We're updating the Ansible community mission statement! Participate in our survey and let us know - What does Ansible mean to you? (https://www.surveymonkey.co.uk/r/DLG9FJN)

You are reading the **latest** (stable) community version of the Ansible documentation. If you are a Red Hat customer, refer to the Ansible Automation Platform Life Cycle (https://access.redhat.com/support/policy/updates/ansible-automation-platform) page for subscription details.

Using Variables

Ansible uses variables to manage differences between systems. With Ansible, you can execute tasks and playbooks on multiple different systems with a single command. To represent the variations among those different systems, you can create variables with standard YAML syntax, including lists and dictionaries. You can define these variables in your playbooks, in your inventory (../inventory_guide/intro_inventory.html#intro-inventory), in re-usable files (playbooks_reuse.html#playbooks-reuse) or roles (playbooks_reuse_roles.html#playbooks-reuse-roles), or at the command line. You can also create variables during a playbook run by registering the return value or values of a task as a new variable.

After you create variables, either by defining them in a file, passing them at the command line, or registering the return value or values of a task as a new variable, you can use those variables in module arguments, in conditional "when" statements (playbooks_conditionals.html#playbooks-conditionals), in templates (playbooks_templating.html#playbooks-templating), and in loops (playbooks_loops.html#playbooks-loops). The ansible-examples github repository (https://github.com/ansible/ansible-examples) contains many examples of using variables in Ansible.

Once you understand the concepts and examples on this page, read about Ansible facts (playbooks_vars_facts.html#vars-and-facts), which are variables you retrieve from remote systems.

Search this site

- Creating valid variable names

Creating valid variable names

Not all strings are valid Ansible variable names. A variable name can only include letters, numbers, and underscores. Python keywords (https://docs.python.org/3/reference/lexical_analysis.html#keywords) or playbook keywords (../reference_appendices/playbooks_keywords.html#playbook-keywords) are not valid variable names. A variable name cannot begin with a number.

Variable names can begin with an underscore. In many programming languages, variables that begin with an underscore are private. This is not true in Ansible. Variables that begin with an underscore are treated exactly the same as any other variable. Do not rely on this convention for privacy or security.

This table gives examples of valid and invalid variable names:

Search this site

Valid variable names	Not valid
`foo`	`*foo` , Python keywords (https://docs.python.org/3/reference/lexical_analy and `lambda`
`foo_env`	playbook keywords (../reference_appendices/playbooks_keywords.html#pla
`foo_port`	`foo-port` , `foo port` , `foo.port`
`foo5` , `_foo`	`5foo` , `12`

Simple variables

Simple variables combine a variable name with a single value. You can use this syntax (and the syntax for lists and dictionaries shown below) in a variety of places. For details about setting variables in inventory, in playbooks, in reusable files, in roles, or at the command line, see Where to set variables.

Defining simple variables

You can define a simple variable using standard YAML syntax. For example:

```
remote_install_path: /opt/my_app_config
```

Referencing simple variables

After you define a variable, use Jinja2 syntax to reference it. Jinja2 variables use double curly braces. For example, the expression `My amp goes to {{ max_amp_value }}` demonstrates the most basic form of variable substitution. You can use Jinja2 syntax in playbooks. For example:

```
ansible.builtin.template:
  src: foo.cfg.j2
  dest: '{{ remote_install_path }}/foo.cfg'
```

In this example, the variable defines the location of a file, which can vary from one system to another.

Note

Ansible allows Jinja2 loops and conditionals in templates (playbooks_templating.html#playbooks-templating) but not in playbooks. You cannot create a loop of tasks. Ansible playbooks are pure machine-parseable YAML.

Search this site

When to quote variables (a YAML gotcha)

If you start a value with `{{ foo }}`, you must quote the whole expression to create valid YAML syntax. If you do not quote the whole expression, the YAML parser cannot interpret the syntax - it might be a variable or it might be the start of a YAML dictionary. For guidance on writing YAML, see the YAML Syntax (../reference_appendices/YAMLSyntax.html#yaml-syntax) documentation.

If you use a variable without quotes like this:

```
- hosts: app_servers
  vars:
      app_path: {{ base_path }}/22
```

You will see: `ERROR! Syntax Error while loading YAML.` If you add quotes, Ansible works correctly:

```
- hosts: app_servers
  vars:
       app_path: "{{ base_path }}/22"
```

Boolean variables

Ansible accepts a broad range of values for boolean variables: `true/false`, `1/0`, `yes/no`, `True/False` and so on. The matching of valid strings is case insensitive. While documentation examples focus on `true/false` to be compatible with `ansible-lint` default settings, you can use any of the following:

Valid values	Description
`True` , `'true'` , `'t'` , `'yes'` , `'y'` , `'on'` , `'1'` , `1` , `1.0`	Truthy values
`False` , `'false'` , `'f'` , `'no'` , `'n'` , `'off'` , `'0'` , `0` , `0.0`	Falsy values

List variables

A list variable combines a variable name with multiple values. The multiple values can be stored as an itemized list or in square brackets `[]`, separated with commas.

Defining variables as lists

You can define variables with multiple values using YAML lists. For example:

Search this site

```
region:
  - northeast
  - southeast
  - midwest
```

Referencing list variables

When you use variables defined as a list (also called an array), you can use individual, specific fields from that list. The first item in a list is item 0, the second item is item 1. For example:

```
region: "{{ region[0] }}"
```

The value of this expression would be “northeast”.

Dictionary variables

A dictionary stores the data in key-value pairs. Usually, dictionaries are used to store related data, such as the information contained in an ID or a user profile.

Defining variables as key:value dictionaries

You can define more complex variables using YAML dictionaries. A YAML dictionary maps keys to values. For example:

```
foo:
  field1: one
  field2: two
```

Referencing key:value dictionary variables

When you use variables defined as a key:value dictionary (also called a hash), you can use individual, specific fields from that dictionary using either bracket notation or dot notation:

```
foo['field1']
foo.field1
```

Both of these examples reference the same value (“one”). Bracket notation always works. Dot notation can cause problems because some keys collide with attributes and methods of python dictionaries. Use bracket notation if you use keys which start and end with two underscores (which are reserved for special meanings in python) or are any of the known public attributes:

Search this site

`add` , `append` , `as_integer_ratio` , `bit_length` , `capitalize` , `center` , `clear` , `conjugate` , `copy` , `count` , `decode` , `denominator` , `difference` , `difference_update` , `discard` , `encode` , `endswith` , `expandtabs` , `extend` , `find` , `format` , `fromhex` , `fromkeys` , `get` , `has_key` , `hex` , `imag` , `index` , `insert` , `intersection` , `intersection_update` , `isalnum` , `isalpha` , `isdecimal` , `isdigit` , `isdisjoint` , `is_integer` , `islower` , `isnumeric` , `isspace` , `issubset` , `issuperset` , `istitle` , `isupper` , `items` , `iteritems` , `iterkeys` , `itervalues` , `join` , `keys` , `ljust` , `lower` , `lstrip` , `numerator` , `partition` , `pop` , `popitem` , `real` , `remove` , `replace` , `reverse` , `rfind` , `rindex` , `rjust` , `rpartition` , `rsplit` , `rstrip` , `setdefault` , `sort` , `split` , `splitlines` , `startswith` , `strip` , `swapcase` , `symmetric_difference` , `symmetric_difference_update` , `title` , `translate` , `union` , `update` , `upper` , `values` , `viewitems` , `viewkeys` , `viewvalues` , `zfill` .

Registering variables

You can create variables from the output of an Ansible task with the task keyword `register`. You can use registered variables in any later tasks in your play. For example:

```yaml
- hosts: web_servers

  tasks:

     - name: Run a shell command and register its output as a variable
       ansible.builtin.shell: /usr/bin/foo
       register: foo_result
       ignore_errors: true

     - name: Run a shell command using output of the previous task
       ansible.builtin.shell: /usr/bin/bar
       when: foo_result.rc == 5
```

For more examples of using registered variables in conditions on later tasks, see Conditionals (playbooks_conditionals.html#playbooks-conditionals). Registered variables may be simple variables, list variables, dictionary variables, or complex nested data structures. The documentation for each module includes a `RETURN` section describing the return values for that module. To see the values for a particular task, run your playbook with `-v` .

Registered variables are stored in memory. You cannot cache registered variables for use in future playbook runs. Registered variables are only valid on the host for the rest of the current playbook run, including subsequent plays within the same playbook run.

Registered variables are host-level variables. When you register a variable in a task with a loop, the registered variable contains a value for each item in the loop. The data structure placed in the variable during the loop will contain a `results` attribute, that is a list of all responses from the module. For a more in-depth example of how this works, see the Loops (playbooks_loops.html#playbooks-loops) section on using register with a loop.

Search this site

Note

If a task fails or is skipped, Ansible still registers a variable with a failure or skipped status, unless the task is skipped based on tags. See Tags (playbooks_tags.html#tags) for information on adding and using tags.

Referencing nested variables

Many registered variables (and facts (playbooks_vars_facts.html#vars-and-facts)) are nested YAML or JSON data structures. You cannot access values from these nested data structures with the simple `{{ foo }}` syntax. You must use either bracket notation or dot notation. For example, to reference an IP address from your facts using the bracket notation:

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

To reference an IP address from your facts using the dot notation:

```
{{ ansible_facts.eth0.ipv4.address }}
```

Transforming variables with Jinja2 filters

Jinja2 filters let you transform the value of a variable within a template expression. For example, the `capitalize` filter capitalizes any value passed to it; the `to_yaml` and `to_json` filters change the format of your variable values. Jinja2 includes many built-in filters (https://jinja.palletsprojects.com/templates/#builtin-filters) and Ansible supplies many more filters. To find more examples of filters, see Using filters to manipulate data (playbooks_filters.html#playbooks-filters).

Where to set variables

You can define variables in a variety of places, such as in inventory, in playbooks, in reusable files, in roles, and at the command line. Ansible loads every possible variable it finds, then chooses the variable to apply based on variable precedence rules.

Defining variables in inventory

You can define different variables for each individual host, or set shared variables for a group of hosts in your inventory. For example, if all machines in the `[Boston]` group use 'boston.ntp.example.com' as an NTP server, you can set a group variable. The How to build your inventory (../inventory_guide/intro_inventory.html#intro-inventory) page has details on setting host variables (../inventory_guide/intro_inventory.html#host-variables) and group variables (../inventory_guide/intro_inventory.html#group-variables) in inventory.

Defining variables in a play

You can define variables directly in a playbook play:

```
- hosts: webservers
  vars:
    http_port: 80
```

When you define variables in a play, they are only visible to tasks executed in that play.

Defining variables in included files and roles

You can define variables in reusable variables files and/or in reusable roles. When you define variables in reusable variable files, the sensitive variables are separated from playbooks. This separation enables you to store your playbooks in a source control software and even share the playbooks, without the risk of exposing passwords or other sensitive and personal data. For information about creating reusable files and roles, see Re-using Ansible artifacts (playbooks_reuse.html#playbooks-reuse).

This example shows how you can include variables defined in an external file:

```
---

- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /vars/external_vars.yml

  tasks:

  - name: This is just a placeholder
    ansible.builtin.command: /bin/echo foo
```

The contents of each variables file is a simple YAML dictionary. For example:

```
---
# in the above example, this would be vars/external_vars.yml
somevar: somevalue
password: magic
```

Search this site

Note

You can keep per-host and per-group variables in similar files. To learn about organizing your variables, see Organizing host and group variables (../inventory_guide/intro_inventory.html#splitting-out-vars).

Defining variables at runtime

You can define variables when you run your playbook by passing variables at the command line using the `--extra-vars` (or `-e`) argument. You can also request user input with a `vars_prompt` (see Interactive input: prompts (playbooks_prompts.html#playbooks-prompts)). When you pass variables at the command line, use a single quoted string, that contains one or more variables, in one of the formats below.

key=value format

Values passed in using the `key=value` syntax are interpreted as strings. Use the JSON format if you need to pass non-string values such as Booleans, integers, floats, lists, and so on.

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

JSON string format

```
ansible-playbook release.yml --extra-vars
'{"version":"1.23.45","other_variable":"foo"}'
ansible-playbook arcade.yml --extra-vars '{"pacman":"mrs","ghosts":
["inky","pinky","clyde","sue"]}'
```

When passing variables with `--extra-vars`, you must escape quotes and other special characters appropriately for both your markup (for example, JSON), and for your shell:

```
ansible-playbook arcade.yml --extra-vars "{\"name\":\"Conan O\'Brien\"}"
ansible-playbook arcade.yml --extra-vars '{"name":"Conan O'\\\''Brien"}'
ansible-playbook script.yml --extra-vars "{\"dialog\":\"He said \\\"I just can\'t get
enough of those single and double-quotes"\!"\\\"\"}"
```

vars from a JSON or YAML file

If you have a lot of special characters, use a JSON or YAML file containing the variable definitions. Prepend both JSON and YAML filenames with @.

Search this site

```
ansible-playbook release.yml --extra-vars "@some_file.json"
ansible-playbook release.yml --extra-vars "@some_file.yaml"
```

Variable precedence: Where should I put a variable?

You can set multiple variables with the same name in many different places. When you do this, Ansible loads every possible variable it finds, then chooses the variable to apply based on variable precedence. In other words, the different variables will override each other in a certain order.

Teams and projects that agree on guidelines for defining variables (where to define certain types of variables) usually avoid variable precedence concerns. We suggest that you define each variable in one place: figure out where to define a variable, and keep it simple. For examples, see Tips on where to set variables.

Some behavioral parameters that you can set in variables you can also set in Ansible configuration, as command-line options, and using playbook keywords. For example, you can define the user Ansible uses to connect to remote devices as a variable with `ansible_user`, in a configuration file with `DEFAULT_REMOTE_USER`, as a command-line option with `-u`, and with the playbook keyword `remote_user`. If you define the same parameter in a variable and by another method, the variable overrides the other setting. This approach allows host-specific settings to override more general settings. For examples and more details on the precedence of these various settings, see Controlling how Ansible behaves: precedence rules (../reference_appendices/general_precedence.html#general-precedence-rules).

Understanding variable precedence

Ansible does apply variable precedence, and you might have a use for it. Here is the order of precedence from least to greatest (the last listed variables override all other variables):

Search this site

1. command line values (for example, `-u my_user`, these are not variables)
2. role defaults (defined in role/defaults/main.yml) [1]
3. inventory file or script group vars [2]
4. inventory group_vars/all [3]
5. playbook group_vars/all [3]
6. inventory group_vars/* [3]
7. playbook group_vars/* [3]
8. inventory file or script host vars [2]
9. inventory host_vars/* [3]
10. playbook host_vars/* [3]
11. host facts / cached set_facts [4]
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in role/vars/main.yml)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts / registered vars
20. role (and include_role) params
21. include params
22. extra vars (for example, `-e "user=my_user"`)(always win precedence)

In general, Ansible gives precedence to variables that were defined more recently, more actively, and with more explicit scope. Variables in the defaults folder inside a role are easily overridden. Anything in the vars directory of the role overrides previous versions of that variable in the namespace. Host and/or inventory variables override role defaults, but explicit includes such as the vars directory or an `include_vars` task override inventory variables.

Ansible merges different variables set in inventory so that more specific settings override more generic settings. For example, `ansible_ssh_user` specified as a group_var is overridden by `ansible_user` specified as a host_var. For details about the precedence of variables set in inventory, see How variables are merged (../inventory_guide/intro_inventory.html#how-we-merge).

Footnotes

[1] Tasks in each role see their own role's defaults. Tasks defined outside of a role see the last role's defaults.

[2] *(1,2)* Variables defined in inventory file or provided by dynamic inventory.

[3] *(1,2,3,4,5,6)* Includes vars added by 'vars plugins' as well as host_vars and group_vars which are added by the default vars plugin shipped with Ansible.

[4] When created with set_facts's cacheable option, variables have the high precedence in the play, but are the same as a host facts precedence when they come from the cache.

Search this site

❗ Note

Within any section, redefining a var overrides the previous instance. If multiple groups have the same variable, the last one loaded wins. If you define a variable twice in a play's `vars:` section, the second one wins.

❗ Note

The previous describes the default config `hash_behaviour=replace`, switch to `merge` to only partially overwrite.

Scoping variables

You can decide where to set a variable based on the scope you want that value to have. Ansible has three main scopes:

- Global: this is set by config, environment variables and the command line
- Play: each play and contained structures, vars entries (vars; vars_files; vars_prompt), role defaults and vars.
- Host: variables directly associated to a host, like inventory, include_vars, facts or registered task outputs

Inside a template, you automatically have access to all variables that are in scope for a host, plus any registered variables, facts, and magic variables.

Tips on where to set variables

You should choose where to define a variable based on the kind of control you might want over values.

Set variables in inventory that deal with geography or behavior. Since groups are frequently the entity that maps roles onto hosts, you can often set variables on the group instead of defining them on a role. Remember: child groups override parent groups, and host variables override group variables. See Defining variables in inventory for details on setting host and group variables.

Set common defaults in a `group_vars/all` file. See Organizing host and group variables (../inventory_guide/intro_inventory.html#splitting-out-vars) for details on how to organize host and group variables in your inventory. Group variables are generally placed alongside your inventory file, but they can also be returned by dynamic inventory (see Working with dynamic inventory (../inventory_guide/intro_dynamic_inventory.html#intro-dynamic-inventory)) or defined in AWX or on Red Hat Ansible Automation Platform (../reference_appendices/tower.html#ansible-platform) from the UI or API:

Search this site

```
---
# file: /etc/ansible/group_vars/all
# this is the site wide default
ntp_server: default-time.example.com
```

Set location-specific variables in `group_vars/my_location` files. All groups are children of the `all` group, so variables set here override those set in `group_vars/all`:

```
---
# file: /etc/ansible/group_vars/boston
ntp_server: boston-time.example.com
```

If one host used a different NTP server, you could set that in a host_vars file, which would override the group variable:

```
---
# file: /etc/ansible/host_vars/xyz.boston.example.com
ntp_server: override.example.com
```

Set defaults in roles to avoid undefined-variable errors. If you share your roles, other users can rely on the reasonable defaults you added in the `roles/x/defaults/main.yml` file, or they can easily override those values in inventory or at the command line. See Roles (playbooks_reuse_roles.html#playbooks-reuse-roles) for more info. For example:

```
---
# file: roles/x/defaults/main.yml
# if no other value is supplied in inventory or as a parameter, this value will be used
http_port: 80
```

Set variables in roles to ensure a value is used in that role, and is not overridden by inventory variables. If you are not sharing your role with others, you can define app-specific behaviors like ports this way, in `roles/x/vars/main.yml`. If you are sharing roles with others, putting variables here makes them harder to override, although they still can by passing a parameter to the role or setting a variable with `-e`:

```
---
# file: roles/x/vars/main.yml
# this will absolutely be used in this role
http_port: 80
```

Search this site

Pass variables as parameters when you call roles for maximum clarity, flexibility, and visibility. This approach overrides any defaults that exist for a role. For example:

```
roles:
   - role: apache
     vars:
        http_port: 8080
```

When you read this playbook it is clear that you have chosen to set a variable or override a default. You can also pass multiple values, which allows you to run the same role multiple times. See Running a role multiple times in one play (playbooks_reuse_roles.html#run-role-twice) for more details. For example:

```
roles:
   - role: app_user
     vars:
        myname: Ian
   - role: app_user
     vars:
       myname: Terry
   - role: app_user
     vars:
       myname: Graham
   - role: app_user
     vars:
       myname: John
```

Variables set in one role are available to later roles. You can set variables in a `roles/common_settings/vars/main.yml` file and use them in other roles and elsewhere in your playbook:

```
roles:
   - role: common_settings
   - role: something
     vars:
       foo: 12
   - role: something_else
```

Note

There are some protections in place to avoid the need to namespace variables. In this example, variables defined in 'common_settings' are available to 'something' and 'something_else' tasks, but tasks in 'something' have foo set at 12, even if 'common_settings' sets foo to 20.

Instead of worrying about variable precedence, we encourage you to think about how easily or how often you want to override a variable when deciding where to set it. If you are not sure what other variables are defined, and you need a particular value, use `--extra-vars` (`-e`) to override all other variables.

Using advanced variable syntax

For information about advanced YAML syntax used to declare variables and have more control over the data placed in YAML files used by Ansible, see Advanced playbook syntax (playbooks_advanced_syntax.html#playbooks-advanced-syntax).

See also

Ansible playbooks (playbooks_intro.html#about-playbooks)
An introduction to playbooks

Conditionals (playbooks_conditionals.html#playbooks-conditionals)
Conditional statements in playbooks

Using filters to manipulate data (playbooks_filters.html#playbooks-filters)
Jinja2 filters and their uses

Loops (playbooks_loops.html#playbooks-loops)
Looping in playbooks

Roles (playbooks_reuse_roles.html#playbooks-reuse-roles)
Playbook organization by roles

General tips (../tips_tricks/ansible_tips_tricks.html#tips-and-tricks)
Tips and tricks for playbooks

Special Variables (../reference_appendices/special_variables.html#special-variables)
List of special variables

User Mailing List (https://groups.google.com/group/ansible-devel)
Have a question? Stop by the google group!

Real-time chat (../community/communication.html#communication-irc)
How to join Ansible chat channels