, Part 1]** Read raw operational data, like LLDP neighborship, with Netmiko
2. **[Lab 1, Part 2]** Read operational data as **structured data**, such as JSON, using Netmiko and TextFSM
3. **[Lab 2, Part 1]** Commit arbitrary configuration with Netmiko based off of LLDP neighbor data we found.

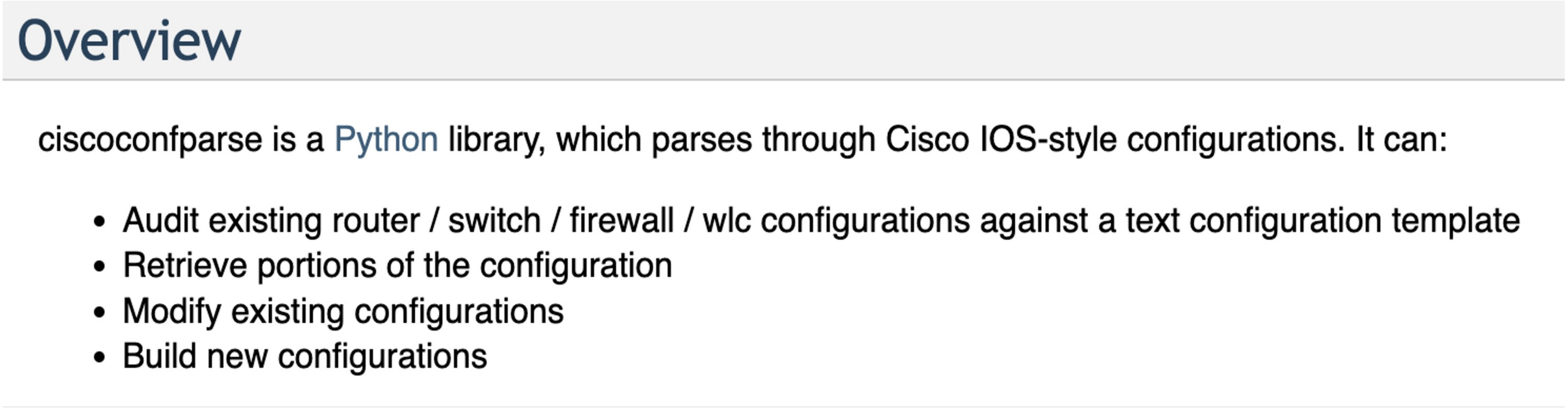
Finally, in [Lab 2, Part 2], we will learn to interpret interface configuration with a **new tool** (CiscoConfParse) and make changes based on logic.

Scenario: A campus DHCP server was re-IP'd and all IP helpers must be updated.

New Tool: What is CiscoConfParse?

Don’t let the name fool you – this works with more than just Cisco IOS (although we will only use it with our Cisco routers in the script.)

From the source:



Overview

ciscoconfparse is a Python library, which parses through Cisco IOS-style configurations. It can:

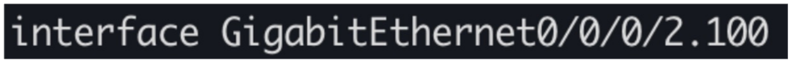
- Audit existing router / switch / firewall / wlc configurations against a text configuration template
- Retrieve portions of the configuration
- Modify existing configurations
- Build new configurations

Picture source: http://www.pennington.net/py/ciscoconfparse/intro.html#what-is-ciscoconfparse-good-for

New Tool: What is CiscoConfParse?

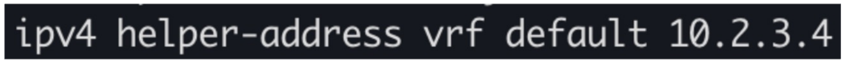
One of the primary focuses of CiscoConfParse is modelling configuration as **parents** and **children**.

This is useful, because we can scan for parents like:



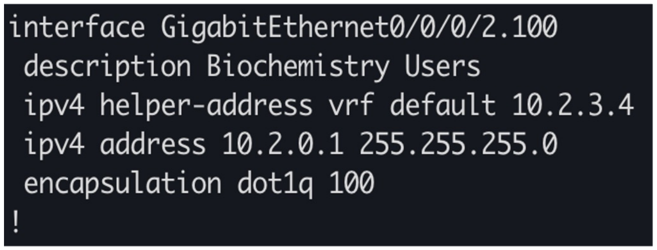
```
interface GigabitEthernet0/0/0/2.100
```

And look for children like:

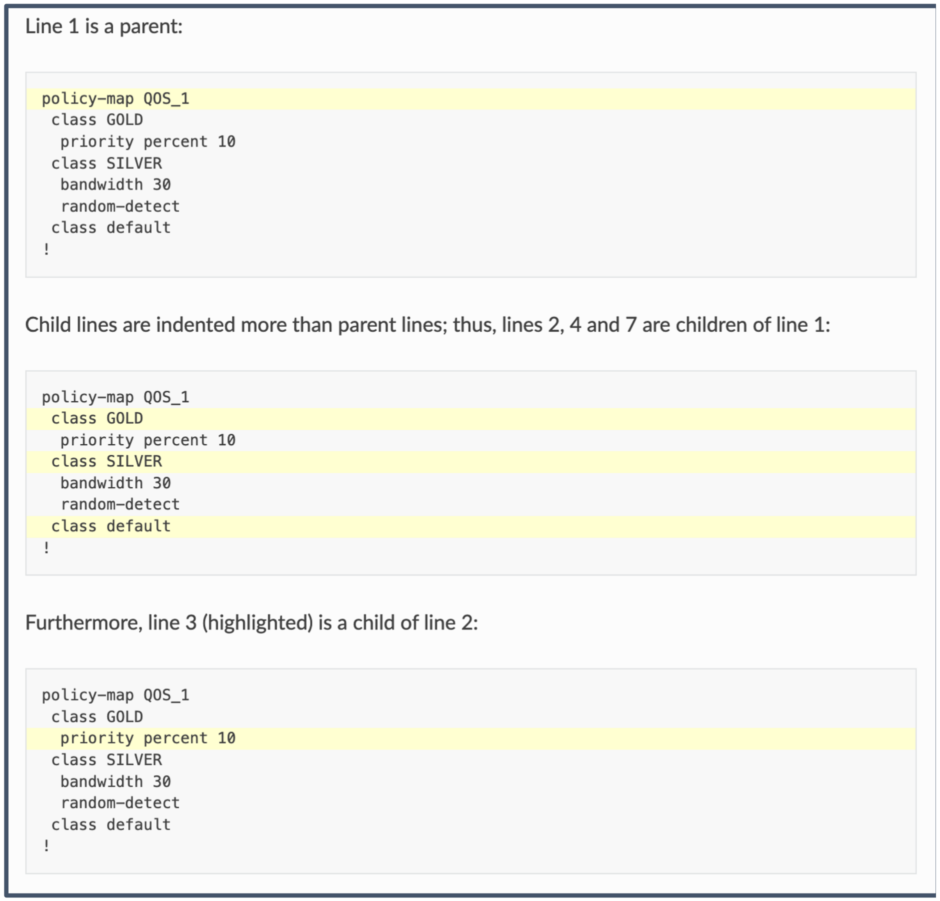


```
ipv4 helper-address vrf default 10.2.3.4
```

Ex:



```
interface GigabitEthernet0/0/0/2.100
 description Biochemistry Users
 ipv4 helper-address vrf default 10.2.3.4
 ipv4 address 10.2.0.1 255.255.255.0
 encapsulation dot1q 100
!
```



Line 1 is a parent:

```
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
```

Child lines are indented more than parent lines; thus, lines 2, 4 and 7 are children of line 1:

```
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
```

Furthermore, line 3 (highlighted) is a child of line 2:

```
policy-map QOS_1
 class GOLD
  priority percent 10
 class SILVER
  bandwidth 30
  random-detect
 class default
!
```

Lab 2, Part 2: A campus DHCP server was re-IP'd and all IP helpers must be updated.

Your IT department has a centralized DHCP server that is used in the majority of user VLANs on campus.

This DHCP server has moved and is now routed from a new subnet.

The old IP helper, **10.2.3.4**, now needs to be **192.168.100.100**.

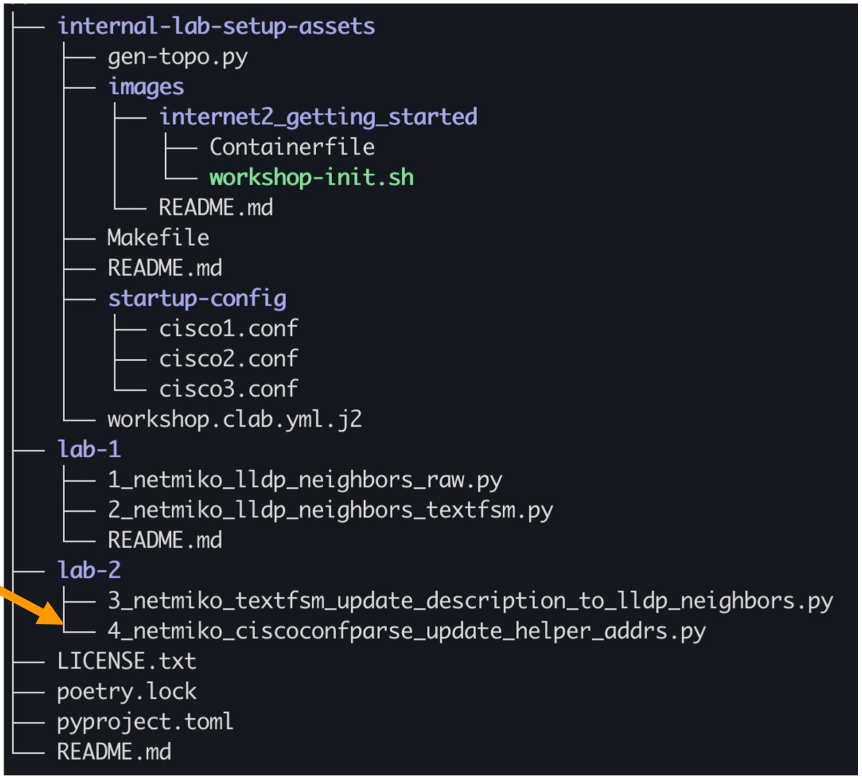
- Subnets that did *not* have the old IP helper should not have the new helper added.
- Other IP helpers must also be ignored.

Lab 2, Part 2: Our Fourth Script

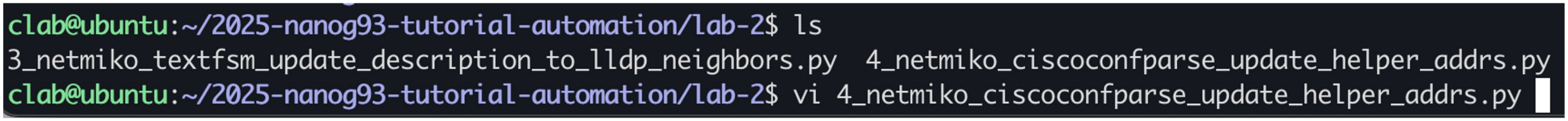
Analyzing and manipulating configuration

From the top folder, **`cd`** into the **`lab-2`** folder

Open with your favorite text editor:
4_netmiko_ciscoconfparse_update_helper_addrs.py



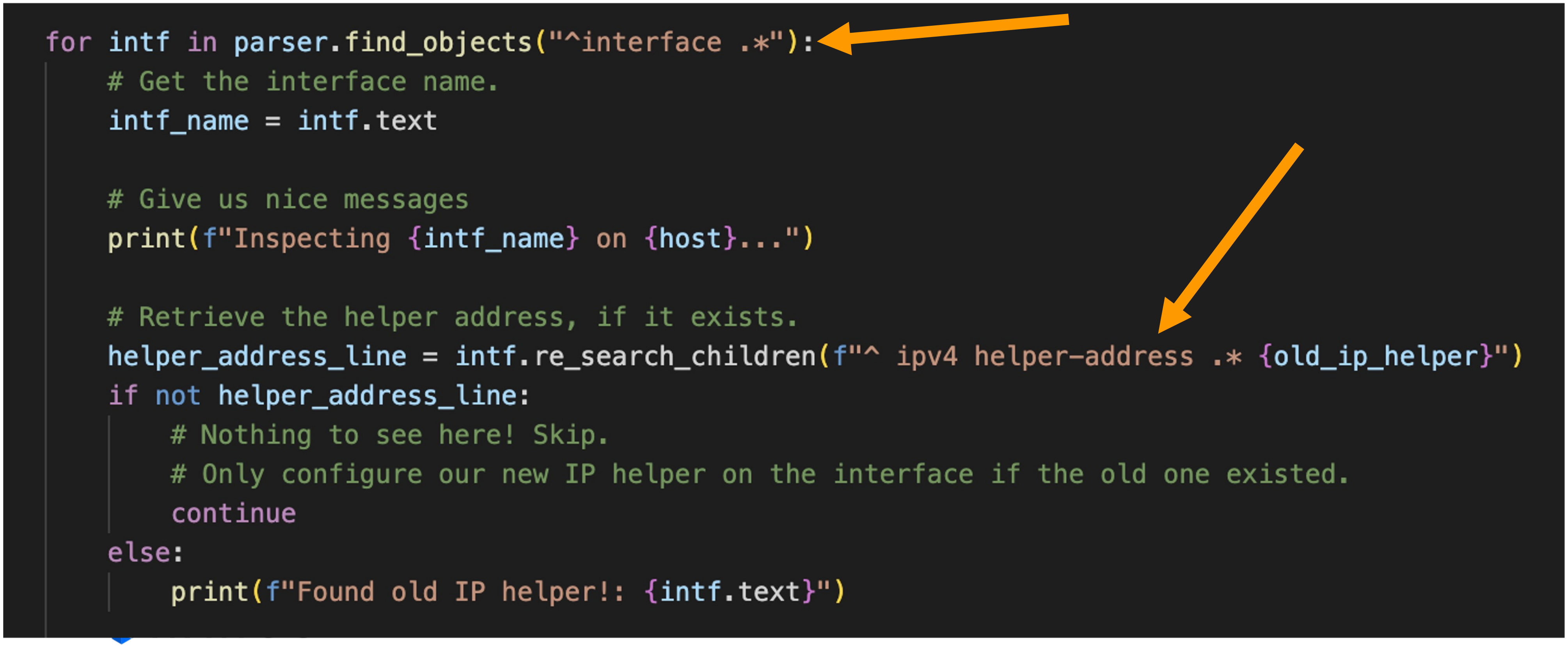
```
├── internal-lab-setup-assets
│   ├── gen-topo.py
│   ├── images
│   │   ├── internet2_getting_started
│   │   │   ├── Containerfile
│   │   │   └── workshop-init.sh
│   │   └── README.md
│   ├── Makefile
│   ├── README.md
│   ├── startup-config
│   │   ├── cisco1.conf
│   │   ├── cisco2.conf
│   │   └── cisco3.conf
│   └── workshop.clab.yml.j2
├── lab-1
│   ├── 1_netmiko_lldp_neighbors_raw.py
│   ├── 2_netmiko_lldp_neighbors_textfsm.py
│   └── README.md
├── lab-2
│   ├── 3_netmiko_textfsm_update_description_to_lldp_neighbors.py
│   └── 4_netmiko_ciscoconfparse_update_helper_addrs.py
├── LICENSE.txt
├── poetry.lock
├── pyproject.toml
└── README.md
```



```
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-2$ ls
3_netmiko_textfsm_update_description_to_lldp_neighbors.py  4_netmiko_ciscoconfparse_update_helper_addrs.py
clab@ubuntu:~/2025-nanog93-tutorial-automation/lab-2$ vi 4_netmiko_ciscoconfparse_update_helper_addrs.py
```

[Shannon does a scrolling walkthrough of the fourth script at this point.]

The Real Magic

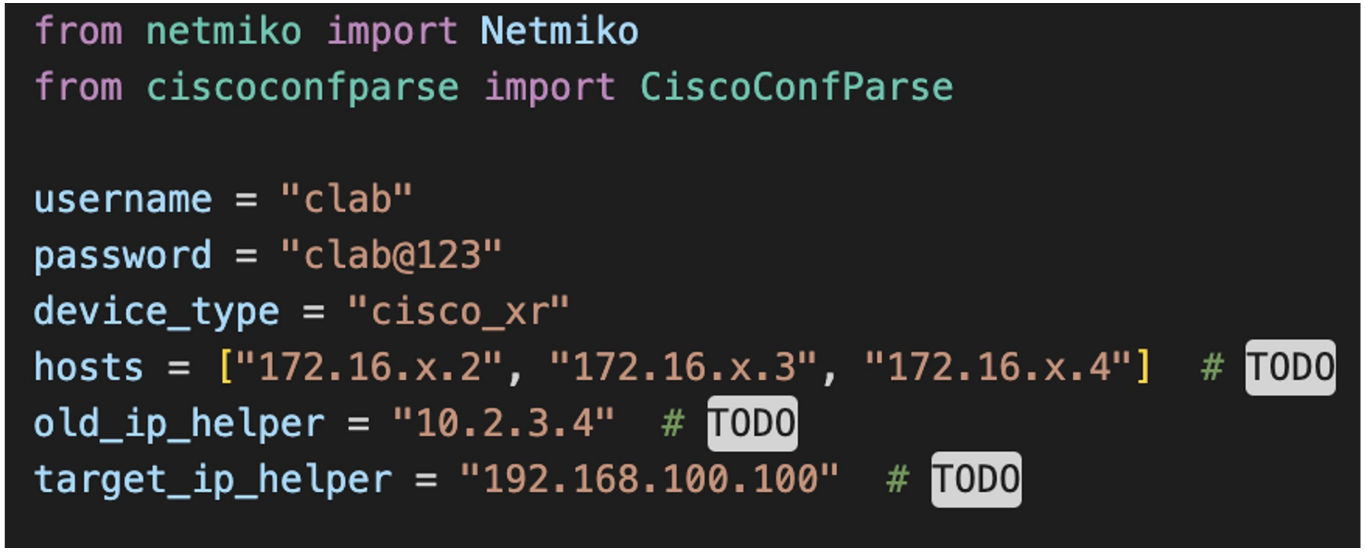


```python
for intf in parser.find_objects("^interface .*"):
    # Get the interface name.
    intf_name = intf.text

    # Give us nice messages
    print(f"Inspecting {intf_name} on {host}...")

    # Retrieve the helper address, if it exists.
    helper_address_line = intf.re_search_children(f"^ ipv4 helper-address .* {old_ip_helper}")
    if not helper_address_line:
        # Nothing to see here! Skip.
        # Only configure our new IP helper on the interface if the old one existed.
        continue
    else:
        print(f"Found old IP helper!: {intf.text}")
```

Lab 2, Part 2: Our Fourth Script

Analyzing and manipulating configuration

Before we go… you guessed it, update the information at the top of the script like so.



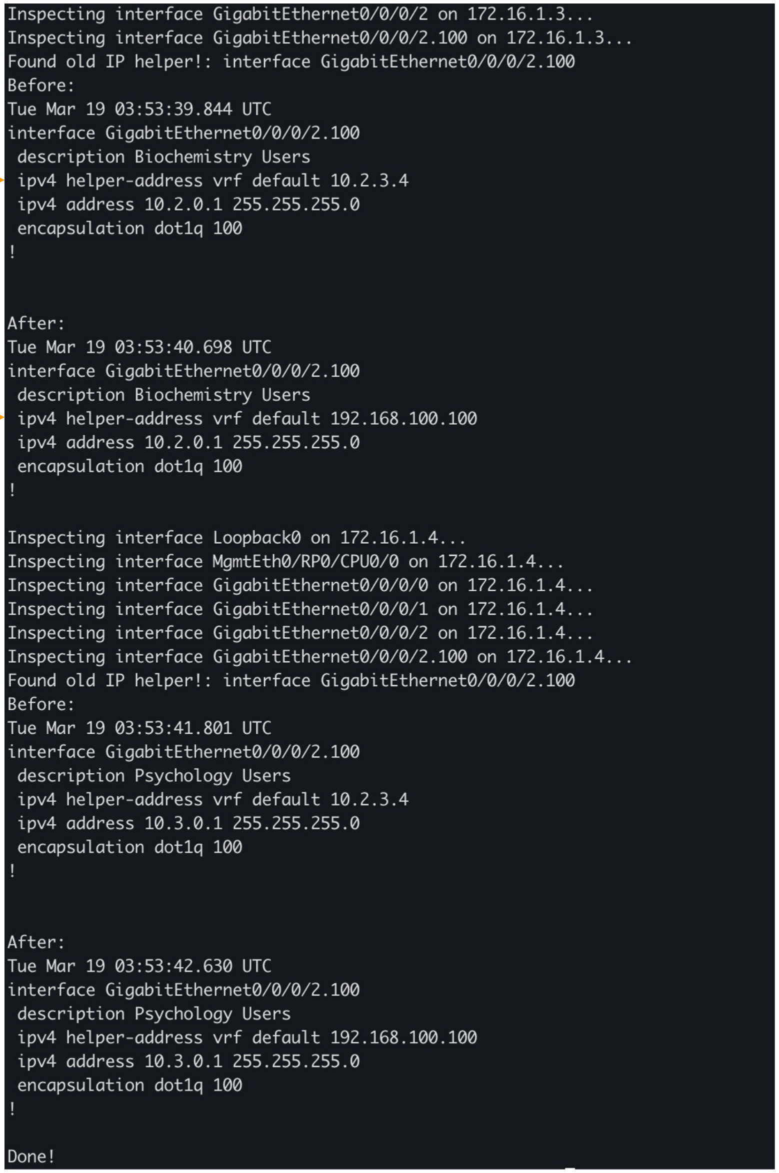
```python
from netmiko import Netmiko
from ciscoconfparse import CiscoConfParse

username = "clab"
password = "clab@123"
device_type = "cisco_xr"
hosts = ["172.16.x.2", "172.16.x.3", "172.16.x.4"]  # TODO
old_ip_helper = "10.2.3.4"  # TODO
target_ip_helper = "192.168.100.100"  # TODO
```

Lab 2, Part 2: Our Fourth Script

Analyzing and manipulating configuration

Then, run the script with:

```
poetry run python
4_netmiko_ciscoconfparse_update_helper_addrs.py
```



```
Inspecting interface GigabitEthernet0/0/0/2 on 172.16.1.3...
Inspecting interface GigabitEthernet0/0/0/2.100 on 172.16.1.3...
Found old IP helper!: interface GigabitEthernet0/0/0/2.100
Before:
Tue Mar 19 03:53:39.844 UTC
interface GigabitEthernet0/0/0/2.100
 description Biochemistry Users
 ipv4 helper-address vrf default 10.2.3.4
 ipv4 address 10.2.0.1 255.255.255.0
 encapsulation dot1q 100
!


After:
Tue Mar 19 03:53:40.698 UTC
interface GigabitEthernet0/0/0/2.100
 description Biochemistry Users
 ipv4 helper-address vrf default 192.168.100.100
 ipv4 address 10.2.0.1 255.255.255.0
 encapsulation dot1q 100
!

Inspecting interface Loopback0 on 172.16.1.4...
Inspecting interface MgmtEth0/RP0/CPU0/0 on 172.16.1.4...
Inspecting interface GigabitEthernet0/0/0/0 on 172.16.1.4...
Inspecting interface GigabitEthernet0/0/0/1 on 172.16.1.4...
Inspecting interface GigabitEthernet0/0/0/2 on 172.16.1.4...
Inspecting interface GigabitEthernet0/0/0/2.100 on 172.16.1.4...
Found old IP helper!: interface GigabitEthernet0/0/0/2.100
Before:
Tue Mar 19 03:53:41.801 UTC
interface GigabitEthernet0/0/0/2.100
 description Psychology Users
 ipv4 helper-address vrf default 10.2.3.4
 ipv4 address 10.3.0.1 255.255.255.0
 encapsulation dot1q 100
!


After:
Tue Mar 19 03:53:42.630 UTC
interface GigabitEthernet0/0/0/2.100
 description Psychology Users
 ipv4 helper-address vrf default 192.168.100.100
 ipv4 address 10.3.0.1 255.255.255.0
 encapsulation dot1q 100
!

Done!
```

Wrap Up

04-FEB-2025

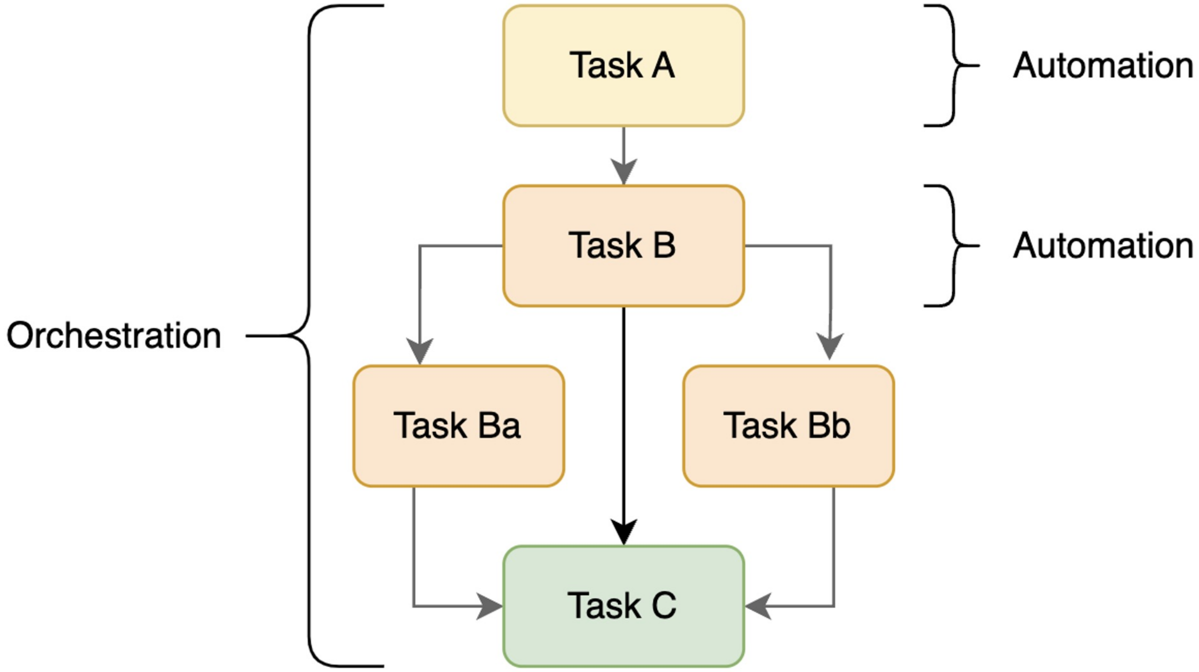
NANOG™

What is Network Automation?

Network Automation refers tc the automation of a singular task or small number of closely related tasks.

Orchestration refers to the transformation of automated tasks into a larger workflow.

All of our examples today fall into **Network Automation**.

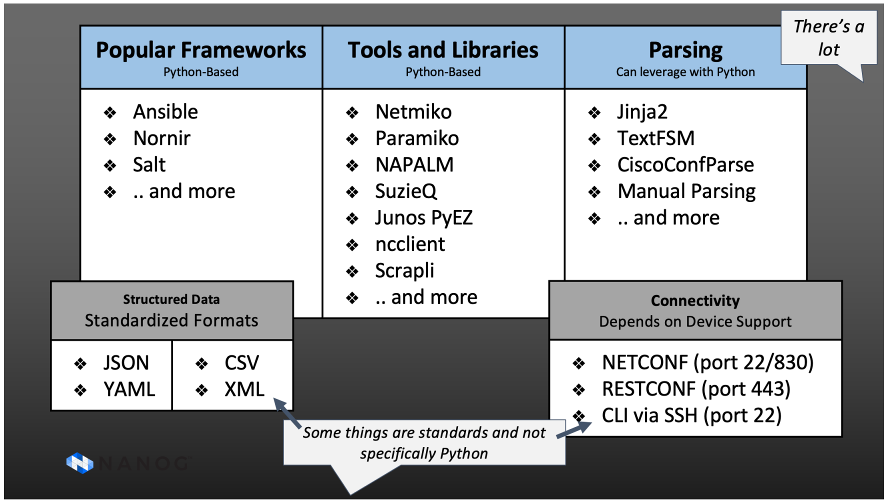


Tooling

We focused on tools that perform, or help with, **CLI Scraping**.

- **Netmiko** for reading and writing to a device.
- **TextFSM** for formatting and standardizing raw output from devices.
- **CiscoConfParse** for analyzing raw Cisco-like configuration and understanding the relationships between different lines.

Remember this?



Popular Frameworks
Python-Based

- Ansible
- Nornir
- Salt
- .. and more

Tools and Libraries
Python-Based

- Netmiko
- Paramiko
- NAPALM
- SuzieQ
- Junos PyEZ
- ncclient
- Scrapli
- .. and more

Parsing
Can leverage with Python

- Jinja2
- TextFSM
- CiscoConfParse
- Manual Parsing
- .. and more

There's a lot

Structured Data
Standardized Formats

- JSON
- YAML
- CSV
- XML

Connectivity
Depends on Device Support

- NETCONF (port 22/830)
- RESTCONF (port 443)
- CLI via SSH (port 22)

Some things are standards and not specifically Python

Ideas to take away

- **Start with one thing, and automate it.**
    - Ideally, make it the most boring, tedious task you have, especially if you have nervous engineers who need encouragement to embrace automation.

- **Yes, your network is special.**
    - Automate for the 70% that isn't special.
    - Automate for the new infrastructure you bring into your network over time.
    - Use automation to standardize your network.

- Brownfield networks do not fit well into trainings, courses and workshops.
    - Instead, **leverage the community around you to share knowledge, strategies, and solutions** to everything that isn't cookie cutter.

Bonus:

You can use scraped, non-operational data to help with an initial population of your Source of Intent!

(Ask me about Source of Intent population)
(It's a can of worms)

* I resisted a meme this long.

Thank you!

sbyrnes@internet2.edu