# Templates for Network Engineers

A day in the life of a Network Engineer using Jinja2 Templates

# Course Overview

**WS:B4 - A day in the life of a Network Engineer using Jinja2 Templates**

**Proctor:** Claudia de Luna, EIA

**Description:** Using templates to create network docs & configurations programmatically is key to any automation strategy; we will generate a wide range of artifacts using Jinja templates.

**Level:** Beginner, Intermediate

**Agenda:**

+ Everyone already has templates, Why are we here?
+ Background on templating languages
+ Revision Control
+ Template format
+ Jinja2 Fundamentals
+ Working Sessions

# Why are we here?

I already have templates!

*I already have a phone!*

## Word or Text

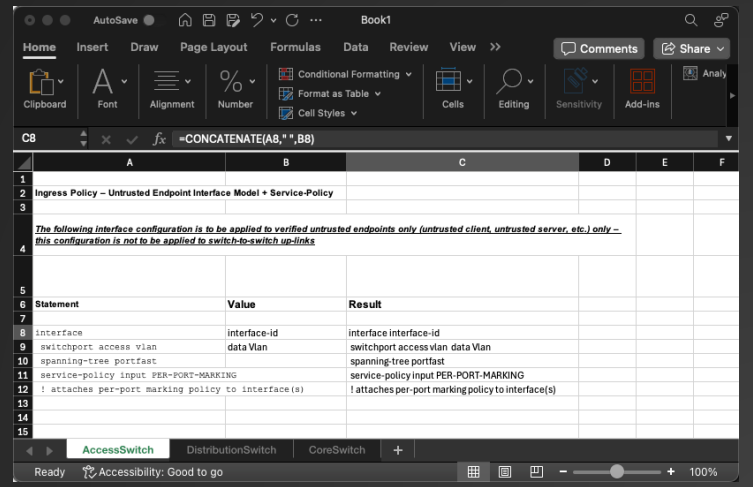
```
interface [interface-id]
 switchport access vlan [data VLAN]
 spanning-tree portfast
 service-policy input PER-PORT-MARKING
 ! attaches per-port marking policy to interface(s)
```

## PowerPoint

Ingress Policy – Untrusted Endpoint

```
interface [interface-id]
 switchport access vlan [data VLAN]
 spanning-tree portfast
 service-policy input PER-PORT-MARKING
 ! attaches per-port marking policy to interface(s)
```

## Excel

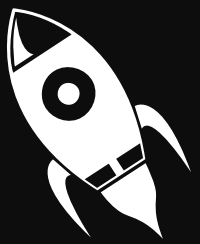



## The last device I configured

# A more precise question

Are your configuration templates

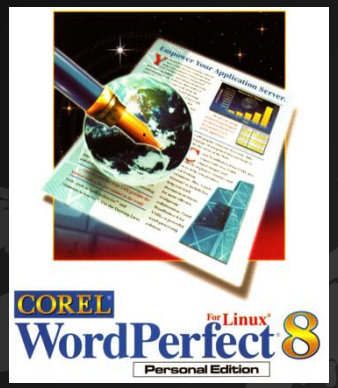
- Under forma revision control in a Revision Control System
- Current
- Maintained under a rigorous process
- Used without exception
- Programmatically consumable



# Templates and Templating Languages

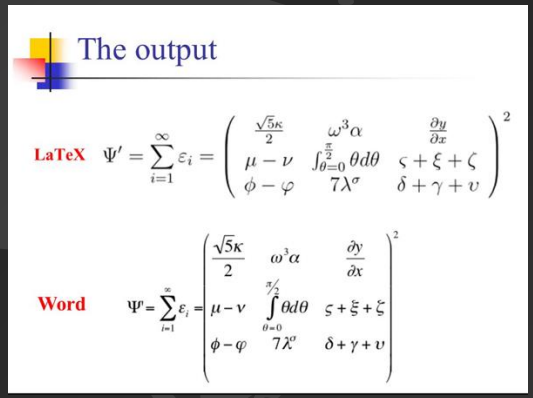
## Mail Merge!

Confess…You have done this!




## Typesetting

TeX/LaTeX for Documents




## Web Development

We are Here!

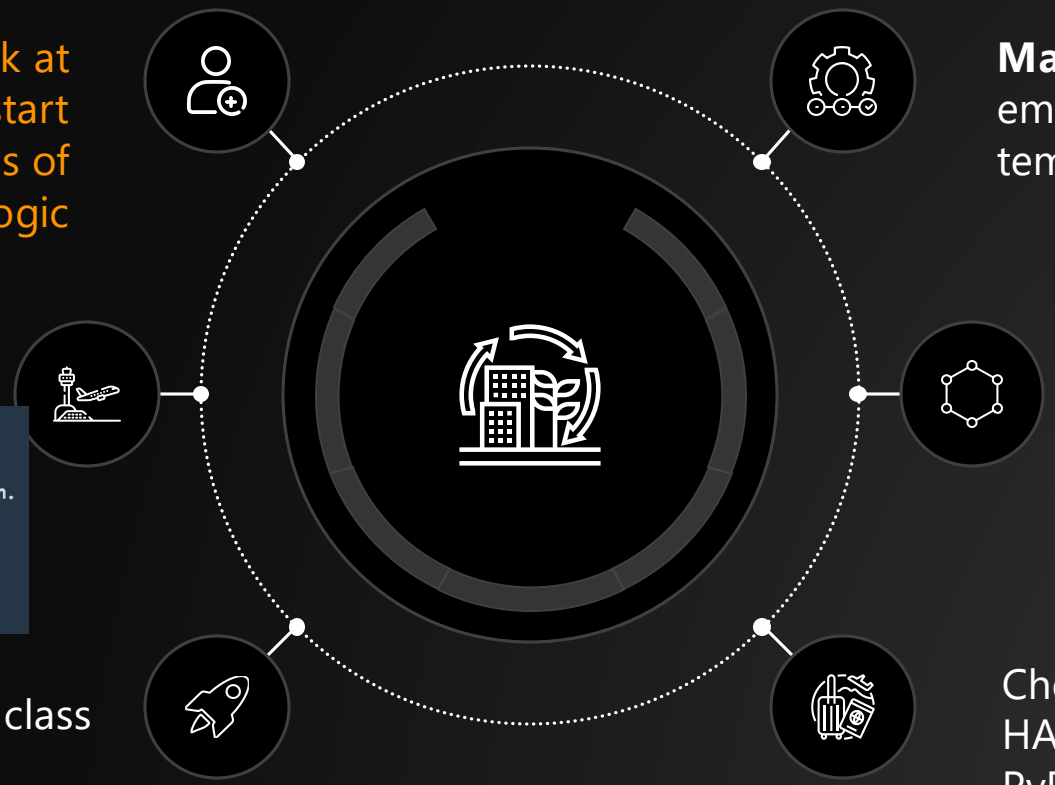
- Programmatically consumable Templates
- Focused on Text including Encoding and Escaping

# WEB DEVELOPMENT

## Templating Landscape (Python)

DE FACTO STANDARD

**Jinja2**- Start here and look at other options when you start to outgrow Jinja in terms of speed and/or complex logic



**Mako** - Superfast and lets you embed Python code in the template

**Mustache/Pystache –** No logic support (Logic free)

Cheetah – Last updated 2021
HAML/PyHAML - Last updated 2019
PyPugJS – Sort of active but very Web Focused

Python f-strings

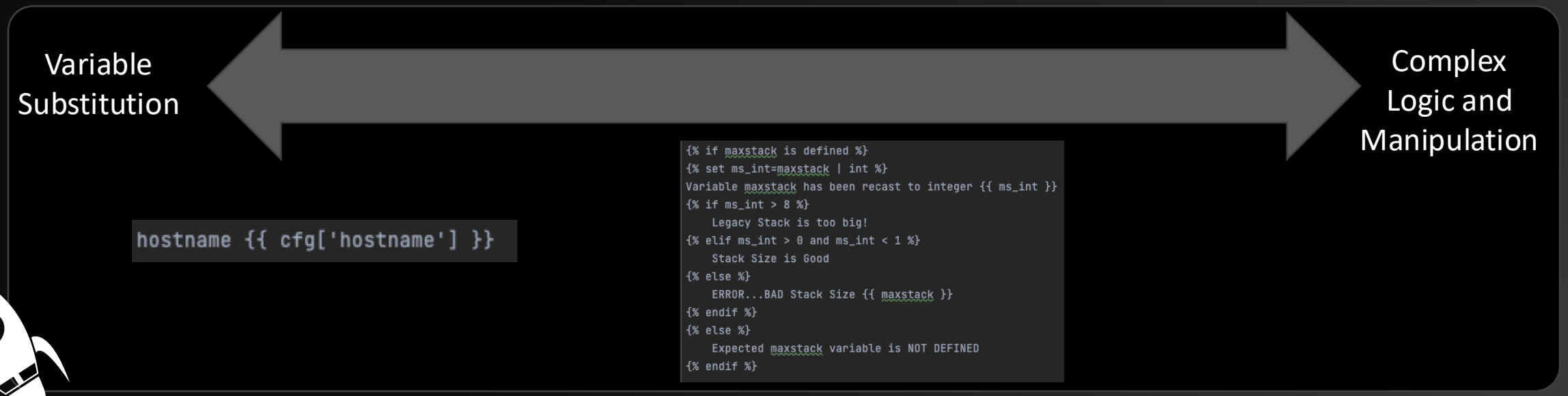
```
% python3
Python 3.9.16 (main, Dec 18 2022, 09:07:38)
[Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> idx = 2
>>> location = "Denver"
>>> rendered = f"Im so happy to be in {location} at AutoCon{idx}"
>>> rendered
'Im so happy to be in Denver at AutoCon2'
```

Python string.Template class

```
>>> import string
>>> template = string.Template("I'm so happy to be in $location for AutoCon$idx")
>>> rendered = template.substitute(location=location, idx=idx)
>>> rendered
"I'm so happy to be in Denver for AutoCon2"
```

# Template Spectrum & Composition

- Where are you in the spectrum
- How do you want to manage your templates?
  - More monolithic (A template for my access EOS)
  - More granular  (Inheritance and Template Blocks)
  - What changes??



# Revision Control

## A Cautionary Tale

"Revision" Numbers Meaningless

Inconsistent revision numbering

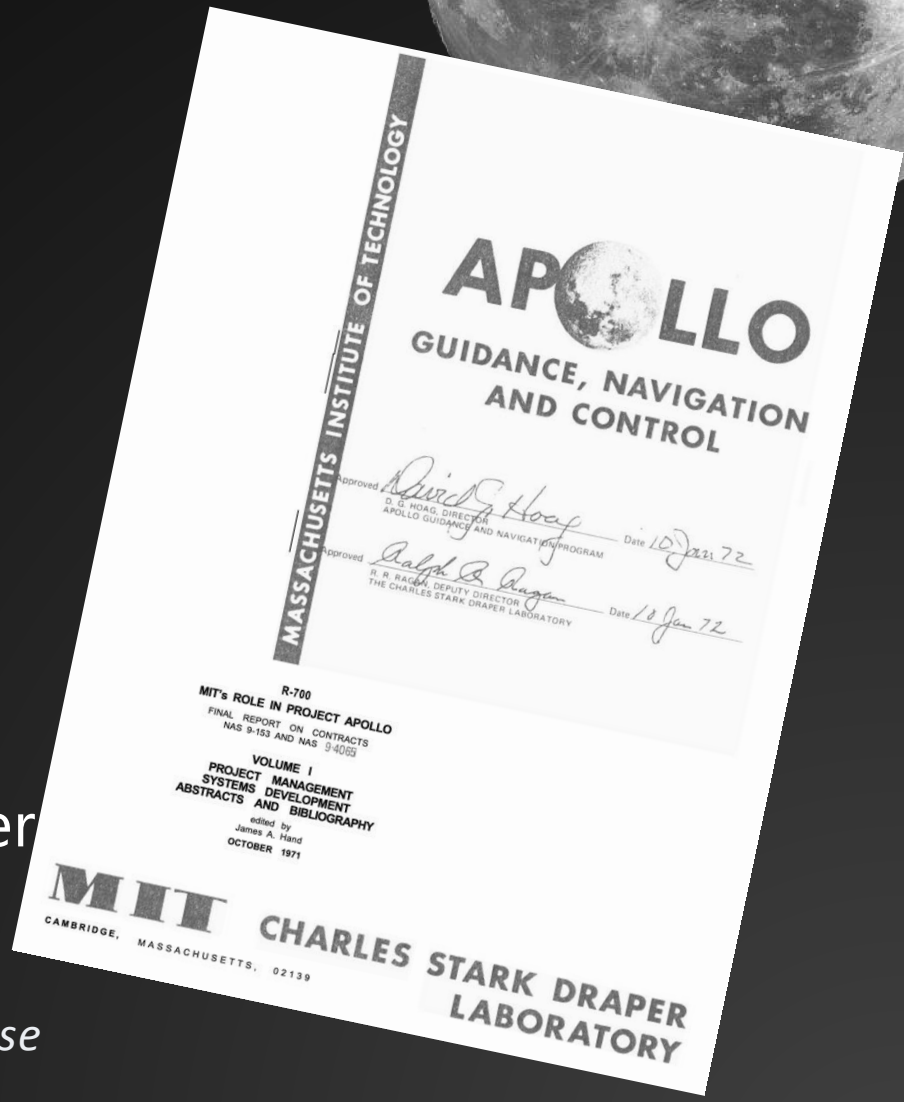
Highest revision number was not the most recent file

- switch_interface_mapping20.py
- switch_interface_mapping21.py
- switch_interface_mapping22.py
- switch_interface_mapping22a.py
- switch_interface_mapping23.py
- switch_interface_mapping24.py
- switch_interface_mapping25.py
- switch_interface_mapping25cdl.py
- switch_interface_mapping26.py
- switch_interface_mapping27.py
- switch_interface_mapping28.py
- switch_interface_mapping29.py
- switch_interface_mapping30cdl.py
- switch_interface_mapping31.py
- switch_interface_mapping32.py
- switch_interface_mapping33.py

# Managing Change

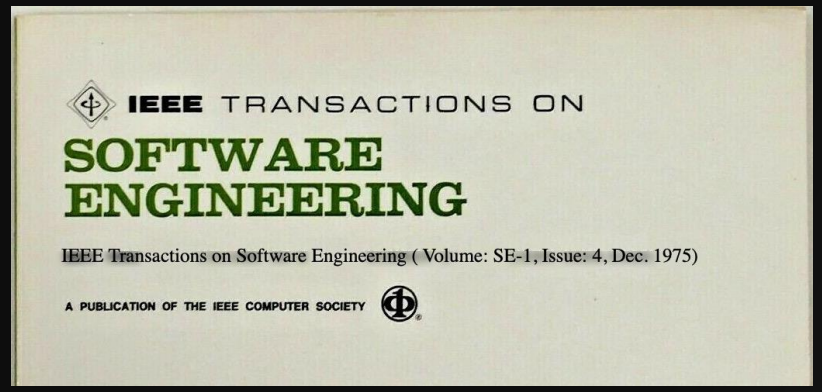
- Source Control
- Version Control
- Revision Control
- Source Control Management (SCM)

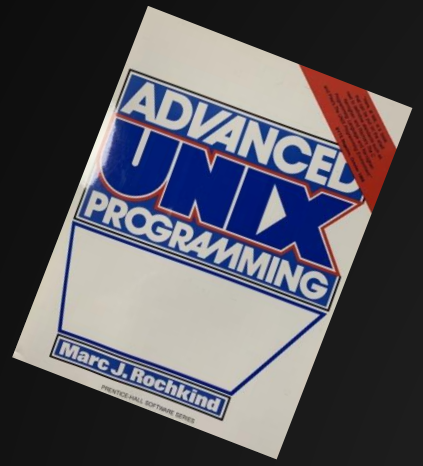
- ~145,000 lines of code in the Apollo Computer
- ~ 40 – 1500 lines of code for one network engineer




*"The specification, formulation, design, coding, testing, and documentation of these programs have been a major undertaking. The change-control activity alone has seen over 750 program change requests processed."*

# Formal Revision Control




*Abstract-The* Source Code Control System (SCCS) is a software tool designed to help programming projects control changes to source code. It provides facilities for storing, updating, and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change, when and where it was made, and why. This paper discusses the SCCS approach to source code control, shows how it is used and explains how it is implemented.

- Storing
- Updating and Versioning
- Retrieving
- Sharing
- Tracking
  - Who made each change
  - When was a change made
  - Why was a change made
  - What changes were made
- 50+ Years of Evolution
  - Saving Efficiently
  - Branching
  - Cloning
  - Forking

# Todays Popular Tools



Our repository is on GitHub

| Local to your System | Remote/Cloud Developer Collaboration & Revision Control |
|---|---|
| **git** | **GitHub (Distributed)** |
| git | GitLab (open source) (Distributed)<br>Bitbucket (Distributed)<br>…many more |
| Mercurial | (Distributed) |
| Apache® Subversion® (Successor to CVS) | Client Server |
| Concurrent Versions System (CVS) | Client/Server |

# Templates & Revision Control

## One Authoritative File

With complete history

In a shareable repository

Consumable by automation

Supporting complex data structures and logic

```
% git log --oneline templates/test_list.j2
f8a007f (HEAD -> main) newline at end of file
54af867 Added debug and output var with out of range index
21a8f46 Added RAR variable output
8a1741f Removing dashes
02f26f0 Renamed to show list and added comments after defining the comment symbol in the Jinja2 env
```

# Jinja2 Fundamentals

**Version 1**

Obsolete

**Version 2**

The current version (3.1.4) still referred to as Jinja2.

# Jinja Workflow - Conceptual

Python

Jinja2

## Template Environment

We must define our templating environment

- Where do we load templates from
- How we are going to load our templates (particularly if using inheritance)
- How we deal with white space
- Do we want debugging
- Do we want to customize

## Load Template

- Once our environment is defined, we just have to pick the template we want to use

Templates are loaded into memory as objects

## Render Template

- Send our data structure to our template for rendering resulting in a specific artifact we can save or pass on to another part of the workflow

Data Manipulation

Save or Send

# Jinja Workflow – Simple Example

Python

Jinja2

## Template Environment

```
# "Environment" is a
string
tmp_str =
"I am in {{ city }}!"
```

## Load Template

```
import jinja2 as j2
# create template object
t = j2.Template(tmp_str)
```

## Render Template

```
result =
t.render(city="Denver"
print(result)
```

```
'I am in Denver'
```

```
city = "Denver"
```

Data Manipulation

Save or Send

# Interactive Python Interpreter

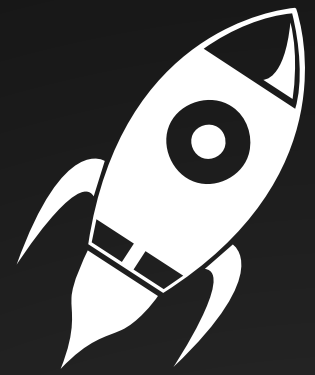
```
Last login: Mon Sep 23 18:53:28 on ttys008
claudiadeluna in ~
% source ~/vEnvs/v3-10-14/venv-ac2jinja/bin/activate
(venv-ac2jinja) claudiadeluna in ~
% python
Python 3.10.14 (main, Jul 15 2024, 11:17:20) [Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> tmp_str = "I am in {{ city }} "
>>> tmp_str
'I am in {{ city }} '
>>> import jinja2 as j2
>>> t = j2.Template(tmp_str)
>>> t
<Template memory:101076ad0>
>>> dir(t)
['__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__g
etattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__ne
w__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref_
_', '_debug_info', '_from_namespace', '_get_default_module', '_get_default_module_async', '_module', '_uptodate', 'block
s', 'debug_info', 'environment', 'environment_class', 'filename', 'from_code', 'from_module_dict', 'generate', 'generate
_async', 'get_corresponding_lineno', 'globals', 'is_up_to_date', 'make_module', 'make_module_async', 'module', 'name', '
new_context', 'render', 'render_async', 'root_render_func', 'stream']
>>> t.name
>>> t.environment
<jinja2.environment.Environment object at 0x1005d7490>
>>> t.render(city="Denver")
'I am in Denver '
>>> city = "Los Angeles"
>>> result = t.render(city=city)
>>> result
'I am in Los Angeles '
>>>
```

# Environment Options

Load your Templates from

- A Python Package
- A Directory
- A String Variable

```
jinja2.Environment(loader=<>)
```



**loader=FileSystemLoader(dir)**

dir is a variable with an absolute or relative path to where the templates are stashed. This method is simple but makes you think about directories.

**loader=PackageLoader(pkg)**

Pkg is a variable with the name of a Python package. This method is highly portable but more complex.

**env.from_string(template_string)**

template_string is a string variable with template code.

# Environment

```
-zsh
/Users/claudiadeluna/Documents/GitHub/ac2_templating_workshop
claudiadeluna in ~/Documents/GitHub/ac2_templating_workshop on main
% clear
claudiadeluna in ~/Documents/GitHub/ac2_templating_workshop on main
% pwd
/Users/claudiadeluna/Documents/GitHub/ac2_templating_workshop
claudiadeluna in ~/Documents/GitHub/ac2_templating_workshop on main
% tree
.
├── move_access_intf.py
├── requirements.txt
└── templates
    ├── all_in_one.j2
    ├── control_structures.j2
    ├── layer2_vlan_template.j2
    ├── move_intf.j2
    ├── test_dict.j2
    ├── test_list.j2
    ├── test_tests.j2
    ├── vlan_list.j2
    └── vlan_lod.j2

2 directories, 11 files
claudiadeluna in ~/Documents/GitHub/ac2_templating_workshop on main
%
```

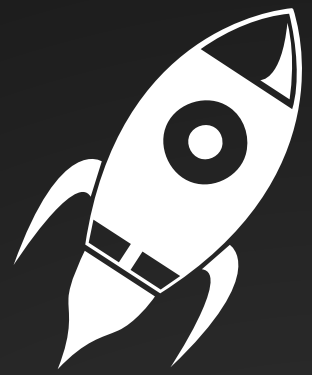
move_access_intf.py

```python
import jinja2


def main():

    # Step 1 - Define the environment containing the templates you want to use
    # Here all the templates will be in a directory called "templates" relative to the current directory
    env_obj = jinja2.Environment(
        loader=jinja2.FileSystemLoader("templates", encoding="utf-8")
    )

    # Step 2 - Load the specific template to be used
    template_obj = env_obj.get_template("move_intf.j2")

    # Template Payload
    cfg_dict = {
        "GigabitEthernet1/0/27": 300
    }

    # Step 3 - Render the template with specific payload
    # The template is expecting a dictionary named "cfg"
    rendered = template_obj.render(cfg=cfg_dict)

    print(rendered)


# Standard call to the main() function.
if __name__ == '__main__':
    main()
```

```python
    env = jinja2.Environment(
        loader=jinja2.FileSystemLoader(search_dir, encoding='utf-8'),
        line_comment_prefix="#",
        keep_trailing_newline=False,  # Default is False
        trim_blocks=True,  # Removes lines
        lstrip_blocks=True,
        undefined=jinja2.runtime.Undefined,  # Default is Undefined - undefined variables render as empty string
    )
```

# Load Template

```
template_file_name="installation_procedure_md_template.j2"
template_obj = jenv_obj.get_template(template_file_name)
```



```python
import jinja2


def main():

    # Step 1 - Define the environment containing the templates you want to use
    # Here all the templates will be in a directory called "templates" relative to the current directory
    env_obj = jinja2.Environment(
        loader=jinja2.FileSystemLoader("templates", encoding="utf-8")
    )

    # Step 2 - Load the specific template to be used
    template_obj = env_obj.get_template("move_intf.j2")

    # Template Payload
    cfg_dict = {
        "GigabitEthernet1/0/27": 300
    }

    # Step 3 - Render the template with specific payload
    # The template is expecting a dictionary named "cfg"
    rendered = template_obj.render(cfg=cfg_dict)

    print(rendered)


# Standard call to the main() function.
if __name__ == '__main__':
    main()
```

```python
def load_jtemplate(jenv_obj, template_file_name):
    """
    This function takes two arguments:
    1. A Jinja2 Environment Object
    2. A Jinja2 Template Filename
    :param jenv_obj: Jinja2 Environment Object
    :param template_file_name: Jinja2 Template Filename
    :return: template_obj which will either be the requested template object or False
    """
    template_obj = False

    try:
        template_obj = jenv_obj.get_template(template_file_name)
    except jinja2.exceptions.TemplateNotFound as fn:
        print(f"ERROR: Template {template_file_name} not found!")
        print(f"Available Templates in the Environment are:")
        for line in jenv_obj.list_templates():
            print(f"\t- {line}")
    except Exception as e:
        print(f"ERROR: {e}")

    return template_obj
```

# Render Template

```python
import jinja2


def main():

    # Step 1 - Define the environment containing the templates you want to use
    # Here all the templates will be in a directory called "templates" relative to the current directory
    env_obj = jinja2.Environment(
        loader=jinja2.FileSystemLoader("templates", encoding="utf-8")
    )

    # Step 2 - Load the specific template to be used
    template_obj = env_obj.get_template("move_intf.j2")

    # Template Payload
    cfg_dict = {
        "GigabitEthernet1/0/27": 300
    }

    # Step 3 - Render the template with specific payload
    # The template is expecting a dictionary named "cfg"
    rendered = template_obj.render(cfg=cfg_dict)

    print(rendered)


# Standard call to the main() function.
if __name__ == '__main__':
    main()
```

Template Variable

Script Variable

**move_intf.j2**

```
! This template expects a dictionary where
the key is an interface id and the value
is the new vlan

{% for intf, new_vlan in cfg.items() %}
interface {{ intf }}
  switchport access vlan {{ new_vlan }}
{% endfor %}
```

```
(venv-ac2jinja) claudiadeluna in ~/Indigo Wire Networks Dropbox/Claudia de Luna/
scripts/python/2024/ac2_templating_workshop on main
% python move_access_intf.py
! This template expects a dictionary where the key is an interface id and the va
lue is the new vlan


interface GigabitEthernet1/0/27
  switchport access vlan 300

(venv-ac2jinja) claudiadeluna in ~/Indigo Wire Networks Dropbox/Claudia de Luna/
scripts/python/2024/ac2_templating_workshop on main
%
```

# Passing payload to your template

## template.render(cfg_data) vs template.render(cfg=cfg_data)

template.render(**cfg_data) aka template.render(cfg_data)

- Unpacking (**) is the default behavior in Jinja2
- so if your cfg_data is a dictionary with one key/value pair `{"hostname": "my_switch"}` if you don't assign your payload to a specific name (like `cfg=cfg_data`), by default, Jinja2 will make the dictionary keys available as top-level variables in the template.

- {{ hostname }} in the template will render to "my_switch"

While using the "unpacking" default behavior can be a bit more streamlined, it inhibits the ability to dump the payload in the template if you are troubleshooting or unsure of the keys or structure.

{{ cfg }}

I find the ability to do this very useful, so I generally always assign my payload (script variable) to a template variable.

# Named vs Unpacked Payload

rendered = template_obj_named.render(cfg=payload_dict)

- Passing script variable to template variable

test_named.j2

```
! Passing Named Payload
{{ cfg }}
```

```
==== Passing named variable ====
! Passing Named Payload
{'list': [1, 2, 3, 4, 5], 'maxstack': 9, 'hostname':
'myswitch-01', 'mydict': {'subnet': '192.168.0.0/24',
 'gw': '192.168.0.1', 'mask': '255.255.255.0'}}
```

- Price: {{ cfg['list'] }} vs {{ list }}

rendered = template_obj_upacked.render(payload_dict)

test_unpacked.j2

```
! Passing unnamed variable
! In standalone Jinja (outside a framework like Django or Flask) there is not a built in way to
access the
! passed variables without naming them

! Referencing the keys directly but you have to know what they are.
! With a named variable you can dump the entire payload with a single command {{ cfg }}
! In this case I know Im passing a dictionary with the keys list, maxstack, hostname, and mydict
{{ list }}
{{ maxstack }}
{{ hostname }}
{{ mydict }}
```

```
==== Passing Un-named and Unpacked variable ====
! Passing unnamed variable
! In standalone Jinja (outside a framework like Django or
Flask) there is not a built in way to access the
! passed variables without naming them

! Referencing the keys directly but you have to know what
they are.
! With a named variable you can dump the entire payload wi
th a single command
! In this case I know Im passing a dictionary with the key
s list, maxstack, hostname, and mydict
[1, 2, 3, 4, 5]
9
myswitch-01
{'subnet': '192.168.0.0/24', 'gw': '192.168.0.1', 'mask':
'255.255.255.0'}
```


Jinja2
Language Syntax

*Undefined variables render as empty strings by default!*

# Variables, Filters, & Tests

## Variables

Outside a control structure

```
{{ var }}
{{ mylist[0] }}
{{ mydict.key }}
{{ mydict[key] }}
```

Inside a control structure

```
{% if var > 1 %}
```

*NOTE the absence of the double curly braces when a variable is part of a control structure.*

## Filters

Jinja comes with built in filters

```
{{ config['hostname']|upper }}
```

| | | | | |
|---|---|---|---|---|
| abs() | float() | lower() | round() | tojson() |
| attr() | forceescape() | map() | safe() | trim() |
| batch() | format() | max() | select() | truncate() |
| capitalize() | groupby() | min() | selectattr() | unique() |
| center() | indent() | pprint() | slice() | upper() |
| default() | int() | random() | sort() | urlencode() |
| dictsort() | join() | reject() | string() | urlize() |
| escape() | last() | rejectattr() | striptags() | wordcount() |
| filesizeformat() | length() | replace() | sum() | wordwrap() |
| first() | list() | reverse() | title() | xmlattr() |

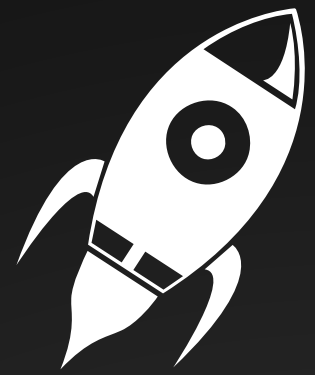
## Tests

Jinja comes with built in tests but its very easy to define your own

```
! Is the variable defined
{% if val is defined %}
    {{ val }} is DEFINED
```

| | | | | |
|---|---|---|---|---|
| boolean() | even() | in() | mapping() | sequence() |
| callable() | false() | integer() | ne() | string() |
| defined() | filter() | iterable() | none() | test() |
| divisibleby() | float() | le() | number() | true() |
| eq() | ge() | lower() | odd() | undefined() |
| escaped() | gt() | lt() | sameas() | upper() |

# Control Structures, Assignments, & Truth



- For Loop

```
{% for intf in intf_list %}
   {{ intf }}
{% endfor %}
```

- IF/ELIF/ELSE

```
{% if ms_int > 8 %}
   Legacy Stack is too big!
{% elif ms_int > 0 and ms_int < 1 %}
   Stack Size is Good
{% else %}
   ERROR...BAD Stack Size {{ maxstack }}
{% endif %}
```

- Assignments

```
{% set ms_int=maxstack | int %}
```

- Truth
  - `true`
  - `false`

```
{% if value == True %}
   Explicitly true
{% endif %}

{% if value is true %}
   Using 'is' test
{% endif %}
```

# Conditions and Operators

Comparison: ==, !=, <, >, <=, >=

Logical: and, or, not

Identity: is, is not

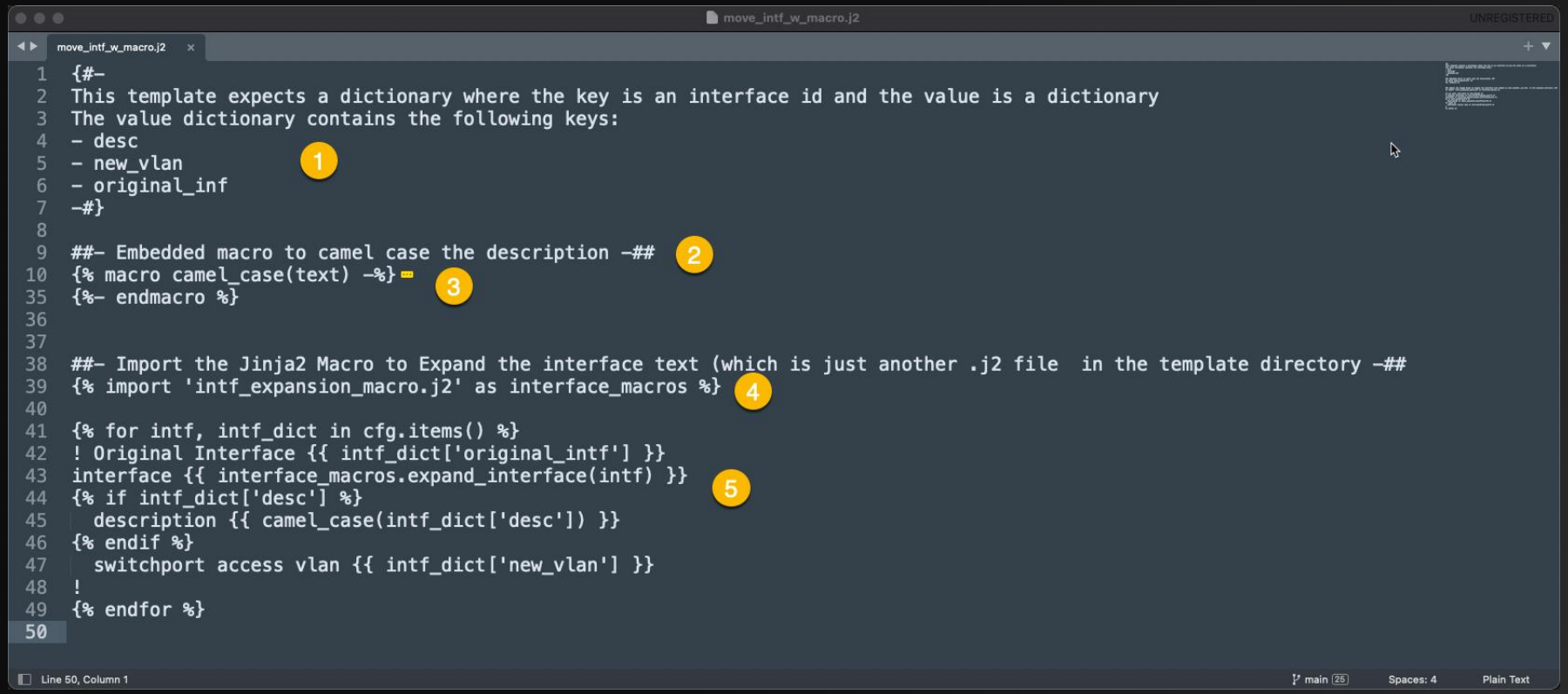
Membership: in, not in

Test: is defined, is undefined, is none

Evaluates to False in Jinja2

- False
- None
- 0 (zero)
- Empty sequences ([], (), {}, set())
- Empty strings ("")
- Special value undefined

# Macros - Templates

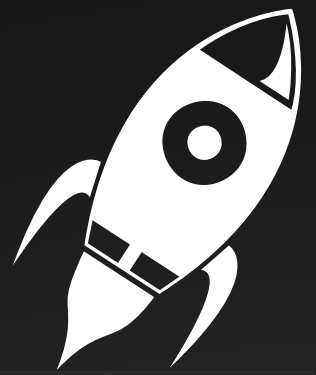



1. Multiline comment with the standard {# #}
2. Single-line comment with '##' configured in environment
3. Embedded macro to turn text into camel case
4. Importing another Jinja2 template file which contains a macro definition
5. More complex cfg data structure which is now a dictionary with values which are also dictionaries

# Macros – Use and Result

```python
# Notice we don't need to import Jinja2 here as we are using the function in utils.py
import utils

def main():

    # Step 1 - Define the environment containing the templates you want to use
    # Here all the templates will be in a directory called "templates" relative to the current directory
    env_obj = utils.jenv_filesystem(line_comment="##", ktn=True, lsb=True, tb=True)

    # Step 2 - Load the specific template to be used
    template_obj = env_obj.get_template("move_intf_w_macro.j2")

    # Template Payload
    # Creating a dictionary for the first 5 GigE interfaces using the GigE shorthand so we can use the
    # interface expansion macro in the template to expand the interface
    cfg_dict = dict()
    desc = "supply kiosk"

    for idx in range(34,39):
        # Cisco UXM switches
        if idx >36:
            intf_prefix = "Te"
        else:
            intf_prefix = "Tw"
        cfg_dict.update({f"{intf_prefix}1/0/{idx}": {"new_vlan": 300, "desc": desc, "original_intf": f"GigE1/0/{idx}"}})

    print(cfg_dict)

    # Step 3 - Render the template with specific payload
    # The template is expecting a dictionary named "cfg"
    desc = ""
    rendered = template_obj.render(cfg=cfg_dict)

    print(rendered)


# Standard call to the main() function.
if __name__ == '__main__':
    main()
```

```
% python move_access_intf_w_macro.py
{'Te1/0/37': {'desc': 'supply kiosk',
              'new_vlan': 300,
              'original_intf': 'GigE1/0/37'},
 'Te1/0/38': {'desc': 'supply kiosk',
              'new_vlan': 300,
              'original_intf': 'GigE1/0/38'},
 'Tw1/0/34': {'desc': 'supply kiosk',
              'new_vlan': 300,
              'original_intf': 'GigE1/0/34'},
 'Tw1/0/35': {'desc': 'supply kiosk',
              'new_vlan': 300,
              'original_intf': 'GigE1/0/35'},
 'Tw1/0/36': {'desc': 'supply kiosk',
              'new_vlan': 300,
              'original_intf': 'GigE1/0/36'}}
```



```
! Original Interface GigE1/0/34
interface TwoGigabitEthernet1/0/34
  description supplyKiosk
  switchport access vlan 300
!
! Original Interface GigE1/0/35
interface TwoGigabitEthernet1/0/35
  description supplyKiosk
  switchport access vlan 300
!
! Original Interface GigE1/0/36
interface TwoGigabitEthernet1/0/36
  description supplyKiosk
  switchport access vlan 300
!
! Original Interface GigE1/0/37
interface TenGigabitEthernet1/0/37
  description supplyKiosk
  switchport access vlan 300
!
! Original Interface GigE1/0/38
interface TenGigabitEthernet1/0/38
  description supplyKiosk
  switchport access vlan 300
!
```

# Macros - Considerations

Macros are handy but…

Its important to think about where you want your logic.

Do you want to be updating your Templates when a new prefix comes out or is it better to update your Python script which likely has other related logic.

What if you want to use the interface expansion logic in one of your Python scripts. Will you be better served with a Python function rather than a Jinja2 macro?

There is no right answer here.

I keep logic to a minimum in my Templates and do the heavy lifting (more complex data manipulation) in the Python script but that is what works for me and the use cases I'm supporting.

# Strategy for Creating Templates

One instance vs N instances (even if N = 1)

## In code

... Doesn't count
Templates in code should be very tactical and targeted. Things like a route statement or adding a vlan to a trunk or setting a new access vlan on an access interface.

## Monolithic

Monolithic templates encompass an entire device configuration including all the specific features enabled for the device.
PRO
- Configuration patterns for a specific model/function are in one template

CON
- Configuration patterns for a specific model/function are in one template. If different teams govern the feature pattern, there may be many making changes to the single template

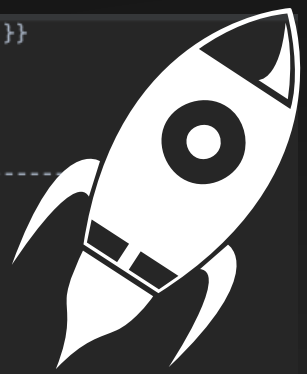
## Modular (Inheritance)

With modular templates, you take on the additional work of putting together a complete configuration for a device but separate features are in their own "feature" template and can be managed individually. The WAN team can manage their BGP template, the Security team can manage their NAC template, etc.
PRO
- Feature templates can be managed by the controlling team

CON
- Added step of building a complete cfg

# Monolithic

A single template for a particular type of device.



```
! Access Switch Configuration for location {{ cfg['location'] }}
! Access Switch Hostname is {{ cfg['hostname'] }}
! Using Template: DNAC Sample Jinja Template Base Config
! Configuration Generated {{ cfg['timestamp'] }}
! ------------------------------------------------------------------
!
no service pad
service timestamps debug datetime msec localtime
service timestamps log datetime msec localtime
service password-encryption
!
hostname {{ cfg['hostname'] }}

!
boot-start-marker
boot-end-marker
!
enable secret {{ cfg['enable_sec'] }}
!
!
!
aaa new-model
!
!
aaa authentication login default group tacacs+ enable none
aaa authentication login console enable none
aaa authentication enable default group tacacs+ enable none
aaa authorization exec default group tacacs+ none
aaa authorization exec console none

aaa authorization commands 15 default group tacacs+ none
!
!
!
aaa session-id common
clock timezone {{ cfg['timezone'] }} {{ cfg['timezone_offset'] }}
{% if cfg['summertime_bool'] %}
clock summer-time {{ cfg['timezone'] }} recurring  {{ cfg['summertime_start_stop'] }}
{% endif %}

```

# Modular using Jinja2 Includes

```python
import jinja2


def load_cfg_data():...


def main():

    # Load configuration payload
    cfg_data = load_cfg_data()

    # Create Jinja2 environment
    env = jinja2.Environment(
        loader=jinja2.FileSystemLoader('templates'),
        trim_blocks=True,
        lstrip_blocks=True
    )

    # Load base template
    template = env.get_template('modular_inc_base.j2')

    # Render configuration
    config = template.render(cfg_data)

    # Write configuration to file
    file_name = f"{cfg_data['hostname']}_router_config.txt"
    with open(file_name, 'w') as f:
        f.write(config)

    print(f"\nSaved configuration to file {file_name} in current directory.\n")


# Standard call to the main() function.
if __name__ == '__main__':
    main()
```

```
{#-
# Directory structure:
# templates/
#   modular_inc_base.j2
#   modular_inc_aaa.j2
#   modular_inc_interfaces.j2
#   modular_inc_ospf.j2
# modular_include_config_generator.py
-#}

! templates/modular_inc_base.j2
hostname {{ hostname }}
!
{% include 'modular_inc_aaa.j2' %}
!
{% include 'modular_inc_interfaces.j2' %}
!
{% include 'modular_inc_ospf.j2' %}
!
end
```

# Modular using Inheritance with Extend

```python
def main():

    # Load configuration payload
    cfg_data = load_cfg_data()

    # Create Jinja2 environment
    env = jinja2.Environment(
        loader=jinja2.FileSystemLoader('templates'),
        trim_blocks=True,
        lstrip_blocks=True
    )

    # Load templates
    base_template = env.get_template('mod_inherit_base.j2')
    aaa_template = env.get_template('mod_inherit_aaa.j2')
    interfaces_template = env.get_template('mod_inherit_interfaces.j2')
    ospf_template = env.get_template('mod_inherit_ospf.j2')

    # Generate configurations
    config_sections = {
        'base': base_template.render(cfg_data),
        'aaa': aaa_template.render(cfg_data),
        'interfaces': interfaces_template.render(cfg_data),
        'ospf': ospf_template.render(cfg_data)
    }

    print(config_sections['base'])
    print(config_sections['aaa'])
    print(config_sections['interfaces'])
    print(config_sections['ospf'])

    # Write configuration to file
    file_name = f"{cfg_data['hostname']}_router_config.txt"
    with open(file_name, 'w') as f:
        f.write(config_sections['base'])
        f.write(config_sections['aaa'])
        f.write(config_sections['interfaces'])
        f.write(config_sections['ospf'])

    print(f"\nSaved configuration to file {file_name} in current directory.\n")
```

```
!
! Base Configuration Template
!
hostname {{ hostname }}
!
{# AAA Configuration Block #}
{% block aaa %}
{% endblock %}
!
{# Interface Configuration Block #}
{% block interfaces %}
{% endblock %}
!
{# OSPF Configuration Block #}
{% block ospf %}
{% endblock %}
!
end
```

```
{% extends "mod_inherit_base.j2" %}

{% block aaa %}
! AAA Template
aaa new-model
aaa authentication login default group tacacs+ local
aaa authorization exec default group tacacs+ local
aaa accounting exec default start-stop group tacacs+
!
tacacs server TACACS1
 address ipv4 {{ tacacs_server }}
 key {{ tacacs_key }}
{% endblock %}
```

```
{% extends "mod_inherit_base.j2" %}

{% block interfaces %}
! Interface Template
{% for interface in interfaces %}
interface {{ interface.name }}
 description {{ interface.description }}
 ip address {{ interface.ip_address }} {{ interface.subnet_mask }}
 {% if interface.get('ospf_area') %}
 ip ospf {{ ospf_process_id }} area {{ interface.ospf_area }}
 {% endif %}
 no shutdown
!
{% endfor %}
{% endblock %}
```

# Creating Templates with AI

This is an area where AI is already proving to be a tremendous time saver.

The containerlab mini project was largely generated by AI with some minor tweaking from me.

This can be quite a time saver however

- don’t be afraid to tell your AI buddy they are wrong!
- double check. (I asked for a 3-tier network topology and got the access switches connected to both core and distribution..I'm just saying…)
- this is like “verbal” coding so be precise in your prompts

# Whitespace

Can be aggravating!  Control structures can add unwanted whitespace and empty lines.

```
rendered = template_obj.render(cfg=cfg_dict2, desc=desc)
```

```
{% for intf, new_vlan in cfg.items() %}
interface {{ intf }}
{% if desc %}
  description {{ desc }}
{% endif %}
  switchport access vlan {{ new_vlan }}
{% endfor %}
```

desc="Digital Signage”

```
interface GigabitEthernet1/0/1
  description Digital Signage
  switchport access vlan 300
interface GigabitEthernet1/0/2
  description Digital Signage
  switchport access vlan 300
interface GigabitEthernet1/0/3
  description Digital Signage
```

desc=””

```
interface GigabitEthernet1/0/1
  switchport access vlan 300
interface GigabitEthernet1/0/2
  switchport access vlan 300
interface GigabitEthernet1/0/3
  switchport access vlan 300
```

```
{% for intf, new_vlan in cfg.items() %}
interface {{ intf }}
{% if desc %}  description {{ desc }}{% endif %}
  switchport access vlan {{ new_vlan }}
{% endfor %}
```

desc="Digital Signage”

```
interface GigabitEthernet1/0/1
  description Digital Signage
  switchport access vlan 300

interface GigabitEthernet1/0/2
  description Digital Signage
  switchport access vlan 300

interface GigabitEthernet1/0/3
  description Digital Signage
  switchport access vlan 300
```

desc=””

```
interface GigabitEthernet1/0/1

  switchport access vlan 300

interface GigabitEthernet1/0/2

  switchport access vlan 300

interface GigabitEthernet1/0/3

  switchport access vlan 300
```

Tips

- Often it will come down to what you can live with and the functional impact of any extraneous whitespace when you push your configs.
- “One liners” can help
- Jinja2 has several white space management options

# Whitespace Control with “-”

The minus sign can be used

{%- ... %} Removes whitespace before the tag

{% ... -%} Removes whitespace after the tag

{%- ... -%} Removes whitespace both before and after the tag

{{- cfg -}} Removes whitespace both before and after the variable

The plus sign preserves white space which would normally be removed. While its use is less common it can be useful as an override particularly if you set whitespace controls when defining your environment.

{%+ ... %} Preserve indentation

```
{% for intf, new_vlan in cfg.items() %}
interface {{ intf }}
{% if desc %}
  description {{ desc }}
{% endif -%}
  switchport access vlan {{ new_vlan }}
{% endfor %}
```

desc="Digital Signage”

```
interface GigabitEthernet1/0/1

  description Digital Signage
switchport access vlan 300

interface GigabitEthernet1/0/2

  description Digital Signage
switchport access vlan 300

interface GigabitEthernet1/0/3

  description Digital Signage
switchport access vlan 300
```

desc=””

```
interface GigabitEthernet1/0/1
switchport access vlan 300

interface GigabitEthernet1/0/2
switchport access vlan 300

interface GigabitEthernet1/0/3
switchport access vlan 300
```

# Whitespace Control in Environment

```
import jinja2

env = jinja2.Environment(
    trim_blocks=True, # Remove first newline after a block  (
    lstrip_blocks=True, # Remove leading spaces and tabs from block tags
    keep_trailing_newline=True, # Preserve trailing newline in templates
)
```

By default these three settings are False.

```
env_obj2 = utils.jenv_filesystem(
  ktn=True,
  lsb=True,
  tb=True
)
```

```
{% for intf, new_vlan in cfg.items() %}
interface {{ intf }}
{% if desc %}
  description {{ desc }}
{% endif %}
  switchport access vlan {{ new_vlan }}
{% endfor %}
```

desc="Digital Signage”

```
==== With white space settings set in env ====

interface GigabitEthernet1/0/1
  description Digital Signage
  switchport access vlan 300
interface GigabitEthernet1/0/2
  description Digital Signage
  switchport access vlan 300
interface GigabitEthernet1/0/3
  description Digital Signage
  switchport access vlan 300
```

desc=””

```
==== With white space settings set in env ====

interface GigabitEthernet1/0/1
  switchport access vlan 300
interface GigabitEthernet1/0/2
  switchport access vlan 300
interface GigabitEthernet1/0/3
  switchport access vlan 300
interface GigabitEthernet1/0/4
  switchport access vlan 300
interface GigabitEthernet1/0/5
  switchport access vlan 300
```

# Did I mention whitespace control is maddening?

- Expect Trial and Error
- In some cases, a separator in your template can help (!)
- Understand what is aesthetics vs functional

```
{% for intf, new_vlan in cfg.items() %}
interface {{ intf }}
{% if desc %}
  description {{ desc }}
{% endif %}
  switchport access vlan {{ new_vlan }}
!
{% endfor %}
```

```
env_obj2 = utils.jenv_filesystem(
  ktn=True,
  lsb=True,
  tb=True
)
```

desc="Digital Signage”

```
==== With white space settings set in env ====

interface GigabitEthernet1/0/1
  description Digital Signage
  switchport access vlan 300
!
interface GigabitEthernet1/0/2
  description Digital Signage
  switchport access vlan 300
!
interface GigabitEthernet1/0/3
  description Digital Signage
  switchport access vlan 300
```

desc=””

```
==== With white space settings set in env ====

interface GigabitEthernet1/0/1
  switchport access vlan 300
!
interface GigabitEthernet1/0/2
  switchport access vlan 300
!
interface GigabitEthernet1/0/3
  switchport access vlan 300
!
```

# Comments

- Multiline {# #}

```
{#
Multi-line comments in a Jinja2 template.
These will not be part of the rendered text.
#}
```

- Single Line {# #} and custom set in environment

```
{# Single line using the default comment #}

If set when establishing the environment
## Another single line comment ##

Even if "line_comment_prefix" is
set, {# #} can be used

Use {#-  -#} to strip the white space!
```

def jenv_filesystem in utils.py

```python
valid_line_comment_prefixes = ["//", "#", ";", "--", "##"]
# In case an invalid or blank line comment is provided to the function
if line_comment not in valid_line_comment_prefixes:
    line_comment = "##"

env = jinja2.Environment(
    loader=jinja2.FileSystemLoader(search_dir, encoding="utf-8"),
    line_comment_prefix=line_comment,  # Only works for single-line comments
```

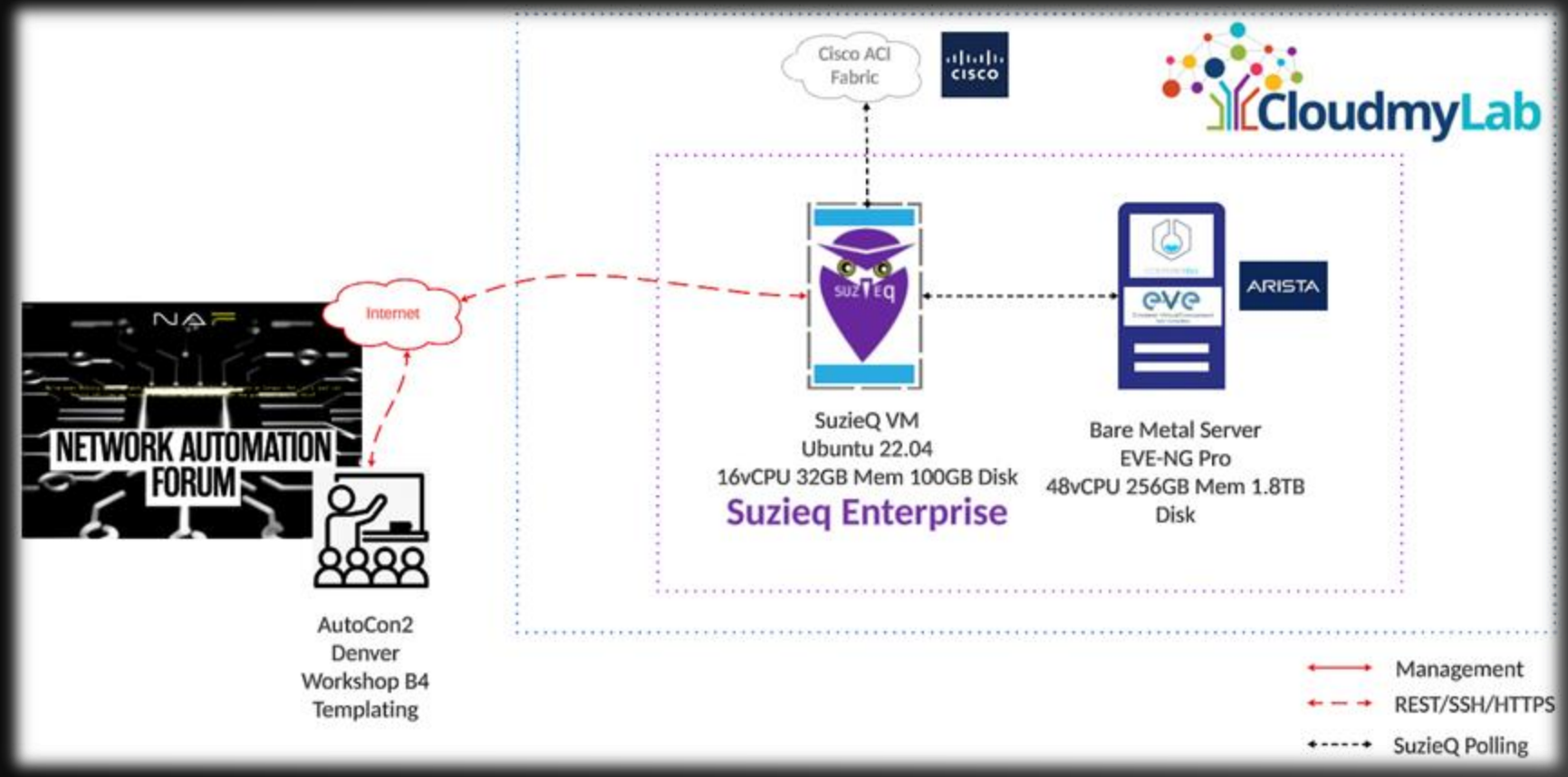
# Course Repository

- General use scripts and templates at the top level used to illustrate various concepts in the presentation
- Utility Script (`util.py`)
  - You will see this imported in most of the mini projects.
  - There are handy functions that can be used across the mini projects and it's a good place for reusable functions.
  - A key function is the **`render_in_one.`**
  - This function takes:
    - A template filename
    - A payload dictionary
    - And optional parameters controlling the template sub-directory and whitespace control in the defined environment
  - And returns
    - The rendered text
- Mini Projects (in project sub-directories)

```python
def render_in_one(
    template_file_name, payload_dict, search_dir="templates", line_comment="#"
):
    """
    Inspired by:
    https://daniel.feldroy.com/posts/jinja2-quick-load-function

    This is the all-in-one version of the Jinja2 process
    1. Create environment (using the jenv_filesystem function)
    2. Load template (using the load_jtemplate function)
    3. Render template with data

    :return: rendered text as a string (class 'str')
    """

    jenv = jenv_filesystem(search_dir=search_dir, line_comment="#")

    jtemplate = load_jtemplate(jenv, template_file_name=template_file_name)

    # The templates must use a variable called "payload_dict"
    return jtemplate.render(cfg=payload_dict)
```

# Course Infrastructure

In addition to your student laptops and our workshop repository…



# Template Exercises

We will be working through the use cases below in class

**Current State Report**
Current State Report

**Clab Topology**
Clab Topology based on LLDP topology

**Configurations**
Monolithic vs Modular (Inheritance)

**Installation Procedure**
Installation Procedure

**Change Request**
Change Request Text
Chang Request (SNOW Optional)

**Reports, Documents, & Data Input**
Verification Report
Statement of Work
Design Document

# Joint Mini Projects

## 01_ Current State Report

Takes existing JSON output from SuzieQ and generates a Markdown report with a diagram showing the BGP sessions some of their attributes.

## 02_Procedure

Takes a YAML file of configuration details and generates an Installation report suitable to give to a local “Smart Hands” for installation of an appliance.

## 03_Configurations

Examples of Monolithic and Modular configurations using Jinja2 features including extends and includes.

# Templating Mini Projects

## 04_ChangeRequest

- This mini project will use a template to generate Change Request text
- It illustrates the type of payload manipulation that winds up being common
- It includes an option to create a Change Request on a developer instance of Service Now using the Python requests module.
- It also introduces the use of environment variables and the built in os module  to set secrets and sensitive information via the command line as environment variables for the script to use.
  - This is handy if you are in a situation were you want to really minimize the use of 3rd party modules ..say you ONLY have python and its built-in modules.

# Templating Mini Projects

## 05_ClabTopology

- This project extracts LLDP topology information for a site and creates a simplified containerlab topology for testing.
- It shows the use of the Mermaid Live editor to illustrate a diagram based on the topology. (Note that Containerlab also does this)
- It also introduces the use of the python-dotenv module and a .env file to store secrets and sensitive information.
  - In this case the REST API token to the Workshop SuzieQ server.

# Templating Mini Projects

## 06_VerificationReport

- This verification report is suitable for post change verification for a new Vlan.
- The script pulls Vlan and Spanning Tree data from the SuzieQ server and performs some common verification steps for a given Vlan.
  - Is it configured where it needs to be?
  - Does it have STP root?
- The use of a local WebApp is shows.  This is actually a Streamlit App that presents the user with curated information to minimize “bad user input and typos”.
- We will discuss the lengthy history of automating this task!

# Templating Mini Projects

## 07_SoW

- This mini project takes “design data” (a list of new Access Points, their location and specifications for the required cabling runs) and develops a Statement of Work suitable for RFQs.
- This shows the “off label” use of SuzieQ to hold design data.

# Templating Mini Projects

## 08_DesignDocument Capstone Project

- Get Hardware, Connectivity, and Subnet/Vlan Design details from the SuzieQ Sever for Site (namespace) DEN_Campus
- Create a Low-Level Design Document Markdown Template in Jinja2
- Determine how you want to represent a topology diagram
  - Diagrams
  - Mermaid
  - Other
- Generate an LLD Design Package with a Document and the corresponding device configurations for each new device.
- Generate the Change Request Text suitable for a change request ticket for the deployment of this new network


SlideModel.com