Upgrade to Ansible 12/ansible-core 2.19 and above

Ansible Style Guide

CfgMgmtCamp Ghent 2026
03.02.2026

Kostiantyn Volenbovskyi
Abacus Umantis AG

Agenda

Upgrade to Ansible 12 and above (ansible-core 2.19 and above)

Ansible Style Guide

- Variables
- Tasks
- (some) Modules
- Jinja, filters, plugins

Installation of Ansible

Option 1: Your Linux distribution is likely to have Ansible

- It is likely to be outdated (even after e.g. `dnf update` ☺)

Option 2: Installation via `pip` (sometimes `pipx`) in a Python virtual environment

- Should be analysed considering Python version on Ansible control node and especially on managed node**s**

Option 2.1 Multiple virtual environments depending on Managed nodes?

Option 3. Ansible Execution Environments

Ansible versions overview (February 2026)

Ansible community has certain ansible-core version as dependency		Requirements			Notes
Ansible community package version	**ansible-core package version**	Control node Python	Managed node Python	Managed node PowerShell	
14	**2.21**	3.12 - 3.14 (* I guess)	3.9 - 3.14 (* I guess)	5.1 (I guess)	Not released, to be released in June 2026
13	**2.20**	3.12 - 3.14	3.9 - 3.14	5.1	
12	**2.19**	3.11 - 3.13	3.8 - 3.13	5.1	
11	**2.18**	3.11 - 3.13	3.8 - 3.13	5.1	EOL in December 2025
10	**2.17**	3.10 - 3.12	3.7 - 3.12	5.1	EOL
9	**2.16**	3.10 - 3.12	2.7 / 3.6 - 3.12	3 - 5.1	EOL Last version compatible with CentOS 7 and AlmaLinux 8, Debian 10 (and SLES12?)
8	**2.15**	3.9 - 3.11	2.7 / 3.5 - 3.11	3 - 5.1	EOL

In previous Ansible versions

- `| succeeded` is not supported anymore, use `is succeeded`
- `include:` is not supported anymore, use include_role (or import_role)
- `pre_tasks`/`post_tasks` in playbooks are not necessary anymore, just use tasks with mix of tasks and e.g. `import_role`
- `is truthy/is falsy` was implemented in Ansible 2.10

Ansible 12 upgrade/Ansible-core 2.19 upgrade

😀 As with any newer Ansible versions: compatibility with newer collections (newer modules/new modules…)

😀 Faster templating leads to faster execution of certain tasks

😀 Better error reporting

😢 Forces to implement conditionals in a certain way

Workaround that will work till ansible-core 2.23: `ALLOW_BROKEN_CONDITIONALS=true` in ansible.cfg

Check:

```
grep "Broken conditionals are currently allowed because" * -B 8 <directory with Ansible logs>*.log
```

😢 Evaluates variables in task attributes even in skipped tasks, eg. `delegate_to`.

😢 Evaluates variables in `import_tasks` that were skipped (?)

😢 Existing code must be changed

- Deprecation is not «quick» when you update often but it is «quick» when you update Ansible rarely
- 1-2 changes are needed per year in «the whole codebase». Sometimes via e.g. «sed» but not always…

😢 Certain improvements are not improvements but regressions ?

Ansible 12 upgrade: conditionals

No more support of conditionals like

(`vip_ip_address` is a string)

~~`when: vip_ip_address`~~

What needs to be done?

- Variables in conditionals (`when:`) should contain one of the following:
  - Strings: `is truthy/is falsy;` in this case `when: vip_ip_address is truthy`
  - Lists/dictionaries: `|length > 0`
  - Booleans or strings that can be converted to booleans: `| bool`.

    (it is not necessary for actual Booleans but it is recommended so that it less risk of confusion)
  - … or there is Ansible test («in», «is») : this results in a Boolean

Conditionals: you will need |bool

Below will result in a string, no matter spelling of true/false etc..

You should use |bool

```
input_verification_phase1: >-
  {%- if trg_solution |d('') is truthy and trg_solution_id|d(0)|int is truthy -%}
  true
  {%- elif src_solution_id |d(0)|int is truthy -%}
  true
  {%- else -%}
  false
  {%- endif -%}
```

Ansible 12 upgrade: delegate_to

- Sometimes a «virtual host» is used
  - «virtual»: it is not in an inventory; nor it is added via `add_host`
  - it is sometimes part of a conditional
- What needs to be done:
  - It can’t be an empty value ☹, it shouldn’t be false
  - These variables in `delegate_to` should then have:
    - `|d (omit)`
    - Or «fake», non-empty string (for example "void_host")

Ansible 12 upgrade: undefined variables as values of task attributes

For example: value of delegate_to can never be undefined variable in all tasks (including skipped ones):

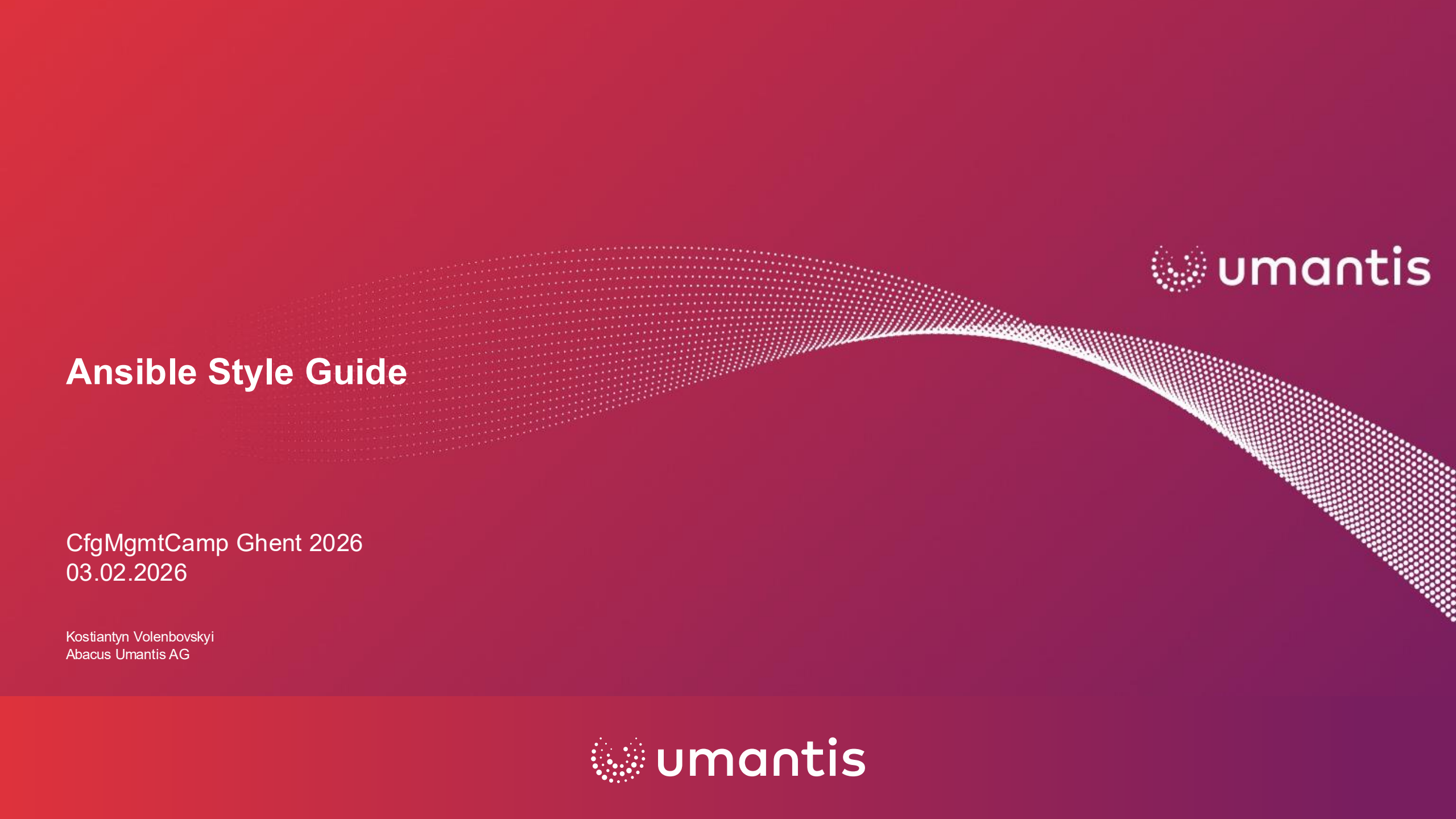
```
[ERROR]: Task failed: 'new_ca_renewalmaster' is undefined

Task failed.
Origin: /repo/roles/umantis.infra_ipaserver/tasks/changefreeipamaster.yml:1:3

1 - name: Change CA renewal master
    ^ column 3

<<< caused by >>>

'new_ca_renewalmaster' is undefined
Origin: home/repo/roles/umantis.infra_ipaserver/tasks/changefreeipamaster.yml:3:16

1 - name: Change CA renewal master
2   shell: "ipa config-mod --ca-renewal-master-server {{ new_ca_renewalmaster }}"
3   delegate_to: "{{ new_ca_renewalmaster }}"
                 ^ column 16

fatal: [freeipahost.domain.com -> {{ new_ca_renewalmaster }}]: FAILED! =>
    changed: false
    msg: 'Task failed: ''new_ca_renewalmaster'' is undefined'
```

References

https://forum.ansible.com/t/python-3-7-impact-on-el8-future-for-el9/6229

https://docs.ansible.com/projects/ansible/latest/reference_appendices/release_and_maintenance.html#support-life

https://docs.adfinis.com/ansible-guide/



Ansible Style Guide

CfgMgmtCamp Ghent 2026
03.02.2026

Kostiantyn Volenbovskyi
Abacus Umantis AG

umantis

Acknowledgements/credits

Thanks to:

- My team: Site Reliability Team of Ansible Umantis

- Ansible core team

- Red Hat team that maintains Automation Good Practices;
- Companies that created their Style Guides: LinuxFabrik; Whitecloud
- Style Guide and other Ansible-related work: Thomas Jungbauer; Felix Fontein, Thomas C. Foulds, George Shuklin; Andreas Sommer; Vladimir Botka; Jeff Geerling…

ABACUS umantis

ABACUS

HR Software for DACH (Germany, Austria, Switzerland)

- Applicant Management/Employee Management, On-/Off-boarding, Expense Management….
- Salary module

Leader in ERP- Software in Switzerland

- The largest independent SW vendor of Business Software for SME, founded in 1985
- Software for Finance, HR, Administration, Production, etc.
- 65‘000 happy customers

Objectives

- To have an Ansible Style guide that:
  - leads to code that is written faster without compromises on quality - understood (results in less cognitive load)/maintained/debugged/reused easier…
  - doesn’t suggest the code/code style that introduces a fault into managed node

also:

- Ansible code that has reasonable performance

(Somewhat) contentious questions

Contentious question	Alternative 1 (preferred by me)	Alternative 2
1. Dictionary notation	With dots (possible in 95% of cases)	With square brackets
2. Prefixing variables	Prefixes with role name so variable name in unique in the codebase	Additional Prefixes like: Internal : double underscore _ Registered variable: r_
3. Format of inventory	INI	YAML
4. Usage of quotes	Only if necessary	All strings are to be quoted
5. Quotes option	- External quotes: "" - Internal quotes: '' - Additional quotes – use YAML multiline - >-	- Single quotes - ''
6. Verb being used as task name	Verbs like: “Install” “Create” “Configure”	“Ensure”

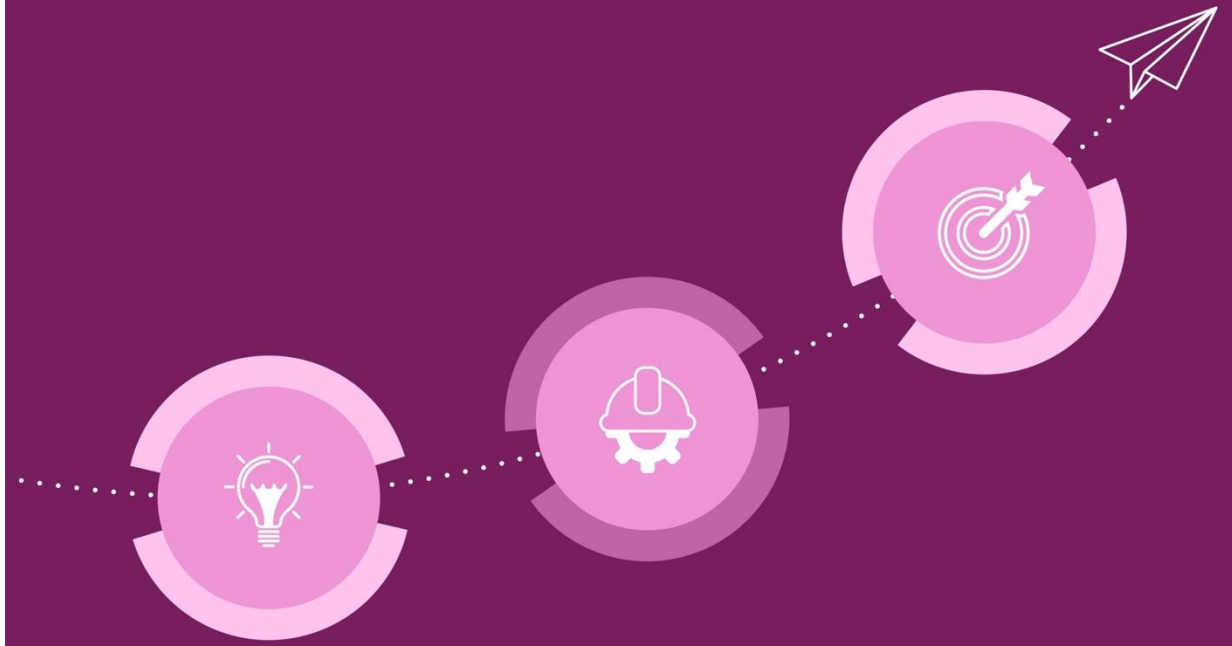
Types of variables

- String
- Integer
- Floats (fairly rare)
- List
  - Typically you use element from the list, like [0], [1]
- Boolean: `true` and `false` (other options are possible though!)
- Dictionaries

```
mydictionary:
  name: rene
  lastname: magritte
```

- List of dictionaries (it is a list but there are dictionaries)

```
mylistofdictionaries:
 - name: rene
   lastname: magritte
 - name: paul
   lastname: van Haver
   stagename: stromae
```



Handling of conditional/boolean values

VAR-7

- Instead of:

```
myapp_upgrade_needed |d(True) == True
```

use:

```
myapp_upgrade_needed |bool
```

Instead of:

```
myapp_upgrade_needed |d(false) == False
```

use

```
not upgrade_needed |d(true)|bool
```

Mixing strings and None: truthy/falsy

VAR-8

- Always use `else` in Jinja conditionals
- You might choose not to put anything after `else` : it will result in None and typically it should be be fine (or you should implement e.g. assert)

```
peer_host: >-
  {%- if other_datacenter |bool -%}
  webproxy01.{{ peer_dns_zone }}
  {%- else -%}
  {%- endif -%}
```

`| bool` : only strings "1", "on", "yes", and "true" (case insensitive), or the numeral 1 return true. So you should use: `when: peer_host is truthy`

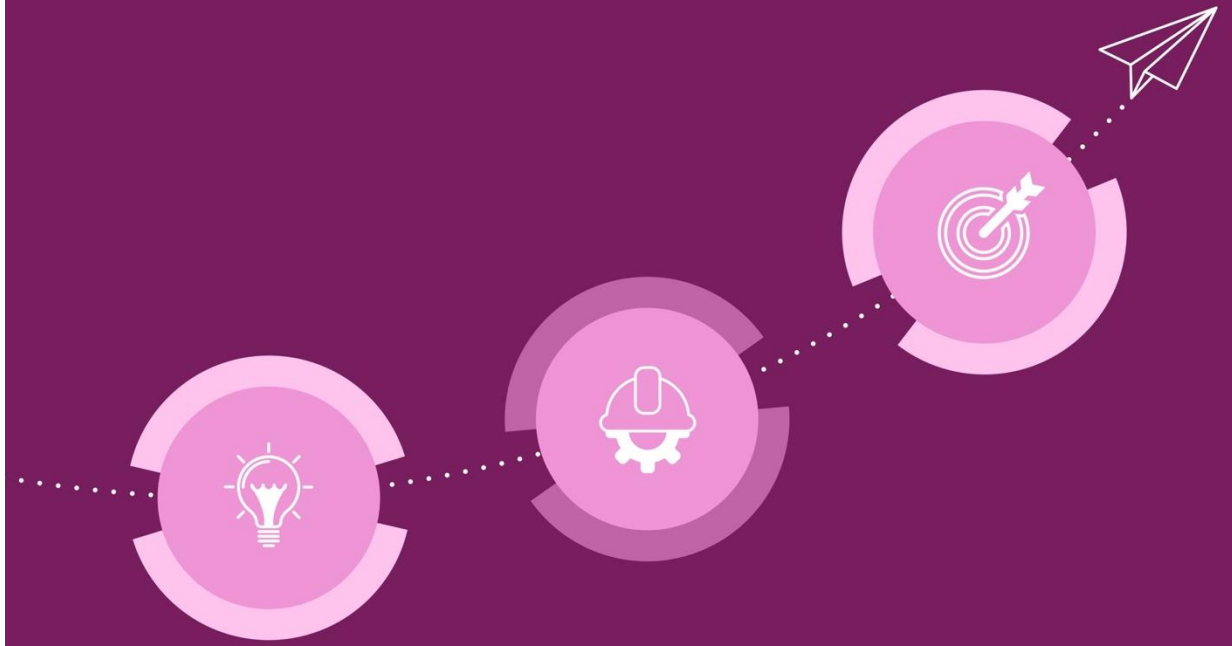
Variables: Jinja

Jinja2 allows to implement:

- Templates:
  - **MODULE-5**
  - templates should be preferred to ‘files’, they are much more future-proof

In templates or in variables:

- Cleaner conditionals (**if/elif/else)** for variables
- Loops using `for` (much faster than **loop:** and typically would give more features; but it is more complex**)**
- Changing values of variables



Conditionals

```
    # Simpler, one-liner:
    vm_net_bridge_name: "{{ 'virtmgmt' if vm_net == 'infra_mgmt' and virt_node |bool else vm_net }}”

    vault_installation_required: "{{ true if installed_vault_version.stdout |d('') != vault_version else false }}”
```

```
    # More complex: Jinja
src_tenant: >-
  {%- if src_tenant_stable.rc == 0 and src_tenant_beta.rc == 1 -%}
  {{ src_tenant_stable }}
  {%- elif src_tenant_stable.rc == 1 and src_tenant_beta.rc == 0 -%}
  {{ src_tenant_beta }}
  {%- else -%}
  {%- endif -%}
```

Integers vs strings

- Integers:
  - implement comparison and arithmetic operations

- In case you don’t need arithmetics:
  - You can’t easily specify integer via extra-vars, so sometimes you use `|int` to cast to integer

Defining default values/ |default filter

VAR-9

- Default value for boolean: `|d(false)` (sometimes you will end up using `|d(true))`
- Default value for a string: `|d('')`
- Default value for list `|d([])`
- Default value for dictionary `|d({})`

List of dictionaries

- **VAR-10**
- Recommended structure of variables that are more complex: **list of dictionaries**
- It is fairly easy to implement many things in list of dictionaries (in `configuration_properties` there are keys with name `value` :
  - loop: just specify `{{ configuration_properties }}` then use `{{ item.value }}`
  - Retrieval of a value: `selectattr`
  - Create list of certain values: `map`
  - More complex: write Jinja

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/subelements_lookup.html

https://docs.ansible.com/ansible/latest/playbook_guide/complex_data_manipulation.html

Nested loops/subelements

- **VAR-11**
- «One variable»: there is a need to have a **list inside a dictionary** that is inside of a list of dictionaries and iterate through that
  - Use: `| subelements` filter
- «Two variables»: there is a need to have **separate** list that should be executed for every element in the first list
  - For example there are 4 databases (`postgres_databases`) and for each one you need to enable 4 Postgres extensions (`postgres_extensions`)
  - Use: `with_nested`

Defining default values/ |default filter

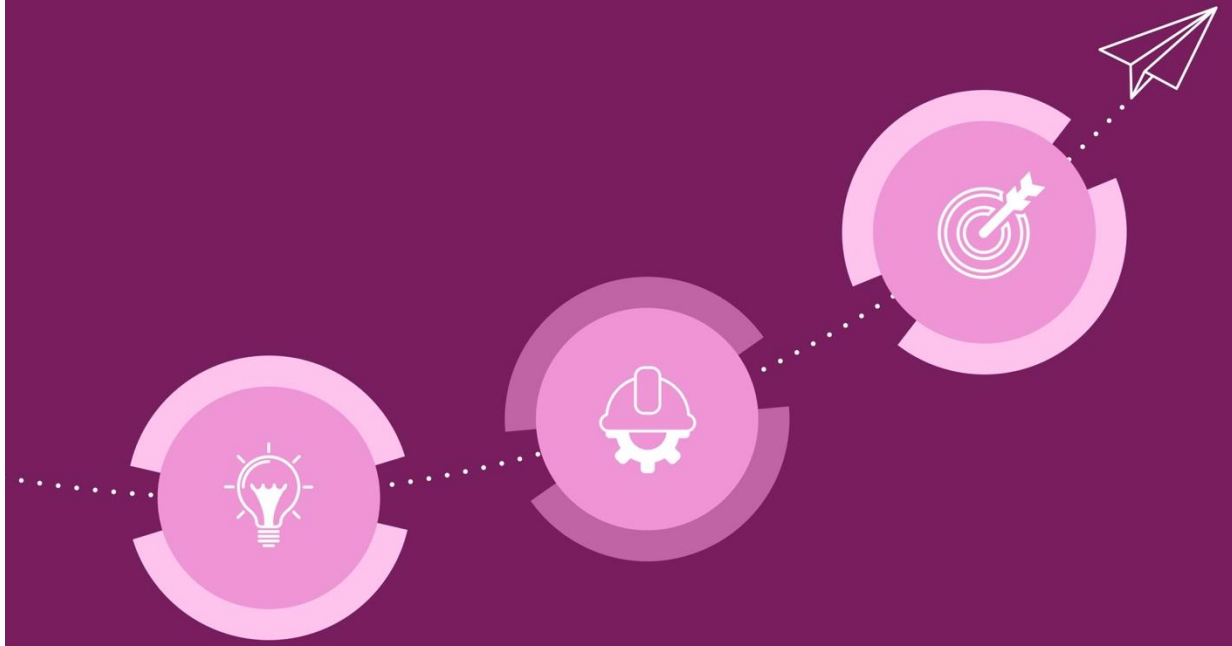
- **VAR-12**
  - **Option 1 recommended** Try to initialize variables in `defaults/main.yml`
    - And then remove «`is defined`» statements; no more «variable not defined» errors
    - No need to use `|d` everywhere
- **Option 2**
  - Use |d a lot

|d will **always** be used on variables coming from «register» statements

Variable definition 101

- `defaults/main.yml`
  - could be overriden by `group_vars` and/or `extra_vars`
- `vars/main.yml`
  - Those are **constants:** they have priority over defaults/group_vars. Please do not override them via `extra vars`
- Then there are:
  - `group_vars` (but remember about DRY and ‘all’ group)
  - Inventory variables
  - `set_fact`

And other places – but do you need them?



vars vs. defaults; to_nice_yaml

- **VAR-13**
- Variable used by a single role
  - Typically: `defaults/main.yml` or `vars/main.yml;`
- Variable used by several roles:
  - Variable used by **two (or 3)** roles: `include_vars`
  - When it is used more than
    - Alternative 1: `group_vars/all/: particular when roles have nothing to do with each other`
    - Alternative 2: more `include_vars`

`vars/main.yml` vs. `defaults/main.yml`

- In case it is overridden by `group_vars/extra_vars`: then it should be in `defaults/main.yml`
  - In case you consider that sometime in future this configuration might change because of SW version change: put it to `defaults/main.yml and not in vars/main.yml`

Additional files. In case, for example variables that contain different messages:

`vars/communication.yml`

It won't be »sourced» automatically, you will need `include_vars`

Use cases of set_fact

Typical use cases:

- «finalizing» the variable, eg. a password
- «propagating» variable to other hosts (`set_fact` with `delegate_facts`)
- it is done in a loop (but maybe you could use Jinja?), `include_tasks` implemented via a loop

Overwrite variable «dynamically»:

- in the beginning of the execution variable has certain value, but you need to get it changed (or you overwrite Ansible built-in variable)

A pitfall, static nature of import_tasks:

set_fact and **when:** you can't use `import_tasks`, you should use `include_tasks`

Recommendations

VAR-13

- A lot of `set_fact` usage can be avoided
  - typically to be replaced by `vars/main.yml`

VAR-14

- Avoid ‘variable undefined’ error by going through checklist:
  - Does a variable have a sensible default?
    - Use `defaults/main.yml or other location`
    - Empty value is sometimes fine but sometimes you need to avoid that («assert») gets into e.g. as script parameter unintentionally
  - Is the variable a «runtime» one (`register`)? Then use **|d** filter

Extra-vars vs. tags

- Extra-vars

VAR-15:

- Extra vars should not be mandatory for Day0/Day1 playbooks

VAR-16:

- Extra vars must be documented, typically should be verified by `assert/fail`

- Tags
  - Shorten the execution time/show clearly which tasks are executed/speed up troubleshooting
  - **TASKS-6**
  - There should be no changes in outcomes of «execute with all tags» vs. «execute with a specific tag»

Debugging

When writing play, use tags and debug statements:

- Tags:
  - Some of the tags are useful in normal operation (less time for executing some exact task)
  - Tags are very useful in troubleshooting
  - Some of the tags are useful also in the maintenance of Ansible code

- Debug

Two main alternatives:

```
debug:
  msg: "{{ variable_to_debug }}”
```

- Typically leave them in code via `verbosity: 1`

```
pause:
  prompt: "{{variable_to_debug }} "
```

Use **to_nice_yaml** filter in most of cases to make nicer output of various eg. dictionaries/lists within eg. «debug» module

Meta subdirectory

- In case you publish to Galaxy: OK!

- Otherwise look at your `meta/main.yml`:
  - Does anyone read/update it?
    - It is not used automatically by most of Ansible functionality
  - Do not use dependencies there, use `import_role`

TASKS2

meta/ directory can be removed completely in most cases

Blocks

TASKS-3

Main use case:

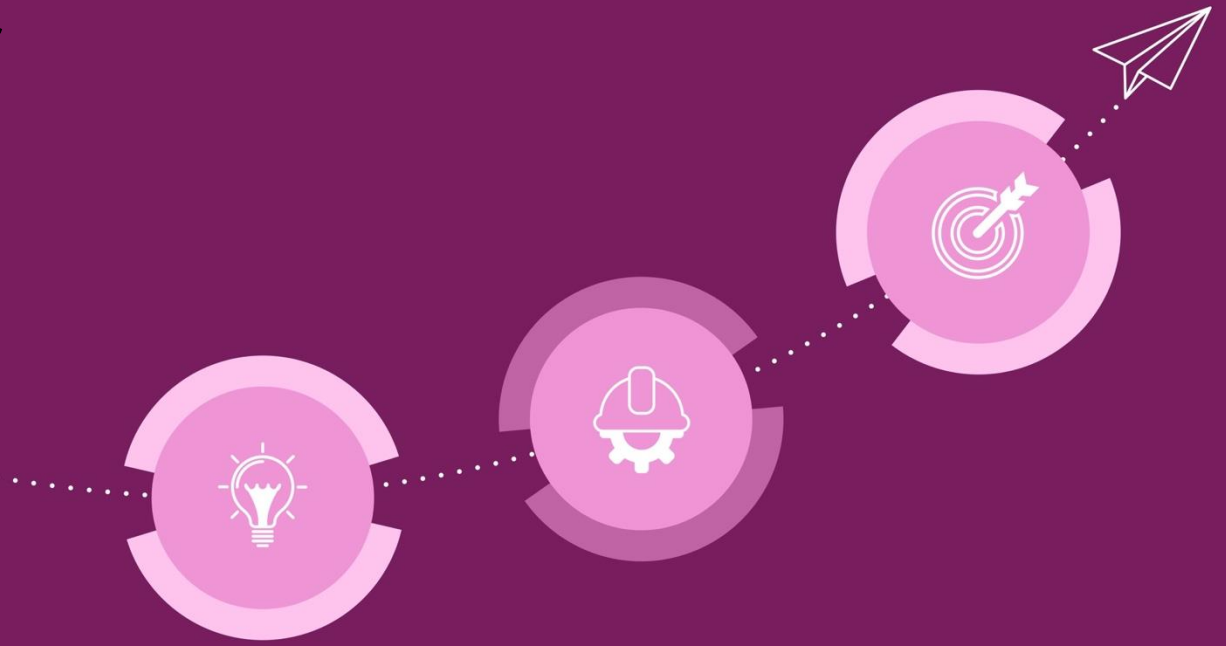
- You must use `block` when you would like to use `rescue:`

Other use cases:

- 2 or 3 tasks with common task attributes: maybe block is better, up to you!
- More than 3 tasks: consider using separate file and `import_tasks`

import_tasks/include_tasks

- **TASKS-4** Prefer `import_tasks` over `include_tasks` in case possible
- `import_tasks` is good because it can decrease the amount of same values for certain attributes over and over, attributes like:
  - `become:`
  - `become_user:`
  - `run_once:`
  - `delegate_to:`
  - `tags:`



Assert vs. fail

MODULE-6

- `ansible.builtin.assert/ansible.builtin.fail` modules implement «fail early» principle and provide user-friendly failure handling
  - One of the main parameters is information that you give within: it is meant to be used by user executing Ansible
  - Any playbook with more complicated input or somewhat «unreliable output» will benefit from `assert/fail`
- For multiple conditions `assert:` seems better than `fail`

Assert

MODULE-7

- `ansible.builtin.assert`

You might skip using `fail_msg` in case name of the task describes the condition

MODULE-3

- In case there are multiple conditions:
  - make sure that they are on separate lines and make sure that `fail_msg` contains text that covers both conditions!

Ansible filters vs. Jinja built-in methods

- Ansible filters
  - **Pipe** notation
- Ansible tests
- Jinja built-in methods
  - Very similar to Ansible filters
  - **Dot** notation

Lookup plugin(s)

- Always executed on Ansible control node (!)
- Functionality of various lookup plugins:
  - Generate random password
  - Check file contents
  - Find files that match the pattern (`find`)
  - Execute some Linux command (so that output of command is in Ansible variable)
    - For example `date` or `dig`

MODULE-4

An example of the value for variable below: 2024-08-16T13:42:45.906+0200

current_date_full: "{{ lookup('pipe','date +%Y-%m-%dT%H:%M:%S.%3N%z') }}”

- Lookup value of variable so that it is passed by value instead of passed by reference

Comparison/search/grep

FILTER_TEST_JINJA-1

- Equal to `==` ; not equal to: `!=` ; Comparison: `>` and `<` , typically with casting to **integer (| int)**
- Check if particular element is a element in a list or substring is part of a string:

  `in`

- Search is done via Ansible **test:**

```
release_jira_description |lower is search('secondary')
```

- regex_search/regex_findall: Ansible filters that extract part of line
  - `k8stool_download.dest | regex_search('.*\.(gz|zip|bz2|xz)$')`
  - `found_rpms_base: "{{ found_issues.meta.issues[0].fields.description | regex_findall('application-(?:package1| package2)_(?:beta|stable).*?\\.rpm')}}"`

Checking if file exists/read file

Ansible Control Node:

FILTER_TEST_JINJA-2

Check if file exists in Ansible control node:

```
hosts_path: /etc/hosts
…
 debug:
     msg: "host file exists"
 when: hosts_path is exists
```

Read file contents , file is on Ansible control node:

```
lookup/'file’:
info_new_dc: "{{ lookup('file', 'somedirec
tory/example.yml') | from_yaml |
d([]) }}"
```

Managed Node:

MODULE-5

Checking if file exists in managed node

- `ansible.builtin.stat`
  - `exists` dictionary key with boolean value

Read file contents, file is on managed node: `slurp`

References

RedHat Automation Good Practices:

https://redhat-cop.github.io/automation-good-practices/#_introduction

Linuxfabrik's Ansible Development Guidelines:

https://github.com/Linuxfabrik/lfops/blob/main/CONTRIBUTING.rst

Ansible Style Guide: Preparation Document, by Thomas Jungbauer

https://github.com/tjungbauer/ansible-style-guide/blob/main/AnsibleStyleGuide.adoc

https://docs.adfinis.com/ansible-guide/

25 Tips for Using Ansible in Large Projects by Thomas C.Foulds

https://thomascfoulds.com/2021/09/29/25-tips-for-using-ansible-in-large-projects.html

Adfinis Ansible Guide:

https://docs.adfinis.com/ansible-guide/

Ansible best practices by Andreas Sommer:

https://andidog.de/blog/2017-04-24-ansible-best-practices

References (2)

Ansible Best Practices by George Shuklin:

https://www.slideshare.net/slideshow/best-practices-for-ansible/95475433#3