Chapter 4. Variables and Facts

Ansible is not a full-fledged programming language, but it does have several programming language features, and one of the most important of these is variable substitution. In this chapter, we’ll cover Ansible’s support for variables in more detail, including a certain type of variable that Ansible calls a *fact*.

Defining Variables in Playbooks

The simplest way to define variables is to put a vars section in your playbook with the names and values of variables. Recall from [Example 2-8](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch02.html#WEB_TLS_PLAYBOOK) that we used this approach to define several configuration-related variables, like this:

vars:

key\_file: /etc/nginx/ssl/nginx.key

cert\_file: /etc/nginx/ssl/nginx.crt

conf\_file: /etc/nginx/sites-available/default

server\_name: localhost

Ansible also allows you to put variables into one or more files, using a section called vars\_files. Let’s say we wanted to take the preceding example and put the variables in a file named *nginx.yml* instead of putting them right in the playbook. We would replace the vars section with a vars\_files that looks like this:

vars\_files:

- nginx.yml

The *nginx.yml* file would look like [Example 4-1](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#nginx_vars_example).

*Example 4-1. nginx.yml*

key\_file: /etc/nginx/ssl/nginx.key

cert\_file: /etc/nginx/ssl/nginx.crt

conf\_file: /etc/nginx/sites-available/default

server\_name: localhost

We’ll see an example of vars\_files in action in [Chapter 6](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch06.html#deploying_mezzanine) when we use it to separate out the variables that contain sensitive information.

As we discussed in [Chapter 3](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch03.html#inventory), Ansible also let you define variables associated with hosts or groups in the inventory file or in separate files that live alongside the inventory file.

Viewing the Values of Variables

For debugging, it’s often handy to be able to view the output of a variable. We saw in [Chapter 2](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch02.html#playbooks_a_beginning) how we could use the debug module to print out an arbitrary message. We can also use it to output the value of the variable. It works like this:

- debug: var=myvarname

We’ll be using this form of the debug module several times in this chapter.

Registering Variables

Often, you’ll find that you need to set the value of a variable based on the result of a task. To do so, we create a *registered variable* using the register clause when invoking a module. [Example 4-2](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#capture_output) shows how we would capture the output of the *whoami* command to a variable named *login*.

*Example 4-2. Capturing the output of a command to a variable*

- name: capture output of whoami command

command: whoami

register: login

In order to use the *login* variable later, we need to know what type of value to expect. The value of a variable set using the register clause is always a dictionary, but the specific keys of the dictionary are different, depending on the module that was invoked.

Unfortunately, the official Ansible module documentation doesn’t contain information about what the return values look like for each module. The module docs do often contain examples that use the register clause, which can be helpful. I’ve found the simplest way to find out what a module returns is to register a variable and then output that variable with the debug module:

Let’s say we run the playbook shown in [Example 4-3](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#whoami_example).

*Example 4-3. whoami.yml*

- name: show return value of command module

hosts: server1

tasks:

- name: capture output of id command

command: id -un

register: login

- debug: var=login

The output of the debug module would look like this:

TASK: [debug var=login] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [server1] => {

"login": {

"changed": true, [1](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#callout_variables_and_facts_CO1-1)

"cmd": [ [2](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#callout_variables_and_facts_CO1-2)

"id",

"-un"

],

"delta": "0:00:00.002180",

"end": "2015-01-11 15:57:19.193699",

"invocation": {

"module\_args": "id -un",

"module\_name": "command"

},

"rc": 0, [3](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#callout_variables_and_facts_CO1-3)

"start": "2015-01-11 15:57:19.191519",

"stderr": "", [4](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#callout_variables_and_facts_CO1-4)

"stdout": "vagrant", [5](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#callout_variables_and_facts_CO1-5)

"stdout\_lines": [ [6](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#callout_variables_and_facts_CO1-6)

"vagrant"

],

"warnings": []

}

}

[1](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#co_variables_and_facts_CO1-1)

The changed key is present in the return value of all Ansible modules, and Ansible uses it to determine whether a state change has occurred. For the command and shell module, this will always be set to true unless overridden with the changed\_when clause, which we cover in [Chapter 7](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch07.html#more_on_playbooks).

[2](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#co_variables_and_facts_CO1-2)

The cmd key contains the invoked command as a list of strings.

[3](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#co_variables_and_facts_CO1-3)

The rc key contains the return code. If it is non-zero, Ansible will assume the task failed to execute.

[4](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#co_variables_and_facts_CO1-4)

The stderr key contains any text written to standard error, as a single string.

[5](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#co_variables_and_facts_CO1-5)

The stdout key contains any text written to standard out, as a single string.

[6](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#co_variables_and_facts_CO1-6)

The stdout\_lines key contains any text written to split by newline. It is a list, where each element of the list is a line of output.

If you’re using the register clause with the command module, you’ll likely want access to the stdout key, as shown in [Example 4-4](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#using_output_command_task).

*Example 4-4. Using the output of a command in a task*

- name: capture output of id command

command: id -un

register: login

- debug: msg="Logged in as user {{ login.stdout }}"

Sometimes it’s useful to do something with the output of a failed task. However, if the task fails, then Ansible will stop executing tasks for the failed host. We can use the ignore\_errors clause, as shown in [Example 4-5](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#example4-5), so Ansible does not stop on the error.

*Example 4-5. Ignoring when a module returns an error*

- name: Run myprog

command: /opt/myprog

register: result

ignore\_errors: True

- debug: var=result

The shell module has the same output structure as the command module, but other modules contain different keys. [Example 4-6](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#apt_package_not_present) shows the output of the aptmodule when installing a package that wasn’t present before.

*Example 4-6. Output of apt module when installing a new package*

ok: [server1] => {

"result": {

"changed": true,

"invocation": {

"module\_args": "name=nginx",

"module\_name": "apt"

},

"stderr": "",

"stdout": "Reading package lists...\nBuilding dependency tree...",

"stdout\_lines": [

"Reading package lists...",

"Building dependency tree...",

"Reading state information...",

"Preparing to unpack .../nginx-common\_1.4.6-1ubuntu3.1\_all.deb ...",

...

"Setting up nginx-core (1.4.6-1ubuntu3.1) ...",

"Setting up nginx (1.4.6-1ubuntu3.1) ...",

"Processing triggers for libc-bin (2.19-0ubuntu6.3) ..."

]

}

}

**ACCESSING DICTIONARY KEYS IN A VARIABLE**

If a variable contains a dictionary, then you can access the keys of the dictionary using either a dot (.) or a subscript ([]). [Example 4-4](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#using_output_command_task) had a variable reference that used the dot notation:

{{ login.stdout }}

We could have used subscript notation instead:

{{ login['stdout'] }}

This rule applies to multiple dereferences, so all of the following are equivalent:

ansible\_eth1['ipv4']['address']

ansible\_eth1['ipv4'].address

ansible\_eth1.ipv4['address']

ansible\_eth1.ipv4.address

I generally prefer the dot notation, unless the key is a string that contains a character that’s not allowed as a variable name, such as a dot, space, or hyphen.

Ansible uses Jinja2 to implement variable dereferencing, so for more details on this topic, see the [Jinja2 documentation on variables](http://jinja.pocoo.org/docs/dev/templates/#variables).

[Example 4-7](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#apt_already_present) shows the output of the apt module when the package was already present on the host.

*Example 4-7. Output of apt module when package already present*

ok: [server1] => {

"result": {

"changed": false,

"invocation": {

"module\_args": "name=nginx",

"module\_name": "apt"

}

}

}

Note that the stdout, stderr, and stdout\_lines keys were present only in the output when the package was not previously installed.

Caution

If your playbooks use registered variables, make sure you know the content of that variable, both for cases where the module changes the host’s state and for when the module doesn’t change the host’s state. Otherwise, your playbook might fail when it tries to access a key in a registered variable that doesn’t exist.

Facts

As we’ve already seen, when Ansible runs a playbook, before the first task runs, this happens:

GATHERING FACTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [servername]

When Ansible gathers facts, it connects to the host and queries the host for all kinds of details about the host: CPU architecture, operating system, IP addresses, memory info, disk info, and more. This information is stored in variables that are called *facts*, and they behave just like any other variable does.

Here’s a simple playbook that will print out the operating system of each server:

- name: print out operating system

hosts: all

gather\_facts: True

tasks:

- debug: var=ansible\_distribution

Here’s what the output looks like for servers running Ubuntu and CentOS.

PLAY [print out operating system] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

GATHERING FACTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [server1]

ok: [server2]

TASK: [debug var=ansible\_distribution] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [server1] => {

"ansible\_distribution": "Ubuntu"

}

ok: [server2] => {

"ansible\_distribution": "CentOS"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

server1 : ok=2 changed=0 unreachable=0 failed=0

server2 : ok=2 changed=0 unreachable=0 failed=0

You can consult the [official Ansible documentation](http://bit.ly/1G9pVfx) for a list of some of the available facts. I maintain a more comprehensive list of facts on [GitHub](http://bit.ly/1G9pX7a).

Viewing All Facts Associated with a Server

Ansible implements fact collecting through the use of a special module called the setup module. You don’t need to call this module in your playbooks because Ansible does that automatically when it gathers facts. However, if you invoke it manually with the ansible command-line tool, like this:

**$** ansible server1 -m setup

Then Ansible will output all of the facts, as shown in [Example 4-8](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#setup_module_output).

*Example 4-8. Output of setup module*

server1 | success >> {

"ansible\_facts": {

"ansible\_all\_ipv4\_addresses": [

"10.0.2.15",

"192.168.4.10"

],

"ansible\_all\_ipv6\_addresses": [

"fe80::a00:27ff:fefe:1e4d",

"fe80::a00:27ff:fe67:bbf3"

],

(many more facts)

Note how the returned value is a dictionary whose key is ansible\_facts and whose value is a dictionary that contains the name and value of the actual facts.

Viewing a Subset of Facts

Because Ansible collects many facts, the setup module supports a filterparameter that lets you filter by fact name by specifying a glob.[1](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7243120) For example:

**$** ansible web -m setup -a 'filter=ansible\_eth\*'

The output would look like this:

web | success >> {

"ansible\_facts": {

"ansible\_eth0": {

"active": true,

"device": "eth0",

"ipv4": {

"address": "10.0.2.15",

"netmask": "255.255.255.0",

"network": "10.0.2.0"

},

"ipv6": [

{

"address": "fe80::a00:27ff:fefe:1e4d",

"prefix": "64",

"scope": "link"

}

],

"macaddress": "08:00:27:fe:1e:4d",

"module": "e1000",

"mtu": 1500,

"promisc": false,

"type": "ether"

},

"ansible\_eth1": {

"active": true,

"device": "eth1",

"ipv4": {

"address": "192.168.33.10",

"netmask": "255.255.255.0",

"network": "192.168.33.0"

},

"ipv6": [

{

"address": "fe80::a00:27ff:fe23:ae8e",

"prefix": "64",

"scope": "link"

}

],

"macaddress": "08:00:27:23:ae:8e",

"module": "e1000",

"mtu": 1500,

"promisc": false,

"type": "ether"

}

},

"changed": false

}

Any Module Can Return Facts

If you look closely at [Example 4-8](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#setup_module_output), you’ll see that the output is a dictionary whose key is ansible\_facts. The use of ansible\_facts in the return value is an Ansible idiom. If a module returns a dictionary that contains ansible\_facts as a key, then Ansible will create variable names in the environment with those values and associate them with the active host.

For modules that return facts, there’s no need to register variables, since Ansible creates these variables for you automatically. For example, the following tasks would use the ec2\_facts module to retrieve Amazon EC2[2](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7359568)facts about a server and then print out the instance id.

- name: get ec2 facts

ec2\_facts:

- debug: var=ansible\_ec2\_instance\_id

The output would look like this.

TASK: [debug var=ansible\_ec2\_instance\_id] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [myserver] => {

"ansible\_ec2\_instance\_id": "i-a3a2f866"

}

Note how we did not need to use the register keyword when invokingec2\_facts, since the returned values are facts. There are several modules that ship with Ansible that return facts. We’ll see another one of them, the dockermodule, in [Chapter 13](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch13.html#ch_docker).

Local Facts

Ansible also provides an additional mechanism for associating facts with a host. You can place one or more files on the host machine in the*/etc/ansible/facts.d* directory. Ansible will recognize the file if it’s:

* In *.ini* format
* In JSON format
* An executable that takes no arguments and outputs JSON on standard out

These facts are available as keys of a special variable named ansible\_local.

For instance, [Example 4-9](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#local_fact) shows a fact file in *.ini* format.

*Example 4-9. /etc/ansible/facts.d/example.fact*

[book]

title=Ansible: Up and Running

author=Lorin Hochstein

publisher=O'Reilly Media

If we copy this file to */etc/ansible/facts.d/example.fact* on the remote host, we can access the contents of the ansible\_local variable in a playbook:

- name: print ansible\_local

debug: var=ansible\_local

- name: print book title

debug: msg="The title of the book is {{ ansible\_local.example.book.title }}"

The output of these tasks looks like this:

TASK: [print ansible\_local] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [server1] => {

"ansible\_local": {

"example": {

"book": {

"author": "Lorin Hochstein",

"publisher": "O'Reilly Media",

"title": "Ansible: Up and Running"

}

}

}

}

TASK: [print book title] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [server1] => {

"msg": "The title of the book is Ansible: Up and Running"

}

Note the structure of value in the ansible\_local variable. Because the fact file is named *example.fact*, the ansible\_local variable is a dictionary that contains a key named “example.”

Using set\_fact to Define a New Variable

Ansible also allows you to set a fact (effectively the same as defining a new variable) in a task using the set\_fact module. I often like to use set\_factimmediately after register to make it simpler to refer to a variable. [Example 4-10](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#SET_FACT_SIMPLIFY) demonstrates how to use set\_fact so that a variable can be referred to as snap instead of snap\_result.stdout.

*Example 4-10. Using set\_fact to simplify variable reference*

- name: get snapshot id

shell: >

aws ec2 describe-snapshots --filters

Name=tag:Name,Values=my-snapshot

| jq --raw-output ".Snapshots[].SnapshotId"

register: snap\_result

- set\_fact: snap={{ snap\_result.stdout }}

- name: delete old snapshot

command: aws ec2 delete-snapshot --snapshot-id "{{ snap }}"

Built-in Variables

Ansible defines several variables that are always available in a playbook, shown in [Table 4-1](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#built_in_variables).

| **Parameter** | **Description** |
| --- | --- |
| hostvars | A dict whose keys are Ansible host names and values are dicts that map variable names to values |
| inventory\_hostname | Name of the current host as known by Ansible |
| group\_names | A list of all groups that the current host is a member of |
| groups | A dict whose keys are Ansible group names and values are a list of hostnames that are members of the group. Includes all and ungrouped groups: {"all": […], "web": […], "ungrouped": […]} |
| play\_hosts | A list of inventory hostnames that are active in the current play |
| ansible\_version | A dict with Ansible version info: {"full": 1.8.2", "major": 1, "minor": 8, "revision": 2, "string": "1.8.2"} |
| *Table 4-1. Built-in variables* | |

The hostvars, inventory\_hostname, and groups variables merit some additional discussion.

hostvars

In Ansible, variables are scoped by host. It only makes sense to talk about the value of a variable relative to a given host.

The idea that variables are relative to a given host might sound confusing, since Ansible allows you to define variables on a group of hosts. For example, if you define a variable in the *vars* section of a play, you are defining the variable for the set of hosts in the play. But what Ansible is really doing is creating a copy of that variable for each host in the group.

Sometimes, a task that’s running on one host needs the value of a variable defined on another host. Consider the scenario where you need to create a configuration file on web servers that contains the IP address of the *eth1*interface of the database server, and you don’t know in advance what this IP address is. This IP address is available as the *ansible\_eth1.ipv4.address* fact for the database server.

The solution is to use the hostvars variable. This is a dictionary that contains all of the variables defined on all of the hosts, keyed by the hostname as known to Ansible. If Ansible has not yet gathered facts on a host, then you will not be able to access its facts using the hostvars variable, unless fact caching is enabled.[3](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7565984)

Continuing our example, if our database server is *db.example.com*, then we could put the following in a configuration template:

{{ hostvars['db.example.com'].ansible\_eth1.ipv4.address }}

This would evaluate to the *ansible\_eth1.ipv4.address* fact associated with the host named *db.example.com*.

inventory\_hostname

The *inventory\_hostname* is the hostname of the current host, as known by Ansible. If you have defined an alias for a host, then this is the alias name. For example, if your inventory contains a line like this:

server1 ansible\_ssh\_host=192.168.4.10

then the *inventory\_hostname* would be server1.

You can output all of the variables associated with the current host with the help of the hostvars and inventory\_hostname variables:

- debug: var=hostvars[inventory\_hostname]

Groups

The groups variable can be useful when you need to access variables for a group of hosts. Let’s say we are configuring a load balancing host, and our configuration file needs the IP addresses of all of the servers in our web group. Our configuration file would contain a fragment that looks like this:

backend web-backend

{% for host in groups.web %}

server {{ host.inventory\_hostname }} {{ host.ansible\_default\_ipv4.address }}:80

{% endfor %}

The generated file would look like this:

backend web-backend

server georgia.example.com 203.0.113.15:80

server newhampshire.example.com 203.0.113.25:80

server newjersey.example.com 203.0.113.38:80

Setting Variables on the Command Line

Variables set by passing -e var=value to ansible-playbook have the highest precedence, which means you can use this to override variables that are already defined. [Example 4-11](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#SETTING_VAR_COMMAND_LINE) shows how to set the variable named token to the value *12345*.

*Example 4-11. Setting a variable from the command-line*

**$** ansible-playbook example.yml -e token=12345

Use the ansible-playbook -e var=value method when you want to want to use a playbook like you would a shell script that takes a command-line argument. The -e flag effectively allows you to pass variables as arguments.

[Example 4-12](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#greet-yml) shows a very simple playbook that outputs a message specified by a variable.

*Example 4-12. greet.yml*

- name: pass a message on the command line

hosts: localhost

vars:

greeting: "you didn't specify a message"

tasks:

- name: output a message

debug: msg="{{ greeting }}"

If we invoke it like this:

**$** ansible-playbook greet.yml -e greeting=hiya

Then the output looks like this:

PLAY [pass a message on the command line] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK: [output a message] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost] => {

"msg": "hiya"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

localhost : ok=1 changed=0 unreachable=0 failed=0

If you want to put a space in the variable, you’ll need two use quotes like this:

**$** ansible-playbook greet.yml -e 'greeting="hi there"'

You’ve got to put single quotes around the entire 'greeting="hi there"' so that the shell interprets that as a single argument to pass to Ansible, and you’ve got to put double quotes around "hi there" so that Ansible treats that message as a single string.

Ansible also allows you to pass a file containing the variables instead of passing them directly on the command line by passing @filename.yml as the argument to -e, for example, if we had a file that looked like [Example 4-13](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#greetvars).

*Example 4-13. greetvars.yml*

greeting: hiya

Then we can pass this file to the command line like this:

**$** ansible-playbook greet.yml -e @greetvars.yml

Precedence

We’ve covered several different ways of defining variables, and it can happen that you define the same variable multiple times for a host, using different values. Avoid this when you can, but if you can’t, then keep in mind Ansible’s precedence rules. When the same variable is defined in multiple ways, the precedence rules determine which value wins.

The basic rules of precedence are:

1. (Highest) ansible-playbook -e var=value
2. Everything else not mentioned in this list
3. On a host or group, either defined in inventory file or YAML file
4. Facts
5. In *defaults/main.yml* of a role.[4](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7776080)

In this chapter, we covered the different ways you can define and access variables and facts. In the next chapter, we’ll focus on a realistic example of deploying an application.

[1](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7243120-marker) A glob is what shells use to match file patterns (e.g., \*.txt).

[2](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7359568-marker) We’ll cover Amazon EC2 in more detail in [Chapter 12](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch12.html#cloud).

[3](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch04.html#idp7565984-marker) See [Chapter 9](https://www.safaribooksonline.com/library/view/ansible-up-and/9781491915318/ch09.html#connections-ssh) for information about fact caching.