Loops

- Comparing loops
- Using loops
  - Iterating over a simple list
  - Iterating over a list of hashes
  - Iterating over a dictionary
  - Registering variables with a loop
  - Retrying a task until a condition is met
  - Looping over inventory
- Ensuring list input for loop: using query rather than lookup
- Adding controls to loops
  - Limiting loop output with label
  - Pausing within a loop
  - Breaking out of a loop
  - Tracking progress through a loop with index_var
  - Extended loop variables
  - Accessing the name of your loop_var
- Nested Loops
  - Iterating over nested lists
  - Stacking loops via include_tasks
  - Until and loop
- Migrating from with_X to loop
  - with_list
  - with_items
  - with_indexed_items
  - with_flattened
  - with_together
  - with_dict
  - with_sequence
  - with_subelements

- with_nested/with_cartesian
- with_random_choice

Ansible offers the `loop`, `with_<lookup>`, and `until` keywords to execute a task multiple times. Examples of commonly-used loops include changing ownership on several files and/or directories with the `file module <file_module>`, creating multiple users with the `user module <user_module>`, and repeating a polling step until a certain result is reached.

> * We added `loop` in Ansible 2.5. as a simpler way to do loops, but we recommend it for most use cases. * We have not deprecated the use of `with_<lookup>` - that syntax will still be valid for the foreseeable future. * `loop` and `with_<lookup>` are mutually exclusive. While it is possible to nest them under `until`, this affects each loop iteration.

Comparing loops

- The normal use case for `until` has to do with tasks that are likely to fail, while `loop` and `with_<lookup>` are meant for repeating tasks with slight variations.
- The `loop` and `with_<lookup>` will run the task once per item in the list used as input, while `until` will rerun the task until a condition is met. For programmers the former are "for loops" and the latter is a "while/until loop".
- The `with_<lookup>` keywords rely on `lookup_plugins` - even `items` is a lookup.
- The `loop` keyword is equivalent to `with_list`, and is the best choice for simple loops.
- The `loop` keyword will not accept a string as input, see `query_vs_lookup`.
- The `until` keyword accepts an 'end conditional' (expression that returns `True` or `False`) that is "implicitly templated" (no need for `{{ }}`), commonly based on the variable you `register` for the task.
- `loop_control` affects both `loop` and `with_<lookup>`, but not `until`, which has its own companion keywords: `retries` and `delay`.
- Generally speaking, any use of `with_*` covered in `migrating_to_loop` can be updated to use `loop`.
- Be careful when changing `with_items` to `loop`, as `with_items` performs implicit single-level flattening. You may need to use `| flatten(1)` with `loop` to match the exact outcome. For example, to get the same output as:

```
with_items:
  - 1
  - [2,3]
  - 4
```

you would need

```
loop: "{{ [1, [2, 3], 4] | flatten(1) }}"
```

- Any `with_*` statement that requires using `lookup` within a loop should not be converted to use the `loop` keyword. For example, instead of doing:

```
loop: "{{ lookup('fileglob', '*.txt', wantlist=True) }}"
```

it is cleaner to keep

```
with_fileglob: '*.txt'
```

Using loops

Iterating over a simple list

Repeated tasks can be written as standard loops over a simple list of strings. You can define the list directly in the task.

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
     - testuser1
     - testuser2
```

You can define the list in a variables file, or in the 'vars' section of your play, then refer to the name of the list in the task.

```
loop: "{{ somelist }}"
```

Either of these examples would be the equivalent of

```
- name: Add user testuser1
  ansible.builtin.user:
    name: "testuser1"
```

```
    state: present
    groups: "wheel"

- name: Add user testuser2
  ansible.builtin.user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

You can pass a list directly to a parameter for some plugins. Most of the packaging modules, like `yum <yum_module>` and `apt <apt_module>`, have this capability. When available, passing the list to a parameter is better than looping over the task. For example

```
- name: Optimal yum
  ansible.builtin.yum:
    name: "{{ list_of_packages }}"
    state: present

- name: Non-optimal yum, slower and may cause issues with interdependencies
  ansible.builtin.yum:
    name: "{{ item }}"
    state: present
  loop: "{{ list_of_packages }}"
```

Check the `module documentation <list_of_module_plugins>` to see if you can pass a list to any particular module's parameter(s).

Iterating over a list of hashes

If you have a list of hashes, you can reference subkeys in a loop. For example:

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

When combining `conditionals <playbooks_conditionals>` with a loop, the `when:` statement is processed separately for each item. See `the_when_statement` for examples.

Iterating over a dictionary

To loop over a dict, use the `dict2items <dict_filter>`:

```
- name: Using dict2items
  ansible.builtin.debug:
    msg: "{{ item.key }}: {{ item.value.ip_address }} {{ item.value.role }}"
  loop: "{{ server_configs | dict2items }}"
  vars:
    server_configs:
      web_01:
        ip_address: "10.1.1.50"
        role: "frontend"
      db_01:
        ip_address: "10.1.1.100"
        role: "backend_db"
```

This example iterates over server_configs and prints the key and selected nested fields.

If the values in the dictionary are themselves dictionaries (for example, each group maps to a dict containing a `gid`), remember that after applying `dict2items` each loop item has two attributes: `item.key` and `item.value`. Access nested fields via `item.value.<field>`.

Registering variables with a loop

You can register the output of a loop as a variable. For example

```
- name: Register loop output as a variable
  ansible.builtin.shell: "echo {{ item }}"
  loop:
    - "one"
    - "two"
  register: echo
```

When you use `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. This differs from the data structure returned when using `register` without a loop. The `changed`/`failed`/`skipped` attribute that's beside the `results` will represent the overall state. `changed`/`failed` will be true if at least one of the

iterations triggered a change/failed, while `skipped` will be true only if all iterations were skipped.


```
{
    "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
  ansible.builtin.fail:
    msg: "The command ({{ item.cmd }}) did not have a 0 return code"
  when: item.rc != 0
  loop: "{{ echo.results }}"
```

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

```
- name: Place the result of the current item in the variable
  ansible.builtin.shell: echo "{{ item }}"
  loop:
    - one
    - two
  register: echo
  changed_when: echo.stdout != "one"
```

2.21

You can also use register projections to access task results without registering a variable, or to register multiple variables at once. See `registered_variables` for details.

The result of the current item during iteration is also accessible in the `result` property of the implicit variable `_task`. This allows for accessing loop item results without registering a variable.

```
- name: Place the result of the current item in the variable
  ansible.builtin.shell: echo "{{ item }}"
  loop:
    - one
    - two
  changed_when: _task.result.stdout != "one"
```

For more details on register projections and the `_task` variable, see `registered_variables`.

To access the full, accumulated loop result list during iteration, the `loop_result` property of the `_task` implicit variable may be used.

```
- name: Run a loop and access individual item output
  ansible.builtin.shell: "{{ item }}"
  register:
    foo_output: _task.loop_result.results[0].stdout
    bar_output: _task.loop_result.results[1].stdout
  loop:
```

```
    - /usr/bin/foo
    - /usr/bin/bar
```

> Register projection expressions are evaluated after the task completes, so `default` is not needed when accessing `_task.loop_result` if the registered variables are only used after the task. However, if a registered projection variable will be used in a conditional during iteration, `default` must be used in the register expression itself because the variable does not exist on the first iteration.
>
> ```
> - name: Using registered projection in conditional during iteration
>   ansible.builtin.shell: "{{ item }}"
>   register:
>     first_output: _task.loop_result.results[0].stdout | default('')  # default needed to
> use first_output during iteration
>   loop: [1, 2, 3]
>   when: first_output != 'skip'
> ```
>
> See `playbooks_conditionals` for examples of using `_task` directly in conditionals.

To access the same functionality as name-only variable registration when registering multiple variables, the `polymorphic_result` property of the `_task` implicit variable can be used. During iteration, it will contain the result of the most recent loop iteration, and afterwards it will contain the full task result. This makes these two tasks equivalent:

```
- name: Run a loop and register a single variable
  ansible.builtin.shell: "{{ item }}"
  register: foo_bar_result
  loop:
    - /usr/bin/foo
    - /usr/bin/bar

- name: Register the same variable in the multi-variable register format
  ansible.builtin.shell: "{{ item }}"
  register:
    foo_bar_result: _task.polymorphic_result
  loop:
    - /usr/bin/foo
    - /usr/bin/bar
```

Retrying a task until a condition is met

1.4

You can use the `until` keyword to retry a task until a certain condition is met. Here's an example:

```
- name: Retry a task until a certain condition is met
  ansible.builtin.shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

This task runs up to 5 times with a delay of 10 seconds between each attempt. If the result of any attempt has "all systems go" in its stdout, the task succeeds. The default value for "retries" is 3 and "delay" is 5.

To see the results of individual retries, run the play with `-vv`.

When you run a task with `until` and register the result as a variable, the registered variable will include a key called "attempts", which records the number of retries for the task.

If `until` is not specified, the task will retry until the task succeeds but at most `retries` times (New in version 2.16).

You can combine the `until` keyword with `loop` or `with_<lookup>`. The result of the task for each element of the loop is registered in the variable and can be used in the `until` condition. Here is an example:

```
- name: Retry combined with a loop
  uri:
    url: "https://{{ item }}.ansible.com"
    method: GET
  register: uri_output
  with_items:
  - "galaxy"
  - "docs"
  - "forum"
  - "www"
  retries: 2
  delay: 1
  until: "uri_output.status == 200"
```

> When you use the `timeout` keyword in a loop, it applies to each attempt of the task action. See `TASK_TIMEOUT <TASK_TIMEOUT>` for more details.

You can use the implicit `_task` variable to access the current result in `until` without registering a variable:

```
- name: Wait for each service to be ready
  ansible.builtin.command: systemctl is-active {{ item }}
  loop:
    - nginx
    - postgresql
    - redis
  retries: 5
  delay: 2
  until: _task.result.rc == 0
```

`registered_variables`
For more information on the `_task` implicit variable and register projections.

Looping over inventory

Normally the play itself is a loop over your inventory, but sometimes you need a task to do the same over a different set of hosts. To loop over your inventory, or just a subset of it, you can use a regular `loop` with the `ansible_play_batch` or `groups` variables.

```
- name: Show all the hosts in the inventory
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ groups['all'] }}"

- name: Show all the hosts in the current play
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ ansible_play_batch }}"
```

There is also a specific lookup plugin `inventory_hostnames` that can be used like this

```
- name: Show all the hosts in the inventory
  ansible.builtin.debug:
    msg: "{{ item }}"
```

```
  loop: "{{ query('inventory_hostnames', 'all') }}"

- name: Show all the hosts matching the pattern, ie all but the group www
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ query('inventory_hostnames', 'all:!www') }}"
```

More information on the patterns can be found in `intro_patterns`.

Ensuring list input for `loop`: using `query` rather than `lookup`

The `loop` keyword requires a list as input, but the `lookup` keyword returns a string of comma-separated values by default. Ansible 2.5 introduced a new Jinja2 function named `query <query>` that always returns a list, offering a simpler interface and more predictable output from lookup plugins when using the `loop` keyword.

You can force `lookup` to return a list to `loop` by using `wantlist=True`, or you can use `query` instead.

The following two examples do the same thing.

```
loop: "{{ query('inventory_hostnames', 'all') }}"

loop: "{{ lookup('inventory_hostnames', 'all', wantlist=True) }}"
```

Adding controls to loops

2.1

The `loop_control` keyword lets you manage your loops in useful ways.

Limiting loop output with `label`

2.2

When looping over complex data structures, the console output of your task can be enormous. To limit the displayed output, use the `label` directive with `loop_control`.

```
- name: Create servers
  digital_ocean:
    name: "{{ item.name }}"
    state: present
  loop:
    - name: server1
      disks: 3gb
      ram: 15Gb
      network:
        nic01: 100Gb
        nic02: 10Gb
        # ...
  loop_control:
    label: "{{ item.name }}"
```

The output of this task will display just the `name` field for each `item` instead of the entire contents of the multi-line `{{ item }}` variable.

> This is for making console output more readable, not protecting sensitive data. If there is sensitive data in `loop`, set `no_log: true` on the task to prevent disclosure.

Pausing within a loop

2.2

To control the time (in seconds) between the execution of each item in a task loop, use the `pause` directive with `loop_control`.

```
# main.yml
- name: Create servers, pause 3s before creating next
  community.digitalocean.digital_ocean:
    name: "{{ item }}"
    state: present
  loop:
    - server1
    - server2
```

```
  loop_control:
    pause: 3
```

Breaking out of a loop

2.18

Use the `break_when` directive with `loop_control` to exit a loop after any item, based on Jinja2 expressions.

```
# main.yml
- name: Use set_fact in a loop until a condition is met
  vars:
    special_characters: "!@#$%^&*(),.?:{}|<>"
    character_set: "digits,ascii_letters,{{ special_characters }}"
    password_policy: '^(?=.*\d)(?=.*[A-Z])(?=.*[{{ special_characters | regex_escape
}}]).{12,}$'
  block:
    - name: Generate a password until it contains a digit, uppercase letter, and special
character (10 attempts)
      set_fact:
        password: "{{ lookup('password', '/dev/null', chars=character_set, length=12) }}"
      loop: "{{ range(0, 10) }}"
      loop_control:
        break_when:
          - password is match(password_policy)

    - fail:
        msg: "Maximum attempts to generate a valid password exceeded"
      when: password is not match(password_policy)
```

When using register projections, you can reference the registered variables in `break_when`:

```
- name: Break after processing two items
  ansible.builtin.debug:
    msg: "Processing {{ item }}"
  loop: [1, 2, 3, 4, 5]
  loop_control:
    break_when: items_processed == 2
```

```
  register:
    items_processed: _task.loop_result.results | length
```

Tracking progress through a loop with `index_var`

2.5

To keep track of where you are in a loop, use the `index_var` directive with `loop_control`. This directive specifies a variable name to contain the current loop index.

```
- name: Count our fruit
  ansible.builtin.debug:
    msg: "{{ item }} with index {{ my_idx }}"
  loop:
    - apple
    - banana
    - pear
  loop_control:
    index_var: my_idx
```

> index_var is 0 indexed.

Extended loop variables

2.8

As of Ansible 2.8, you can get extended loop information using the `extended` option to loop control. This option will expose the following information.

Variable	Description
`ansible_loop.allitems`	The list of all items in the loop
`ansible_loop.index`	The current iteration of the loop. (1 indexed)
`ansible_loop.index0`	The current iteration of the loop. (0 indexed)
`ansible_loop.revindex`	The number of iterations from the end of the loop (1 indexed)

`ansible_loop.revindex0`	The number of iterations from the end of the loop (0 indexed)
`ansible_loop.first`	`True` if first iteration
`ansible_loop.last`	`True` if last iteration
`ansible_loop.length`	The number of items in the loop
`ansible_loop.previtem`	The item from the previous iteration of the loop. Undefined during the first iteration.
`ansible_loop.nextitem`	The item from the following iteration of the loop. Undefined during the last iteration.

```
loop_control:
  extended: true
```

> When using `loop_control.extended` more memory will be utilized on the control node. This is a result of `ansible_loop.allitems` containing a reference to the full loop data for every loop. When serializing the results for display in callback plugins within the main ansible process, these references may be dereferenced causing memory usage to increase.

2.14

To disable the `ansible_loop.allitems` item, to reduce memory consumption, set `loop_control.extended_allitems: false`.

```
loop_control:
  extended: true
  extended_allitems: false
```

Accessing the name of your loop_var

2.8

As of Ansible 2.8, you can get the name of the value provided to `loop_control.loop_var` using the `ansible_loop_var` variable

For role authors, writing roles that allow loops, instead of dictating the required `loop_var` value, you can gather the value through the following

```
"{{ lookup('vars', ansible_loop_var) }}"
```

Nested Loops

While we are using `loop` in these examples, the same applies to `with_<lookup>`.

Iterating over nested lists

The simplest way to 'nest' loops is to avoid nesting loops, just format the data to achieve the same result. You can use Jinja2 expressions to iterate over complex lists. For example, a loop can combine nested lists, which simulates a nested loop.

```
- name: Give users access to multiple databases
  community.mysql.mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: true
    password: "foo"
  loop: "{{ ['alice', 'bob'] | product(['clientdb', 'employeedb', 'providerdb']) | list }}"
```

Stacking loops via include_tasks

2.1

You can nest two looping tasks using `include_tasks`. However, by default, Ansible sets the loop variable `item` for each loop. This means the inner, nested loop will overwrite the value of `item` from the outer loop. To avoid this, you can specify the name of the variable for each loop using `loop_var` with `loop_control`.

```
# main.yml
- include_tasks: inner.yml
  loop:
    - 1
    - 2
    - 3
  loop_control:
    loop_var: outer_item

# inner.yml
- name: Print outer and inner items
```

```
  ansible.builtin.debug:
    msg: "outer item={{ outer_item }} inner item={{ item }}"
  loop:
    - a
    - b
    - c
```

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

Until and loop

The `until` condition will apply per `item` of the `loop`:

```
- debug: msg={{item}}
  loop:
    - 1
    - 2
    - 3
  retries: 2
  until: item > 2
```

This will make Ansible retry the first 2 items twice, then fail the item on the 3rd attempt, then succeed at the first attempt on the 3rd item, in the end failing the task as a whole.

```
[started TASK: debug on localhost]
FAILED - RETRYING: [localhost]: debug (2 retries left).Result was: {
    "attempts": 1,
    "changed": false,
    "msg": 1,
    "retries": 3
}
FAILED - RETRYING: [localhost]: debug (1 retries left).Result was: {
    "attempts": 2,
    "changed": false,
    "msg": 1,
    "retries": 3
}
```

```
failed: [localhost] (item=1) => {
    "msg": 1
}
FAILED - RETRYING: [localhost]: debug (2 retries left).Result was: {
    "attempts": 1,
    "changed": false,
    "msg": 2,
    "retries": 3
}
FAILED - RETRYING: [localhost]: debug (1 retries left).Result was: {
    "attempts": 2,
    "changed": false,
    "msg": 2,
    "retries": 3
}
failed: [localhost] (item=2) => {
    "msg": 2
}
ok: [localhost] => (item=3) => {
    "msg": 3
}
fatal: [localhost]: FAILED! => {"msg": "One or more items failed"}
```

Migrating from with_X to loop

In most cases, loops work best with the `loop` keyword instead of `with_X` style loops. The `loop` syntax is usually best expressed using filters instead of more complex use of `query` or `lookup`.

These examples show how to convert many common `with_` style loops to `loop` and filters.

with_list

`with_list` is directly replaced by `loop`.

```
- name: with_list
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_list:
```

```
    - one
    - two

- name: with_list -> loop
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop:
    - one
    - two
```

with_items

`with_items` is replaced by `loop` and the `flatten` filter.

```
- name: with_items
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_items: "{{ items }}"

- name: with_items -> loop
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
```

with_indexed_items

`with_indexed_items` is replaced by `loop`, the `flatten` filter and `loop_control.index_var`.

```
- name: with_indexed_items
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  with_indexed_items: "{{ items }}"

- name: with_indexed_items -> loop
  ansible.builtin.debug:
    msg: "{{ index }} - {{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
  loop_control:
```

```
    index_var: index
```

with_flattened

`with_flattened` is replaced by `loop` and the `flatten` filter.

```
- name: with_flattened
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_flattened: "{{ items }}"

- name: with_flattened -> loop
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten }}"
```

with_together

`with_together` is replaced by `loop` and the `zip` filter.

```
- name: with_together
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  with_together:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_together -> loop
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ list_one|zip(list_two)|list }}"
```

Another example with complex data

```
- name: with_together -> loop
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }} - {{ item.2 }}"
  loop: "{{ data[0]|zip(*data[1:])|list }}"
```

```
  vars:
    data:
      - ['a', 'b', 'c']
      - ['d', 'e', 'f']
      - ['g', 'h', 'i']
```

with_dict

`with_dict` can be substituted by `loop` and either the `dictsort` or `dict2items` filters.

```
- name: with_dict
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  with_dict: "{{ dictionary }}"

- name: with_dict -> loop (option 1)
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ dictionary|dict2items }}"

- name: with_dict -> loop (option 2)
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ dictionary|dictsort }}"
```

with_sequence

`with_sequence` is replaced by `loop` and the `range` function, and potentially the `format` filter.

```
- name: with_sequence
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_sequence: start=0 end=4 stride=2 format=testuser%02x

- name: with_sequence -> loop
  ansible.builtin.debug:
    msg: "{{ 'testuser%02x' | format(item) }}"
  loop: "{{ range(0, 4 + 1, 2)|list }}"
```

The range of the loop is exclusive of the end point.

with_subelements

`with_subelements` is replaced by `loop` and the `subelements` filter.

```
- name: with_subelements
  ansible.builtin.debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  with_subelements:
    - "{{ users }}"
    - mysql.hosts

- name: with_subelements -> loop
  ansible.builtin.debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  loop: "{{ users|subelements('mysql.hosts') }}"
```

with_nested/with_cartesian

`with_nested` and `with_cartesian` are replaced by loop and the `product` filter.

```
- name: with_nested
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  with_nested:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_nested -> loop
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ list_one|product(list_two)|list }}"
```

with_random_choice

`with_random_choice` is replaced by just use of the `random` filter, without need of `loop`.

```
- name: with_random_choice
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_random_choice: "{{ my_list }}"

- name: with_random_choice -> loop (No loop is needed here)
  ansible.builtin.debug:
    msg: "{{ my_list|random }}"
  tags: random
```

`about_playbooks`
An introduction to playbooks

`playbooks_reuse_roles`
Playbook organization by roles

`tips_and_tricks`
Tips and tricks for playbooks

`playbooks_conditionals`
Conditional statements in playbooks

`playbooks_variables`
All about variables

`Communication<communication>`
Got questions? Need help? Want to share your ideas? Visit the Ansible communication guide

---

□□□ #2
□ □□□ □□□ 2026-10-31 14:40:40 UTC
□ □□□ □□□ 2026-10-31 14:40:41 UTC