Ansible Tutorial Part 7 – Implementing Task Control

Implementing Task Controls

Objectives

- Implement loops to write efficient tasks to control when to run tasks.
- Implement a task that runs only when another task changes the managed host.
- Control what happens when a task fails and what conditions cause a task to fail.

Writing Loops and Conditional Tasks

- Loop à is the code segment which gets executed repeatedly.
- Looping à is the process.
- Ansible supports iterating a task using loop keyword.
- A simple loop will iterate a task over a list of items.
- The loop variable item holds the value used during each iteration.
- The list used by loop variable can be provided by a variable.

Using Register Variables with Loops– The register keyword / statement is used to capture the output of a task that loops.

Running Tasks Conditionally– Conditional execution of Ansible tasks.

- ansible_cpu à "x86_64" max_memory à 512   min_memory < 128
- Variable exists à min_memory is defined.
- Variable does not exist à min_memory not defined.

Task Iteration with Loops

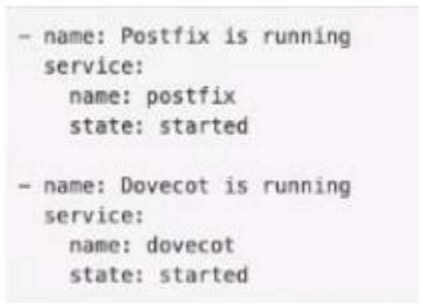
Using loops saves administrators from the need to write multiple tasks that use the same module. For example, instead of writing five tasks to ensure five users exist, you can write one task that iterates over a list if five users to ensure they all exist.

Ansible supports iterating a task over a set of items using the **loop** keyword. You can configure loops to repeat a task using each item in a list , the contents of each of the files in a list, a generated sequence of numbers or using more complicated structures. This section covers simple loops that iterate over a list of items.

Simple Loops

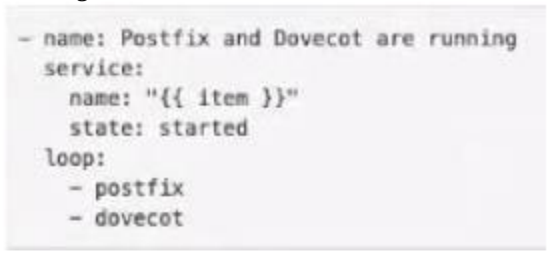
A simple loop iterates a task over a list of items. The **loop** keyword is added to the task and takes as a value the list of items over which the task should be iterated. The loop variable **item** holds the value used during each iteration.

Consider the following snippet that uses the **service** module twice in order to ensure two network services are running –



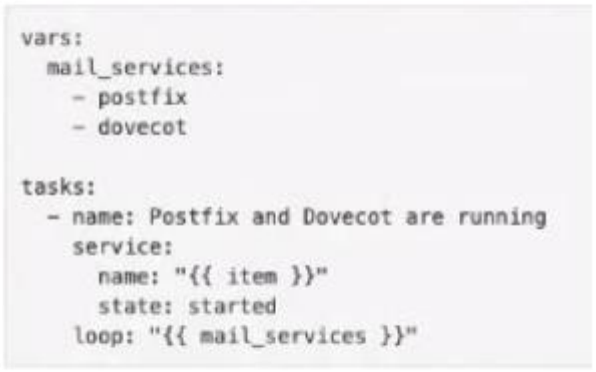
```
- name: Postfix is running
  service:
    name: postfix
    state: started

- name: Dovecot is running
  service:
    name: dovecot
    state: started
```

These two tasks can rewritten to use a simple loop so that only one task is needed to ensure both services are running –



```
- name: Postfix and Dovecot are running
  service:
    name: "{{ item }}"
    state: started
  loop:
    - postfix
    - dovecot
```

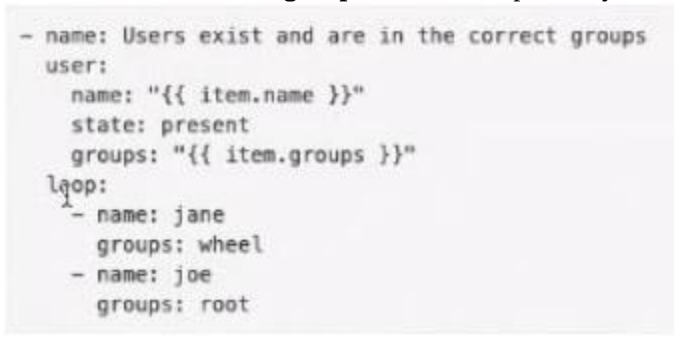
The list used by **loop** can be provided by a variable. In the following example the variable **mail_services** contains the list of services that need to be running –



```
vars:
  mail_services:
    - postfix
    - dovecot

tasks:
  - name: Postfix and Dovecot are running
    service:
      name: "{{ item }}"
      state: started
    loop: "{{ mail_services }}"
```

Loops over a list of Hashes or Directories

The **loop** list does not need to be a list if simple values. In the following example, each item in the list is actually a hash or a directory. Each hash or directory in the example has two keys, **name** and **groups** and the value of each key in the current **item** loop variable can be retrieved with the **item.name** and **item.groups** variables respectively.



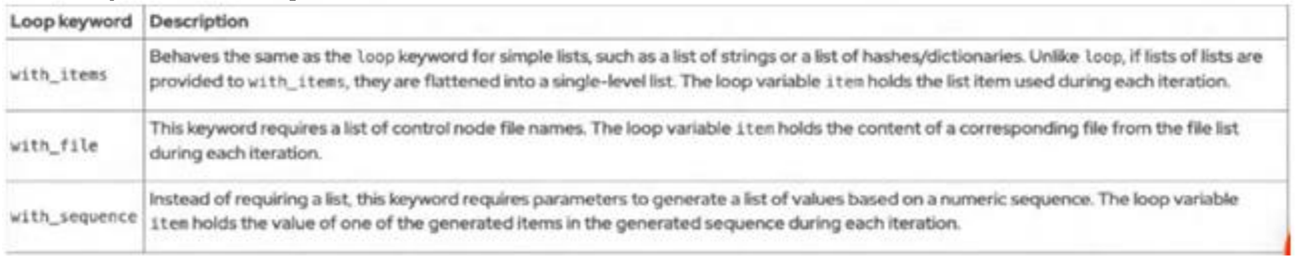
```
- name: Users exist and are in the correct groups
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - name: jane
      groups: wheel
    - name: joe
      groups: root
```

The outcome of the preceding task is that the user jane is present and a member of the group wheel and that the user joe is present and a member of the group root.

Earlier style Loop keywords

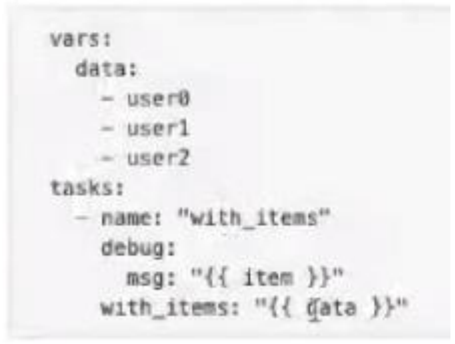
Before Ansible 2.5, most playbooks used a different syntax for loops. Multiple loop keywords were provided which were prefixed with **with_**followed by the name of an Ansible look-up plugin (an advanced feature not

covered in detail in this course). This syntax for looping is very common in existing playbooks but will probably be deprecated at some point in the future.
A few examples are listed in the table below –
Earlier Style Ansible Loops –



Loop keyword	Description
with_items	Behaves the same as the loop keyword for simple lists, such as a list of strings or a list of hashes/dictionaries. Unlike loop, if lists of lists are provided to with_items, they are flattened into a single-level list. The loop variable item holds the list item used during each iteration.
with_file	This keyword requires a list of control node file names. The loop variable item holds the content of a corresponding file from the file list during each iteration.
with_sequence	Instead of requiring a list, this keyword requires parameters to generate a list of values based on a numeric sequence. The loop variable item holds the value of one of the generated items in the generated sequence during each iteration.

An example of **with_items** in a playbook is shown below –

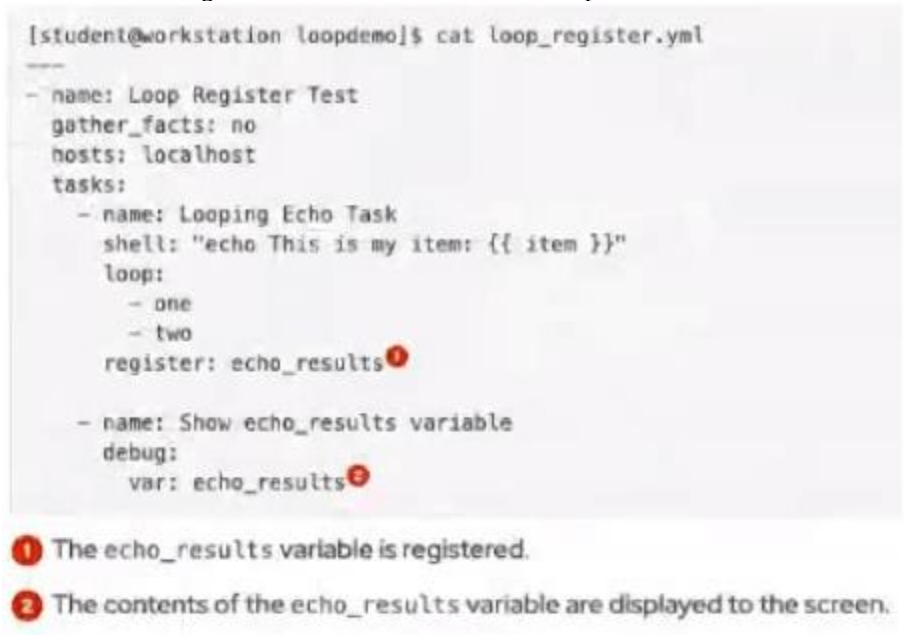


```
vars:
  data:
    - user0
    - user1
    - user2
tasks:
  - name: "with_items"
    debug:
      msg: "{{ item }}"
    with_items: "{{ data }}"
```

Important – Since Ansible 2.5, the recommended way to write loops is to use the **loop** keyword. However, you should still understand the old syntax, especially **with_items** because it is widely used in existing playbooks. You are likely to encounter playbooks and roles that continue to use with_* keywords for looping.

Using Register Variables with Loops

The **register** keyword can also capture the output of a task that loops. The following snippet shows the structure of the register variable from a task that loops –

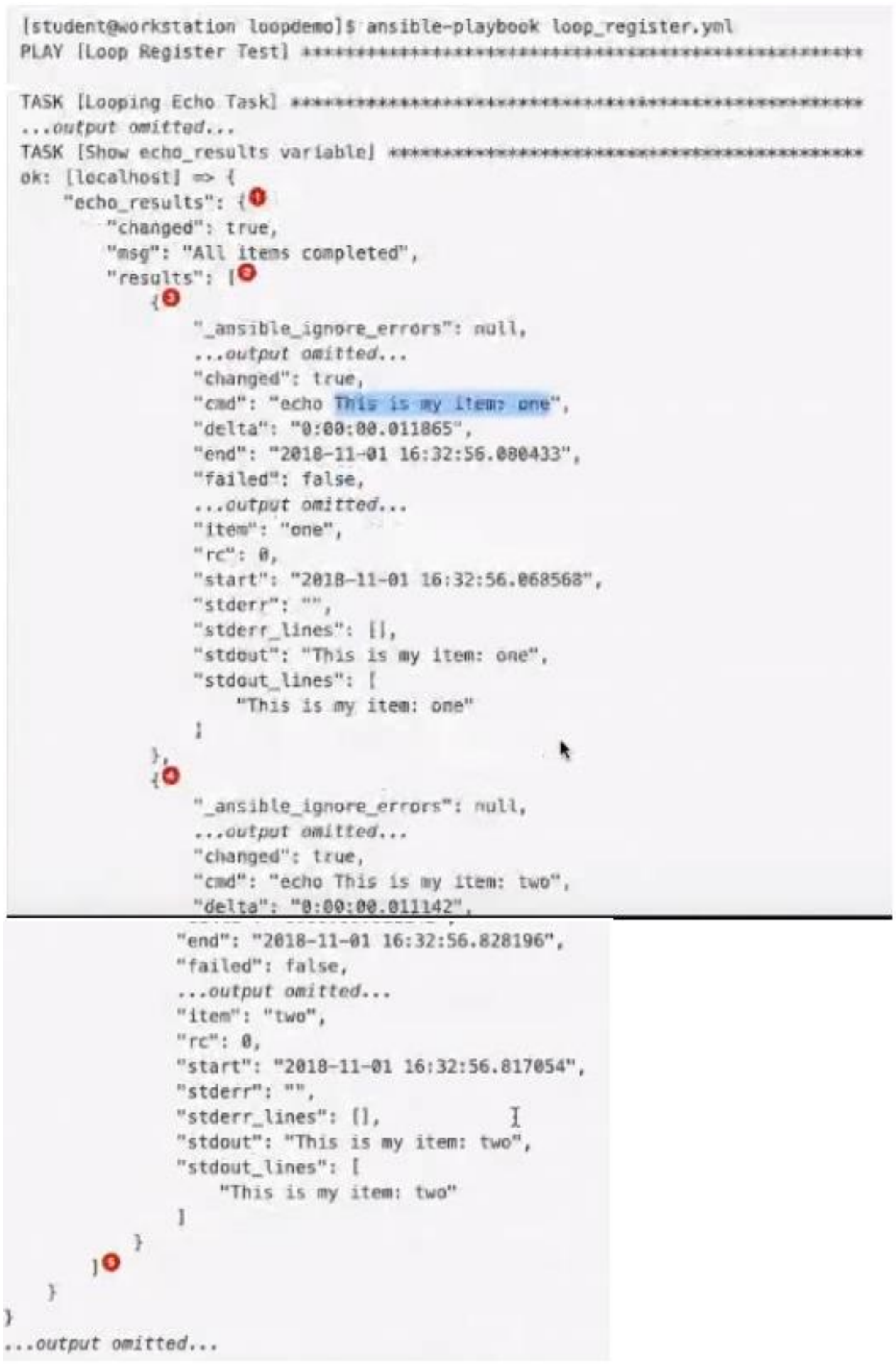


```
[student@workstation loopdemo]$ cat loop_register.yml
---
- name: Loop Register Test
  gather_facts: no
  hosts: localhost
  tasks:
    - name: Looping Echo Task
      shell: "echo This is my item: {{ item }}"
      loop:
        - one
        - two
      register: echo_results1

    - name: Show echo_results variable
      debug:
        var: echo_results2
```

1 The echo_results variable is registered.

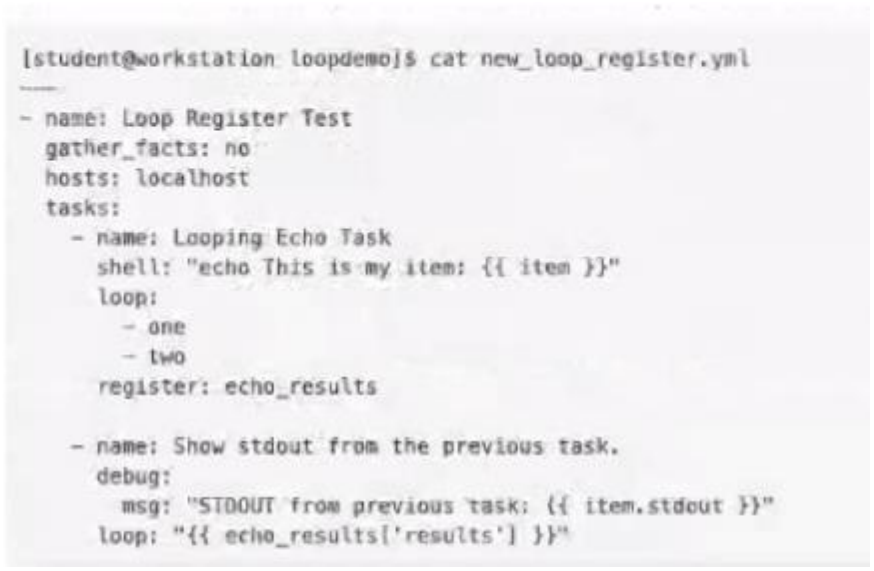
2 The contents of the echo_results variable are displayed to the screen.

Running the above playbook yields the following output –



```
[student@workstation loopdemo]$ ansible-playbook loop_register.yml
PLAY [Loop Register Test] ****************************************************

TASK [Looping Echo Task] *****************************************************
...output omitted...
TASK [Show echo_results variable] ********************************************
ok: [localhost] => {
    "echo_results": {
        "changed": true,
        "msg": "All items completed",
        "results": [
            {
                "_ansible_ignore_errors": null,
                ...output omitted...
                "changed": true,
                "cmd": "echo This is my item: one",
                "delta": "0:00:00.011865",
                "end": "2018-11-01 16:32:56.080433",
                "failed": false,
                ...output omitted...
                "item": "one",
                "rc": 0,
                "start": "2018-11-01 16:32:56.068568",
                "stderr": "",
                "stderr_lines": [],
                "stdout": "This is my item: one",
                "stdout_lines": [
                    "This is my item: one"
                ]
            },
            {
                "_ansible_ignore_errors": null,
                ...output omitted...
                "changed": true,
                "cmd": "echo This is my item: two",
                "delta": "0:00:00.011142",
                "end": "2018-11-01 16:32:56.828196",
                "failed": false,
                ...output omitted...
                "item": "two",
                "rc": 0,
                "start": "2018-11-01 16:32:56.817054",
                "stderr": "",
                "stderr_lines": [],
                "stdout": "This is my item: two",
                "stdout_lines": [
                    "This is my item: two"
                ]
            }
        ]
    }
}
...output omitted...
```

In the above, the results key contains a list. Below the playbook is modified such that the second task iterates over this list



```
[student@workstation loopdemo]$ cat new_loop_register.yml
---
- name: Loop Register Test
  gather_facts: no
  hosts: localhost
  tasks:
    - name: Looping Echo Task
      shell: "echo This is my item: {{ item }}"
      loop:
        - one
        - two
      register: echo_results

    - name: Show stdout from the previous task.
      debug:
        msg: "STDOUT from previous task: {{ item.stdout }}"
      loop: "{{ echo_results['results'] }}"
```

Running Tasks Conditionally

Ansible can use **conditionals** to execute tasks or plays when certain conditions are met. For example, a conditional can be used to determine available memory on a managed host before Ansible installs or configures as a service.

Conditionals allow administrators to differentiate between managed hosts and assign them functional roles based on the conditions that they meet. Playbook variables, registered variables and Ansible facts can all be tested with conditionals. Operators to compare strings, numeric data and Boolean values are available.

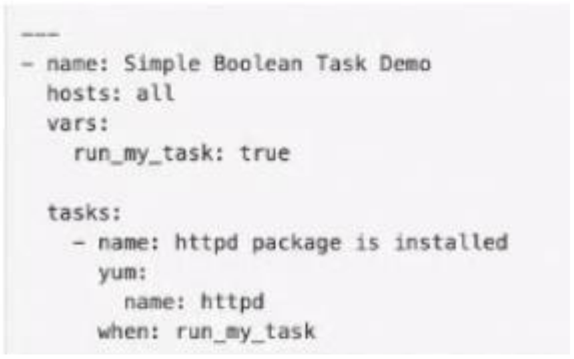
The following scenarios illustrate the use of conditionals in Ansible –

- A hard limit can be defined in a variable (for example **min_memory)** and compared against the available memory on a managed host.
- The output of a command can be captured and evaluated by Ansible to determine whether or not a task completed before taking further action. For example if a program fails, then a batch is skipped.
- Use Ansible facts to determine the managed host network configuration and decide which template file to send (for example , network bonding or trunking).
- The number of CPUs can be evaluated to determine how to properly tune a web server.
- Compare a registered variable with a predefined variable to determine if a service changed. For example, test the MD5 checksum of a service configuration file to see if the service is changed.

Conditional Task Syntax

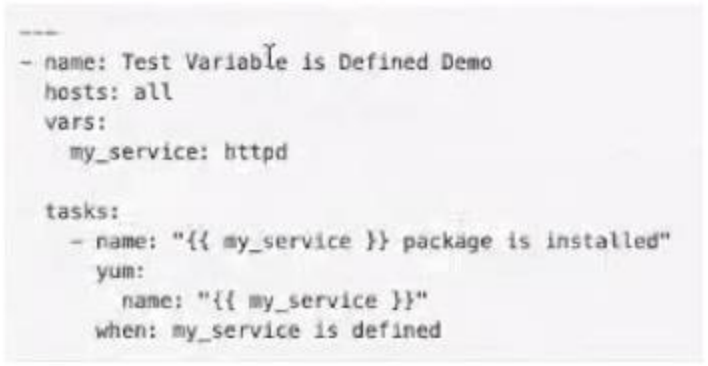
The **when** statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.

One of the simplest conditions that can be tested is whether a Boolean variable is true or false. The when statement in the following example causes the task to run only if **run_my_task** is true –



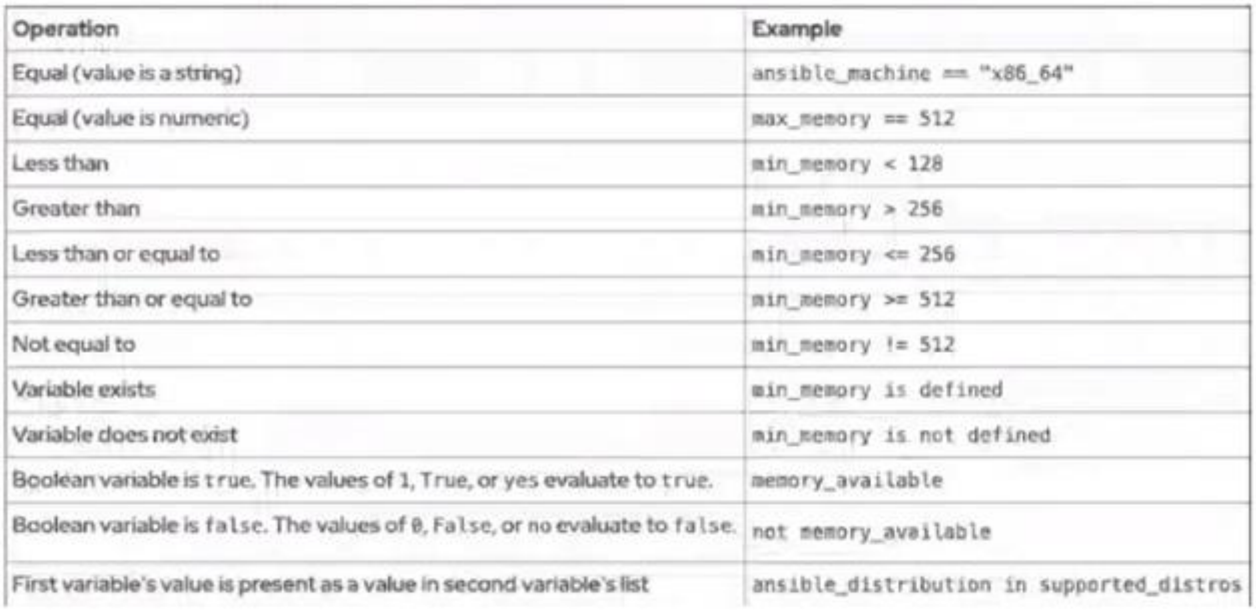
```
---
- name: Simple Boolean Task Demo
  hosts: all
  vars:
    run_my_task: true

  tasks:
    - name: httpd package is installed
      yum:
        name: httpd
      when: run_my_task
```

The next example is a bit more sophisticated and test whether the **my_service** variable has a value. If it does, the value of my_service is used as the name of the package to install. If the my_service variable is not defined, then the task is skipped without an error –



```
---
- name: Test Variable is Defined Demo
  hosts: all
  vars:
    my_service: httpd

  tasks:
    - name: "{{ my_service }} package is installed"
      yum:
        name: "{{ my_service }}"
      when: my_service is defined
```

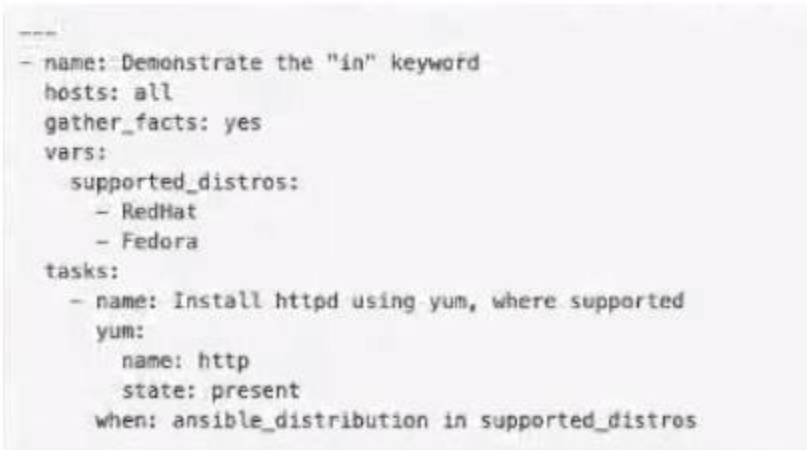
The following table shows some of the operations that administrators can use when working with conditionals –



Operation	Example
Equal (value is a string)	`ansible_machine == "x86_64"`
Equal (value is numeric)	`max_memory == 512`
Less than	`min_memory < 128`
Greater than	`min_memory > 256`
Less than or equal to	`min_memory <= 256`
Greater than or equal to	`min_memory >= 512`
Not equal to	`min_memory != 512`
Variable exists	`min_memory is defined`
Variable does not exist	`min_memory is not defined`
Boolean variable is `true`. The values of `1`, `True`, or `yes` evaluate to `true`.	`memory_available`
Boolean variable is `false`. The values of `0`, `False`, or `no` evaluate to `false`.	`not memory_available`
First variable's value is present as a value in second variable's list	`ansible_distribution in supported_distros`

The last entry in the preceding table might be confusing at first. The following example illustrates how it works.

In the example, the **ansible_distribution** variable is a fact determined during the Gathering Facts task and identifies the managed host's Operating System distribution. The variable **supported_distros** was created by the playbook author and contains a list of Operating System distributions that the playbook supports. If the **ansible_distribution** is in the **supported_distros** list, the conditional passes and the task runs.



```
---
- name: Demonstrate the "in" keyword
  hosts: all
  gather_facts: yes
  vars:
    supported_distros:
      - RedHat
      - Fedora
  tasks:
    - name: Install httpd using yum, where supported
      yum:
        name: http
        state: present
      when: ansible_distribution in supported_distros
```

Testing Multiple Conditions

One **when** statement can be used to evaluate multiple conditionals. To do so, conditionals can be combined with either the **and** or **or** keywords and grouped with parentheses.
The following snippets show some examples of how to express multiple conditions –

- If a conditional statement should be met when either condition is true , then you should use the **or** For example the following condition is met if the machine is running either Red Hat Enterprise Linux or Fedora –

when: ansible_distribution == “Redhat” or ansible_distribution == “Fedora”

- With the **and** operation, both conditions have to be true for the entire conditional statement to be met. For example, the following condition is met if the remote host is a Red Hat Enterprise Linux 7.5 host and the installed kernel is the specified version –

when: ansible_distribution_version ==”7.5” and ansible_kernel == “3.10.0-327.el7.x86_64”

The **when** keyword also supports using a list to describe a list of conditions. When a list is provided to the when keyword, all of the conditions are combined using the and operation. The example below demonstrates another way to combine multiple conditional statements using the and operator –

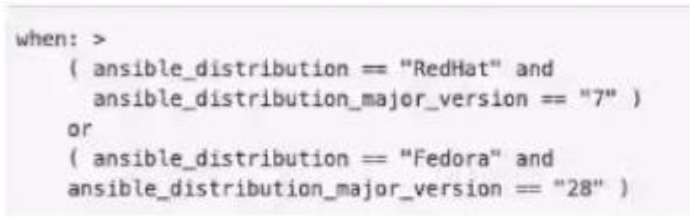
when:

- ansible_distribution_version == “7.5”
- ansible_kernel == “3.10.0-327.el7.x86_64”

This format improves readability, a key goal of well-written Ansible Playbooks.

- More complex conditional statements can be expressed by grouping conditions with parentheses. This ensures that they are correctly interpreted.

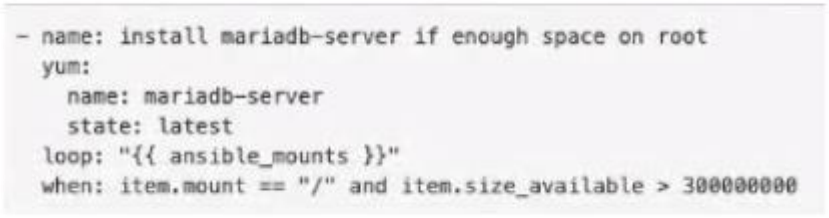
For example , the following conditional statement is met if the machine is running either Red Hat Enterprise Linux 7 or Fedora 28. This example uses the greater-than character (>) so that the long conditional can be split over multiple lines in the playbook, to make it easier to read.



```
when: >
    ( ansible_distribution == "RedHat" and
      ansible_distribution_major_version == "7" )
    or
    ( ansible_distribution == "Fedora" and
    ansible_distribution_major_version == "28" )
```

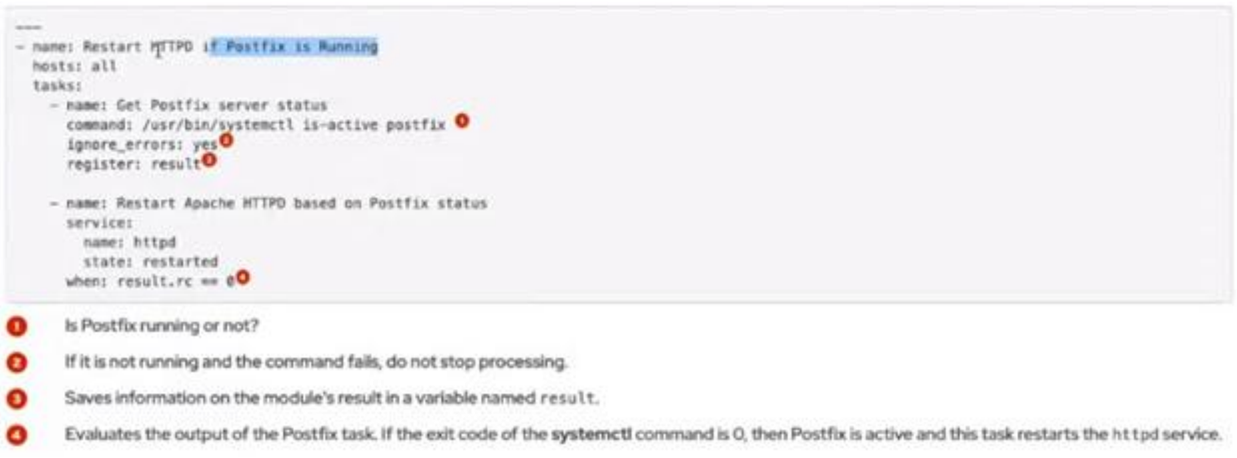
Combining Loops and Conditional Tasks

You can combine Loops and Conditionals.
In the following example the maria-db server package is installed by the yum module if there is a file system mounted on the / with more than 300 MB free. The **ansible_mounts** fact is a list of dictionaries, each one representing facts about one mounted file system. The loop iterates over each dictionary in the list and the conditional statement is not met unless a dictionary is found representing a mounted file system where both conditions are true –



```
- name: install mariadb-server if enough space on root
  yum:
    name: mariadb-server
    state: latest
  loop: "{{ ansible_mounts }}"
  when: item.mount == "/" and item.size_available > 300000000
```

Important– When you use **when** with **loop** for a task, the when statement is checked for each item.
Here is another example that combines conditionals and register variables. The following annotated playbook restarts the httpd service only if the postfix service is running –



```
---
- name: Restart HTTPD if Postfix is Running
  hosts: all
  tasks:
    - name: Get Postfix server status
      command: /usr/bin/systemctl is-active postfix (1)
      ignore_errors: yes(2)
      register: result(3)

    - name: Restart Apache HTTPD based on Postfix status
      service:
        name: httpd
        state: restarted
      when: result.rc == 0(4)
```

1. Is Postfix running or not?
2. If it is not running and the command fails, do not stop processing.
3. Saves information on the module's result in a variable named `result`.
4. Evaluates the output of the Postfix task. If the exit code of the **systemctl** command is 0, then Postfix is active and this task restarts the `httpd` service.

.