## The When Statement

Sometimes you will want to skip a particular step on a particular host. This could be something as simple as not installing a certain package if the operating system is a particular version, or it could be something like performing some cleanup steps if a filesystem is getting full.

This is easy to do in Ansible with the when clause, which contains a raw Jinja2 expression without double curly braces (see Variables). It’s actually pretty simple:

tasks:

- name: "shut down Debian flavored systems"

command: /sbin/shutdown -t now

when: ansible\_os\_family == "Debian"

*# note that Ansible facts and vars like ansible\_os\_family can be used*

*# directly in conditionals without double curly braces*

You can also use parentheses to group conditions:

tasks:

- name: "shut down CentOS 6 and Debian 7 systems"

command: /sbin/shutdown -t now

when: (ansible\_distribution == "CentOS" and ansible\_distribution\_major\_version == "6") or

(ansible\_distribution == "Debian" and ansible\_distribution\_major\_version == "7")

Multiple conditions that all need to be true (a logical ‘and’) can also be specified as a list:

tasks:

- name: "shut down CentOS 6 systems"

command: /sbin/shutdown -t now

when:

- ansible\_distribution == "CentOS"

- ansible\_distribution\_major\_version == "6"

A number of Jinja2 “filters” can also be used in when statements, some of which are unique and provided by Ansible. Suppose we want to ignore the error of one statement and then decide to do something conditionally based on success or failure:

tasks:

- command: /bin/false

register: result

ignore\_errors: True

- command: /bin/something

when: result|failed

*# In older versions of ansible use |success, now both are valid but succeeded uses the correct tense.*

- command: /bin/something\_else

when: result|succeeded

- command: /bin/still/something\_else

when: result|skipped

**Note**

the filters have been updated in 2.1 so both success and succeeded work (fail/failed, etc).

Note that was a little bit of foreshadowing on the ‘register’ statement. We’ll get to it a bit later in this chapter.

As a reminder, to see what facts are available on a particular system, you can do:

ansible hostname.example.com -m setup

Tip: Sometimes you’ll get back a variable that’s a string and you’ll want to do a math operation comparison on it. You can do this like so:

tasks:

- shell: echo "only on Red Hat 6, derivatives, and later"

when: ansible\_os\_family == "RedHat" and ansible\_lsb.major\_release|int >= 6

**Note**

the above example requires the lsb\_release package on the target host in order to return the ansible\_lsb.major\_release fact.

Variables defined in the playbooks or inventory can also be used. An example may be the execution of a task based on a variable’s boolean value:

vars:

epic: true

Then a conditional execution might look like:

tasks:

- shell: echo "This certainly is epic!"

when: epic

or:

tasks:

- shell: echo "This certainly isn't epic!"

when: not epic

If a required variable has not been set, you can skip or fail using Jinja2’s defined test. For example:

tasks:

- shell: echo "I've got '**{{** foo **}}**' and am not afraid to use it!"

when: foo is defined

- fail: msg="Bailing out. this play requires 'bar'"

when: bar is undefined

This is especially useful in combination with the conditional import of vars files (see below). As the examples show, you don’t need to use {{ }} to use variables inside conditionals, as these are already implied.

## Loops and Conditionals

Combining when with with\_items (see Loops), be aware that the when statement is processed separately for each item. This is by design:

tasks:

- command: echo **{{** item **}}**

with\_items: [ 0, 2, 4, 6, 8, 10 ]

when: item > 5

If you need to skip the whole task depending on the loop variable being defined, used the |default filter to provide an empty iterator:

- command: echo **{{** item **}}**

with\_items: "**{{** mylist**|default([])** **}}**"

when: item > 5

If using with\_dict which does not take a list:

- command: echo **{{** item.key **}}**

with\_dict: "**{{** mydict**|default({})** **}}**"

when: item.value > 5

## Loading in Custom Facts

It’s also easy to provide your own facts if you want, which is covered in Developing Modules. To run them, just make a call to your own custom fact gathering module at the top of your list of tasks, and variables returned there will be accessible to future tasks:

tasks:

- name: gather site specific fact data

action: site\_facts

- command: /usr/bin/thingy

when: my\_custom\_fact\_just\_retrieved\_from\_the\_remote\_system == '1234'

## Applying ‘when’ to roles, imports, and includes

Note that if you have several tasks that all share the same conditional statement, you can affix the conditional to a task include statement as below. All the tasks get evaluated, but the conditional is applied to each and every task:

- import\_tasks: tasks/sometasks.yml

when: "'reticulating splines' in output"

**Note**

In versions prior to 2.0 this worked with task includes but not playbook includes. 2.0 allows it to work with both.

Or with a role:

- hosts: webservers

roles:

- { role: debian\_stock\_config, when: ansible\_os\_family == 'Debian' }

You will note a lot of ‘skipped’ output by default in Ansible when using this approach on systems that don’t match the criteria. Read up on the ‘group\_by’ module in the About Modules docs for a more streamlined way to accomplish the same thing.

When used with include\_\* tasks instead of imports, the conditional is applied \_only\_ to the include task itself and not any other tasks within the included file(s). A common situation where this distinction is important is as follows:

*# include a file to define a variable when it is not already defined*

*# main.yml*

- include\_tasks: other\_tasks.yml

when: x is not defined

*# other\_tasks.yml*

- set\_fact:

x: foo

- debug:

var: x

In the above example, if import\_tasks had been used instead both included tasks would have also been skipped. With include\_tasksinstead, the tasks are executed as expected because the conditional is not applied to them.

## Conditional Imports

**Note**

This is an advanced topic that is infrequently used. You can probably skip this section.

Sometimes you will want to do certain things differently in a playbook based on certain criteria. Having one playbook that works on multiple platforms and OS versions is a good example.

As an example, the name of the Apache package may be different between CentOS and Debian, but it is easily handled with a minimum of syntax in an Ansible Playbook:

---

- hosts: all

remote\_user: root

vars\_files:

- "vars/common.yml"

- [ "vars/**{{** ansible\_os\_family **}}**.yml", "vars/os\_defaults.yml" ]

tasks:

- name: make sure apache is started

service: name=**{{** apache **}}** state=started

**Note**

The variable ‘ansible\_os\_family’ is being interpolated into the list of filenames being defined for vars\_files.

As a reminder, the various YAML files contain just keys and values:

---

*# for vars/CentOS.yml*

apache: httpd

somethingelse: 42

How does this work? If the operating system was ‘CentOS’, the first file Ansible would try to import would be ‘vars/CentOS.yml’, followed by ‘/vars/os\_defaults.yml’ if that file did not exist. If no files in the list were found, an error would be raised. On Debian, it would instead first look towards ‘vars/Debian.yml’ instead of ‘vars/CentOS.yml’, before falling back on ‘vars/os\_defaults.yml’. Pretty simple.

To use this conditional import feature, you’ll need facter or ohai installed prior to running the playbook, but you can of course push this out with Ansible if you like:

*# for facter*

ansible -m yum -a "pkg=facter state=present"

ansible -m yum -a "pkg=ruby-json state=present"

# for ohai

ansible -m yum -a "pkg=ohai state=present"

Ansible’s approach to configuration – separating variables from tasks, keeps your playbooks from turning into arbitrary code with ugly nested ifs, conditionals, and so on - and results in more streamlined & auditable configuration rules – especially because there are a minimum of decision points to track.

## Selecting Files And Templates Based On Variables

**Note**

This is an advanced topic that is infrequently used. You can probably skip this section.

Sometimes a configuration file you want to copy, or a template you will use may depend on a variable. The following construct selects the first available file appropriate for the variables of a given host, which is often much cleaner than putting a lot of if conditionals in a template.

The following example shows how to template out a configuration file that was very different between, say, CentOS and Debian:

- name: template a file

template: src=**{{** item **}}** dest=/etc/myapp/foo.conf

with\_first\_found:

- files:

- **{{** ansible\_distribution **}}**.conf

- default.conf

paths:

- search\_location\_one/somedir/

- /opt/other\_location/somedir/

## Register Variables

Often in a playbook it may be useful to store the result of a given command in a variable and access it later. Use of the command module in this way can in many ways eliminate the need to write site specific facts, for instance, you could test for the existence of a particular program.

The ‘register’ keyword decides what variable to save a result in. The resulting variables can be used in templates, action lines, or whenstatements. It looks like this (in an obviously trivial example):

- name: test play

hosts: all

tasks:

- shell: cat /etc/motd

register: motd\_contents

- shell: echo "motd contains the word hi"

when: motd\_contents.stdout.find('hi') != -1

As shown previously, the registered variable’s string contents are accessible with the ‘stdout’ value. The registered result can be used in the “with\_items” of a task if it is converted into a list (or already is a list) as shown below. “stdout\_lines” is already available on the object as well though you could also call “home\_dirs.stdout.split()” if you wanted, and could split by other fields:

- name: registered variable usage as a with\_items list

hosts: all

tasks:

- name: retrieve the list of home directories

command: ls /home

register: home\_dirs

- name: add home dirs to the backup spooler

file: path=/mnt/bkspool/**{{** item **}}** src=/home/**{{** item **}}** state=link

with\_items: "**{{** home\_dirs.stdout\_lines **}}**"

*# same as with\_items: "***{{** home\_dirs.stdout.split**()** **}}***"*

As shown previously, the registered variable’s string contents are accessible with the ‘stdout’ value. You may check the registered variable’s string contents for emptiness:

- name: check registered variable for emptiness

hosts: all

tasks:

- name: list contents of directory

command: ls mydir

register: contents

- name: check contents for emptiness

debug: msg="Directory is empty"

when: contents.stdout == ""

**Standard Loops**

To save some typing, repeated tasks can be written in short-hand like so:

- name: add several users

user:

name: "**{{** item **}}**"

state: present

groups: "wheel"

with\_items:

- testuser1

- testuser2

If you have defined a YAML list in a variables file, or the ‘vars’ section, you can also do:

with\_items: "**{{** somelist **}}**"

The above would be the equivalent of:

- name: add user testuser1

user:

name: "testuser1"

state: present

groups: "wheel"

- name: add user testuser2

user:

name: "testuser2"

state: present

groups: "wheel"

The yum and apt modules use with\_items to execute fewer package manager transactions.

Note that the types of items you iterate over with ‘with\_items’ do not have to be simple lists of strings. If you have a list of hashes, you can reference subkeys using things like:

- name: add several users

user:

name: "**{{** item.name **}}**"

state: present

groups: "**{{** item.groups **}}**"

with\_items:

- { name: 'testuser1', groups: 'wheel' }

- { name: 'testuser2', groups: 'root' }

Also be aware that when combining *when* with *with\_items* (or any other loop statement), the *when* statement is processed separately for each item. See The When Statement for an example.

Loops are actually a combination of things *with\_* + *lookup()*, so any lookup plugin can be used as a source for a loop, ‘items’ is lookup.

Please note that with\_items flattens the first depth of the list it is provided and can yield unexpected results if you pass a list which is composed of lists. You can work around this by wrapping your nested list inside a list:

*# This will run debug three times since the list is flattened*

- debug:

msg: "**{{** item **}}**"

vars:

nested\_list:

- - one

- two

- three

with\_items: "**{{** nested\_list **}}**"

*# This will run debug once with the three items*

- debug:

msg: "**{{** item **}}**"

vars:

nested\_list:

- - one

- two

- three

with\_items:

- "**{{** nested\_list **}}**"

**Nested Loops**

Loops can be nested as well:

- name: give users access to multiple databases

mysql\_user:

name: "**{{** item**[**0**]** **}}**"

priv: "**{{** item**[**1**]** **}}**.\*:ALL"

append\_privs: yes

password: "foo"

with\_nested:

- [ 'alice', 'bob' ]

- [ 'clientdb', 'employeedb', 'providerdb' ]

As with the case of ‘with\_items’ above, you can use previously defined variables.:

- name: here, 'users' contains the above list of employees

mysql\_user:

name: "**{{** item**[**0**]** **}}**"

priv: "**{{** item**[**1**]** **}}**.\*:ALL"

append\_privs: yes

password: "foo"

with\_nested:

- "**{{** users **}}**"

- [ 'clientdb', 'employeedb', 'providerdb' ]

**Looping over Hashes**

New in version 1.5.

Suppose you have the following variable:

---

users:

alice:

name: Alice Appleworth

telephone: 123-456-7890

bob:

name: Bob Bananarama

telephone: 987-654-3210

And you want to print every user’s name and phone number. You can loop through the elements of a hash using with\_dict like this:

tasks:

- name: Print phone records

debug:

msg: "User **{{** item.key **}}** is **{{** item.value.name **}}** (**{{** item.value.telephone **}}**)"

with\_dict: "**{{** users **}}**"

**Looping over Files**

with\_file iterates over the content of a list of files, *item* will be set to the content of each file in sequence. It can be used like this:

---

- hosts: all

tasks:

*# emit a debug message containing the content of each file.*

- debug:

msg: "**{{** item **}}**"

with\_file:

- first\_example\_file

- second\_example\_file

Assuming that first\_example\_file contained the text “hello” and second\_example\_file contained the text “world”, this would result in:

TASK [debug msg={{ item }}] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost] => (item=hello) => {

"item": "hello",

"msg": "hello"

}

ok: [localhost] => (item=world) => {

"item": "world",

"msg": "world"

}

**Looping over Fileglobs**

with\_fileglob matches all files in a single directory, non-recursively, that match a pattern. It calls Python’s glob library, and can be used like this:

---

- hosts: all

tasks:

*# first ensure our target directory exists*

- name: Ensure target directory exists

file:

dest: "/etc/fooapp"

state: directory

*# copy each file over that matches the given pattern*

- name: Copy each file over that matches the given pattern

copy:

src: "**{{** item **}}**"

dest: "/etc/fooapp/"

owner: "root"

mode: 0600

with\_fileglob:

- "/playbooks/files/fooapp/\*"

**Note**

When using a relative path with with\_fileglob in a role, Ansible resolves the path relative to the *roles/<rolename>/files* directory.

**Looping over Filetrees**

with\_filetree recursively matches all files in a directory tree, enabling you to template a complete tree of files on a target system while retaining permissions and ownership.

The filetree lookup-plugin supports directories, files and symlinks, including SELinux and other file properties. Here is a complete list of what each file object consists of:

* src
* root
* path
* mode
* state
* owner
* group
* seuser
* serole
* setype
* selevel
* uid
* gid
* size
* mtime
* ctime

If you provide more than one path, it will implement a with\_first\_found logic, and will not process entries it already processed in previous paths. This enables the user to merge different trees in order of importance, or add role\_vars specific paths to influence different instances of the same role.

Here is an example of how we use with\_filetree within a role. The web/ path is relative to either roles/<role>/files/ or files/:

---

- name: Create directories

file:

path: /web/**{{** item.path **}}**

state: directory

mode: '**{{** item.mode **}}**'

with\_filetree: web/

when: item.state == 'directory'

- name: Template files

template:

src: '**{{** item.src **}}**'

dest: /web/**{{** item.path **}}**

mode: '**{{** item.mode **}}**'

with\_filetree: web/

when: item.state == 'file'

- name: Recreate symlinks

file:

src: '**{{** item.src **}}**'

dest: /web/**{{** item.path **}}**

state: link

force: yes

mode: '**{{** item.mode **}}**'

with\_filetree: web/

when: item.state == 'link'

The following properties are also available:

* root: allows filtering by original location
* path: contains the relative path to root
* uidi, gid: force-create by exact id, rather than by name
* size, mtime, ctime: filter out files by size, mtime or ctime

**Looping over Parallel Sets of Data**

Suppose you have the following variable data:

---

alpha: [ 'a', 'b', 'c', 'd' ]

numbers: [ 1, 2, 3, 4 ]

...and you want the set of ‘(a, 1)’ and ‘(b, 2)’. Use ‘with\_together’ to get this:

tasks:

- debug:

msg: "**{{** item.0 **}}** and **{{** item.1 **}}**"

with\_together:

- "**{{** alpha **}}**"

- "**{{** numbers **}}**"

**Looping over Subelements**

Suppose you want to do something like loop over a list of users, creating them, and allowing them to login by a certain set of SSH keys.

In this example, we’ll assume you have the following defined and loaded in via “vars\_files” or maybe a “group\_vars/all” file:

---

users:

- name: alice

authorized:

- /tmp/alice/onekey.pub

- /tmp/alice/twokey.pub

mysql:

password: mysql-password

hosts:

- "%"

- "127.0.0.1"

- "::1"

- "localhost"

privs:

- "\*.\*:SELECT"

- "DB1.\*:ALL"

- name: bob

authorized:

- /tmp/bob/id\_rsa.pub

mysql:

password: other-mysql-password

hosts:

- "db1"

privs:

- "\*.\*:SELECT"

- "DB2.\*:ALL"

You could loop over these subelements like this:

- name: Create User

user:

name: "**{{** item.name **}}**"

state: present

generate\_ssh\_key: yes

with\_items:

- "**{{** users **}}**"

- name: Set authorized ssh key

authorized\_key:

user: "**{{** item.0.name **}}**"

key: "**{{** lookup**(**'file'**,** item.1**)** **}}**"

with\_subelements:

- "**{{** users **}}**"

- authorized

Given the mysql hosts and privs subkey lists, you can also iterate over a list in a nested subkey:

- name: Setup MySQL users

mysql\_user:

name: "**{{** item.0.name **}}**"

password: "**{{** item.0.mysql.password **}}**"

host: "**{{** item.1 **}}**"

priv: "**{{** item.0.mysql.privs **|** **join(**'/'**)** **}}**"

with\_subelements:

- "**{{** users **}}**"

- mysql.hosts

Subelements walks a list of hashes (aka dictionaries) and then traverses a list with a given (nested sub-)key inside of those records.

Optionally, you can add a third element to the subelements list, that holds a dictionary of flags. Currently you can add the ‘skip\_missing’ flag. If set to True, the lookup plugin will skip the lists items that do not contain the given subkey. Without this flag, or if that flag is set to False, the plugin will yield an error and complain about the missing subkey.

The authorized\_key pattern is exactly where it comes up most.

**Looping over Integer Sequences**

with\_sequence generates a sequence of items. You can specify a start value, an end value, an optional “stride” value that specifies the number of steps to increment the sequence, and an optional printf-style format string.

Arguments should be specified as key=value pair strings.

A simple shortcut form of the arguments string is also accepted: [start-]end[/stride][:format].

Numerical values can be specified in decimal, hexadecimal (0x3f8) or octal (0600). Negative numbers are not supported. This works as follows:

---

- hosts: all

tasks:

*# create groups*

- group:

name: "evens"

state: present

- group:

name: "odds"

state: present

*# create some test users*

- user:

name: "**{{** item **}}**"

state: present

groups: "evens"

with\_sequence: start=0 end=32 format=testuser%02x

*# create a series of directories with even numbers for some reason*

- file:

dest: "/var/stuff/**{{** item **}}**"

state: directory

with\_sequence: start=4 end=16 stride=2

*# a simpler way to use the sequence plugin*

*# create 4 groups*

- group:

name: "group**{{** item **}}**"

state: present

with\_sequence: count=4

**Random Choices**

The ‘random\_choice’ feature can be used to pick something at random. While it’s not a load balancer (there are modules for those), it can somewhat be used as a poor man’s load balancer in a MacGyver like situation:

- debug:

msg: "**{{** item **}}**"

with\_random\_choice:

- "go through the door"

- "drink from the goblet"

- "press the red button"

- "do nothing"

One of the provided strings will be selected at random.

At a more basic level, they can be used to add chaos and excitement to otherwise predictable automation environments.

**Do-Until Loops**

New in version 1.4.

Sometimes you would want to retry a task until a certain condition is met. Here’s an example:

- shell: /usr/bin/foo

register: result

until: result.stdout.find("all systems go") != -1

retries: 5

delay: 10

The above example run the shell module recursively till the module’s result has “all systems go” in its stdout or the task has been retried for 5 times with a delay of 10 seconds. The default value for “retries” is 3 and “delay” is 5.

The task returns the results returned by the last task run. The results of individual retries can be viewed by -vv option. The registered variable will also have a new key “attempts” which will have the number of the retries for the task.

**Note**

If the “until” parameter isn’t defined, the value for the “retries” parameter is forced to 1.

**Finding First Matched Files**

**Note**

This is an uncommon thing to want to do, but we’re documenting it for completeness. You probably won’t be reaching for this one often.

This isn’t exactly a loop, but it’s close. What if you want to use a reference to a file based on the first file found that matches a given criteria, and some of the filenames are determined by variable names? Yes, you can do that as follows:

- name: INTERFACES | Create Ansible header for /etc/network/interfaces

template:

src: "**{{** item **}}**"

dest: "/etc/foo.conf"

with\_first\_found:

- "**{{** ansible\_virtualization\_type **}}**\_foo.conf"

- "default\_foo.conf"

This tool also has a long form version that allows for configurable search paths. Here’s an example:

- name: some configuration template

template:

src: "**{{** item **}}**"

dest: "/etc/file.cfg"

mode: 0444

owner: "root"

group: "root"

with\_first\_found:

- files:

- "**{{** inventory\_hostname **}}**/etc/file.cfg"

paths:

- ../../../templates.overwrites

- ../../../templates

- files:

- etc/file.cfg

paths:

- templates

**Iterating Over The Results of a Program Execution**

**Note**

This is an uncommon thing to want to do, but we’re documenting it for completeness. You probably won’t be reaching for this one often.

Sometimes you might want to execute a program, and based on the output of that program, loop over the results of that line by line. Ansible provides a neat way to do that, though you should remember, this is always executed on the control machine, not the remote machine:

- name: Example of looping over a command result

shell: "/usr/bin/frobnicate **{{** item **}}**"

with\_lines:

- "/usr/bin/frobnications\_per\_host --param **{{** inventory\_hostname **}}**"

Ok, that was a bit arbitrary. In fact, if you’re doing something that is inventory related you might just want to write a dynamic inventory source instead (see Dynamic Inventory), but this can be occasionally useful in quick-and-dirty implementations.

Should you ever need to execute a command remotely, you would not use the above method. Instead do this:

- name: Example of looping over a REMOTE command result

shell: "/usr/bin/something"

register: command\_result

- name: Do something with each result

shell: "/usr/bin/something\_else --param **{{** item **}}**"

with\_items:

- "**{{** command\_result.stdout\_lines **}}**"

**Looping Over A List With An Index**

**Note**

This is an uncommon thing to want to do, but we’re documenting it for completeness. You probably won’t be reaching for this one often.

New in version 1.3.

If you want to loop over an array and also get the numeric index of where you are in the array as you go, you can also do that. It’s uncommonly used:

- name: indexed loop demo

debug:

msg: "at array position **{{** item.0 **}}** there is a value **{{** item.1 **}}**"

with\_indexed\_items:

- "**{{** some\_list **}}**"

**Using ini file with a loop**

New in version 2.0.

The ini plugin can use regexp to retrieve a set of keys. As a consequence, we can loop over this set. Here is the ini file we’ll use:

**[section1]**

value1**=**section1/value1

value2**=**section1/value2

**[section2]**

value1**=**section2/value1

value2**=**section2/value2

Here is an example of using with\_ini:

- debug:

msg: "**{{** item **}}**"

with\_ini:

- value[1-2]

- section: section1

- file: "lookup.ini"

- re: true

And here is the returned value:

{

"changed": false,

"msg": "All items completed",

"results": [

{

"invocation": {

"module\_args": "msg=\"section1/value1\"",

"module\_name": "debug"

},

"item": "section1/value1",

"msg": "section1/value1",

"verbose\_always": true

},

{

"invocation": {

"module\_args": "msg=\"section1/value2\"",

"module\_name": "debug"

},

"item": "section1/value2",

"msg": "section1/value2",

"verbose\_always": true

}

]

}

**Flattening A List**

**Note**

This is an uncommon thing to want to do, but we’re documenting it for completeness. You probably won’t be reaching for this one often.

In rare instances you might have several lists of lists, and you just want to iterate over every item in all of those lists. Assume a really crazy hypothetical datastructure:

----

# file: roles/foo/vars/main.yml

packages\_base:

- [ 'foo-package', 'bar-package' ]

packages\_apps:

- [ ['one-package', 'two-package' ]]

- [ ['red-package'], ['blue-package']]

As you can see the formatting of packages in these lists is all over the place. How can we install all of the packages in both lists?:

- name: flattened loop demo

yum:

name: "**{{** item **}}**"

state: present

with\_flattened:

- "**{{** packages\_base **}}**"

- "**{{** packages\_apps **}}**"

That’s how!

**Using register with a loop**

After using register with a loop, the data structure placed in the variable will contain a results attribute that is a list of all responses from the module.

Here is an example of using register with with\_items:

- shell: "echo **{{** item **}}**"

with\_items:

- "one"

- "two"

register: echo

This differs from the data structure returned when using register without a loop:

{

"changed": true,

"msg": "All items completed",

"results": [

{

"changed": true,

"cmd": "echo \"one\" ",

"delta": "0:00:00.003110",

"end": "2013-12-19 12:00:05.187153",

"invocation": {

"module\_args": "echo \"one\"",

"module\_name": "shell"

},

"item": "one",

"rc": 0,

"start": "2013-12-19 12:00:05.184043",

"stderr": "",

"stdout": "one"

},

{

"changed": true,

"cmd": "echo \"two\" ",

"delta": "0:00:00.002920",

"end": "2013-12-19 12:00:05.245502",

"invocation": {

"module\_args": "echo \"two\"",

"module\_name": "shell"

},

"item": "two",

"rc": 0,

"start": "2013-12-19 12:00:05.242582",

"stderr": "",

"stdout": "two"

}

]

}

Subsequent loops over the registered variable to inspect the results may look like:

- name: Fail if return code is not 0

fail:

msg: "The command (**{{** item.cmd **}}**) did not have a 0 return code"

when: item.rc != 0

with\_items: "**{{** echo.results **}}**"

During iteration, the result of the current item will be placed in the variable:

- shell: echo "**{{** item **}}**"

with\_items:

- one

- two

register: echo

changed\_when: echo.stdout != "one"

**Looping over the inventory**

If you wish to loop over the inventory, or just a subset of it, there is multiple ways. One can use a regular with\_items with the ansible\_play\_batch or groups variables, like this:

*# show all the hosts in the inventory*

- debug:

msg: "**{{** item **}}**"

with\_items:

- "**{{** groups**[**'all'**]** **}}**"

*# show all the hosts in the current play*

- debug:

msg: "**{{** item **}}**"

with\_items:

- "**{{** ansible\_play\_batch **}}**"

There is also a specific lookup plugin inventory\_hostnames that can be used like this:

*# show all the hosts in the inventory*

- debug:

msg: "**{{** item **}}**"

with\_inventory\_hostnames:

- all

*# show all the hosts matching the pattern, ie all but the group www*

- debug:

msg: "**{{** item **}}**"

with\_inventory\_hostnames:

- all:!www

More information on the patterns can be found on Patterns

**Loop Control**

New in version 2.1.

In 2.0 you are again able to use *with\_* loops and task includes (but not playbook includes). This adds the ability to loop over the set of tasks in one shot. Ansible by default sets the loop variable *item* for each loop, which causes these nested loops to overwrite the value of *item* from the “outer” loops. As of Ansible 2.1, the *loop\_control* option can be used to specify the name of the variable to be used for the loop:

*# main.yml*

- include\_tasks: inner.yml

with\_items:

- 1

- 2

- 3

loop\_control:

loop\_var: outer\_item

*# inner.yml*

- debug:

msg: "outer item=**{{** outer\_item **}}** inner item=**{{** item **}}**"

with\_items:

- a

- b

- c

**Note**

If Ansible detects that the current loop is using a variable which has already been defined, it will raise an error to fail the task.

New in version 2.2.

When using complex data structures for looping the display might get a bit too “busy”, this is where the C(label) directive comes to help:

- name: create servers

digital\_ocean:

name: "**{{** item.name **}}**"

state: present

with\_items:

- name: server1

disks: 3gb

ram: 15Gb

network:

nic01: 100Gb

nic02: 10Gb

...

loop\_control:

label: "**{{**item.name**}}**"

This will now display just the ‘label’ field instead of the whole structure per ‘item’, it defaults to ‘”{{item}}”’ to display things as usual.

New in version 2.2.

Another option to loop control is C(pause), which allows you to control the time (in seconds) between execution of items in a task loop.:

*# main.yml*

- name: create servers, pause 3s before creating next

digital\_ocean:

name: "**{{** item **}}**"

state: present

with\_items:

- server1

- server2

loop\_control:

pause: 3

**Loops and Includes in 2.0**

Because *loop\_control* is not available in Ansible 2.0, when using an include with a loop you should use *set\_fact* to save the “outer” loops value for *item*:

*# main.yml*

- include\_tasks: inner.yml

with\_items:

- 1

- 2

- 3

*# inner.yml*

- set\_fact:

outer\_item: "**{{** item **}}**"

- debug:

msg: "outer item=**{{** outer\_item **}}** inner item=**{{** item **}}**"

with\_items:

- a

- b

- c