Provenance

At AutoCon2, NAF and EIA hosted a Jinja2 (Beginner to Intermediate) Templating workshop.

Attendee feedback suggested the workshop was more intermediate than beginner and so for AutoCon3 we have put together this Introductory Jinja2 course.

It is available to the public.

If you want to dive deeper into the power of templating and look at real world uses that are not related to generating configuration we hope you join us for the actual Workshop!

NAF AutoCon3 Workshops

Workshop Details

- Explore real-world use cases
- Take away lessons learned and expert recommendations
- Share in the collective experience and use cases of the course attendees

AUTOCON 3

WORKSHOP

JINJA2 IN ACTION: 10 REAL-WORLD PYTHON TEMPLATE PATTERNS

Dive into the power of Jinja2, a Python templating engine, and its role in streamlining network automation workflows. This workshop is designed for participants with a foundational understanding of Python, REST APIs, and Jinja.

Proctor:

CLAUDIA DE LUNA

AGENDA:

- Jinja Essentials
- Background on Templating Languages
- Revision Control
- Template Format Options
- Use Case Working Sessions

SECURE YOUR SPOT

TUESDAY 27 MAY, 2025

Level: Intermediate

09:00-13:00

NETWORKAUTOMATION.FORUM/AUTOCON3#WORKSHOP

WS:C4

Course Requirements

1. Browser & Internet Connection
    - TTL255 J2Live
    - PacketCoders J2Render
2. Basic Python Knowledge
3. Google Colab account (Free) or Python Interpreter
4. UV installed (Installing UV)
5. Clone or Download this repository ac3 intro to jinja2

Patterns & Templates

...have been around for a long time!

Web Development has given us some amazing tools to manipulate text (HTML) and documents. Variable substitution is its simplest form.

Enter the Python module **Jinja2**

Why should I care about this?

As Network Engineers we use templates daily.
- Configurations
- Outage emails
- Change Requests
- Work Instructions
- Anything that is text

Jinja2 is the de facto standard in Network Automation workflows.

Why are we here?

You may be saying **"..but I already have templates!"**

I've see templates in:

- Microsoft Word
- Notepad (Text)
- PowerPoint
- Excel
- **The last device I configured**

What problem are we trying to solve?

How many times have you seen something like this?

```
interface e1/0/1
  description <Enter interface description here>
```

We have to give up the idea of people filling values manually.

Humans should be out of the variable subsitution business

- It does not scale
- It is prone to errors
- It allows inconsitency
- It kills reproducability
- It takes too much time
- Its not very rewarding (in and of itself)

What is Jinja2?

Jinja2 is an open source text-based templating engine (module) for the Python programming language.

It gained adoption due to is flexibility and Python-like syntax.

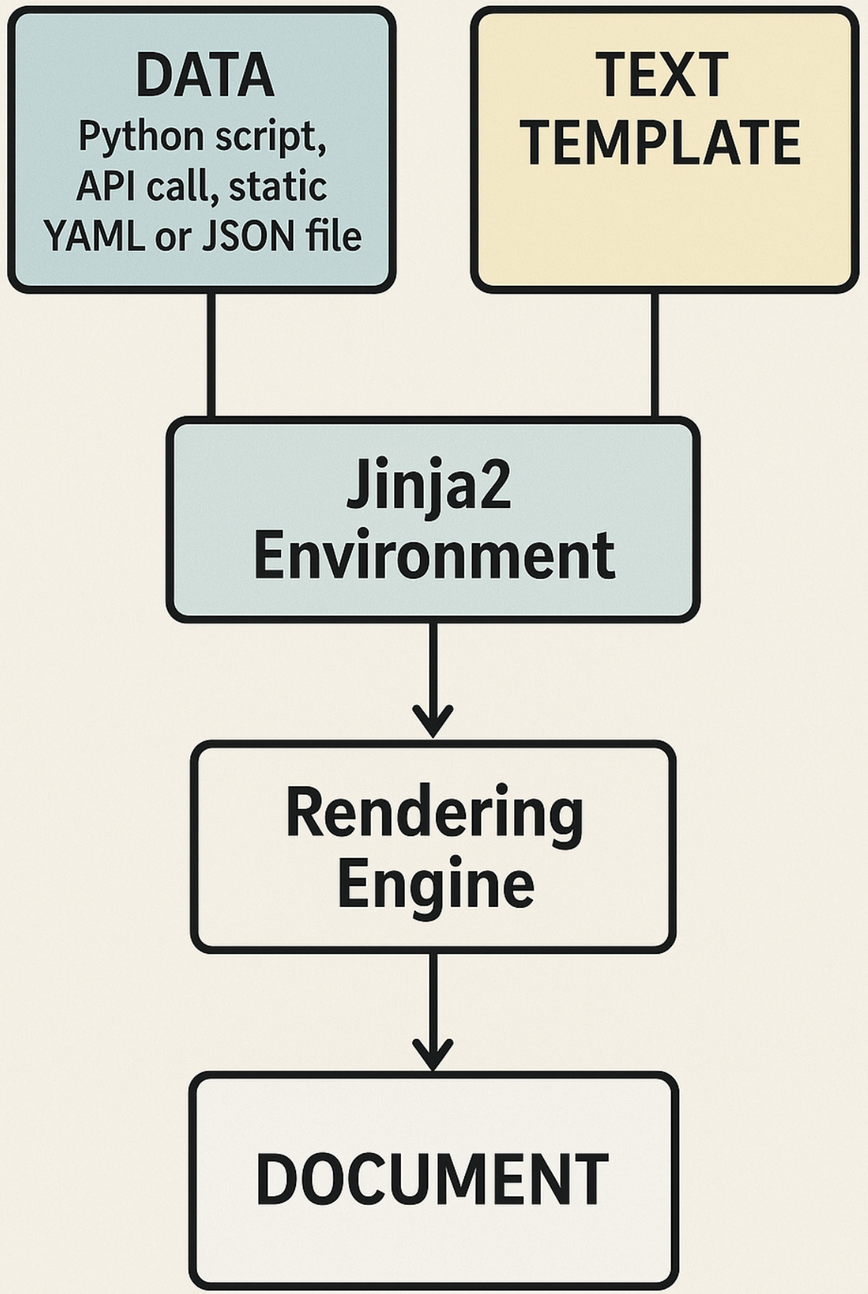
While it has its roots as a web/HTML templating language, it is well-suited for generating a wide range of text output formats.

Jinja2 Versions

- Version 1
  Obsolete
- Version 2 <-- **We are here**
  The current version (3.1.6) is still referred to as Jinja2

Jinja2 Components

The Jinja2 Module
The Execution Environment
- **Template**
- **Data**



Templates

- are typically text files with a .j2 extension
- they can also be variables in a python script

```
% cat simple_template.j2
! Simple Template
{{ my_var }}

% cat vlan_list.j2
! Layer 2 Vlan Template
! using a List of Lists

{% for line in mylistvar %}
vlan {{ line[0] }}
  name {{ line[1] }}
{% endfor %}
```

control_structures.j2 — Edited

```
! Control Structures
{# Typical Comments #}

! For Loops
{% for intf in intf_list %}
    {{ intf }}
{% endfor %}


! If/else if/ else

{% if maxstack is defined %}
{% set ms_int=maxstack | int %}
Variable maxstack has been recast to integer {{ ms_int }}
{% 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 %}
{% else %}
    Expected maxstack variable is NOT DEFINED
{% endif %}

! Filter
{% filter upper %}
    This text becomes uppercase
{% endfilter %}
```

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

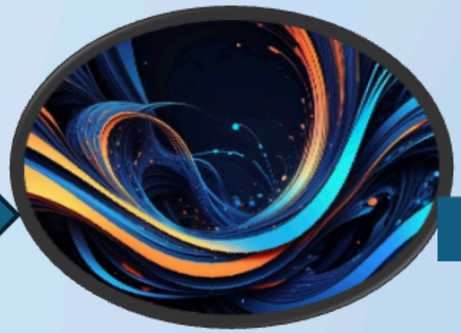
Load Template

- 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

PREVIEW

Exercise 1 - On Line

Przemek Rogala's J2Live Online Jinja2 Parser and Renderer by TTL255

Packet Coders Jinja2 Renderer

Execution Environment

Ultimately Jinja2 will be a tool in your automation workflow
- Python
- Ansible
- Vendor tools like Cisco Catalyst Center (DNAC)
- Other Automation Environments

LANGUAGE SYNTAX

Variables by Themselves

Outside a control structure (jinja2 block)

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

Note: Undefined variables render as empty strings by default!

Variables in a Jinja Statement

Outside of a control structure (jinja2 block)

```
{{ var }}
```

Inside a control structure

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

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

Control Structures (jinja2 block)

Comparison between Python and Jinja2 {% %}

- In Python, blocks use indentation and may or may not have a close statement
- In Jinja2, blocks use {% ... %} and {% end... %} (they generally have a close statement)

Control Structure	Python Syntax	Jinja2 Syntax	Notes
If	`if condition:` `...`	`{% if condition %}...{% endif %}`	Very similar; `{% %}` encl control.

If-Else	`if condition:`
	`...`
	`else:`
	`...`
		`{% if condition %}...{% else %}...{% endif %}`	Same logic, different blo

Elif (Else if)	`elif condition:`	`{% elif condition %}`	Jinja2 uses `{% elif %}` `%}` .
For loop	`for item in list:` `...`	`{% for item in list %}...{% endfor %}`	Same logic; `{% endfor` loop.

For-Else loop	`for item in list:`
	`...`
	`else:`
	`...`
		`{% for item in list %}...{% else %}...{% endfor %}`	Jinja2 supports `for-els` Python.

While loop	`while condition:` `...`	*Not supported directly*	Must simulate using `{%` conditions.
Break	`break`	`{% break %}`	Available in Jinja2 >= 2.1
Continue	`continue`	`{% continue %}`	Available in Jinja2 >= 2.1
With (scoping)	`with open('file') as f:` `...`	`{% with var=value %}...{% endwith %}`	Different usage; Jinja2 fo variables.
Set Variable	`x = 5`	`{% set x = 5 %}`	Jinja2 requires `{% set %` assignment.

Control Structures - Loops

For Loop

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

Control Structures - Conditionals

IF/ELIF/ELSE

```
{% if ms_int > 8 %}
    Legacy stack is too big!
{% elif ms_int > 0 and ms_int < 9 %}
    Stack Size is Good
{% else %}
    ERROR...BAD Max Stack Value {{ ms_int }}
{% endif %}
```

Assignments

Define a variable within the Template

```
{% set myvalue = 2 %}
```

Assignment and cast to integer (using a built-in J2 Filter)

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

Filters

List of Builtin Filters

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

Tests

Jinja comes with built in tests.
It is very easy to define your own

```
! Is the variable defined
{% if val is defined %}
The variable val exists and
has a value of {{ val }}
{% endif %}
```

List of Builtin Tests

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()

Conditions and Operators

Jinja2 Expression Categories and Operators

Category	Operators
Comparison	==, !=, <, >, <=, >=
Logical	and, or, not
Identity	is, is not
Membership	in, not in
Test	is defined, is undefined, is none

Symbol	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

False in Jinja2

What evaluates to False in Jinja2?
- false
- none
- 0 (zero)
- Empty sequences ([], (), {}, set())
- Empty strings ("")
- Special value undefined

Type	Python Falsy	Jinja2 Falsy
`None`	✅	✅
`False`	✅	✅
`0` / `0.0`	✅	✅
`''` (empty string)	✅	✅
`[]`, `{}`, `()` (empty collections)	✅	✅
`undefined`	❌ (doesn't exist)	✅
`missing attribute/variable`	Raises error	Evaluates to `False` (but emits a warning or silent fail based on config)

Truth may not behave as expected

These are not interchangeable across contexts:
- In Python code, using true will result in a NameError
- In Jinja2 templates, using True will not be recognized as a boolean
- In Jinja2 type matters

Be Careful With Truth

```
{% if value is true %}
    Using 'is' test works on a boolean
{% else %}
    Fails: checks for exact boolean true
{% endif %}

{% if value %}
    "Truthy" true
{% endif %}
```

Python vs Jinja2

Concept	Python	Jinja2
True value	True	true
False value	False	false
None/null value	None	none

WORKFLOW ENVIRONMENT

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

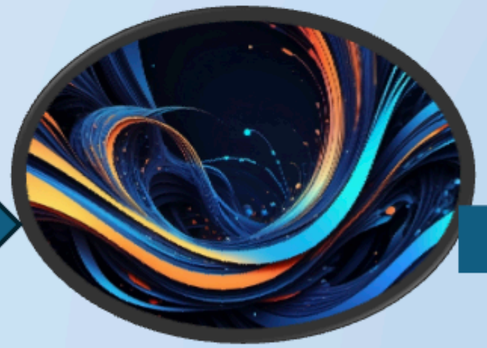
Load Template

- 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 Sen

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.

Jinja2 Environment Loaders

```
# env_examples
.
├── j2_env_loader_examples.py
├── app_templates
│   ├── __init__.py
│   └── simple_template.j2
├── dir1
│   └── dir2
│       └── simple_template.j2
└── templates
    └── simple_template.j2
```

Quick Summary of Requirements:

Loader Type	Requirements
String Loader	Nothing needed on disk.
Filesystem Loader	`./templates/simple_template.j2` must exist.
Package Loader	Python package `app_templates/` with `simple_template.j2` and `__init__.py`.
Anywhere Loader	Template at `./dir1/dir2/simple_template.j2` or adjust the full path accordingly.

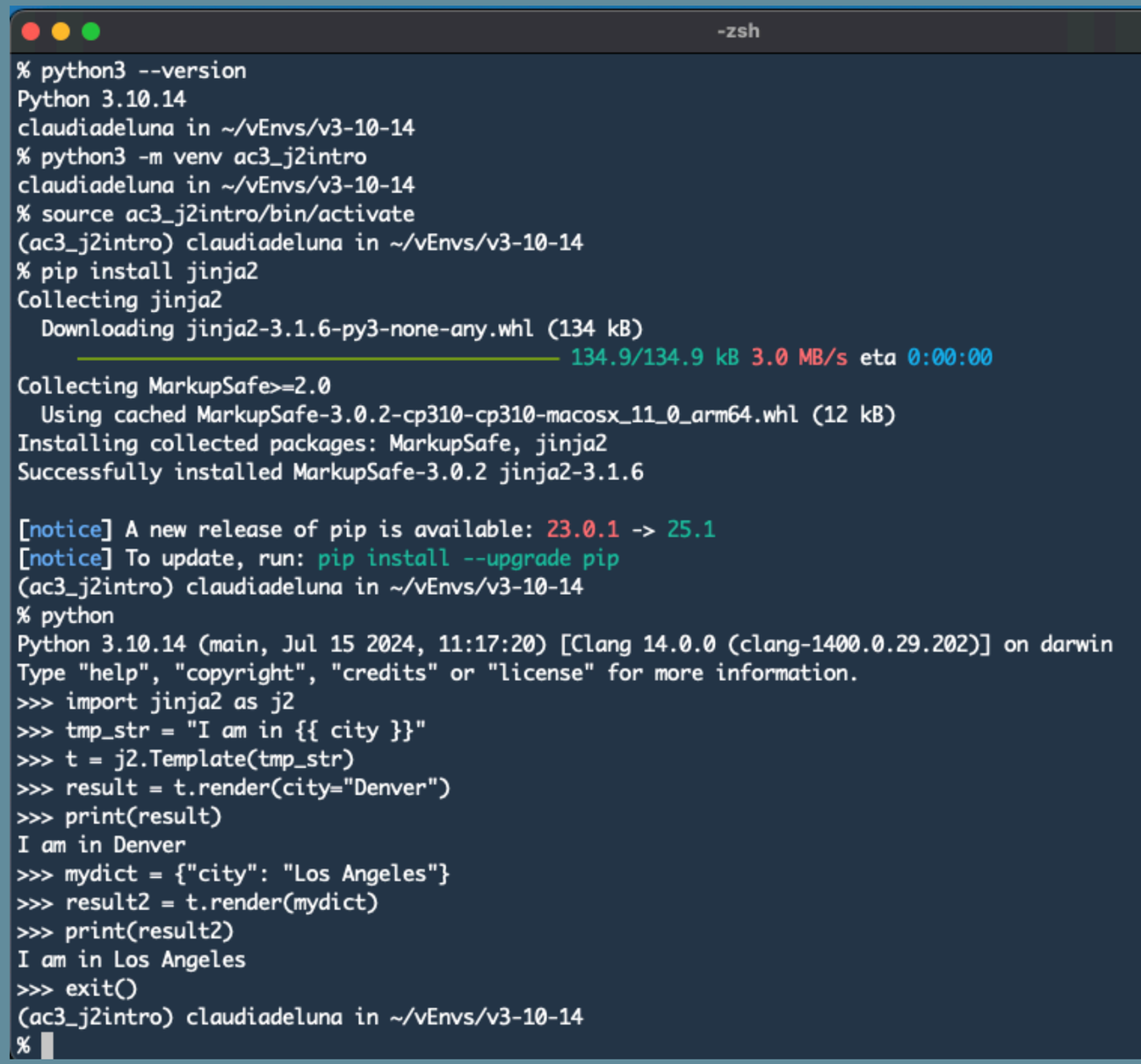
Exercise 2 - Load and Render via CLI

Python Interactive Interpreter or REPL (Read, Evaluate, Print, and Loop)

```
% uv run python
Using CPython 3.11.12
Creating virtual environment at: .venv
Installed 2 packages in 4ms
Python 3.11.12 (main, Apr  9 2025, 03:49:53) [Clang 20.1.0 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import jinja2 as j2
>>> exit()
```

OR

Google Colab



Exercise 3 - Load and Render in Python

Template (Environment) Loading

```
claudiadeluna in ~/ac3_intro_to_jinja/exercise3 on main
% uv run j2_env_loader_examples.py

--- Loading from STRING ---
Hello World!
Available templates: (not applicable for string-based templates)

--- Loading from FILESYSTEM ---
Available templates: ['simple_template.j2', 'test_named.j2']
Hello from Filesystem!
This is the simple_template.j2 rendered by Jinja2.

--- Loading from PACKAGE ---
Available templates: ['__init__.py', '__pycache__/__init__.cpython-311.pyc', 'simple_template.j2']
Hello from Package!
This is the simple_template.j2 rendered by Jinja2.

--- Loading from INDIVIDUAL FILE PATH ---
Available templates: (not applicable for direct file loading)
Hello from Anywhere!
This is the simple_template.j2 rendered by Jinja2.
claudiadeluna in ~/ac3_intro_to_jinja/exercise3 on main
%
```

PAYLOAD DATA

Passing Data to Templates

Getting the data into your template and rendering can be confusing at first.

By the time you are ready to render your template, your payload must be in the data structure expected by the template.

Render with Data

Your Template gets the data in the rendering step.

```
rendered_template_output = template_obj.render(cfg=payload_dict)
```

As the complexity of your payload increases, so will your template complexity.

```
[ ] import jinja2 as j2
```

Define the Template (in a string in this case)

```
[ ] tmp_str = "I am in {{ city }}"
```

Initialize the Jinja2 Template Object in Memory

```
[ ] t = j2.Template(tmp_str)
```

Pass the Data to the Template object and render

```
[ ] result = t.render(city="Denver")
```

```
[ ] print(result)
```

```
I am in Denver
```

```
[5] import jinja2 as j2
```

Define the Template (in a string in this case)

```
[6] tmp_str = "payload passed to string template is {{ payload }}"
```

Initialize the Jinja2 Template Object in Memory

```
[7] t = j2.Template(tmp_str)
```

Pass the Data to the Template object and render

```
[13] payload_from_python = list(range(0,5))
```

```
[14] result = t.render(payload=payload_from_python)
```

```
[15] print(result)
```

```
payload passed to string template is [0, 1, 2, 3, 4]
```

Python Unpacking Behavior **

This works with lists and dictionaries in Python
When working with Jinja2 focus on putting your payload in dictionaries so you can be very explicit about picking out the exact data you want because you can referenc either by name.

`fruits` (as dictionary)	** Becomes → **	Regular function arguments
`{ "fruit1": "mango", ... }`	`**fruits`	`fruit1="mango", fruit2="strawberry"` etc.

Jinja 2 Unpacking Behavior

```
template.render(cfg_data) # equivalent to 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”}`
{{ hostname }} in the template will render to **my_switch**
{{ cfg_data["hostname"] }} in the template will not be defined

Jinja 2 "Named" Data

```
template.render(cfg=cfg_data)
```

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 (vs. key/value pairs).

```
Given a dictionary {"fruit": "mango"}
It is the difference between accessing your data as a dictionary
{{ cfg['fruit']}}
or accessing your data as a top level variable
{{ fruit }}
```

Jinja 2 Unpacking Behavior

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

```
{{ cfg }}
```

Reccomendation

The ability to send your entire payload in a variable (dictionary) is very useful, so I generally always assign my payload a variable:

script variable **"cfg_data"** passed to template variable **"cfg"**

You can make the template variable short to minimize typing.

```
template.render(cfg=cfg_data)
```

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'}
```

Exercise 4 - Data

uv run payload_named_vs_unpacked.py

```
% uv run payload_named_vs_unpacked.py
dict_keys(['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 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 }}

Rendered Template:
! 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
! In this case I know Im passing a dictionary with the keys 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'}

================================================================================

----- Passing named variable -----
! Passing Named Payload
{{ cfg }}


Rendered Template:
! 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'}}
```

WHITE SPACE

Spaces and Line Feeds can be Tricky in Jinja2

Control structures can add unwanted whitespace and empty lines.

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
  - + and -
  - Environment knobs (Trim and Lstrip)

Whitespace control with "-"

The minus sign can be used to control whitespace

```
{%- ... %}  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
```

Whitespace control with "+"

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

```
{%+ ... %} Preserve indentation
```

Whitespace control during Environment defintion

```
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

Exercise 5 - Whitespace

Using an on line editor, lets play around with white space

REAL WORLD EXAMPLE

Lets put it all together

Lets generate switch configurations for an MLAG pair along with summary text which can be included in a Change Request ticket.

1. Template
2. Payload
3. Python script

Exercise 6 - Use Case

```
% cd exercise6

% uv run gen_sw_configs.py

================================================================================
Generating Switch Configs and Change Request Ticket Summary

There are 3 templates available in the FileSystemLoader environment ['templates']:
        - switch_config.j2
        - switch_summary.j2
        - switch_summary_oneliner.j2

--------------------------------------------------------------------------------
Generated config for switch-01 at ./output_configs/switch-01_config.txt
Generated config for switch-02 at ./output_configs/switch-02_config.txt

Completed generating configs for 2 switches.

--------------------------------------------------------------------------------
Ticket Summary:

Switch Deployment Summary:
- Hostname: switch-01
- Management IP: 192.168.1.10/24


Switch Deployment Summary:
- Hostname: switch-02
- Management IP: 192.168.1.11/24


================================================================================
```

Conclusion

- Any text is fair game for Jinja2
- Don't limit yourself to configurations
- Once you set up your little "template" factory your focus will be on the data and it's template.
- Re-use your environment and rendering steps (turn them into functions!)

** Hope to see you in Prague where the fun really begins!**

EXTRAS

Creating Templates with AI

This is an area where "AI" (next word prediction at scale) can be a time saver if you know your fundamentals.

So:
- don’t be afraid to tell your AI buddy they are wrong!
- double check and test, test, test.
- this is like “verbal” coding so be precise in your prompts

Great on line resources to get us started

Recommended

https://j2live.ttl255.com/
https://nebula.packetcoders.io/j2-render/