University of Colorado Boulder

Network Management and Automation

DevOps: Jinja2 & netcopa

Levi Perigo, Ph.D.
University of Colorado Boulder
Department of Computer Science
Network Engineering

Review

- **Automation**
- **Orchestration**
- **Virtualization**
- **NMAS**
- **NSOT**

DevOps Tools

- **Python-YAML-Ansible-Jinja2**

- **NSOT**
- **NMAS**
- **IAC**
- **CI/CD**

Jinja2

- Template engine - A general purpose templating language
  - – When and why do we use templates?

- Library for Python – designed to be user-friendly

- Works with Python 2.6.x, 2.7.x and >= 3.3

- Install -> pip install Jinja2

Features

- Sandboxed execution
- AutoEscaping
  - *Improves security by blocking unescaped strings*
- Template inheritance
- Easy to debug and configurable syntax

- Licensed under a BSD License
- Created by Armin Ronacher (creator of the Flask web framework for Python)
- Flask comes packaged with Jinja
- Modelled after Django’s templates
- By convention, they live in the /templates directory

Templates

- A Jinja template is simply a text file
  - *Can generate any text-based format: HTML, XML, CSV, LaTeX, etc.*

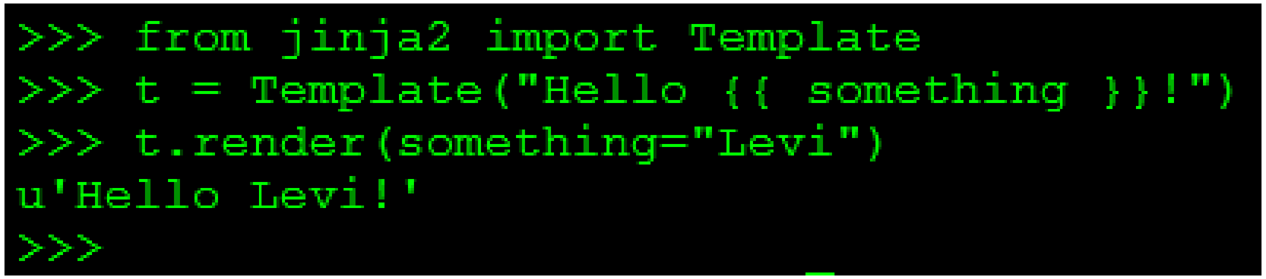
- No specific extension
  - *.j2, .txt, .html, .xml or any other is fine*

- Contains variables and/or expressions
  - *Replaced by values passed when the templates are **rendered***

- **CLI Templates**
  - Set of re-usable device configuration commands
  - Ability to parameterize select elements (variables) of config.
  - Control logic statements
  - What can you do with these templates?
    - ***IAC & NSOT***

Examples

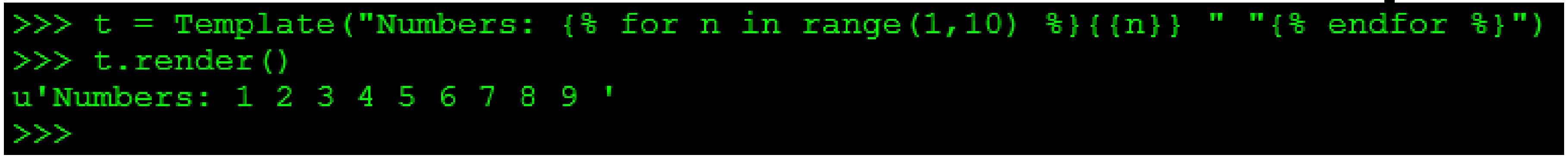
Replacing variable by value passed in Braces {}-



```
>>> from jinja2 import Template
>>> t = Template("Hello {{ something }}!")
>>> t.render(something="Levi")
u'Hello Levi!'
>>>
```

Loop expression-

To close the ‘for’ block

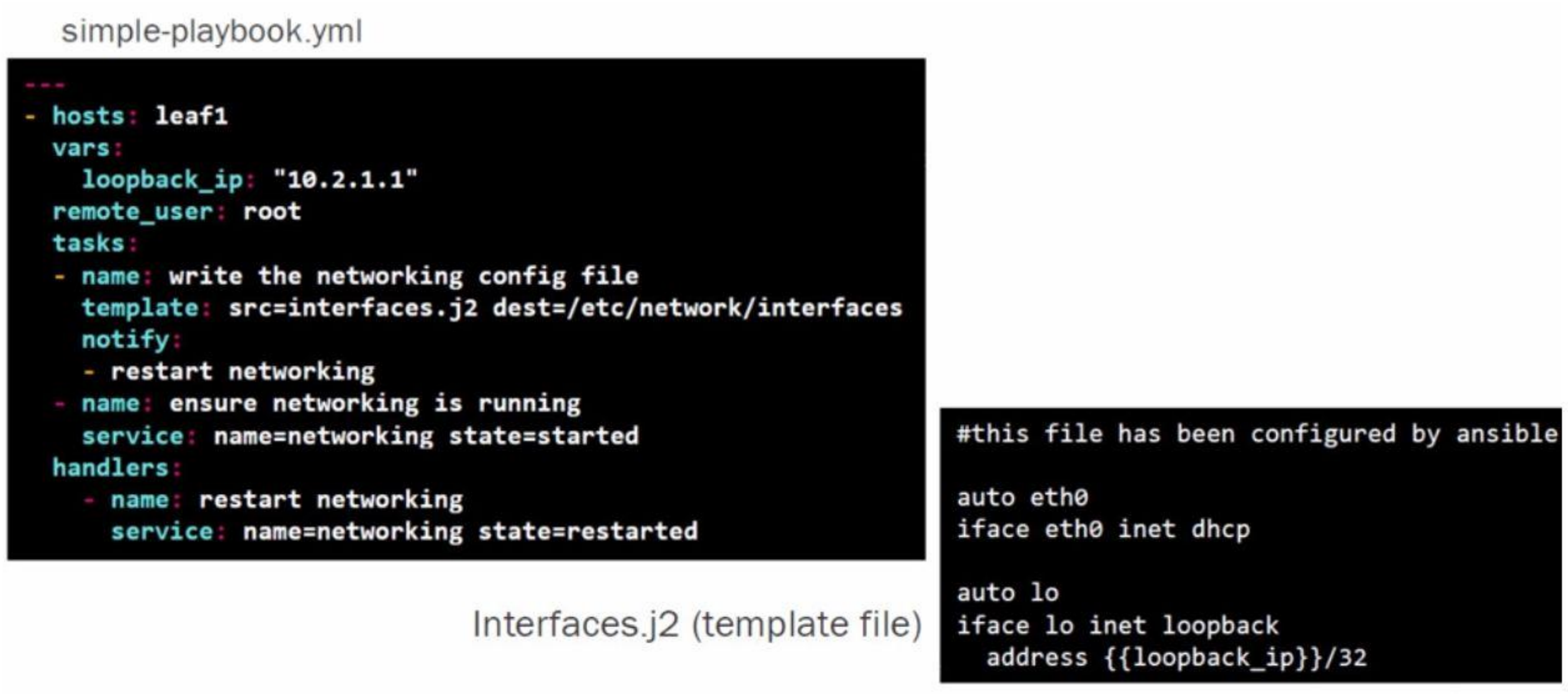


```
>>> t = Template("Numbers: {% for n in range(1,10) %}{{n}} " "{% endfor %}")
>>> t.render()
u'Numbers: 1 2 3 4 5 6 7 8 9 '
>>>
```

University of Colorado
Boulder

Playbook – Linux Example (static)

- **YAML**
  - Spacing is specific (like Python)
    - ***Spacing vs tabs vs indentation etc.***
- **To run Playbook**
  - ansible-playbook simple-playbook.yml



simple-playbook.yml

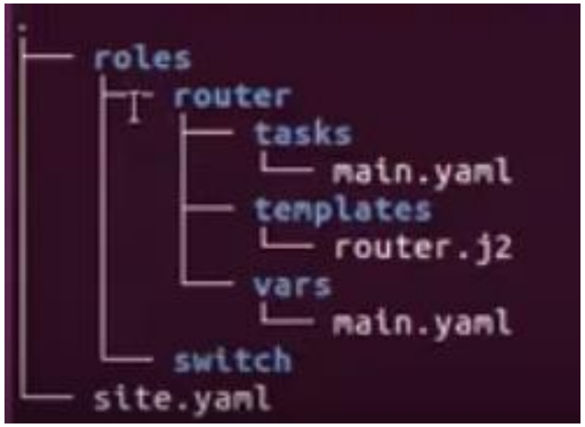
```yaml
---
- hosts: leaf1
  vars:
    loopback_ip: "10.2.1.1"
  remote_user: root
  tasks:
  - name: write the networking config file
    template: src=interfaces.j2 dest=/etc/network/interfaces
    notify:
    - restart networking
  - name: ensure networking is running
    service: name=networking state=started
  handlers:
    - name: restart networking
      service: name=networking state=restarted
```

Interfaces.j2 (template file)

```
#this file has been configured by ansible

auto eth0
iface eth0 inet dhcp

auto lo
iface lo inet loopback
  address {{loopback_ip}}/32
```

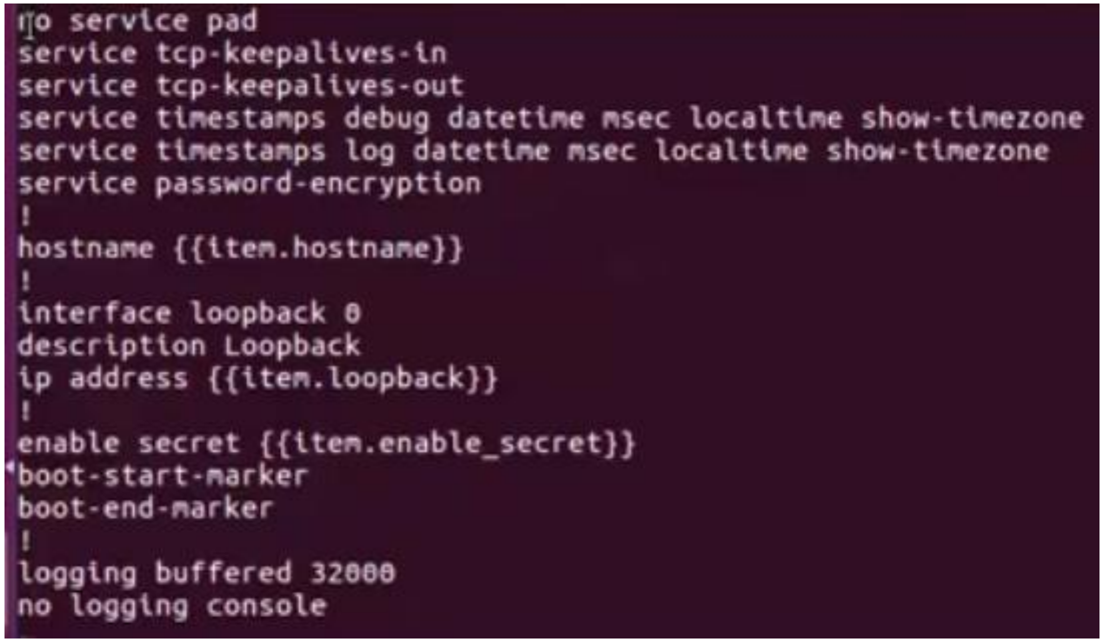
Example 2

Tree



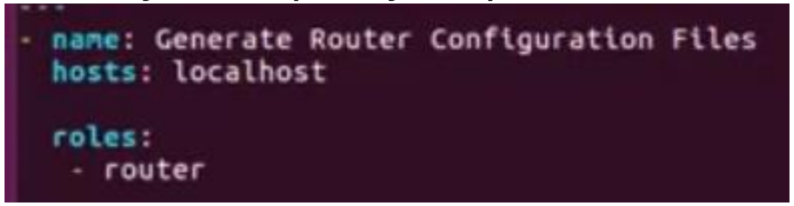
```
.
├── roles
│   ├─┬ router
│   │ ├── tasks
│   │ │   └── main.yaml
│   │ ├── templates
│   │ │   └── router.j2
│   │ └── vars
│   │     └── main.yaml
│   └── switch
└── site.yaml
```

3. Templates (router.j2)



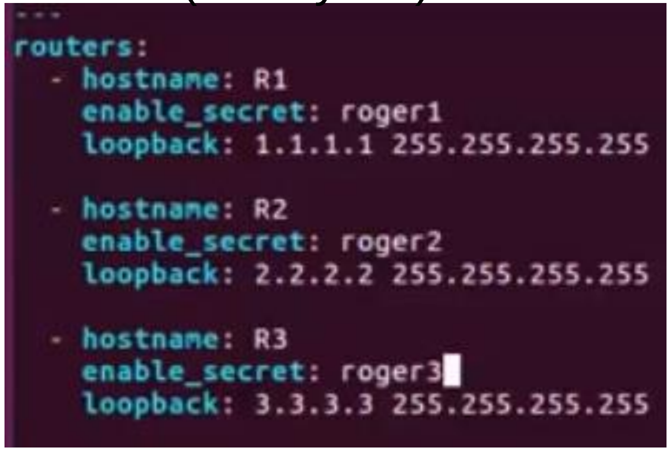
```
no service pad
service tcp-keepalives-in
service tcp-keepalives-out
service timestamps debug datetime msec localtime show-timezone
service timestamps log datetime msec localtime show-timezone
service password-encryption
!
hostname {{item.hostname}}
!
interface loopback 0
description Loopback
ip address {{item.loopback}}
!
enable secret {{item.enable_secret}}
boot-start-marker
boot-end-marker
!
logging buffered 32000
no logging console
```

1. Playbook (site.yaml)



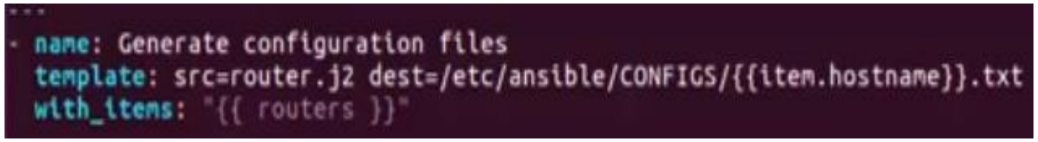
```
---
- name: Generate Router Configuration Files
  hosts: localhost

  roles:
   - router
```

4. vars (main.yaml)



```
---
routers:
  - hostname: R1
    enable_secret: roger1
    loopback: 1.1.1.1 255.255.255.255

  - hostname: R2
    enable_secret: roger2
    loopback: 2.2.2.2 255.255.255.255

  - hostname: R3
    enable_secret: roger3
    loopback: 3.3.3.3 255.255.255.255
```

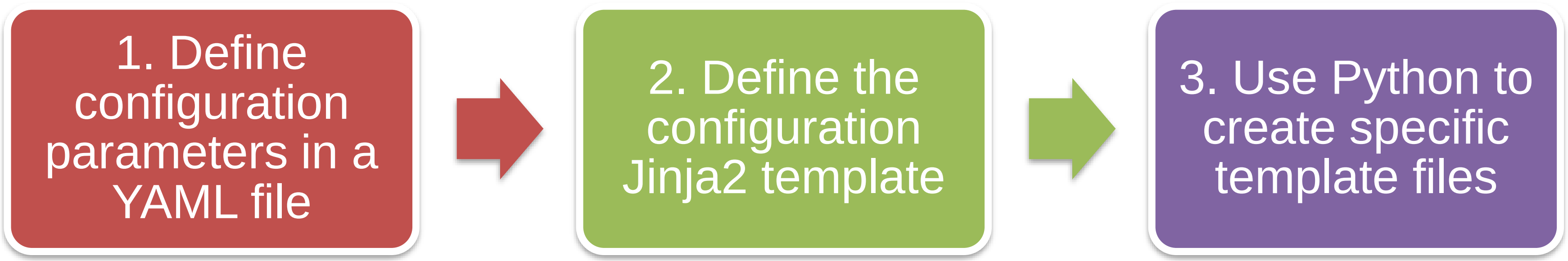
2. Tasks – build cfg from vars and temp (main.yaml)



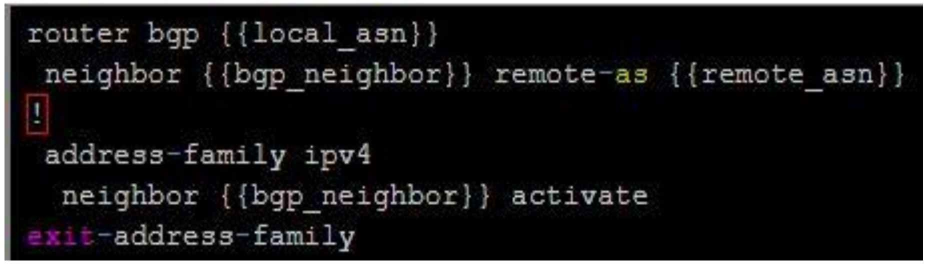
```
---
- name: Generate configuration files
  template: src=router.j2 dest=/etc/ansible/CONFIGS/{{item.hostname}}.txt
  with_items: "{{ routers }}"
```

5. Push config. To device

Jinja2 with YAML & Python


1. Define configuration parameters in a YAML file
2. Define the configuration Jinja2 template
3. Use Python to create specific template files

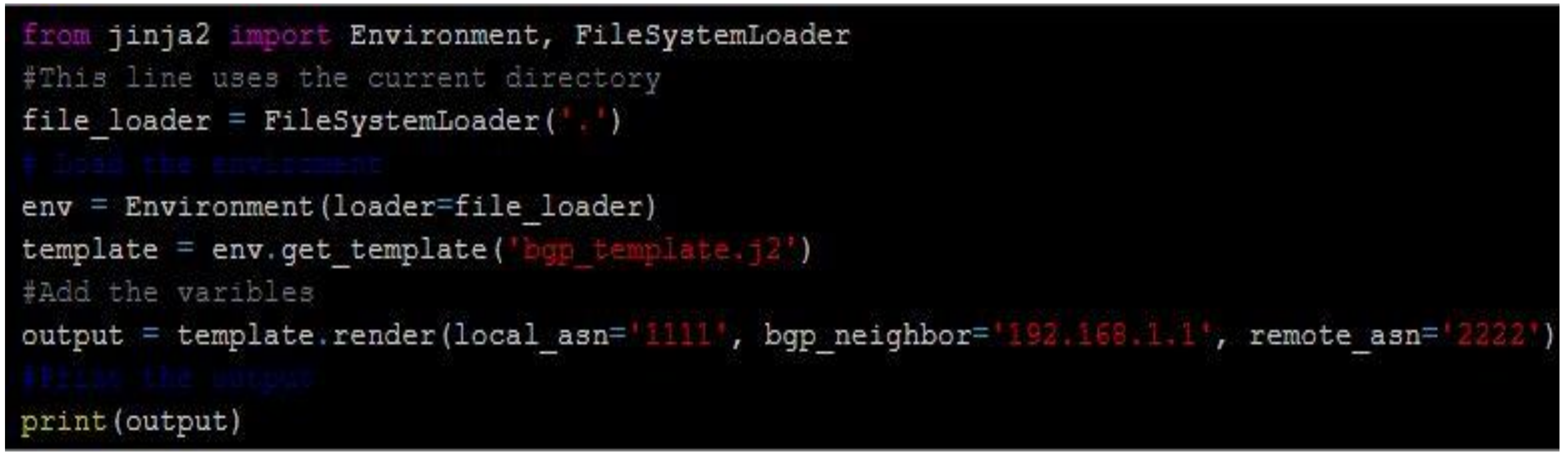
Use Case 1 (Jinja2 > Py > CFG (“static”))



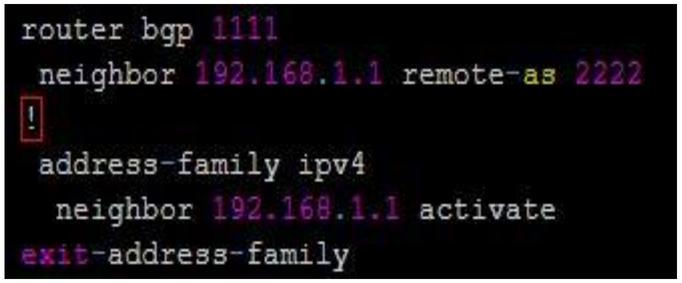
```
router bgp {{local_asn}}
 neighbor {{bgp_neighbor}} remote-as {{remote_asn}}
!
 address-family ipv4
  neighbor {{bgp_neighbor}} activate
exit-address-family
```

bgp_template.j2

bgp_generate_cfg.py



```python
from jinja2 import Environment, FileSystemLoader
#This line uses the current directory
file_loader = FileSystemLoader('.')
# Load the environment
env = Environment(loader=file_loader)
template = env.get_template('bgp_template.j2')
#Add the varibles
output = template.render(local_asn='1111', bgp_neighbor='192.168.1.1', remote_asn='2222')
#Print the output
print(output)
```

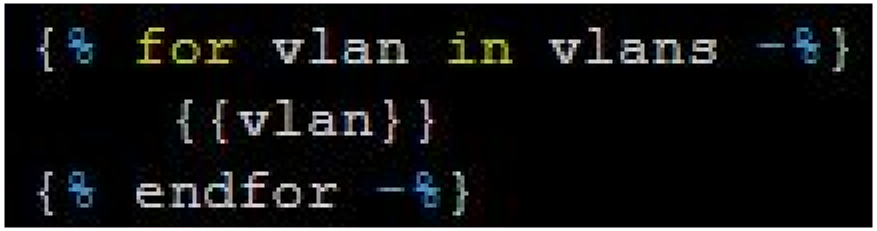


```
router bgp 1111
 neighbor 192.168.1.1 remote-as 2222
!
 address-family ipv4
  neighbor 192.168.1.1 activate
exit-address-family
```

bgp_cfg.txt

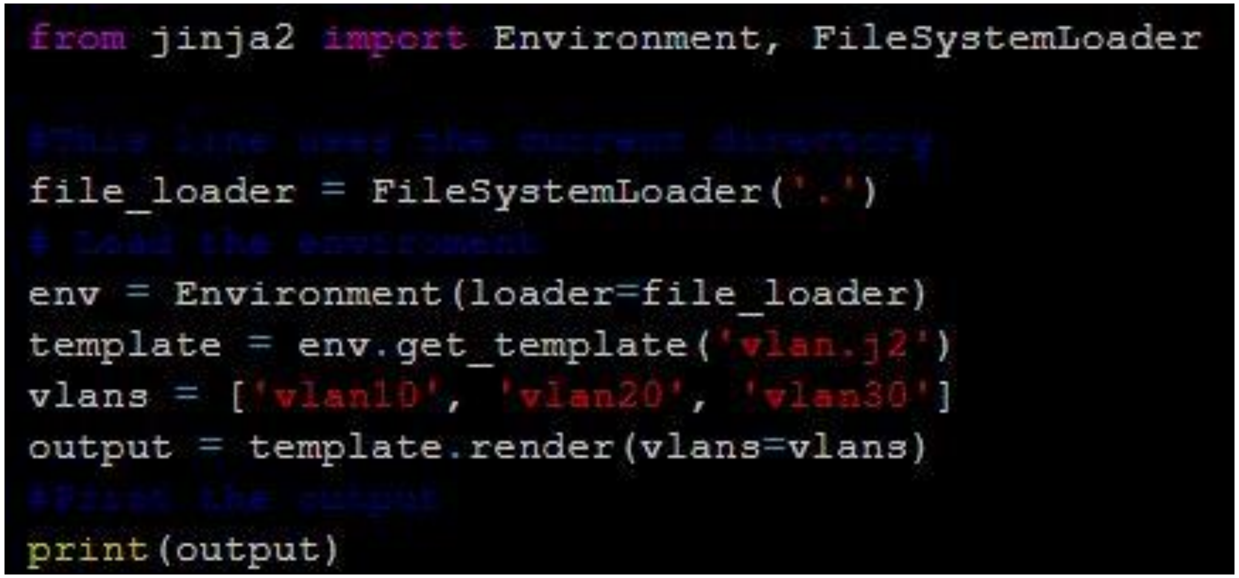
University of Colorado
Boulder

Use Case 2 – Loops (“static”)



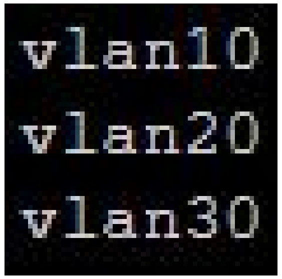
```
{% for vlan in vlans -%}
    {{vlan}}
{% endfor -%}
```

- vlan.j2
  - the ‘-’ before the “%” removes lines before/after



```
from jinja2 import Environment, FileSystemLoader

#This line uses the current directory
file_loader = FileSystemLoader('.')
# Load the environment
env = Environment(loader=file_loader)
template = env.get_template('vlan.j2')
vlans = ['vlan10', 'vlan20', 'vlan30']
output = template.render(vlans=vlans)
#Print the output
print(output)
```

- vlan_loop.py

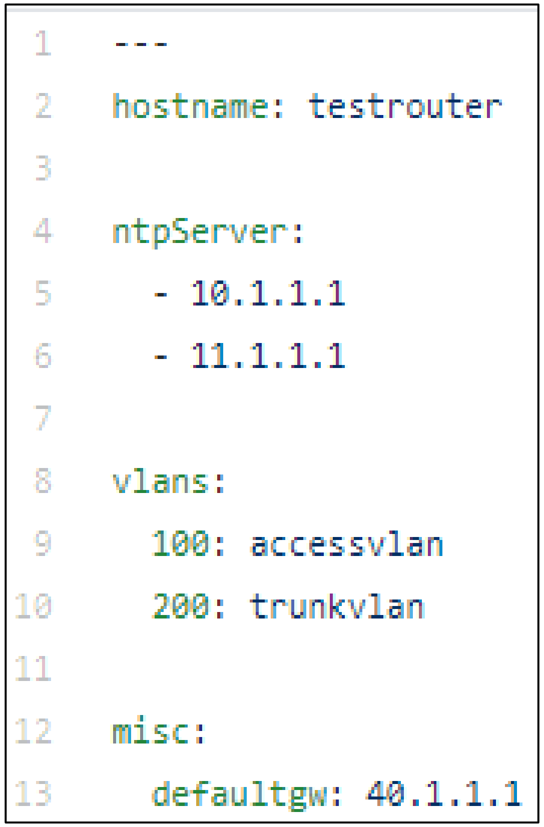


```
vlan10
vlan20
vlan30
```

- vlan_loop_cfg.txt

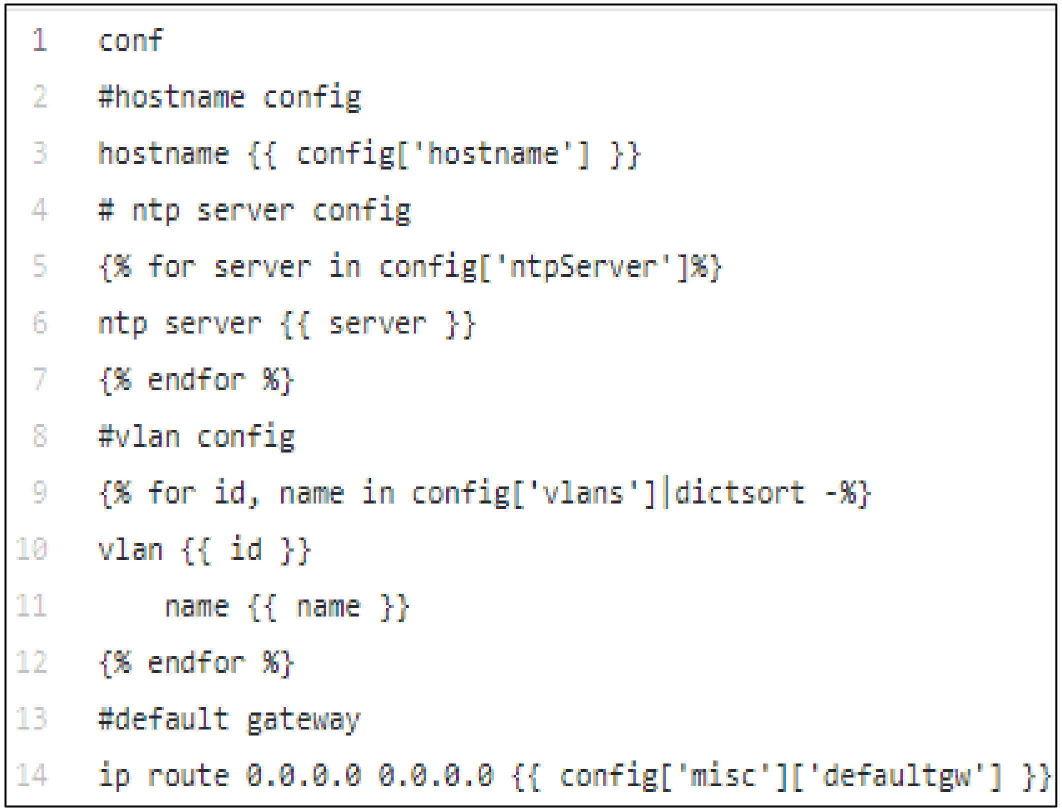
Use Case 3 – Putting it all together

1. baseline_data.yaml

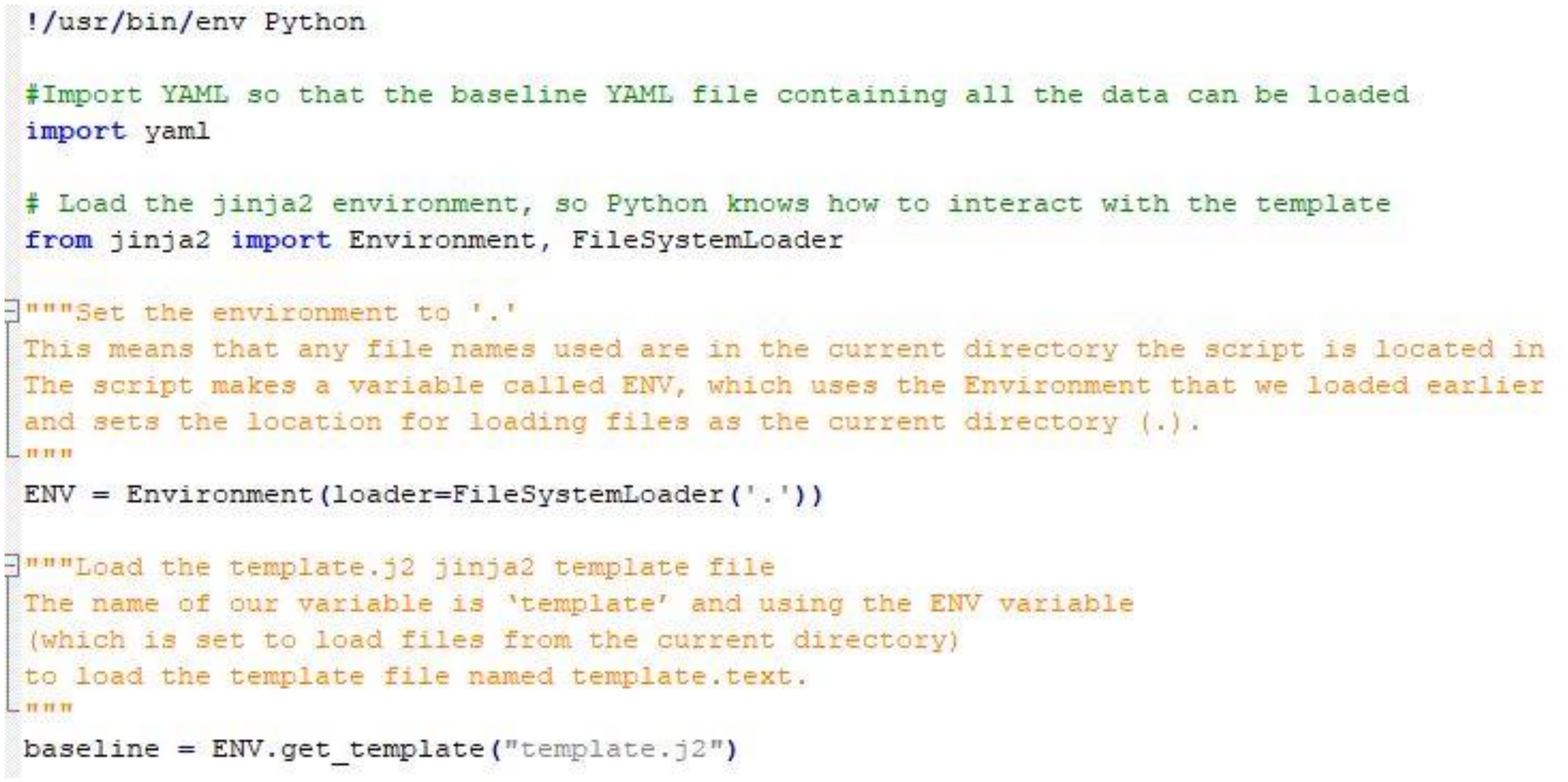


```yaml
---
hostname: testrouter

ntpServer:
  - 10.1.1.1
  - 11.1.1.1

vlans:
  100: accessvlan
  200: trunkvlan

misc:
  defaultgw: 40.1.1.1
```

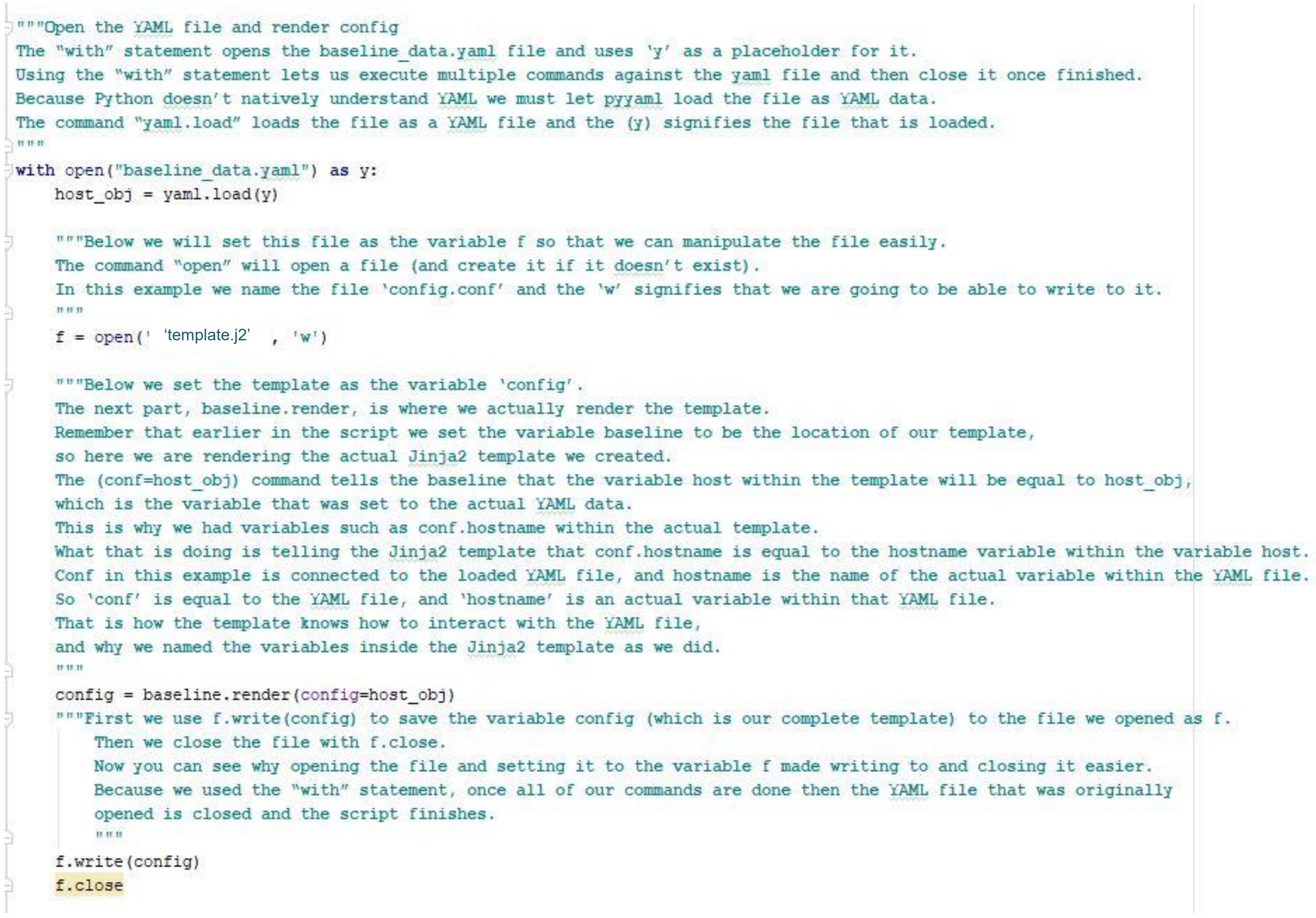
2. template.j2



```
conf
#hostname config
hostname {{ config['hostname'] }}
# ntp server config
{% for server in config['ntpServer']%}
ntp server {{ server }}
{% endfor %}
#vlan config
{% for id, name in config['vlans']|dictsort -%}
vlan {{ id }}
    name {{ name }}
{% endfor %}
#default gateway
ip route 0.0.0.0 0.0.0.0 {{ config['misc']['defaultgw'] }}
```



```
!/usr/bin/env Python

#Import YAML so that the baseline YAML file containing all the data can be loaded
import yaml

# Load the jinja2 environment, so Python knows how to interact with the template
from jinja2 import Environment, FileSystemLoader

"""Set the environment to '.'
This means that any file names used are in the current directory the script is located in
The script makes a variable called ENV, which uses the Environment that we loaded earlier
and sets the location for loading files as the current directory (.).
"""
ENV = Environment(loader=FileSystemLoader('.'))

"""Load the template.j2 jinja2 template file
The name of our variable is ‘template’ and using the ENV variable
(which is set to load files from the current directory)
to load the template file named template.text.
"""
baseline = ENV.get_template("template.j2")
```



```
"""Open the YAML file and render config
The "with" statement opens the baseline_data.yaml file and uses 'y' as a placeholder for it.
Using the "with" statement lets us execute multiple commands against the yaml file and then close it once finished.
Because Python doesn't natively understand YAML we must let pyyaml load the file as YAML data.
The command "yaml.load" loads the file as a YAML file and the (y) signifies the file that is loaded.
"""
with open("baseline_data.yaml") as y:
    host_obj = yaml.load(y)

    """Below we will set this file as the variable f so that we can manipulate the file easily.
    The command "open" will open a file (and create it if it doesn't exist).
    In this example we name the file 'config.conf' and the 'w' signifies that we are going to be able to write to it.
    """
    f = open(' 'template.j2' , 'w')

    """Below we set the template as the variable 'config'.
    The next part, baseline.render, is where we actually render the template.
    Remember that earlier in the script we set the variable baseline to be the location of our template,
    so here we are rendering the actual Jinja2 template we created.
    The (conf=host_obj) command tells the baseline that the variable host within the template will be equal to host_obj,
    which is the variable that was set to the actual YAML data.
    This is why we had variables such as conf.hostname within the actual template.
    What that is doing is telling the Jinja2 template that conf.hostname is equal to the hostname variable within the variable host.
    Conf in this example is connected to the loaded YAML file, and hostname is the name of the actual variable within the YAML file.
    So 'conf' is equal to the YAML file, and 'hostname' is an actual variable within that YAML file.
    That is how the template knows how to interact with the YAML file,
    and why we named the variables inside the Jinja2 template as we did.
    """
    config = baseline.render(config=host_obj)
    """First we use f.write(config) to save the variable config (which is our complete template) to the file we opened as f.
        Then we close the file with f.close.
        Now you can see why opening the file and setting it to the variable f made writing to and closing it easier.
        Because we used the "with" statement, once all of our commands are done then the YAML file that was originally
        opened is closed and the script finishes.
        """
    f.write(config)
    f.close
```

netcopa (Network Configuration Parser)

- netcopa is an engine which implements a template-based state machine for parsing semi-formatted text and storing it as structured data in YAML

- https://github.com/cidrblock/netcopa

- Python 2.7+

Start with text- Finish with YAML-

```
!
interface GigabitEthernet1/3
 switchport access vlan 267
 switchport mode access
 switchport voice vlan 867
 spanning-tree portfast
 spanning-tree bpduguard enable
 service-policy input company-user-access-450x
 service-policy output company-user-access-dbl
!
```

```
interfaces:
  GigabitEthernet1/3:
    name: GigabitEthernet1/3
    service_policies:
    - direction: input
      name: company-user-access-450x
    - direction: output
      name: company-user-access-dbl
    spanning-tree:
      bpduguard: true
      portfast: true
    switchport:
      access:
        vlan: 267
      mode:
      - access
      voice:
        vlan: 867
```

YAML file can then be used with Ansible/J2 to configure the device(s).

University of Colorado
Boulder

Lab

- **Taking this to the next level!**
  - Manually - Create Jinja2 template
  - Python - Read Network Inventory (CSV)
    - ***Python - Create Ansible Playbook YAML file***
    - ***Ansible - Generate configuration file***
  - Python
    - ***Send configuration file to routers***

    - ***ZTP & IAC***
      - CSV > yaml > J2 > Ansible > cfg
        - » Python > network

Questions?