# Scenario Syntax

This article describes the syntax for the scenarios and all of its operators.

All of the syntax described needs to be placed under the **[scenario]** header.

## Variables

Variables can store data, like a string or information returned from commands. This could either be a list or a hash.

Here is more information on scenario command syntax. These variables can be read, manipulated and interacted with.

### List Variables

A standard variable is a list variable, containing zero or more items. It is of the form `<list>`. It can be assigned as follows:

> Legacy scenario syntax used an equal sing “=”. This is still backwards compatible.

```
<list> := "value"  # assign list one literal 'value'
<list> := <item>   # assign list the value of item
<list> := <list2>  # assign list the values of list2
<list> := command [option arguments list]   # assign list the result values
of the command

<list> += "value"  # append 'value' to the list
<list> += <item>   # append the value of item to the list
<list> += <list2>  # append the values from list2 to list
<list> += command [option arguments list]   # append to list the result
values of the command

<list> -= "value"  # remove 'value' from the list
<list> -= <item>   # remove value of item from the list
<list> -= <list2>  # remove values of list2 from list
<list> -= command [option arguments list]   # remove command result(s) from
list
```

Lists can be assigned as parameters for commands:

```
LogAction -n <node> -a Command_job -m "<node> is reachable"
<interfaces> := Parse_run -n <node> -t Interfaces_<vendor_type> -c <context>
-v "<if_name>"
```

If these variables contain more than one value, the first value of the list will be taken.

## Relations

Relations can be resolved and their result will be parsed into the scenario.

```
<vlan_id> := <Vlan_id@Port_subnets>
log_action -n <node> -a Command_job -m "A vlan id is: <vlan_id@vlans>"
log_action -n <node> -a Command_job -m "Vlan 1007 has net name:
<net_name@vlans:1007>"
log_action -n <node> -a Command_job -m "Vlan 1007 has net name:
<net_name@vlans:vlan_id=1007>"
```

Note that when resolving the scenario, the tasker first parses relations, and then it parses hash variables. So relations have priority over hash variables.

## Hash Variables

> Before 7.1.0 the syntax was <@hash>

Some commands return hashes instead of lists (parse_cmd for example), denoted as <%hash>. The syntax for hash variables is similar to that of relations, but with subtle differences. A hash variable is initialised as follows:

```
<%hash>       := hash_command [option arguments list]
<%interfaces> := Parse_cmd -n <node> -t Show_interface_brief -r "show
interface brief"

<%hash>       := cmd_exec -n <node> # Will fail!
```

“interfaces” in the above line is the hash name. Hashes in essence are either hashes of lists, or hashes of hashes of lists. More information on how to return what kind of hash in the command parsing template syntax as an example.

To access variables of the simplest version, a hash of lists:

```
<%aaa> := Parse_cmd -n <node> -t Show_aaa_accounting -r "show aaa
accounting"
<aaa> := <aaa_acc_default%aaa>
```

`aaa_acc_default` in this case is the attribute of the hash with name `aaa`. This is a list, so therefore it can be assigned one-on-one to the list variable <aaa>.

Hashes can be more complex than that, though. A hash of a hash can be accessed as follows:

```
<%cdp_variables> := Parse_cmd -n <node> -t Show_cdp_all -r "show cdp all"
log_action -n <node> -a Parse_cmd_test -m "show cdp all: Ethernet1/1 is
<Ethernet1/1.if_status%cdp_variables>"
```

Here, `cdp_variables` is the hash name, this hash contains a number of keys, amongst which is `Ethernet1/1`. Key names can contain special characters, like /, \, and spaces, but not ., @, %, < or

>. We then look for the list in the attribute named `if_status`. This too is a list. Hash names and attribute names can only contain alphanumeric characters and underscores.

To get a list of all keys, the keyword `keys` can be used:

```
<%cdp_variables> := Parse_cmd -n <node> -t Show_cdp_interface_ethernet -r
"show cdp interface Ethernet1/1"
<devices> := keys <%cdp_variables>
foreach <device> in <devices>
    <vlan> := <device.device_vlan%cdp_variables>
    log_action -n <node> -a Parse_cmd_test -m "show cdp interface
Ethernet1/1: <device> has vlan <vlan>"
endeach
```

> Note: This example previously showed an incorrect assignment of a keyed hash value: `<vlan> := «device>.device_vlan%cdp_variables>` This 'nested' use of the hash-key is incorrect.

Details on foreach can be found below, but note that a key in calling a hash variable can be either a literal or the contents of a list variable. If the length of this list variable is bigger than 1, the first item of this list is taken.

### Error

If a scenario command fails, this is stored in the `<error>` variable. The `<error>` variable is equivalent to `error`, but the former is better practice. The error variable can then be checked within an if statement, for example:

```
reachable -n <node>
if <error>
    log_action -n <node> -a Command_job -m "<node> is not reachable"
    stop
endif
log_action -n <node> -a Command_job -m "<node> is reachable"
```

More details on the error can be found in the job logs.

## Conditionals and Loops

### if <condition>

Conditional execution. This can be nested. Conditions have the form “`<variable> <operator> <variable>`” or “`<variable>`”.
`<variable>` can be either a list variable, or a list in a hash variable, if the list has more than one items, the first item is used for comparison. `<operator>` is one of ==, <>, >=, <=, >, <, !=.

Beyond that, variables can also be extended with extra operations:

```
    if ! <variable>            Negation
    if not <variable>          Negation
    if count <variable>        Count the number of values in <variable>
    if <error>                 The result of the last executed command
```

Note that <error> always has the value 1 or 0. You can compare it against another variable this way.

Examples:

```
if <variable> == "abc.com"
if not <variable> == "abc.com"
if <variable> != "abc.com"
if <variableA> == <variableB>
if <variableA>
```

Notes:

- Strings should be quoted in a condition and be on the right side
- <Variables> don't need quotes in conditions.
    - Unless it contains one or more of these special characters: [ | ] ! @ # $ ^ & + = ' “ : ; < >

## else

Follows an if-statement. If this if fails, the code below else gets executed.

## endif

Close the current if or else condition.

## foreach <item> in <list>

Loop through a list and run code for each item in this list, marked between a foreach and endeach statement. Foreach loops can be nested alongside if statements.

```
foreach <item> in <list>
    # do something
endeach
```

<item> is a list variable, containing only one variable. It works just like any other list variables. <list> can also be a list from a hash variable:

```
foreach <item> in <key.attribute%hash>
    # do something
endeach
```

### endeach

Marks the end of the current foreach loop.

### next

Abort execution of the current foreach loop and move on to the next item in the list.

### last

Jump out of the current foreach loop.

## List Operations

Lists can be manipulated by simple operations.

Note these are some additional, advanced, commands that operate on lists and require options. They are included in the Scenario Commands details page.

These include the commands: “sort”, “like”, “concat”, “grep” and “keys”.

### reverse

Take a list, and return it in reverse order.

```
<revlist> := reverse <list>
```

### push

Push item to end of list. Same as `<list> += <item>`

```
<list> := push <item>
<list> := push 2
<list> := push <key.attribute%hash>
```

### pop

Assign the last value of a list to the variable item and remove from it from the list.

```
<item> := pop <list>
<item> := pop <key.attribute%hash>
```

### unshift

Insert an item to front of the list. Same as `<list> := <item>`; `<list> += <list2>`

```
<list> := unshift <item>
<list> := unshift 2
<list> := unshift <key.attribute%hash>
```

### shift

Assign the first value from the list to item and remove it from the list.

```
<item> := shift <list>
<item> := shift <key.attribute%hash>
```

### null

Assign an empty list. If `<list>` already exists it will clear its values.

```
<list> := null
```

## Other

### description

Set the description of the scenario. If multiple descriptions are set in a scenario, the first one is taken. This also applies when the description is set via an API call. Once set, it is not changed.

```
Description This is a scenario
Description <node> test scenario
```

### stop

Quit scenario. The scenario will have failed and the job status will list ABORTED.

Make sure to read the Scenario Commands details on Error handling.

Here the usage of “STOP” and “END” is clarified and how the “**stop on-error**” can provide a default job ABORT on any error and a “**stop default**” will resume normal error handling.

### end

Quit scenario, signal that the scenario succeeded. It may be used to halt a scenario at any point and set the job status to SUCCESSFUL. All scenarios have the END implicitly added at the bottom resulting in a SUCCESSFUL job by default.

### task

It is currently possible to use a named scenario as a 'task' inside a scenario. It gets substituted with its contents before being parsed. This is supported till a maximum of 50 levels deep.

```
task := another_scenario
```

### comments

Comments can be added to the scenario using the '#' sign. They are to be used on their own lines only.

```
# some comment
if <var1> == <var2>
  # it is expected to enter here.
  do_something
endif
```

## Examples

Several examples can be found on the scenarios example page