ttp

Release 0.4.0

Nov 17, 2020

Contents

CHAPTER 1

Overview

TTP is a Python module that allows fast performance parsing of semi-structured text data using templates. TTP was developed to enable programmatic access to data produced by CLI of networking devices, but, it can be used to parse any semi-structured text that contains distinctive repetition patterns.

In the simplest case ttp takes two files as an input - data that needs to be parsed and parsing template, returning results structure with extracted information.

Same data can be parsed by several templates producing results accordingly, templates are easy to create and users encouraged to write their own ttp templates.

1.1 Motivation

While networking devices continue to develop API capabilities, there is a big footprint of legacy and not-so devices in the field, these devices are lacking of any well developed API to retrieve structured data, the closest they can get is SNMP and CLI text output. Moreover, even if some devices have API capable of representing their configuration or state data in the format that can be consumed programmatically, in certain cases, the amount of work that needs to be done to make use of these capabilities outweighs the benefits or value of produced results.

There are a number of tools available to parse text data, but, author of TTP believes that parsing data is only part of the work flow, where the ultimate goal is to make use of the actual data.

Say we have configuration files and we want to create a report of all IP addresses configured on devices together with VRFs and interface descriptions, report should have csv format. To do that we have (1) collect data from various inputs and maybe sort and prepare it, (2) parse that data, (3) format it in certain way and (4) save it somewhere or pass to other program(s). TTP has built-in capabilities to address all of these steps to produce desired outcome.

1.2 Core Functionality

TTP has a number of systems built into it:

- groups system - help to define results hierarchy and data processing functions with filtering

- parsing system - uses regular expressions derived out of templates to parse and process data
- input system - used to define various input data sources, help to retrieve data, prepare it and map to the groups for parsing
- output system - allows to format parsing results and return them to certain destinations
- macro - inline Python code that can be used to process results and extend TTP functionality, having access to _ttp_ dictionary containing all groups, match, inputs, outputs functions
- lookup tables - helps to enreach results with additional additional information or cross reference results across different templates or groups to combine them
- template variables - variables store, accessible during template execution for caching or retrieving values
- templates tags - to define several independent templates within single file, used to define results forming mode
- CLI tool - allows to execute templates directly
- Lazy loader system - TTP only imports function it uses within the templates, that significantly increase start time
- Multiprocessing - to spun up several Python processes to increase parsing performance
- Logging system - helps to troubleshoot and debug TTP

CHAPTER 2

Installation

Using pip:

```
pip install ttp
```

Or clone from GitHub, unzip, navigate to folder and run:

```
python setup.py install or python -m pip install .
```

2.1 Additional dependencies

TTP mainly uses built-in libraries. However, additional modules need to be installed on the system for certain features to work.

Group Functions

- *cerberus* - requires Cerberus library

Input Sources

- *Netmiko* - requires Netmiko library
- *Nornir* - requires Nornir library

Output Formatters

- *yaml* - requires PyYAML module
- *tabulate* - requires tabulate module
- *jinja2* - requires Jinja2 module
- *excel* - requires openpyxl
- *N2G* - requires N2G module

Output Functions

- *deepdiff* - requires deepdiff library

Lookup Tables

- INI lookup tables - requires configparser
- *geoip2 database* - requires GeoIP2

CHAPTER 3

Quick start

TTP can be used as a module, as a CLI tool or as a script.

3.1 As a module

Sample code:

```
from ttp import ttp

data_to_parse = """
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Vlan778
 description CPE_Acces_Vlan
 ip address 2002::fd37/124
 ip vrf CPE1
!
"""

ttp_template = """
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
"""

# create parser object and parse data using template:
parser = ttp(data=data_to_parse, template=ttp_template)
parser.parse()

# print result in JSON format
```

(continues on next page)

(continued from previous page)

```
results = parser.result(format='json')[0]
print(results)
[
    [
        {
            "description": "Router-id-loopback",
            "interface": "Loopback0",
            "ip": "192.168.0.113",
            "mask": "24"
        },
        {
            "description": "CPE_Acces_Vlan",
            "interface": "Vlan778",
            "ip": "2002::fd37",
            "mask": "124",
            "vrf": "CPE1"
        }
    ]
]

# or in csv format
csv_results = parser.result(format='csv')[0]
print(csv_results)
description,interface,ip,mask,vrf
Router-id-loopback,Loopback0,192.168.0.113,24,
CPE_Acces_Vlan,Vlan778,2002::fd37,124,CPE1
```

3.2 As a CLI tool

Sample command to run in terminal:

```
ttp --data "path/to/data_to_parse.txt" --template "path/to/ttp_template.txt" --
↪outputter json

[
    [
        {
            "description": "Router-id-loopback",
            "interface": "Loopback0",
            "ip": "192.168.0.113",
            "mask": "24"
        },
        {
            "description": "CPE_Acces_Vlan",
            "interface": "Vlan778",
            "ip": "2002::fd37",
            "mask": "124",
            "vrf": "CPE1"
        }
    ]
]
```

Where file `path/to/data_to_parse.txt` contains:

```
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Vlan778
 description CPE_Acces_Vlan
 ip address 2002::fd37/124
 ip vrf CPE1
!
```

And file `path/to/ttp_template.txt` contains:

```
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
```

CHAPTER 4

Match Variables

Match variables used as names (keys) for information (values) that needs to be extracted from text data. Match variables placed within { { and } } double curly brackets. For instance:

```
<group name="interfaces">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans }}
</group>
```

Match variables are `interface` and `trunk_vlans` will store matching values extracted from this sample data:

```
interface GigabitEthernet3/4
 switchport trunk allowed vlan add 771,893
    !
interface GigabitEthernet3/5
 switchport trunk allowed vlan add 138,166-173
```

After parsing, TTP will produce this result:

```
[
    {
        "interfaces": {
            "interface": "GigabitEthernet3/4",
            "trunk_vlans": "771,893"
        },
        {
            "interface": "GigabitEthernet3/5",
            "trunk_vlans": "138,166-173"
        }
    }
]
```

Match variables can reference various function to process data during parsing, indicators to change parsing logic or regular expression patterns to use for data parsing. Match variables combined with groups can help to define the way how data parsed, processed and structured.

4.1 Match Variables reference

4.1.1 Indicators

Indicators or directives can be used to change parsing logic or indicate certain events.

Table 1: indicators

Name	Description
*_exact_*	Threats digits as is without replacing them with `\d+` pattern
*_exact_space_*	Leave space characters in place without replacing them with `r(\\ +)` pattern
*_start_*	Explicitly indicates start of the group
*_end_*	Explicitly indicates end of the group
*_line_*	If present any line will be matched
ignore	Substitute string at given position with regular expression without matching results
*_headers_*	To dynamically form regex for parsing fixed-width, one-line text tables

exact

```
{{ name | _exact_ }}
```

By default all digits in template replaced with 'd+' pattern, if `_exact_` present in any match variable within that line, digits are unchanged and used for parsing as is.

Example

Sample Data:

```
vrf VRF-A
 address-family ipv4 unicast
  maximum prefix 1000 80
 !
 address-family ipv6 unicast
  maximum prefix 300 80
 !
```

If Template:

```
<group name="vrfs">
vrf {{ vrf }}
 <group name="ipv4_config">
 address-family ipv4 unicast {{ _start_ }}
  maximum prefix {{ limit }} {{ warning }}
 </group>
</group>
```

Result will be:

```
{
    "vrfs": {
        "ipv4_config": [
            {
                "limit": "1000",
                "warning": "80"
            },
```

(continues on next page)

(continued from previous page)

```
            {
                "limit": "300",
                "warning": "80"
            }
        ],
        "vrf": "VRF-A"
    }
}
```

As you can see ipv6 part of vrf configuration was matched as well and we got undesirable results, one of the possible solutions would be to use _exact_ directive to indicate that "ipv4" should be matches exactly.

If Template:

```
<group name="vrfs">
vrf {{ vrf }}
 <group name="ipv4_config">
 address-family ipv4 unicast {{ _start_ }}{{ _exact_ }}
  maximum prefix {{ limit }} {{ warning }}
 !{{ _end_ }}
 </group>
</group>
```

Result will be:

```
{
    "vrfs": {
        "ipv4_config": {
            "limit": "1000",
            "warning": "80"
        },
        "vrf": "VRF-A"
    }
}
```

_exact_space_

`{{ name | _exact_space_ }}`

By default all space characters in template replaced with '\ +' pattern, if _exact_space_ present, space characters will stay unchanged and will be used for parsing as is.

start

`{{ name | _start_ }}` or {{ _start_ }}

This directive can be used to explicitly indicate start of the group by matching certain line or if we have multiple lines that can indicate start of the same group.

Example-1

In this example line "————————————-" can serve as an indicator of the beginning of the group, but we do not have any match variables defined in it.

Sample data:

```
switch-a#show cdp neighbors detail
-------------------------
Device ID: switch-b
Entry address(es):
  IP address: 131.0.0.1

-------------------------
Device ID: switch-c
Entry address(es):
  IP address: 131.0.0.2
```

Template:

```
<group name="cdp_peers">
------------------------- {{ _start_ }}
Device ID: {{ peer_hostname }}
Entry address(es):
  IP address: {{ peer_ip }}
</group>
```

Result:

```
{
    "cdp_peers": [
        {
            "peer_hostname": "switch-b",
            "peer_ip": "131.0.0.1"
        },
        {
            "peer_hostname": "switch-c",
            "peer_ip": "131.0.0.2"
        }
    ]
}
```

Example-2

In this example, two different lines can serve as an indicator of the start for the same group.

Sample Data:

```
interface Tunnel2422
 description cpe-1
!
interface GigabitEthernet1/1
 description core-1
```

Template:

```
<group name="interfaces">
interface Tunnel{{ if_id }}
interface GigabitEthernet{{ if_id | _start_ }}
 description {{ description }}
</group>
```

Result will be:

```
{
    "interfaces": [
        {
            "description": "cpe-1",
            "if_id": "2422"
        },
        {
            "description": "core-1",
            "if_id": "1/1"
        }
    ]
}
```

end

`{{ name | _end_ }}` or `{{ _end_ }}`

Explicitly indicates the end of the group. If line was matched that has _end_ indicator assigned - that will trigger processing and saving group results into results tree. The purpose of this indicator is to optimize parsing performance allowing TTP to determine the end of the group faster and eliminate checking of unrelated text data.

line

`{{ name | _line_ }}`

This indicator serves double purpose, first of all, special regular expression will be used to match any line in text, moreover, additional logic will be incorporated for such a cases when same portion of text data was matched by _line_ and other regular expression simultaneously. Main use case for _line_ indicator is to match and collect data that not been matched by other match variables.

All TTP match variables function can be used together with _line_ indicator, for instance `contains` function can be used to filter results.

TTP will assign only last line matched by _line_ to match variable, if multiple lines needs to be saved, `joinmatches` function can be used.

> **Warning:** _line_ expression is computation intensive and can take longer time to process, it is recommended to use _end_ indicator together with _line_ whenever possible to minimize performance impact. In addition, having as clear source data as possible also helps, as it allows to avoid false positives - unnecessary matches.

Example

Let's say we want to match all port-security related configuration on the interface and save it into port_security_cfg variable.

Template:

```
<input load="text">
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Gi0/37
 description CPE_Acces
```

(continues on next page)

(continued from previous page)

```
 switchport port-security
 switchport port-security maximum 5
 switchport port-security mac-address sticky
!
</input>

<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
 {{ port_security_cfg | _line_ | contains("port-security") | joinmatches }}
! {{ _end_ }}
</group>
```

Results:

```
[[{   'description': 'Router-id-loopback',
      'interface': 'Loopback0',
      'ip': '192.168.0.113',
      'mask': '24'},
  {   'description': 'CPE_Acces',
      'interface': 'Gi0/37',
      'port_security_cfg': 'switchport port-security\n'
                           'switchport port-security maximum 5\n'
                           'switchport port-security mac-address sticky'}
                                          ]]
```

ignore

`{{ ignore }}` or `{{ ignore("value") }}`

`value` can be of:

- regular expression string - regex to use to substitute portion of the string, default is `\S+`, meaning any non-space character one or more times.
- template variable - name of template variable that contains regular expression to use
- built in re pattern - name of regex pattern to use, for example *WORD*

Note: Reference template variable if ignore pattern contains `|` (pipe) character, as pipe character used by TTP to separate match variable functions and cannot be used in inline regex.

Primary use case of this indicator is to ignore changing alpha-numerical characters or ignore portion of the line. For example consider this data:

```
FastEthernet0/0 is up, line protocol is up
  Hardware is Gt96k FE, address is c201.1d00.0000 (bia c201.1d00.1234)
  MTU 1500 bytes, BW 100000 Kbit/sec, DLY 1000 usec,
FastEthernet0/1 is up, line protocol is up
  Hardware is Gt96k FE, address is b20a.1e00.8777 (bia c201.1d00.1111)
  MTU 1500 bytes, BW 100000 Kbit/sec, DLY 1000 usec,
```

Example-1

What if only need to extract bia MAC address within parenthesis, below template will **not** work for all cases:

```
{{ interface }} is up, line protocol is up
  Hardware is Gt96k FE, address is c201.1d00.0000 (bia {{MAC}})
  MTU {{ mtu }} bytes, BW 100000 Kbit/sec, DLY 1000 usec,
```

Result:

```
[
    [
        {
            "MAC": "c201.1d00.1234",
            "interface": "FastEthernet0/0",
            "mtu": "1500"
        },
        {
            "interface": "FastEthernet0/1",
            "mtu": "1500"
        }
    ]
]
```

As we can see MAC address for FastEthernet0/1 was not matched due to the fact that "c201.1d00.0000" text was used in template, to fix it we need to ignore MAC address before parenthesis as it keeps changing across the source data:

```
{{ interface }} is up, line protocol is up
  Hardware is Gt96k FE, address is {{ ignore }} (bia {{MAC}})
  MTU {{ mtu }} bytes, BW 100000 Kbit/sec, DLY 1000 usec,
```

Result:

```
[
    [
        {
            "MAC": "c201.1d00.1234",
            "interface": "FastEthernet0/0",
            "mtu": "1500"
        },
        {
            "MAC": "c201.1d00.1111",
            "interface": "FastEthernet0/1",
            "mtu": "1500"
        }
    ]
]
```

Example-2

In this example template variable "pattern_var" used together with ignore, that variable reference regular expression pattern that contains pipe symbol.

Template:

```
<input load="text">
FastEthernet0/0 is up, line protocol is up
  Hardware is Gt96k FE, address is c201.1d00.0000 (bia c201.1d00.1234)
  MTU 1500 bytes, BW 100000 Kbit/sec, DLY 1000 usec,
FastEthernet0/1 is up, line protocol is up
```

(continues on next page)

(continued from previous page)

```
  Hardware is Gt96k FE, address is b20a.1e00.8777 (bia c201.1d00.1111)
  MTU 1500 bytes, BW 100000 Kbit/sec, DLY 1000 usec,
</input>

<vars>
pattern_var = "\S+|\d+"
</vars>

<group name="interfaces">
{{ interface }} is up, line protocol is up
  Hardware is Gt96k FE, address is {{ ignore("pattern_var") }} (bia {{MAC}})
  MTU {{ mtu }} bytes, BW 100000 Kbit/sec, DLY 1000 usec,
</group>
```

Results:

```
[
    [
        {
            "interfaces": [
                {
                    "MAC": "c201.1d00.1234",
                    "interface": "FastEthernet0/0",
                    "mtu": "1500"
                },
                {
                    "MAC": "c201.1d00.1111",
                    "interface": "FastEthernet0/1",
                    "mtu": "1500"
                }
            ]
        }
    ]
]
```

headers

```
head1 head2 ... headN {{ _headers_ }}
```

This indicator uses headers line to dynamically form regular expression to parse fixed-width, one-line text tables.

Column width calculated based on header items length, variables names dynamically formed out of header items. As a result there are a number of restrictions:

- headers line must match original data to calculate correct columns width
- header items must be separated by at least one space character
- header items must be left-aligned to indicate beginning of the column
- header items cannot contain spaces, replace them with underscore
- header items must be valid Python identifies to form variables names
- match variable functions not supported for header items, instead, group functions and macro can be used for processing

Example-1

Template:

```
<input load="text">
Port      Name               Status       Vlan       Duplex  Speed Type
Gi0/1     PIT-VDU213         connected    18         a-full  a-100 10/100/1000BaseTX
Gi0/3     PIT-VDU212         notconnect   18           auto   auto 10/100/1000BaseTX
Gi0/4                        connected    18         a-full  a-100 10/100/1000BaseTX
Gi0/5                        notconnect   18           auto   auto 10/100/1000BaseTX
Gi0/15                       connected    trunk        full   1000 1000BaseLX SFP
Gi0/16    pitrs2201 te1/1/4  connected    trunk        full   1000  1000BaseLX SFP
</input>

<group>
Port      Name               Status       Vlan       Duplex  Speed Type   {{ _headers_
↪ }}
</group>
```

Result:

```
[[[{'Duplex': 'a-full',
    'Name': 'PIT-VDU213',
    'Port': 'Gi0/1',
    'Speed': 'a-100',
    'Status': 'connected',
    'Type': '10/100/1000BaseTX',
    'Vlan': '18'},
   {'Duplex': 'a-full',
    'Name': '',
    'Port': 'Gi0/4',
    'Speed': 'a-100',
    'Status': 'connected',
    'Type': '10/100/1000BaseTX',
    'Vlan': '18'},
   {'Duplex': 'full',
    'Name': 'pitrs2201 te1/1/4',
    'Port': 'Gi0/16',
    'Speed': '1000',
    'Status': 'connected',
    'Type': '1000BaseLX SFP',
    'Vlan': 'trunk'}]]]
```

Example-2

Header can be indented by a number of spaces or tabs, but each tab replaced with 4 space characters to calculate column width.

Template:

```
<input load="text">
   Network            Next Hop            Metric     LocPrf     Weight Path
*>e11.11.1.111/32     12.123.12.1              0                    0 65000 ?
*>e222.222.222.2/32   12.123.12.1              0                    0 65000 ?
*>e333.33.333.333/32  12.123.12.1              0                    0 65000 ?
</input>

<group>
   Network            Next_Hop            Metric     LocPrf     Weight Path  {{ _
↪headers_ }}
</group>
```

Result:

```
[[[{'LocPrf': '',
    'Metric': '0',
    'Network': '*>e11.11.1.111/32',
    'Next_Hop': '12.123.12.1',
    'Path': '65000 ?',
    'Weight': '0'},
   {'LocPrf': '',
    'Metric': '0',
    'Network': '*>e222.222.222.2/32',
    'Next_Hop': '12.123.12.1',
    'Path': '65000 ?',
    'Weight': '0'},
   {'LocPrf': '',
    'Metric': '0',
    'Network': '*>e333.33.333.333/32',
    'Next_Hop': '12.123.12.1',
    'Path': '65000 ?',
    'Weight': '0'}]]]
```

4.1.2 Functions

TTP contains a set of TTP match variables functions that can be applied to match results to transform them in a desired way or validate and filter match results.

Action functions act upon match result to transform into desired state.

Table 2: Action functions

Name	Description
chain	add functions from chain variable
record	Save match result to variable with given name, which can be referenced by actions
let	Assigns provided value to match variable
truncate	truncate match results
joinmatches	join matches using provided character
resub	replace old patter with new pattern in match using re substitute method
join	join match using provided character
append	append provided string to the end of match result
prepend	prepend provided string at the beginning of match result
print	print match result to terminal
unrange	unrange match result using given parameters
set	set result to specific value based if certain string was matched
replaceall	run replace against match for all given values
resuball	run re substitute against match for all given values
lookup	find match value in lookup table and return result
rlookup	find rlookup table key in match result and return associated values
gpvlookup	Glob Patterns Values lookup uses glob patterns testing against match result
geoip_lookup	Uses GeoIP2 database to lookup ASN, Country or City information
item	returns item at given index on match result
macro	runs match result against macro function
to_list	creates empty list nd appends match result to it
to_int	transforms result to integer
to_str	transforms result to python string

Continued on next page

Table 2 – continued from previous page

Name	Description
to_ip	transforms result to python ipaddress module IPvXAddress or IPvXInterface object
to_net	transforms result to python ipaddress module IPvXNetwork object
to_cidr	transforms netmask to cidr (prefix length) notation
ip_info	produces a dictionary with information about give ip address or subnet
dns	performs DNS forward lookup
rdns	performs DNS reverse lookup
sformat	string format using python string format method
uptimeparse	function to parse uptime string
mac_eui	transforms mac string into EUI format
count	function to count matches
void	returns False on results validation, allowing to skip them
to_float	converts match variable value to float integer
to_unicode	if script run by python2, converts string to unicode

Condition functions can perform various checks with match results and returns either True or False depending on check results.

Table 3: Condition functions

Name	Description
equal	check if match is equal to provided value
notequal	check if match is not equal to provided value
startswith_re	checks if match starts with certain string using regular expression
endswith_re	checks if match ends with certain string using regular expression
contains_re	checks if match contains certain string using regular expression
contains	checks if match contains certain string patterns
notstartswith_re	checks if match not starts with certain string using regular expression
notendswith_re	checks if match not ends with certain string using regular expression
exclude_re	checks if match not contains certain string using regular expression
exclude	checks if match not contains certain string
isdigit	checks if match is digit string e.g. '42'
notdigit	checks if match is not digit string
greaterthan	checks if match is greater than given value
lessthan	checks if match is less than given value
is_ip	tries to convert match result to ipaddress object and returns True if so, False otherwise
cidr_match	transforms result to ipaddress object and checks if it overlaps with given prefix

Python built-ins

Apart from functions provided by ttp, python objects built-in functions can be used as well. For instance string *upper* method can be used to convert match into upper case, or list *index* method to return index of certain value.

Example

Data:

```
interface Tunnel2422
 description cpe-1
!
interface GigabitEthernet1/1
 description core-1
```

Template:

```
<group name="interfaces">
interface {{ interface | upper }}
 description {{ description | split('-') }}
</group>
```

Result:

```
{
    "interfaces": [
        {
            "description": ["cpe", "1"],
            "interface": "TUNNEL2422"
        },
        {
            "description": ["core", "1"],
            "interface": "GIGABITETHERNET1/1"
        }
    ]
}
```

chain

```
{{ name | chain(variable_name) }}
```

- variable_name (mandatory) - string containing variable name

Sometime when many functions needs to be run against match result the template can become difficult to read, in addition if same set of functions needs to be run against several matches and changes needs to be done to the set of functions it can become difficult to maintain such a template.

To solve above problem *chain* function can be used. Value supplied to that function must reference a valid variable name, that variable should contain string of functions names that should be used for match result, alternatively variable can reference a list of items, each item is a string representing function to run.

Example-1

chain referencing variable that contains string of functions separated by pipe symbol.

Data:

```
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166-173
 switchport trunk allowed vlan add 400,401,410
```

Template:

```
<vars>
vlans = "unrange(rangechar='-', joinchar=',') | split(',') | join(':') | joinmatches(
↪':')"
</vars>

<group name="interfaces">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans | chain('vlans') }}
</group>
```

Result:

```
{
    "interfaces": {
        "interface": "GigabitEthernet3/3",
        "trunk_vlans": "138:166:167:168:169:170:171:172:173:400:401:410"
    }
}
```

Example-2

chain referencing variable that contains list of strings, each string is a function.

Data:

```
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166-173
 switchport trunk allowed vlan add 400,401,410
```

Template:

```
<vars>
vlans = [
   "unrange(rangechar='-', joinchar=',')",
   "split(',')",
   "join(':')",
   "joinmatches(':')"
]
</vars>

<group name="interfaces">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans | chain('vlans') }}
</group>
```

Result:

```
{
    "interfaces": {
        "interface": "GigabitEthernet3/3",
        "trunk_vlans": "138:166:167:168:169:170:171:172:173:400:401:410"
    }
}
```

record

`{{ name | record(var_name) }}`

- var_name (mandatory) - template variable name that should be used to record match result

Record match results in template variable with given name. That recorded variable can be referenced within other functions such as *set* or retrieved from _ttp_ dictionary within macro.

Variables are recorded in two scopes:

1. Per-Input scope - all groups that parse this particular input will have access to recorded variable; variable stored in `_ttp_["parser_object"].vars` dictionary
2. Global scope - variable available from any group at any template; variable stored in `_ttp_["global_vars"]` dictionary

Warning: record results override one another, meaning if several match variable record result in same template variable, match variable that was matched later will override previous match result.

Example

Template:

```
<input load="text" name="in1">
myswitch1#show run int
interface Vlan778
 ip vrf forwarding VRF_NAME_1
 ip address 2002:fd37::91/124
!
</input>

<input load="text" name="in2">
myswitch2#show run int
interface Vlan779
 description some description input2
!
interface Vlan780
 switchport port-security mac 4
!
</input>

<group name="interfaces" input="in1">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 ip vrf forwarding {{ vrf | record("VRF") }}
 switchport port-security mac {{ sec_mac }}
</group>

<group name="interfaces" input="in2">
interface {{ interface }}
 description {{ description | ORPHRASE | record("my_description") }}
 switchport port-security mac {{ sec_mac }}
 {{ my_vrf | set("VRF") }}
 {{ my_descript | set("my_description") }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "interface": "Vlan778",
            "ip": "2002:fd37::91",
            "mask": "124",
            "vrf": "VRF_NAME_1"
        }
    },
    {
        "interfaces": [
            {
                "description": "some description input2",
                "interface": "Vlan779",
                "my_descript": "some description input2",
```

(continues on next page)

(continued from previous page)

```
                "my_vrf": "VRF_NAME_1"
            },
            {
                "interface": "Vlan780",
                "my_descript": "some description input2",
                "my_vrf": "VRF_NAME_1",
                "sec_mac": "4"
            }
        ]
    }
]
```

In above example `{{ my_vrf | set("VRF") }}` uses "VRF" variable from Global scope, while `{{ my_descript | set("my_description") }}` retrieves "my_description" variable value from per-input scope.

let

`{{ variable | let(var_name, value) }}` or `{{ variable | let(value) }}`

- value (mandatory) - a string containing value to be assigned to variable

Statically assigns provided value to variable with name var_name, if single argument provided, that argument considered to be a value and will be assigned to match variable replacing match result.

Example

Template:

```
<input load="text">
interface Loopback0
 description Management
 ip address 192.168.0.113/24
!
</input>

<group name="interfaces">
interface {{ interface }}
 description {{ description | let("description_undefined") }}
 ip address {{ ip | contains("24") | let("netmask", "255.255.255.0") }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "description": "description_undefined",
            "interface": "Loopback0",
            "ip": "192.168.0.113/24",
            "netmask": "255.255.255.0"
        }
    }
]
```

truncate

```
{{ name | truncate(count) }}
```

- count (mandatory) - integer to count the number of words to remove

Splits match result using ” “(space) char and joins it back up to truncate value. This function can be useful to shorten long match results.

Example

If match is “foo bar foo-bar” and truncate(2) will produce “foo bar”.

joinmatches

```
{{ name | joinmatches(char) }}
```

- char (optional) - character to use to join matches, default is new line ‘\n’

Join results from different matches into a single result string using provider character or string.

Example

Data:

```
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166,173
 switchport trunk allowed vlan add 400,401,410
```

Template:

```
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans | joinmatches(',') }}
```

Result:

```
{
    "interface": "GigabitEthernet3/3"
    "trunkVlans": "138,166,173,400,401,410"
}
```

resub

```
{{ name | resub(old, new, count) }}
```

- old (mandatory) - pattern to be replaced, can reference template variable name
- new (mandatory) - pattern to be replaced with
- count(optional) - digit, default is 1, indicates count of replacements to do

Performs re.sub(old, new, match, count) on match result and returns produced value

Example

Data:

```
interface GigabitEthernet3/3
```

Template is:

```
interface {{ interface | resub(old = '^GigabitEthernet'), new = 'Ge'}}
```

Result:

```
{
    "interface": "Ge3/3"
}
```

join

```
{{ name | match(char) }}
```

- char (mandatory) - character to use to join match

Run joins against match result using provided character and return string

Example-1:

Match is a string here and running join against it will insert '.' in between each character

Data:

```
description someimportantdescription
```

Template is:

```
description {{ description | join('.') }}
```

Result:

```
{
    "description": "s.o.m.e.i.m.p.o.r.t.a.n.t.d.e.s.c.r.i.p.t.i.o.n"
}
```

Example-2:

After running split function match result transformed into list object, running join against list will produce string with values separated by ":" character

Data:

```
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166,173,400,401,410
```

Template:

```
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans | split(',') | join(':') }}
```

Result:

```
{
    "interface": "GigabitEthernet3/3"
    "trunkVlans": "138:166:173:400:401:410"
}
```

append

```
{{ name | append(string) }}
```

- string (mandatory) - string to append

Appends string to match result and returns produced value

Example

Data:

```
interface Ge3/3
```

Template is:

```
interface {{ interface | append(' - non production') }}
```

Result:

```
{
    "interface": "Ge3/3 - non production"
}
```

prepend

```
{{ name | prepend(string) }}
```

- string (mandatory) - string to prepend

Prepends string to match result and returns produced value

print

```
{{ name | print }}
```

Will print match result to terminal as is at the given position, can be used for debugging purposes

Example

Data:

```
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166,173
```

Template:

```
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans | split(',') | print | join(':')␣
↪print }}
```

Results printed to terminal:

```
['138', '166', '173']  <--First print statement
138:166:173            <--Second print statement
```

unrange

```
{{ name | unrange('rangechar', 'joinchar') }}
```

- rangechar (mandatory) - character to indicate range
- joinchar (mandatory) - character used to join range items

If match result has integer range in it, this function can be used to extend that range to specific values, For instance if range is 100-105, after passing that result through this function result '101,102,103,104,105' will be produced. That is useful to extend trunk vlan ranges configured on interface.

Example

Data:

```
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166,170-173
```

Template:

```
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans | unrange(rangechar='-', joinchar=',
↪') }}
```

Result:

```
{
    "interface": "GigabitEthernet3/3"
    "trunkVlans": "138,166,170,171,172,173"
}
```

set

```
{{ name | set('var_set_value') }}
```

- var_set_value (mandatory) - string to set as a value for variable, can be a name of template variable.

Not all configuration statements have variables or values associated with them, but can serve as an indicator if particular feature disabled or enabled, to match such a cases *set* function can be used. This function allows to assign "var_set_value" to match variable, "var_set_value" considered to be a reference to template variable name, if no template variable with "var_set_value" found, "var_set_value" itself will be assigned to match variable.

It is also possible to use *set* function to introduce arbitrary key-value pairs in match result if set function used without any text in front of it.

Example-1

Conditional set function - set only will be invoked in case if preceding line matched. In below example " switchport trunk encapsulation dot1q" line will be searched for, if found, "encap" variable will have "dot1q" value set.

Data:

```
interface GigabitEthernet3/4
 switchport mode access
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport nonegotiate
 shutdown
```

(continues on next page)

(continued from previous page)

```
!
interface GigabitEthernet3/7
 switchport mode access
 switchport mode trunk
 switchport nonegotiate
!
```

Template:

```
<vars>
mys_set_var = "my_set_value"
</vars>

<group name="interfacesset">
interface {{ interface }}
 switchport mode access {{ mode_access | set("True") }}
 switchport trunk encapsulation dot1q {{ encap | set("dot1q") }}
 switchport mode trunk {{ mode | set("Trunk") }} {{ vlans | set("all_vlans") }}
 shutdown {{ disabled | set("True") }} {{ test_var | set("mys_set_var") }}
!{{ _end_ }}
</group>
```

Result:

```
{
    "interfacesset": [
        {
            "disabled": "True",
            "encap": "dot1q",
            "interface": "GigabitEthernet3/4",
            "mode": "Trunk",
            "mode_access": "True",
            "test_var": "my_set_value",
            "vlans": "all_vlans"
        },
        {
            "interface": "GigabitEthernet3/7",
            "mode": "Trunk",
            "mode_access": "True",
            "vlans": "all_vlans"
        }
    ]
}
```

Note: Multiple set statements are supported within the line, however, no other variables can be specified except with *set*, as match performed based on the string preceding variables with *set* function, for instance below will not work: `switchport mode {{ mode }} {{ switchport_mode | set('Trunk') }} {{ trunk_vlans | set('all') }}`

Example-2

Unconditional set - in this example "interface_role" will be statically set to "Uplink", but value for "provider" variable will be taken from template variable "my_var" and set to "L2VC".

Data:

```
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
!
```

Template:

```
<vars>
my_var = "L2VC"
</vars>

<group>
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
  {{ interface_role | set("Uplink") }}
  {{ provider | set("my_var") }}
!{{_end_}}
</group>
```

Result:

```
[
    {
        "description": "Management",
        "interface": "Vlan777",
        "interface_role": "Uplink",
        "ip": "192.168.0.1",
        "mask": "24",
        "provider": "L2VC",
        "vrf": "MGMT"
    }
]
```

replaceall

```
{{ name | replaceall('value1', 'value2', ..., 'valueN') }}
```

- value (mandatory) - string to replace in match

Run string replace method on match with *new* and *old* values derived using below rules.

Case 1 If only one value given *new* set to '' empty value, if several values specified *new* set to first value

Example-1.1 With *new* set to '' empty value

Data:

```
interface GigabitEthernet3/3
interface GigEthernet5/7
interface GeEthernet1/5
```

Template:

```
interface {{ interface | replaceall('Ethernet') }}
```

Result:

```
{'interface': 'Gigabit3/3'}
{'interface': 'Gig5/7'}
{'interface': 'Ge1/5'}
```

Example-1.2 With *new* set to 'Ge'

Data:

```
interface GigabitEthernet3/3
interface GigEth5/7
interface Ethernet1/5
```

Template:

```
interface {{ interface | replaceall('Ge', 'GigabitEthernet', 'GigEth', 'Ethernet') }}
```

Result:

```
{'interface': 'Ge3/3'}
{'interface': 'Ge5/7'}
{'interface': 'Ge1/5'}
```

Case 2 If value found in variables that variable used, if variable value is a list, function will iterate over list and for each item run replace where *new* set either to "" empty or to first value and *old* equal to each list item

Example-2.1 With *new* set to 'GE' value

Data:

```
interface GigabitEthernet3/3
interface GigEthernet5/7
interface GeEthernet1/5
```

Template:

```
<vars load="python">
intf_replace = ['GigabitEthernet', 'GigEthernet', 'GeEthernet']
</vars>

<group name="ifs">
interface {{ interface | replaceall('GE', 'intf_replace') }}
<group>
```

Result:

```
{
    "ifs": [
        {
            "interface": "GE3/3"
        },
        {
            "interface": "GE5/7"
        },
        {
            "interface": "GE1/5"
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Example-2.2 With *new* set to '' empty value

Data:

```
interface GigabitEthernet3/3
interface GigEthernet5/7
interface GeEthernet1/5
```

Template:

```
<vars load="python">
intf_replace = ['GigabitEthernet', 'GigEthernet', 'GeEthernet']
</vars>

<group name="ifs">
interface {{ interface | replaceall('intf_replace') }}
<group>
```

Result:

```
{
    "ifs": [
        {
            "interface": "3/3"
        },
        {
            "interface": "5/7"
        },
        {
            "interface": "1/5"
        }
    ]
}
```

Case 3 If value found in variables that variable used, if variable value is a dictionary, function will iterate over dictionary items and set *new* to item key and *old* to item value.

- If item value is a list, function will iterate over list and run replace using each entry as *old* value
- If item value is a string, function will use that string as *old* value

Example-3.1 With dictionary values as lists

Data:

```
interface GigabitEthernet3/3
interface GigEthernet5/7
interface GeEthernet1/5
interface Loopback1/5
interface TenGigabitEth3/3
interface TeGe5/7
interface 10GE1/5
```

Template:

```
<vars load="python">
intf_replace = {
                'Ge': ['GigabitEthernet', 'GigEthernet', 'GeEthernet'],
                'Lo': ['Loopback'],
                'Te': ['TenGigabitEth', 'TeGe', '10GE']
                }
</vars>

<group name="ifs">
interface {{ interface | replaceall('intf_replace') }}
<group>
```

Result:

```
{
    "ifs": [
        {
            "interface": "Ge3/3"
        },
        {
            "interface": "Ge5/7"
        },
        {
            "interface": "Ge1/5"
        },
        {
            "interface": "Lo1/5"
        },
        {
            "interface": "Te3/3"
        },
        {
            "interface": "Te5/7"
        }
    ]
}
```

resuball

`{{ name | resuball('value1', 'value2', ..., 'valueN') }}`

- value(mandatory) - string to replace in match, can reference template variable name.

Same as *replaceall* but instead of string replace this function runs python re substitute method, allowing the use of regular expression to match *old* values.

Example

If *new* set to "Ge" and *old* set to "GigabitEthernet", running string replace against "TenGigabitEthernet" match will produce "Ten" as undesirable result, to overcome that problem regular expressions can be used. For instance, regex "^GigabitEthernet" will only match "GigabitEthernet3/3" as "^" symbol indicates beginning of the string and will not match "GigabitEthernet" in "TenGigabitEthernet".

Data:

```
interface GigabitEthernet3/3
interface TenGigabitEthernet3/3
```

Template:

```
<vars load="python">
intf_replace = {
                'Ge': ['^GigabitEthernet'],
                'Te': ['^TenGigabitEthernet']
                }
</vars>

<group name="ifs">
interface {{ interface | resuball('intf_replace') }}
<group>
```

Result:

```
{
    "ifs": [
        {
            "interface": "Ge3/3"
        },
        {
            "interface": "Ge5/7"
        },
        {
            "interface": "Ge1/5"
        },
        {
            "interface": "Lo1/5"
        },
        {
            "interface": "Te3/3"
        },
        {
            "interface": "Te5/7"
        }
    ]
}
```

lookup

`{{ name | lookup('name', 'group', 'template', 'add_field') }}`

- name - name of lookup tag and dot-separated path to data within which to perform lookup
- group - dot-separated path to group results to use for lookup
- template - dot-separated path to template results to use for lookup
- add_field - default is False, can be set to string that will indicate name of the new field

Lookup function takes match result value and performs lookup on that value in lookup data structure. Lookup data is a dictionary where keys checked if they are equal to math result.

If lookup was unsuccessful no changes introduces to match result, if it was successful we have two option on what to do with found values: * if add_field is False - match result replaced with found values * if add_field is not False - string passed as add_field value used as a name for additional field that will be added to group match results

Warning: if one group uses results of another group for lookup, these groups must use separate inputs, groups that parse same input data, cannot use each other results for lookup, this is due to the way how TTP combines results on a per-input basis.

Example-1 *add_field* set to False

In this example, as 65101 will be looked up in the lookup table and replaced with found values

Data:

```
router bgp 65100
  neighbor 10.145.1.9
    remote-as 65101
  !
  neighbor 192.168.101.1
    remote-as 65102
```

Template:

```
<lookup name="ASNs" load="csv">
ASN,as_name,as_description
65100,Customer_1,Private ASN for CN451275
65101,CPEs,Private ASN for FTTB CPEs
</lookup>

<group name="bgp_config">
router bgp {{ bgp_as }}
 <group name="peers">
  neighbor {{ peer }}
    remote-as {{ remote_as | lookup('ASNs') }}
 </group>
</group>
```

Result:

```
{
    "bgp_config": {
        "bgp_as": "65100",
        "peers": [
            {
                "peer": "10.145.1.9",
                "remote_as": {
                    "as_description": "Private ASN for FTTB CPEs",
                    "as_name": "CPEs"
                }
            },
            {
                "peer": "192.168.101.1",
                "remote_as": "65102"
            }
        ]
    }
}
```

Example-2 With additional field

Data:

```
router bgp 65100
  neighbor 10.145.1.9
    remote-as 65101
  !
  neighbor 192.168.101.1
    remote-as 65102
```

Template:

```
<lookup name="ASNs" load="csv">
ASN,as_name,as_description
65100,Customer_1,Private ASN for CN451275
65101,CPEs,Private ASN for FTTB CPEs
</lookup>

<group name="bgp_config">
router bgp {{ bgp_as }}
 <group name="peers">
  neighbor {{ peer }}
    remote-as {{ remote_as | lookup('ASNs', add_field='asn_details') }}
 </group>
</group>
```

Result:

```
{
    "bgp_config": {
        "bgp_as": "65100",
        "peers": [
            {
                "asn_details": {
                    "as_description": "Private ASN for FTTB CPEs",
                    "as_name": "CPEs"
                },
                "peer": "10.145.1.9",
                "remote_as": "65101"
            },
            {
                "peer": "192.168.101.1",
                "remote_as": "65102"
            }
        ]
    }
}
```

Example-3

This example uses group “interfaces_data” results to perform lookup and add additional data in results produced by “arp” group

Template:

```
<input name="interfaces_data" load="text">
interface FastEthernet2.13
 description Customer CPE interface
 ip address 10.12.13.1 255.255.255.0
 vrf forwarding CPE-VRF
!
```

(continues on next page)

(continued from previous page)

```
interface GigabitEthernet2.13
 description Customer CPE interface
 ip address 10.12.14.1 255.255.255.0
 vrf forwarding CUST1
!
</input>

<group name="interfaces.{{ interface }}" input="interfaces_data">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ subnet | PHRASE | to_ip | network | to_str }}
 vrf forwarding {{ vrf }}
</group>

<input name="arp_data" load="text">
Protocol  Address     Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.2         98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.14.3        131   0150.7685.14d5  ARPA   GigabitEthernet2.13
</input>

<group name="arp" input="arp_data">
Internet  {{ ip }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface | lookup(group=
↪"interfaces", add_field="subnet_info") }}
</group>
```

Results:


```
[
    [
        {
            "interfaces": {
                "FastEthernet2.13": {
                    "description": "Customer CPE interface",
                    "subnet": "10.12.13.0/24",
                    "vrf": "CPE-VRF"
                },
                "GigabitEthernet2.13": {
                    "description": "Customer CPE interface",
                    "subnet": "10.12.14.0/24",
                    "vrf": "CUST1"
                }
            }
        },
        {
            "arp": [
                {
                    "age": "98",
                    "interface": "FastEthernet2.13",
                    "ip": "10.12.13.2",
                    "mac": "0950.5785.5cd1",
                    "subnet_info": {
                        "description": "Customer CPE interface",
                        "subnet": "10.12.13.0/24",
                        "vrf": "CPE-VRF"
                    }
                },
                {
```


(continues on next page)

(continued from previous page)

```
                    "age": "131",
                    "interface": "GigabitEthernet2.13",
                    "ip": "10.12.14.3",
                    "mac": "0150.7685.14d5",
                    "subnet_info": {
                        "description": "Customer CPE interface",
                        "subnet": "10.12.14.0/24",
                        "vrf": "CUST1"
                    }
                }
            ]
        }
    ]
]
```

Example-4

In this example, second template uses template "interfaces_data" results to perform lookup by denoting name of the template and path to lookup data in "interfaces_data.interfaces" lookup function template argument.

Template:

```
<template name="interfaces_data">
<input load="text">
interface FastEthernet2.13
 description Customer CPE interface
 ip address 10.12.13.1 255.255.255.0
 vrf forwarding CPE-VRF
!
interface GigabitEthernet2.13
 description Customer CPE interface
 ip address 10.12.14.1 255.255.255.0
 vrf forwarding CUST1
!
</input>

<group name="interfaces.{{ interface }}">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ subnet | PHRASE | to_ip | network | to_str }}
 vrf forwarding {{ vrf }}
</group>
</template>

<template>
<input load="text">
Protocol  Address     Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.2        98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.14.3       131   0150.7685.14d5  ARPA   GigabitEthernet2.13
</input>

<group name="arp">
Internet  {{ ip }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface |␣
↪lookup(template="interfaces_data.interfaces", add_field="subnet_info") }}
</group>
</template>
```

Results:

```
[
    [
        {
            "interfaces": {
                "FastEthernet2.13": {
                    "description": "Customer CPE interface",
                    "subnet": "10.12.13.0/24",
                    "vrf": "CPE-VRF"
                },
                "GigabitEthernet2.13": {
                    "description": "Customer CPE interface",
                    "subnet": "10.12.14.0/24",
                    "vrf": "CUST1"
                }
            }
        }
    ],
    [
        {
            "arp": [
                {
                    "age": "98",
                    "interface": "FastEthernet2.13",
                    "ip": "10.12.13.2",
                    "mac": "0950.5785.5cd1",
                    "subnet_info": {
                        "description": "Customer CPE interface",
                        "subnet": "10.12.13.0/24",
                        "vrf": "CPE-VRF"
                    }
                },
                {
                    "age": "131",
                    "interface": "GigabitEthernet2.13",
                    "ip": "10.12.14.3",
                    "mac": "0150.7685.14d5",
                    "subnet_info": {
                        "description": "Customer CPE interface",
                        "subnet": "10.12.14.0/24",
                        "vrf": "CUST1"
                    }
                }
            ]
        }
    ]
]
```


rlookup

```
{{ name | rlookup('name', 'add_field') }}
```

- name(mandatory) - rlookup table name and dot-separated path to data within which to perform search
- add_field(optional) - default is False, can be set to string that will indicate name of the new field

This function searches rlookup table keys in match value. rlookup table is a dictionary data where keys checked if they are equal to math result.

If lookup was unsuccessful no changes introduces to match result, if it was successful we have two options: * if add_field is False - match Result replaced with found values * if add_field is not False - string passed as add_field used as a name for additional field to be added to group results, value for that new field is a data from lookup table

Example

In this example, bgp neighbors descriptions set to hostnames of peering devices, usually hostnames tend to follow some naming convention to indicate physical location of device or its network role, in below example, naming convention is *<state>-<city>-<role><num>*

Data:

```
router bgp 65100
  neighbor 10.145.1.9
    description vic-mel-core1
  !
  neighbor 192.168.101.1
    description qld-bri-core1
```

Template:

```
<lookup name="locations" load="ini">
[cities]
-mel- : 7 Name St, Suburb A, Melbourne, Postal Code
-bri- : 8 Name St, Suburb B, Brisbane, Postal Code
</lookup>

<group name="bgp_config">
router bgp {{ bgp_as }}
 <group name="peers">
  neighbor {{ peer }}
    description {{ remote_as | rlookup('locations.cities', add_field='location') }}
 </group>
</group>
```

Result:

```
{
    "bgp_config": {
        "bgp_as": "65100",
        "peers": [
            {
                "description": "vic-mel-core1",
                "location": "7 Name St, Suburb A, Melbourne, Postal Code",
                "peer": "10.145.1.9"
            },
            {
                "description": "qld-bri-core1",
                "location": "8 Name St, Suburb B, Brisbane, Postal Code",
                "peer": "192.168.101.1"
            }
        ]
    }
}
```

gpvlookup

```
{{ name | gpvlookup('name', 'add_field', 'record', 'multimatch') }}
```

- name - name of lookup tag and dot-separated path to data within which to perform lookup
- add_field - default is False, can be set to string that will indicate name of the new field to add with lookup results
- record - default is False, if True will record lookup results in TTP global and parsing object variables for reference by 'set' function
- multimatch - default is False, will return first match only as lookup result, if True will iterate over all pasterns and return all found lookup matches

Glob Patterns Values Lookup (gpvookup) function takes match result value and performs lookup on it using lookup data structure. This function can be useful to classify matching results and en-reach parsing output with additional information.

Lookup data is a dictionary of key value pairs, where value is a list of Unix glob patterns to check, if at least one pattern matches, key added to found values list. Found values list is a result produced by this function.

If lookup was unsuccessful no changes introduces to match result, if it was successful we have two option on what to do with found values: * if add_field is False - match result replaced with found values list * if add_field is not False - string passed as add_field value used as a name for additional field that will be added to group match results

If record set to True, gpvlookup function will record found values list in TTP parser and global variables scopes.

Example-1

Basic example of gpvlookup usage. Here matched hostnames got classified by network domain based on glob patterns matching against them.

Template:

```
<input load="text">
hostname DC1-SW-2
hostname A1-CORP-SW-2
hostname WIFI-CORE-RT-1
hostname DC2-CORP-FW-02
</input>

<lookup name="domains" load="python">
{
    "NETWORK_DOMAINS": {
        "corporate": ["*CORP*", "WIFI-*"],
        "datacentre": ["DC1-*", "DC2-*"]
    }
}
</lookup>

<group name="devices">
hostname {{ hostname | gpvlookup("domains.NETWORK_DOMAINS", add_field="Network Domains
↪") }}
</group>
```

Results:

```
[
    [
        {
            "devices": [
                {
                    "Network Domains": [
                        "datacentre"
                    ],
```

(continues on next page)

(continued from previous page)

```
                    "hostname": "DC1-SW-2"
                },
                {
                    "Network Domains": [
                        "corporate"
                    ],
                    "hostname": "A1-CORP-SW-2"
                },
                {
                    "Network Domains": [
                        "corporate"
                    ],
                    "hostname": "WIFI-CORE-RT-1"
                },
                {
                    "Network Domains": [
                        "corporate"
                    ],
                    "hostname": "DC2-CORP-FW-02"
                }
            ]
        }
    ]
]
```

Because lookup data is actually a dictionary, first match will be non-deterministic. For instance, in above example hostname DC2-CORP-FW-02 was matched by "corporate" patterns, but not by "datacentre" patterns, even though "datacentre" patterns would produce positive match as well.

Example-2

In this example multimatch used to collect all matches, in addition to that values found by lookup will be recorded in variable "domain" using "record" argument.

Template:

```
<input load="text">
hostname DC1-WIFI-CORE-RT-1
!
interface Lo0
 ip address 5.3.3.3/32
</input>

<input load="text">
hostname WIFI-CORE-RT-1
!
interface Lo0
 ip address 6.3.3.3/32
</input>

<lookup name="domains" load="python">
{
    "NETWORK_DOMAINS": {
        "corporate": ["*WIFI-*"],
        "datacentre": ["DC1-*"]
    }
}
```

(continues on next page)

(continued from previous page)

```
</lookup>

<group void="">
hostname {{ hostname | gpvlookup("domains.NETWORK_DOMAINS", multimatch=True, record=
↪"domain") }}
</group>

<group name="device.{{ interface }}">
interface {{ interface }}
 ip address {{ ip }}
 {{ domain | set(domain) }}
</group>
```

Results:

```
[
    [
        {
            "device": {
                "Lo0": {
                    "domain": [
                        "corporate",
                        "datacentre"
                    ],
                    "ip": "5.3.3.3/32"
                }
            }
        },
        {
            "device": {
                "Lo0": {
                    "domain": [
                        "corporate"
                    ],
                    "ip": "6.3.3.3/32"
                }
            }
        }
    ]
]
```

Group function “void” used to deny match results for this particular group to make output cleaner.

geoip_lookup

```
{{ name | geoip_lookup(db_name, add_field) }}
```

- db_name - Name of the input that contains GeoIP2 database OS absolute path, supporteddatabases are ASN, Country or City
- add_field - default is “geoip_lookup”, can be set to string that will indicate name of new field to use for lookup results

geoip_lookup function use GeoIP2 databases to create Python geoip2 module lookup objects that can be used to enreach results output with information about BGP ASN, Country or City associated with given IP address. db_name reference to lookup tag name with database type separated by dot, such as *lookup_tag_name.database_name*, reference *geoip2 database* on how to properly structure lookup tag.

This function need valid IPv4 orIPv6 address as an input to perfrom lookup against.

Prerequisites

Relies on Python geoip2 module, hence it need to be installed on the system.

Example

Template:

```
<input load="text">
interface Lo0
 ip address 123.209.0.1 32
</input>

<lookup name="geoip2_test" database="geoip2">
citY    = 'C:/path/to/GeoLite2-City.mmdb'
AsN     = 'C:/path/to/GeoLite2-ASN.mmdb'
Country = 'C:/path/to/GeoLite2-Country.mmdb'
</lookup>

<group name="intf_with_city_data">
interface {{ interface }}
 ip address {{ ip | geoip_lookup(db_name="geoip2_test.citY", add_field="city_data") }}
↪ {{ mask }}
</group>

<group name="intf_with_asn_data">
interface {{ interface }}
 ip address {{ ip | geoip_lookup("geoip2_test.AsN", add_field="asn_data") }} {{ mask }
↪}
</group>

<group name="intf_with_country_data">
interface {{ interface }}
 ip address {{ ip | geoip_lookup("geoip2_test.Country", "country_data") }} {{ mask }}
</group>
```

Results:

```
[
    [
        {
            "intf_with_asn_data": {
                "asn_data": {
                    "ASN": 1221,
                    "network": "123.209.0.0/16",
                    "organization": "Telstra Corporation Ltd"
                },
                "interface": "Lo0",
                "ip": "123.209.0.1",
                "mask": "32"
            },
            "intf_with_city_data": {
                "city_data": {
                    "accuracy_radius": 100,
                    "city": "Olinda",
                    "continent": "Oceania",
                    "country": "Australia",
```

(continues on next page)

(continued from previous page)


```
                    "country_iso_code": "AU",
                    "latitude": -37.8596,
                    "longitude": 145.3711,
                    "network": "123.209.0.0/19",
                    "postal_code": "3788",
                    "state": "Victoria",
                    "state_iso_code": "VIC"
                },
                "interface": "Lo0",
                "ip": "123.209.0.1",
                "mask": "32"
            },
            "intf_with_country_data": {
                "country_data": {
                    "continent": "Oceania",
                    "continent_code": "OC",
                    "country": "Australia",
                    "country_iso_code": "AU",
                    "network": "123.208.0.0/14"
                },
                "interface": "Lo0",
                "ip": "123.209.0.1",
                "mask": "32"
            }
        }
    ]
]
```


startswith_re

```
{{ name | startswith_re('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

Python re search used to evaluate if match value starts with given string pattern, returns True if so and False otherwise

endswith_re

```
{{ name | endswith_re('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

Python re search used to evaluate if match value ends with given string pattern, returns True if so and False otherwise

contains_re

```
{{ name | contains_re('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

Python re search used to evaluate if match value contains given string pattern, returns True if so and False otherwise

contains

```
{{ name | contains('pattern1, pattern2, ... , patternN') }}
```

- patternN - string pattern to check or name of variable from <vars> tag.

This function evaluates if match value contains at least one of the given patterns, returns True if so and False otherwise.

Example

contains can be used to filter group results based on filtering start REs, for instance, if we have configuration of networking device and we want to extract information only about *Vlan* interfaces.

Data:

```
interface Vlan123
 description Desks vlan
 ip address 192.168.123.1 255.255.255.0
!
interface GigabitEthernet1/1
 description to core-1
!
interface Vlan222
 description Phones vlan
 ip address 192.168.222.1 255.255.255.0
!
interface Loopback0
 description Routing ID loopback
```

Template:

```
<group name="SVIs">
interface {{ interface | contains('Vlan') }}
 description {{ description | ORPHRASE}}
 ip address {{ ip }} {{ mask }}
</group>
```

Result:

```
{
    "SVIs": [
        {
            "description": "Desks vlan",
            "interface": "Vlan123",
            "ip": "192.168.123.1",
            "mask": "255.255.255.0"
        },
        {
            "description": "Phones vlan",
            "interface": "Vlan222",
            "ip": "192.168.222.1",
            "mask": "255.255.255.0"
        }
    ]
}
```

If first line in the group contains match variables it is considered start re, if start re condition check result evaluated to *False*, all the matches that belong to this group will be filtered. In example above line "interface {{ interface | contains('Vlan') }}" is a start re, hence if "interface" variable match will not contain "Vlan", group results will be discarded.

notstartswith_re

```
{{ name | notstartswith_re('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

Python re search used to evaluate if match value starts with given string pattern, returns False if so and True otherwise

notendswith_re

```
{{ name | notendswith_re('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

Python re search used to evaluate if match value ends with given string pattern, returns False if so and True otherwise

exclude_re

```
{{ name | exclude_re('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

Python re search used to evaluate if match value contains given string pattern, returns False if so and True otherwise

exclude

```
{{ name | exclude('pattern') }}
```

- pattern(mandatory) - string pattern to check or name of variable from <vars> tag.

This function evaluates if match value contains given string pattern, returns False if so and True otherwise.

equal

```
{{ name | equal('value') }}
```

- value(mandatory) - string pattern to check or name of variable from <vars> tag.

This function evaluates if match is equal to given value, returns True if so and False otherwise

notequal

```
{{ name | notequal('value') }}
```

- value(mandatory) - string pattern to check or name of variable from <vars> tag.

This function evaluates if match is equal to given value, returns False if so and True otherwise

isdigit

```
{{ name | isdigit }}
```

This function checks if match is a digit, returns True if so and False otherwise

notdigit

```
{{ name | notdigit }}
```

This function checks if match is digit, returns False if so and True otherwise

greaterthan

```
{{ name | greaterthan('value') }}
```

- value(mandatory) - integer value to compare with

This function checks if match and supplied value are digits and performs comparison operation, if match is bigger than given value returns True and False otherwise

lessthan

```
{{ name | lessthan('value') }}
```

- value(mandatory) - integer value to compare with

This function checks if match and supplied value are digits and performs comparison, if match is smaller than provided value returns True and False otherwise

item

```
{{ name | item(item_index) }}
```

- item_index(mandatory) - integer, index of item to return

Return item value at given index of iterable. If match result (iterable) is string, *item* returns letter at given index, if match been transformed to list by the moment *item* function runs, returns list item at given index. item_index can be positive or negative digit, same rules as for retrieving list items applies e.g. if item_index is -1, last item will be returned.

In addition, ttp preforms index out of range checks, returning last or first item if item_index exceeds length of match result.

macro

```
{{ name | macro(macro_name) }}
```

- macro_name(mandatory) - name of macro function to pass match result through

Macro brings Python language capabilities to match results processing and validation during ttp module execution, as it allows to run custom functions against match results. Macro functions referenced by their name in match variable definitions or as a group *macro* attribute.

Warning: macro uses python `exec` function to parse code payload without imposing any restrictions, hence it is dangerous to run templates from untrusted sources as they can have macro defined in them that can be used to execute any arbitrary code on the system.

Macro function must accept only one attribute to hold match results, for match variable data supplied to macro function is a match result string.

For match variables, depending on data returned by macro function, ttp will behave differently according to these rules:

- If macro returns True or False - original data unchanged, macro handled as condition functions, invalidating result on False and keeps processing result on True
- If macro returns None - data processing continues, no additional logic associated
- If macro returns single item - that item replaces original data supplied to macro and processed further
- If macro return tuple of two elements - fist element must be string - match result, second - dictionary of additional fields to add to results

Note: Macro function contained within `<macro>` tag, each function loaded and saved into the dictionary of function name and function object, as a result cross referencing macro functions is not supported.

Example

In this example macro functions referenced in match variables.

Template:

```
<input load="text">
interface Vlan123
 description Desks vlan
 ip address 192.168.123.1 255.255.255.0
!
interface GigabitEthernet1/1
 description to core-1
!
interface Vlan222
 description Phones vlan
 ip address 192.168.222.1 255.255.255.0
!
interface Loopback0
 description Routing ID loopback
!
</input>

<macro>
def check_if_svi(data):
    if "Vlan" in data:
        return data, {"is_svi": True}
    else:
       return data, {"is_svi": False}

def check_if_loop(data):
    if "Loopback" in data:
        return data, {"is_loop": True}
    else:
       return data, {"is_loop": False}
</macro>

<macro>
def description_mod(data):
    # To revert words order in descripotion
    words_list = data.split(" ")
    words_list_reversed = list(reversed(words_list))
```

(continues on next page)

(continued from previous page)

```
    words_reversed = " ".join(words_list_reversed)
    return words_reversed
</macro>

<group name="interfaces_macro">
interface {{ interface | macro("check_if_svi") | macro("check_if_loop") }}
 description {{ description | ORPHRASE | macro("description_mod")}}
 ip address {{ ip }} {{ mask }}
</group>
```

Result:


```
[
    {
        "interfaces_macro": [
            {
                "description": "vlan Desks",
                "interface": "Vlan123",
                "ip": "192.168.123.1",
                "is_loop": false,
                "is_svi": true,
                "mask": "255.255.255.0"
            },
            {
                "description": "core-1 to",
                "interface": "GigabitEthernet1/1",
                "is_loop": false,
                "is_svi": false
            },
            {
                "description": "vlan Phones",
                "interface": "Vlan222",
                "ip": "192.168.222.1",
                "is_loop": false,
                "is_svi": true,
                "mask": "255.255.255.0"
            },
            {
                "description": "loopback ID Routing",
                "interface": "Loopback0",
                "is_loop": true,
                "is_svi": false
            }
        ]
    }
]
```


to_list

```
{{ name | to_list }}
```

to_list transform match result in python list object in such a way that if match result is a string, empty lit will be created and result will be appended to it, if match result not a string by the time to_list function runs, this function does nothing.

Example

Template:

```
<input load="text" name="test1-18">
interface GigabitEthernet1/1
 description to core-1
 ip address 192.168.123.1 255.255.255.0
!
</input>
<group name="interfaces_functions_test1_18"
input="test1-18"
output="test1-18"
>
interface {{ interface }}
 description {{ description | ORPHRASE | split(" ") | to_list }}
 ip address {{ ip | to_list }} {{ mask }}
</group>
```

Result:

```
[{
    "interfaces_functions_test1_18": {
        "description": [
            "to",
            "core-1"
        ],
        "interface": "GigabitEthernet1/1",
        "ip": [
            "192.168.123.1"
        ],
        "mask": "255.255.255.0"
    }
}]
```

to_str

`{{ name | to_str }}`

This function transforms match result to string object running python `str(match_result)` built-in function, that is useful for such a cases when match result been transformed to some other object during processing and it needs to be converted back to string.

to_int

`{{ name | to_int }}`

This function will try to transforms match result into integer object running python `int(match_result)` built-in function, if it fails to do so, execution will continue, results will not e invalidated. to_int is useful if you need to convert string representation of integer in actual integer object to run mathematical operation with it.

to_ip

`{{ name | to_ip }}` or `{{ name | to_ip("ipv4") }}`

- to_ip(version) - uses python ipaddress module to transform match result in one of ipaddress supported objects, by default will use ipaddress module built-in logic to determine version of IP address, optionally version can

be provided using *ipv4* or *ipv6* arguments to create IPv4Address or IPv6Address ipaddress module objects. In addition ttp does the check to detect if slash "/" present - e.g. 137.168.1.3/27 - in match result or space " " present in match result - e.g. 137.168.1.3 255.255.255.224, if so it will create IPInterface, IPv4Interface or IPv6Interface object depending on provided arguments.

After match result transformed into ipaddress' IPaddress or IPInterface object, built-in functions and attributes of these objects can be called using match variable functions chains.

Note: reference ipaddress module documentation for complete list of functions and attributes

Example

It is often that devices use "ip address 137.168.1.3 255.255.255.224" syntaxes to configure interface's IP addresses, let's assume we need to convert it to "137.168.1.3/27" representation and vice versa.

Template:

```
<input load="text">
interface Loopback0
 ip address 1.0.0.3 255.255.255.0
!
interface Vlan777
 ip address 192.168.0.1/24
!
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip | PHRASE | to_ip | with_prefixlen }}
 ip address {{ ip | to_ip | with_netmask }}
</group>
```

Result:

```
[
    {
        "interfaces": [
            {
                "interface": "Loopback0",
                "ip": "1.0.0.3/24"
            },
            {
                "interface": "Vlan777",
                "ip": "192.168.0.1/255.255.255.0"
            }
        ]
    }
]
```

with_prefixlen and with_netmask are python ipaddress module IPv4Interface object's built-in functions.

to_net

```
{{ name | to_net }}
```

This function leverages python built-in ipaddress module to transform match result into IPNetwork object provided that match
or fe80:ab23::/64.

Example

Let's assume we need to get results for private routes only from below data, to_net can be used to transform match result into network object together with IPNetwork built-in function is_private to filter results.

Template:

```
<input load="text">
RP/0/0/CPU0:XR4#show route
i L2 10.0.0.2/32 [115/20] via 10.0.0.2, 00:41:40, tunnel-te100
i L2 172.16.0.3/32 [115/10] via 10.1.34.3, 00:45:11, GigabitEthernet0/0/0/0.34
i L2 1.1.23.0/24 [115/20] via 10.1.34.3, 00:45:11, GigabitEthernet0/0/0/0.34
</input>

<group name="routes">
{{ code }} {{ subcode }} {{ net | to_net | is_private | to_str }} [{{ ad }}/{{ metric␣
↪}}] via {{ nh_ip }}, {{ age }}, {{ nh_interface }}
</group>
```

Result:

```
[
    {
        "routes": [
            {
                "ad": "115",
                "age": "00:41:40",
                "code": "i",
                "metric": "20",
                "net": "10.0.0.2/32",
                "nh_interface": "tunnel-te100",
                "nh_ip": "10.0.0.2",
                "subcode": "L2"
            },
            {
                "ad": "115",
                "age": "00:45:11",
                "code": "i",
                "metric": "10",
                "net": "172.16.0.3/32",
                "nh_interface": "GigabitEthernet0/0/0/0.34",
                "nh_ip": "10.1.34.3",
                "subcode": "L2"
            }
        ]
    }
]
```

is_private check invalidated public 1.1.23.0/24 subnet and only private networks were included in results.

to_cidr

```
{{ name | to_cidr }}
```

Function to convert subnet mask in prefix length representation, for instance if match result is "255.255.255.0", to_cidr function will return "24"

ip_info

```
{{ name | ip_info }}
```

Python ipaddress module helps to convert plain text string into IP addresses objects, as part of that process ipaddress module calculates a lot of additional information, ip_info function retrieves that information from that object and returns it in dictionary format.

Example

Below loopback0 IP address will be converted to IPv4Address object and ip_info will return information about that IP only, for other interfaces ttp will be able to create IPInterface objects, that apart from IP details contains information about network.

Template:

```
<input load="text">
interface Loopback0
 ip address 1.0.0.3 255.255.255.0
!
interface Vlan777
 ip address 192.168.0.1/24
!
interface Vlan777
 ip address fe80::fd37/124
!
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip | to_ip | ip_info }} {{ mask }}
 ip address {{ ip | to_ip | ip_info }}
</group>
```

Result:

```
[
    {
        "interfaces": [
            {
                "interface": "Loopback0",
                "ip": {
                    "compressed": "1.0.0.3",
                    "exploded": "1.0.0.3",
                    "ip": "1.0.0.3",
                    "is_link_local": false,
                    "is_loopback": false,
                    "is_multicast": false,
                    "is_private": false,
                    "is_reserved": false,
                    "is_unspecified": false,
                    "max_prefixlen": 32,
                    "version": 4
                },
                "mask": "255.255.255.0"
            },
            {
                "interface": "Vlan777",
                "ip": {
```

(continues on next page)

(continued from previous page)


```
                    "broadcast_address": "192.168.0.255",
                    "compressed": "192.168.0.1/24",
                    "exploded": "192.168.0.1/24",
                    "hostmask": "0.0.0.255",
                    "hosts": 254,
                    "ip": "192.168.0.1",
                    "is_link_local": false,
                    "is_loopback": false,
                    "is_multicast": false,
                    "is_private": true,
                    "is_reserved": false,
                    "is_unspecified": false,
                    "max_prefixlen": 32,
                    "netmask": "255.255.255.0",
                    "network": "192.168.0.0/24",
                    "network_address": "192.168.0.0",
                    "num_addresses": 256,
                    "prefixlen": 24,
                    "version": 4,
                    "with_hostmask": "192.168.0.1/0.0.0.255",
                    "with_netmask": "192.168.0.1/255.255.255.0",
                    "with_prefixlen": "192.168.0.1/24"
                }
            },
            {
                "interface": "Vlan777",
                "ip": {
                    "broadcast_address": "fe80::fd3f",
                    "compressed": "fe80::fd37/124",
                    "exploded": "fe80:0000:0000:0000:0000:0000:0000:fd37/124",
                    "hostmask": "::f",
                    "hosts": 14,
                    "ip": "fe80::fd37",
                    "is_link_local": true,
                    "is_loopback": false,
                    "is_multicast": false,
                    "is_private": true,
                    "is_reserved": false,
                    "is_unspecified": false,
                    "max_prefixlen": 128,
                    "netmask": "ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff0",
                    "network": "fe80::fd30/124",
                    "network_address": "fe80::fd30",
                    "num_addresses": 16,
                    "prefixlen": 124,
                    "version": 6,
                    "with_hostmask": "fe80::fd37/::f",
                    "with_netmask": "fe80::fd37/
↪ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff0",
                    "with_prefixlen": "fe80::fd37/124"
                }
            }
        ]
    }
]
```

is_ip

```
{{ name | is_ip }}
```

is_ip function tries to convert provided match result in Python ipaddress module IPAddress or IPInterface object, if that happens without any exceptions (errors), is_ip returns True and False otherwise.

Example

Template:

```
<input load="text">
interface Loopback0
 ip address 192.168.0.113/24
!
interface Loopback1
 ip address 192.168.1.341/24
!
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip | is_ip }}
</group>
```

Result:

```
[
    {
        "interfaces": [
            {
                "interface": "Loopback0",
                "ip": "192.168.0.113/24"
            },
            {
                "interface": "Loopback1"
            }
        ]
    }
]
```

192.168.1.341/24 match result was invalidated as it is not a valid IP address.

cidr_match

```
{{ name | cidr_match(prefix) }}
```

- `prefix` - IPv4 or IPv6 prefix string, for instance '10.0.0.0/16' or name of <vars> tag variable.

This function allows to convert provided prefix in ipaddress IPNetwork object and convert match_result into IPInterface object, after that, cidr_match will run *overlaps* check to see if provided prefix and match result ip address overlapping, returning Trueif so and False otherwise, allowing to filter match results based on that.

Example-1

In example below, IP of Loopback1 interface is not overlapping with 192.168.0.0/16 range, hence it will be invalidated.

Template:

```
<input load="text">
interface Loopback0
 ip address 192.168.0.113/24
!
interface Loopback1
 ip address 10.0.1.251/24
!
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip | cidr_match("192.168.0.0/16") }}
</group>
```

Result:

```
[{
    "interfaces": [
        {
            "interface": "Loopback0",
            "ip": "192.168.0.113/24"
        },
        {
            "interface": "Loopback1"
        }
    ]
}]
```

Example-1

In example below, cidr_match references <vars> tag variable - subnet

Template:

```
<input load="text">
interface Lo0
ip address 124.171.238.50 32
!
interface Lo1
ip address 1.1.1.1 32
</input>

<vars>
subnet="1.1.1.0/24"
</vars>

<group contains="ip">
interface {{ interface }}
ip address {{ ip | cidr_match(subnet) }} {{ mask }}
</group>
```

Result:

```
[
    [
        {
            "interface": "Lo1",
            "ip": "1.1.1.1",
```

(continues on next page)

(continued from previous page)

```
            "mask": "32"
        }
    ]
]
```

dns

```
{{ name | dns(record='A', timeout=1, servers=[], add_field=False) }}
```

This function performs forward DNS lookup of match results and returns sorted list of IP addresses returned by DNS.

Prerequisites: dnspython needs to be installed

Options:

- `record` - by default perform 'A' lookup, any dnspython supported record can be given, e.g. 'AAAA' for IPv6 lookup
- `timeout` - default is 1 second, amount of time to wait for response, overall lifetime of operation will be set to number of servers multiplied by timeout
- `servers` - comma separated string of DNS servers to use for lookup, by default uses DNS servers configured on machine running the code
- `add_field` - boolean or string, if string, its value will be used as a key for DNS lookup results, if False - DNS lookup results will replace match results

If DNS will fail for whatever reason, match results will be returned without any modifications.

Example

Template:

```
<input load="text">
interface GigabitEthernet3/11
 description wikipedia.org
!
</input>

<group name="interfaces">
interface {{ interface }}
 description {{ description | dns }}
</group>

<group name="interfaces_dnsv6">
interface {{ interface }}
 description {{ description | dns(record='AAAA') }}
</group>

<group name="interfaces_dnsv4_google_dns">
interface {{ interface }}
 description {{ description | dns(record='A', servers='8.8.8.8') }}
</group>

<group name="interfaces_dnsv6_add_field">
interface {{ interface }}
 description {{ description | dns(record='AAAA', add_field='IPs') }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "description": [
                "103.102.166.224"
            ],
            "interface": "GigabitEthernet3/11"
        },
        "interfaces_dnsv4_google_dns": {
            "description": [
                "103.102.166.224"
            ],
            "interface": "GigabitEthernet3/11"
        },
        "interfaces_dnsv6": {
            "description": [
                "2001:df2:e500:ed1a::1"
            ],
            "interface": "GigabitEthernet3/11"
        },
        "interfaces_dnsv6_add_field": {
            "IPs": [
                "2001:df2:e500:ed1a::1"
            ],
            "description": "wikipedia.org",
            "interface": "GigabitEthernet3/11"
        }
    }
]
```

rdns

`{{ name | dns(timeout=1, servers=[], add_field=False) }}`

This function performs reverse DNS lookup of match results and returns FQDN obtained from DNS.

Prerequisites: dnspython needs to be installed

Arguments:

- `timeout` - default is 1 second, amount of time to wait for response, overall lifetime of operation will be set to number of servers multiplied by timeout
- `servers` - comma separated string of DNS servers to use for lookup, by default uses DNS servers configured on machine running the code
- `add_field` - boolean or string, if string, its value will be used as a key for DNS lookup results, if False - DNS lookup results will replace match results

If DNS will fail for whatever reason, match results will be returned without any modifications.

Example

Template:

```
<input load="text">
interface GigabitEthernet3/11
```

(continues on next page)

(continued from previous page)

```
 ip address 8.8.8.8 255.255.255.255
!
</input>

<group name="interfaces_rdns">
interface {{ interface }}
 ip address {{ ip | rdns }} {{ mask }}
</group>

<group name="interfaces_rdns_google_server">
interface {{ interface }}
 ip address {{ ip | rdns(servers='8.8.8.8') }} {{ mask }}
</group>

<group name="interfaces_rdns_add_field">
interface {{ interface }}
 ip address {{ ip | rdns(add_field='FQDN') }} {{ mask }}
</group>
```

Result:

```
[
    {
        "interfaces_rdns_add_field": {
            "FQDN": "dns.google",
            "interface": "GigabitEthernet3/11",
            "ip": "8.8.8.8",
            "mask": "255.255.255.255"
        },
        "interfaces_rdnsv4": {
            "interface": "GigabitEthernet3/11",
            "ip": "dns.google",
            "mask": "255.255.255.255"
        },
        "interfaces_rdnsv4_google_server": {
            "interface": "GigabitEthernet3/11",
            "ip": "dns.google",
            "mask": "255.255.255.255"
        }
    }
]
```

sformat

```
{{ name | sformat("value") }}
```

- value - string to format with match result or name of variable for from <vars> tag.

sformat allows to embed match result within arbitrary string using syntaxis supported by python built-in format function.

Example

Template:

```
<input load="text">
interface Vlan778
 ip address 2002:fd37::91/124
!
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip | sformat("ASN 65100 IP - {}") }}
</group>
```

Results:

```
[
    {
        "interfaces": {
            "interface": "Vlan778",
            "ip": "ASN 65100 IP - 2002:fd37::91/124"
        }
    }
]
```

uptimeparse

`{{ name | uptimeparse }}` or `{{ name | uptimeparse(format="seconds|dict") }}`

This function can be used to parse text strings of below format to extract uptime information:

```
2 years, 5 months, 27 weeks, 3 days, 10 hours, 46 minutes
27 weeks, 3 days, 10 hours, 46 minutes
10 hours, 46 minutes
1 minutes
```

Arguments:

- `format` - default is seconds, optional argument to specify format of returned results, if seconds - integer, number of seconds will be returned, if dict - will return a dictionary of extracted time

Example

Template:

```
<input load="text">
device-hostame uptime is 27 weeks, 3 days, 10 hours, 46 minutes, 10 seconds
</input>

<group name="uptime-1-seconds">
device-hostame uptime is {{ uptime | PHRASE | uptimeparse }}
</group>

<group name="uptime-2-dictionary">
device-hostame uptime is {{ uptime | PHRASE | uptimeparse(format="dict") }}
</group>
```

Results:

```
[
    {
        "uptime-1-seconds": {
            "uptime": 16627570
        },
        "uptime-2-dictionary": {
            "uptime": {
                "days": "3",
                "hours": "10",
                "mins": "46",
                "secs": "10",
                "weeks": "27"
            }
        }
    }
]
```

mac_eui

```
{{ name | mac_eui }}
```

This function normalizes mac address representation format by deleting `-:.` characters from mac address string and converting it into aa:bb:cc:dd:ee:ff. It also handles the case when mac address trailing zeros stripped by device in show commands output, by staffing zeros to make mac address 12 symbols long, e.g. aabb.ccdd.ee will be converted to aa:bb:cc:dd:ee:00

count

```
{{ name | count(var="per_input_counter", globvar="global_counter") }}
```

- var - string, name of per input variable to store count results
- globvar - string, name of global variable to store count results across several input datums

This function introduces counting capabilities, allowing to increase counter variable on every successful match. There are two types of count variables supported - per input and global, as the names imply, per input variable has input significance, while global variable can help to count matches across several inputs.

Example

Let's say we need to count a number of interfaces in up state for each device and across all devices.

Template:

```
<input name="device-1" load="text">
device-1#show ip int brief
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/2     unassigned      YES unset  up                    up
GigabitEthernet0/3     unassigned      YES unset  up                    up
GigabitEthernet0/4     unassigned      YES unset  down                  down
</input>

<input name="device-2" load="text">
device-2#show ip int brief
Interface              IP-Address      OK? Method Status                Protocol
Vlan20                 172.29.50.3     YES NVRAM  down                  down
Vlan41                 172.29.52.34    YES NVRAM  up                    up
```

(continues on next page)

(continued from previous page)

```
GigabitEthernet0/1     unassigned      YES unset  down                    down
</input>

<vars name="counters">
interfaces_up = 0
</vars>

<group name="interfaces*">
{{ interface }}  {{ ip }}  YES {{ ignore }}  {{ status | equal("up") | count(var=
↪"interfaces_up", globvar="overall_interfaces_up") }}   {{ protocol }}
</group>

<output macro="add_glob_counters"/>

<macro>
def add_glob_counters(data):
    data.append({ "overall_interfaces_up": _ttp_["global_vars"]["overall_interfaces_up
↪"] })
</macro>
```

Results:


```
[
    [
        {
            "counters": {
                "interfaces_up": 2
            },
            "interfaces": [
                {
                    "interface": "GigabitEthernet0/2",
                    "ip": "unassigned",
                    "protocol": "up",
                    "status": "up"
                },
                {
                    "interface": "GigabitEthernet0/3",
                    "ip": "unassigned",
                    "protocol": "up",
                    "status": "up"
                }
            ]
        },
        {
            "counters": {
                "interfaces_up": 1
            },
            "interfaces": [
                {
                    "interface": "Vlan41",
                    "ip": "172.29.52.34",
                    "protocol": "up",
                    "status": "up"
                }
            ]
        },
        {
```


(continues on next page)

(continued from previous page)

```
            "overall_interfaces_up": 3
        }
    ]
]
```

void

```
{{ name | void }}
```

The purpose of this function is to return False invalidating match results for this variable.

to_float

```
{{ name | to_float }}
```

This function tries to convert integer expressed as int (e.g. 2) or as a string (e,f, “45”) to python integer of float type, e.g. 2 will be converted to 2.0

to_unicode

```
{{ name | to_unicode }}
```

If python2 used to run TTP script, this function will try to convert match variable value to unicode string, e.g. string “abc” will become u”abc”

4.1.3 Regex Patterns

Regexes are in the heart of TTP, but they hidden from user, match patterns or regex formatters can be used to explicitly specify regular expressions that should be used for parsing.

By convention, regex patterns written in upper case, but it is not a hard requirement and custom patterns can use any names.

Table 4: indicators

Name	Description
re	allows to specify regular expression to use for match variable
WORD	matches single word
PHRASE	matches a collection of words separated by single space character
ORPHRASE	matches phrase or single word
*_line_*	matches any line
ROW	matches text-table data with space as column delimiter
DIGIT	matches single number
IP	matches IPv4 address
PREFIX	matches IPv4 prefix
IPV6	matches IPv6 address
PREFIXV6	matches IPv6 prefix
MAC	matches MAC address

re

```
{{ name | re("regex_value") }}
```

- regex_value - regular expression value, this value either substituted with re pattern or used as is.

Regular expression value searched using below sequence.

1. Template variables checked to to find variable names equal to regex_value
2. Built-in regex patterns searched using regex_value
3. regex_value used as is

Example

Template:

```
<vars>
# template variable with custom regular expression:
GE_INTF = "GigabitEthernet\S+"
</vars>

<input load="text">
Protocol  Address      Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.1         98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.13.3        131   0150.7685.14d5  ARPA   GigabitEthernet2.13
Internet  10.12.13.4        198   0950.5C8A.5c41  ARPA   GigabitEthernet2.17
</input>

<group>
Internet  {{ ip | re("IP")}}  {{ age | re("\d+") }}   {{ mac }}  ARPA   {{ interface␣
↪| re("GE_INTF") }}
</group>
```

Results:

```
[
    [
        {
            "age": "131",
            "interface": "GigabitEthernet2.13",
            "ip": "10.12.13.3",
            "mac": "0150.7685.14d5"
        },
        {
            "age": "198",
            "interface": "GigabitEthernet2.17",
            "ip": "10.12.13.4",
            "mac": "0950.5C8A.5c41"
        }
    ]
]
```

In this example group line:

```
Internet {{ ip | re("IP")}} {{ age | re("\d+") }} {{ mac }} ARPA {{
interface | re("GE_INTF") }}
```

transformed into this regular expression:

```
'\nInternet\ +(?P<ip>(?:(?:[0-9]{1,3}\.){3}[0-9]{1,3}))\ +(?P<age>(?:\d+))\
+(?P<mac>(?:\S+))\ +ARPA\ +(?P<interface>(?:GigabitEthernet\S+))[\t ]*(?=\n)'
```

using built-in IP pattern for *ip*, `\d+` inline regex for *age* and custom `GE_INTF` pattern for *interface* match variable.

Warning: inline definition of regular expressions delimited by `|` pipe character is not supported due to TTP uses pipe to separate match variable arguments. In other words, this `{{ name | re("re1|re2|re3") }}` is not supported. Workaround - reference template variable with required regular expression.

Example

Using template variable with multiple regular expression delimited by `|` pipe character

Template:

```
<input load="text">
Protocol  Address     Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.1        98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.13.2        98   0950.5785.5cd2  ARPA   Loopback0
Internet  10.12.13.3       131   0150.7685.14d5  ARPA   GigabitEthernet2.13
Internet  10.12.13.4       198   0950.5C8A.5c41  ARPA   GigabitEthernet2.17
</input>

<vars>
INTF_RE = r"GigabitEthernet\\S+|Fast\\S+"
</vars>

<group name="arp_test">
Internet  {{ ip | re("IP")}}  {{ age | re(r"\\d+") }}   {{ mac }}  ARPA   {{␣
↪interface | re("INTF_RE") }}
</group>
```

Result:

```
[[{'arp_test': [{'age': '98',
                 'interface': 'FastEthernet2.13',
                 'ip': '10.12.13.1',
                 'mac': '0950.5785.5cd1'},
                {'age': '131',
                 'interface': 'GigabitEthernet2.13',
                 'ip': '10.12.13.3',
                 'mac': '0150.7685.14d5'},
                {'age': '198',
                 'interface': 'GigabitEthernet2.17',
                 'ip': '10.12.13.4',
                 'mac': '0950.5C8A.5c41'}]}]]
```

`INTF_RE` - variable contains several regular expression separate by `|` character

Another technique to associate match variable with multiple regular expressions, is to reference `re("regex_value")` several times. Sample template:

```
<input load="text">
Protocol  Address     Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.1        98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.13.2        98   0950.5785.5cd2  ARPA   Loopback0
Internet  10.12.13.3       131   0150.7685.14d5  ARPA   GigabitEthernet2.13
```

(continues on next page)

(continued from previous page)

```
Internet  10.12.13.4       198   0950.5C8A.5c41  ARPA   GigabitEthernet2.17
</input>

<group name="arp_test">
Internet  {{ ip }}  {{ age }}   {{ mac }}  ARPA   {{ interface | re(r
↪"GigabitEthernet\\S+") | re(r"Fast\\S+") }}
</group>
```

Results:

```
[[{'arp_test': [{'age': '98',
                 'interface': 'FastEthernet2.13',
                 'ip': '10.12.13.1',
                 'mac': '0950.5785.5cd1'},
                {'age': '131',
                 'interface': 'GigabitEthernet2.13',
                 'ip': '10.12.13.3',
                 'mac': '0150.7685.14d5'},
                {'age': '198',
                 'interface': 'GigabitEthernet2.17',
                 'ip': '10.12.13.4',
                 'mac': '0950.5C8A.5c41'}]}]]
```

WORD

```
{{ name | WORD }}
```

WORD pattern helps to match single word - collection of characters excluding any space, tab or new line characters.

PHRASE

```
{{ name | PHRASE }}
```

This pattern matches any phrase - collection of words separated by **single** space character, such as "word1 word2 word3".

ORPHRASE

```
{{ name | ORPHRASE }}
```

In many cases data that needs to be extracted can be either a single word or a phrase, the most prominent example - various descriptions, such as interface descriptions, BGP peers descriptions etc. ORPHRASE allows to match and extract such a data.

Example

Template:

```
<input load="text">
interface Loopback0
 description Router id - OSPF, BGP
 ip address 192.168.0.113/24
!
interface Vlan778
```

(continues on next page)

(continued from previous page)

```
 description CPE_Acces_Vlan
 ip address 2002::fd37/124
!
</input>

<group>
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ ip }}/{{ mask }}
</group>
```

Result:

```
[
    [
        {
            "description": "Router id - OSPF, BGP",
            "interface": "Loopback0",
            "ip": "192.168.0.113",
            "mask": "24"
        },
        {
            "description": "CPE_Acces_Vlan",
            "interface": "Vlan778",
            "ip": "2002::fd37",
            "mask": "124"
        }
    ]
]
```

line

```
{{ name | _line_ }}
```

Matches any line within text data, check *_line_* indicators section for more details.

ROW

```
{{ name | ROW }}
```

Helps to match row-like lines of text - words separated by a number of spaces.

Example

Template:

```
<input load="text">
Pesaro# show ip vrf detail Customer_A
VRF Customer_A; default RD 100:101
  Interfaces:
    Loopback101      Loopback111      Vlan707
</input>

<group name="vrfs">
VRF {{ vrf }}; default RD {{ rd }}
```

(continues on next page)

(continued from previous page)

```
<group name="interfaces">
  Interfaces: {{ _start_ }}
    {{ intf_list | ROW }}
</group>
</group>
```

Results:

```
[
    {
        "vrfs": {
            "interfaces": {
                "intf_list": "Loopback101      Loopback111      Vlan707"
            },
            "rd": "100:101",
            "vrf": "Customer_A"
        }
    }
]
```

Line ” Loopback101 Loopback111 Vlan707” was matched by `ROW` regular expression.

DIGIT

```
{{ name | DIGIT }}
```

Matches any single number, such as 1 or 123 or 0012300.

IP

```
{{ name | IP }}
```

This regex pattern can match IPv4 addresses, for instance *192.168.134.251*. But this pattern does not perform IP address validation, as a result this text also will be matched *321.751.123.999*. Condition check function *is_ip* can be used to validate IP addresses.

PREFIX

```
{{ name | PREFIX }}
```

Matches IPv4 prefix, such as *192.168.0.1/24*, but also will match *999.321.192.6/99*, make sure to use *is_ip* function to validate prefixes if required.

IPV6

```
{{ name | IPV6 }}
```

Performs match on IPv6 addresses, for example *2001:ABC0::FE31* address, but will also match incorrect IPv6 *2002::fd37::91* address as well, make sure to use *is_ip* function to validate IPv6 addresses.

PREFIXV6

```
{{ name | PREFIXV6 }}
```

Matches IPv6 prefix, such as *2001:ABC0::FE31/64*, but will also match *2002::fd37::91/124*, make sure to use *is_ip* function to validate prefixes if required.

MAC

```
{{ name | MAC }}
```

MAC addresses will be matched by this regular expression pattern, such as:

- aa:bb:cc:dd:11:33
- aa.bb.cc.dd.11.33
- aabb:ccdd:1133
- aabb.ccdd.1133

CHAPTER 5

Groups

Groups are the core component of ttp together with match variables. Group is a collection of regular expressions derived from template, groups denoted using XML group tag (<g>, <grp>, <group>) and can be nested to form hierarchy. Parsing results for each group combined into a single datum - dictionary, that dictionary merged with bigger set of results data.

As ttp was developed primarily for parsing semi-structured configuration data of various network elements, groups concept stems from the fact that majority of configuration data can be divided in distinctive pieces of information, each of which can denote particular property or feature configured on device, moreover, it is not uncommon that these pieces of information can be broken down into even smaller pieces of repetitive data. TTP helps to combine regular expressions in groups for the sake of parsing small, repetitive pieces of text data.

For example, this is how industry standard CLI configuration data for interfaces might look like:

```
interface Vlan163
 description [OOB management]
 ip address 10.0.10.3 255.255.255.0
!
interface GigabitEthernet6/41
 description [uplink to core]
 ip address 192.168.10.3 255.255.255.0
```

It is easy to notice that there is a lot of data which is the same and there is a lot of information which is different as well, if we would say that overall device's interfaces configuration is a collection of repetitive data, with interfaces being a smallest available datum, we can outline it in ttp template below and use it parse valuable information from text data:

```
<group name="interfaces">
interface {{ interface }}
 description {{ description | PHRASE }}
 ip address {{ ip }} {{ mask }}
</group>
```

After parsing this configuration data with that template results will be:

```
[
    {
        "interfaces": [
            {
                "description": "[OOB management]",
                "interface": "Vlan163",
                "ip": "10.0.10.3",
                "mask": "255.255.255.0"
            },
            {
                "description": "[uplink to core]",
                "interface": "GigabitEthernet6/41",
                "ip": "192.168.10.3",
                "mask": "255.255.255.0"
            }
        ]
    }
]
```

As a result each interfaces group produced separate dictionary and all interfaces dictionaries were combined in a list under *interfaces* key which is derived from group name.

5.1 Group reference

5.1.1 Attributes

Each group tag (<g>, <grp>, <group>) can have a number of attributes, they used during module execution to provide desired results. Attributes can be mandatory or optional. Each attribute is a string of data formatted in certain way.

Table 1: group attributes

Attribute	Description
name	Uniquely identifies group(s) within template and specifies results path location
input	Name of input tag or OS path string to files location
default	Contains default value that should be set for all variables if nothing been matched
method	Indicates parsing method, supported values are *group* or *table*
output	Specify group specific outputs to run group result through

name

```
name="path_string"
```

- path_string (mandatory) - this is the only attribute that *must* be set for each group as it used to form group path - path is a dot separated string that indicates group results placement in results structure.

More on name attribute: Group Name Attribute

input

```
input="input1, input2, ... inputN"
```

- inputN (optional) - comma separated string that contains name(s) of the input tag(s) that should be used to source data for this group, alternatively input string value can reference Operating System fully qualified or relative

path to location of text file(s) that should be parsed by this group. OS relative path should be accompanied with template base_path attribute, that attribute will be perpended to group input to form fully qualified path.

Input attribute of the group considered to be more specific in case if group name referenced in input *groups* attribute, as a result several groups can share same name, but reference different inputs with different set of data to be parsed.

Note: Input attributed only supported at top group, nested groups input attributes are ignored.

Example-1

Template:

```
<input name="test1" load="text">
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166-173
</input>

<group name="interfaces" input="test1">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "interface": "GigabitEthernet3/3",
            "trunk_vlans": "138,166-173"
        }
    }
]
```

Example-2

In this example several inputs define, by default groups set to 'all' for them, moreover, groups have identical name attribute. In this case group's *input* attribute helps to define which input should be parsed by which group.

Template:

```
<input name="input_1" load="text">
interface GigabitEthernet3/11
 description input_1_data
 switchport trunk allowed vlan add 111,222
!
</input>

<input name="input_2" load="text">
interface GigabitEthernet3/22
 description input_2_data
 switchport trunk allowed vlan add 222,888
!
</input>

<group name="interfaces.trunks" input="input_1">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans }}
 description {{ description | ORPHRASE }}
```

(continues on next page)

(continued from previous page)

```
 {{ group_id | set("group_1") }}
!{{ _end_ }}
</group>

<group name="interfaces.trunks" input="input_2">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans }}
 description {{ description | ORPHRASE }}
 {{ group_id | set("group_2") }}
!{{ _end_ }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "trunks": {
                "description": "input_1_data",
                "group_id": "group_1",
                "interface": "GigabitEthernet3/11",
                "trunk_vlans": "111,222"
            }
        }
    },
    {
        "interfaces": {
            "trunks": {
                "description": "input_2_data",
                "group_id": "group_2",
                "interface": "GigabitEthernet3/22",
                "trunk_vlans": "222,888"
            }
        }
    }
]
```

default

`default="value"`

- value (optional) - string that should be used as a default value for all variables within this group.

Example-1

Template:

```
<input name="test1" load="text">
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166-173
</input>

<group name="interfaces" input="test1" default="some_default_value">
interface {{ interface }}
 description {{ description }}
 switchport trunk allowed vlan add {{ trunk_vlans }}
```

(continues on next page)

(continued from previous page)

```
 ip address {{ ip }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "description": "some_default_value",
            "interface": "GigabitEthernet3/3",
            "ip": "some_default_value",
            "trunk_vlans": "138,166-173"
        }
    }
]
```

Because `default` value used for group start regexes, if no matches produced by group, default values will be saved at group path, same is true for child groups

Example-2

Group with no matches but default values.

Template:

```
<input load="text">
device-hostame uptime is 27 weeks, 3 days, 10 hours, 46 minutes, 10 seconds
</input>

<group name="uptime**">
device-hostame uptime is {{ uptime | PHRASE }}
    <group name="software">
     software version {{ version | default("uncknown") }}
    </group>
</group>

<group name="domain" default="Uncknown">
Default domain is {{ fqdn }}
</group>
```

Result:

```
[
    [
        {
            "domain": {
                "fqdn": "Uncknown"
            },
            "uptime": {
                "uptime": "27 weeks, 3 days, 10 hours, 46 minutes, 10 seconds",
                "software": {
                    "version": "uncknown"
                }
            }
        }
    ]
]
```

In above example in input there is not data to match by group `domain`, this group default values were saved in results. Same is for child group `software` - no data to match in input, hence default values appears in results, because match variable `software` is start RE.

method

`method="value"`

- value (optional) - [group | table] default is *group*. If method it *group* only first regular expression in group considered as group-start-re, in addition template lines that contain *_start_* indicator also used as group-start-re.

On the other hand, if method set to *table* each and every regular expression in the group considered as group-start-re, that is very useful if semi-table data structure parsed, and we have several variations of row.

Example

In this example arp table needs to be parsed, but to match all the variations we have to define several template expressions.

Data:

```
CSR1Kv-3-lab#show ip arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.1.13.1              98   0050.5685.5cd1  ARPA   GigabitEthernet2.13
Internet  10.1.13.3               -   0050.5685.14d5  ARPA   GigabitEthernet2.13
```

Template:

This is the template with default method *group*:

```
<group name="arp">
Internet  {{ ip }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface }}
Internet  {{ ip }}  -                   {{ mac }}  ARPA   {{ interface| _start_}}
</group>
```

This is functionally the same template but with method *table*:

```
<group name="arp" method="table">
Internet  {{ ip }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface }}
Internet  {{ ip }}  -                   {{ mac }}  ARPA   {{ interface }}
</group>
```

Result:

```
[
    {
        "arp": [
            {
                "age": "98",
                "interface": "GigabitEthernet2.13",
                "ip": "10.1.13.1",
                "mac": "0050.5685.5cd1"
            },
            {
                "interface": "GigabitEthernet2.13",
                "ip": "10.1.13.3",
                "mac": "0050.5685.14d5"
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    }
]
```

output

```
output="output1, output2, ... , outputN"
```

- outputN - comma separated string of output tag names that should be used to run group results through. The sequence of outputs provided *are preserved* and run run in specified order, meaning that output2 will run only after output1.

Note: only top group supports output attribute, nested groups' output attributes are ignored.

5.1.2 Functions

Group functions can be applied to group results to transform them in a desired way, functions can also be used to validate and filter match results.

Condition functions help to evaluate group results and return *False* or *True*, if *False* returned, group results will be discarded.

Table 2: group functions

Name	Description
containsall	checks if group result contains matches for all given variables
contains	checks if group result contains match at least for one of given variables
macro	Name of the macros function to run against group result
functions or chain	String containing list of functions to run this group results through
to_ip	transforms given values in ipaddress IPAddress object
exclude	invalidates group results if **any** of given keys present in group
excludeall	invalidates group results if **all** given keys present in group
del	delete given keys from group results
sformat	format provided string with match result and/or template variables
itemize	produce list of items extracted out of group match results dictionary
cerberus	filter results using Cerberus validation engine
void	invalidates group results, allowing to skip them
str_to_unicode	converts Python2 str srings in unicode strings
equal	verifies that key's value is equal to provided value
to_int	converts given keys to integer (int or float) or tries to convert all match result values
contains_val	check if certain key contains certain value, return True if so and False otherwise
exclude_val	check if certain key contains certain value, return False if so and True otherwise
record	save (record) variable value in results object and global scope dictionaries
set	get value from results object variables dictionary and assign it to variable
expand	expand match variable dot separated name to dictionary
validate	add Cerberus validation information to results without filtering them

containsall

```
containsall="variable1, variable2, variableN"
```

- **variable (mandatory) - a comma-separated string that contains match variable names. This function** checks if group results contain specified variable, if at least one variable not found in results, whole group result discarded

Example

For instance we want to get results only for interfaces that has IP address configured on them **and** vrf, all the rest of interfaces should not make it to results.

Data:

```
interface Port-Chanel11
  description Storage Management
!
interface Loopback0
  description RID
  ip address 10.0.0.3/24
!
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
```

Template:

```
<group name="interfaces" containsall="ip, vrf">
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

Result:

```
{
    "interfaces": {
        "description": "Management",
        "interface": "Vlan777",
        "ip": "192.168.0.1",
        "mask": "24",
        "vrf": "MGMT"
    }
}
```

contains

```
contains="variable1, variable2, variableN"
```

- **variable (mandatory) - a comma-separated string that contains match variable names. This function** checks if group results contains *any* of specified variable, if no variables found in results, whole group result discarded, if at least one variable found in results, this check is satisfied.

Example

For instance we want to get results only for interfaces that has IP address configured on them **or** vrf.

Data:

```
interface Port-Chanel11
  description Storage Management
!
interface Loopback0
  description RID
  ip address 10.0.0.3/24
!
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
```

Template:

```
<group name="interfaces" contains="ip, vrf">
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

Result:

```
{
    "interfaces": [
        {
            "description": "RID",
            "interface": "Loopback0",
            "ip": "10.0.0.3",
            "mask": "24"
        },
        {
            "description": "Management",
            "interface": "Vlan777",
            "ip": "192.168.0.1",
            "mask": "24",
            "vrf": "MGMT"
        }
    ]
}
```

macro

`macro="name1, name2, ... , nameN"`

- nameN - comma separated string of macro functions names that should be used to run group results through. The sequence is *preserved* and macros executed in specified order, in other words macro named name2 will run after macro name1.

Macro brings Python language capabilities to group results processing and validation during TTP module execution, as it allows to run custom python functions. Macro functions referenced by their name in group tag definitions.

Macro function must accept only one attribute to hold group match results.

Depending on data returned by macro function, TTP will behave differently according to these rules:

- If macro returns True or False - original data unchanged, macro handled as condition functions, invalidating result on False and keeps processing result on True
- If macro returns None - data processing continues, no additional logic associated
- If macro returns single item - that item replaces original data supplied to macro and processed further

Example

Template:

```
<input load="text">
interface GigabitEthernet1/1
 description to core-1
!
interface Vlan222
 description Phones vlan
!
interface Loopback0
 description Routing ID loopback
!
</input>

<macro>
def check_if_svi(data):
    if "Vlan" in data["interface"]:
        data["is_svi"] = True
    else:
        data["is_svi"] = False
    return data

def check_if_loop(data):
    if "Loopback" in data["interface"]:
        data["is_loop"] = True
    else:
        data["is_loop"] = False
    return data
</macro>

<macro>
def description_mod(data):
    # function to revert words order in descripotion
    words_list = data.get("description", "").split(" ")
    words_list_reversed = list(reversed(words_list))
    words_reversed = " ".join(words_list_reversed)
    data["description"] = words_reversed
    return data
</macro>

<group name="interfaces_macro" macro="description_mod, check_if_svi, check_if_loop">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ ip }} {{ mask }}
</group>
```

Result:

```
[
    {
```

(continues on next page)

(continued from previous page)

```
        "interfaces_macro": [
            {
                "description": "core-1 to",
                "interface": "GigabitEthernet1/1",
                "is_loop": false,
                "is_svi": false
            },
            {
                "description": "vlan Phones",
                "interface": "Vlan222",
                "is_loop": false,
                "is_svi": true
            },
            {
                "description": "loopback ID Routing",
                "interface": "Loopback0",
                "is_loop": true,
                "is_svi": false
            }
        ]
    }
]
```

functions or chain

```
functions="function1('attributes') | function2('attributes') | ... |
functionN('attributes')"
```

```
chain="function1('attributes') | function2('attributes') | ... |
functionN('attributes')"
```

```
chain="template_variable_name"
```

- functionN - name of the group function together with it's attributes
- template_variable_name - template variable that contains pipe-separated string of functions or a list

`chain` and `functions` attributes are doing exactly the same, just two different names to reference same functionality, hence can be used interchangeably.

The advantages of using string or list of functions versus defining them directly in the group tag are:

- it allows to define sequence of functions to run group results through and that order will be honored
- chain of functions can also reference template variable that contains string or list of functions strings, that allows to reuse same chain across several groups
- improved readability as multiple functions definitions can go to template variable

For instance we have two below group definitions:

Group1:

```
<group name="interfaces_macro" functions="contains('ip') | macro('description_mod') |␣
↪macro('check_if_svi') | macro('check_if_loop')">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ ip }} {{ mask }}
</group>
```

Group2:

```
<group name="interfaces_macro" contains="ip" macro="description_mod, check_if_svi,␣
↪check_if_loop">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ ip }} {{ mask }}
</group>
```

While above groups have same set of functions defined, for Group1 function will run in provided order, while for Group2 order is undefined due to the fact that XML tag attributes loaded in python dictionary, meaning that key-value mappings are unordered.

Warning: pipe 'I' symbol must be used to separate function names, not comma

Example-1

Using functions within group tag.

Template:

```
<input load="text">
interface GigabitEthernet1/1
 description to core-1
 ip address 192.168.123.1 255.255.255.0
!
interface Vlan222
 description Phones vlan
!
interface Loopback0
 description Routing ID loopback
 ip address 192.168.222.1 255.255.255.0
!
</input>

<macro>
def check_if_svi(data):
    if "Vlan" in data["interface"]:
        data["is_svi"] = True
    else:
        data["is_svi"] = False
    return data

def check_if_loop(data):
    if "Loopback" in data["interface"]:
        data["is_loop"] = True
    else:
        data["is_loop"] = False
    return data
</macro>

<macro>
def description_mod(data):
    # To revert words order in descripotion
    words_list = data.get("description", "").split(" ")
    words_list_reversed = list(reversed(words_list))
    words_reversed = " ".join(words_list_reversed)
```

(continues on next page)

(continued from previous page)

```
    data["description"] = words_reversed
    return data
</macro>

<group name="interfaces_macro" functions="contains('ip') | macro('description_mod') |␣
↪macro('check_if_svi') | macro('check_if_loop')">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 ip address {{ ip }} {{ mask }}
</group>
```

Result:

```
[
    {
        "interfaces_macro": [
            {
                "description": "core-1 to",
                "interface": "GigabitEthernet1/1",
                "ip": "192.168.123.1",
                "is_loop": false,
                "is_svi": false,
                "mask": "255.255.255.0"
            },
            {
                "description": "loopback ID Routing",
                "interface": "Loopback0",
                "ip": "192.168.222.1",
                "is_loop": true,
                "is_svi": false,
                "mask": "255.255.255.0"
            }
        ]
    }
]
```

Example-2

Using template variables to chain functions.

Template:

```
<input load="text">
interface Port-Chanel11
  vlan 10
interface Loopback0
  vlan 20
  description test loopback0
interface Loopback1
  vlan 30
  description test loopback1
</input>

<vars>
chain1 = [
    "del(vlan) | set('set_value', 'set_key')",
    "contains_val(interface, 'Loop')",
```

(continues on next page)

(continued from previous page)

```
    "macro('test_macro')",
    "macro('test_macro1, test_macro2')",
    "macro(test_macro3, test_macro4)",
]
</vars>

<macro>
def test_macro(data):
    data["test_macro"] = "DONE"
    return data

def test_macro1(data):
    data["test_macro1"] = "DONE"
    return data

def test_macro2(data):
    data["test_macro2"] = "DONE"
    return data

def test_macro3(data):
    data["test_macro3"] = "DONE"
    return data

def test_macro4(data):
    data["test_macro4"] = "DONE"
    return data
</macro>

<group chain="chain1">
interface {{ interface }}
  vlan {{ vlan | to_int }}
  description {{ description | ORPHRASE }}
</group>
```

Result:

```
[[[{'description': 'test loopback0',
    'interface': 'Loopback0',
    'set_key': 'set_value',
    'test_macro': 'DONE',
    'test_macro1': 'DONE',
    'test_macro2': 'DONE',
    'test_macro3': 'DONE',
    'test_macro4': 'DONE'},
   {'description': 'test loopback1',
    'interface': 'Loopback1',
    'set_key': 'set_value',
    'test_macro': 'DONE',
    'test_macro1': 'DONE',
    'test_macro2': 'DONE',
    'test_macro3': 'DONE',
    'test_macro4': 'DONE'}]]]
```

to_ip

`functions="to_ip(ip_key='X', mask_key='Y')"` or `to_ip="'X', 'Y'"` or `to_ip="ip_key='X', mask_key='Y'"`

- ip_key - name of the key that contains IP address string
- mask_key - name of the key that contains mask string

This functions can help to construct ipaddress IpAddress object out of ip_key and mask_key values, on success this function will return ipaddress object assigned to ip_key.

Example

Template:

```
<input load="text">
interface Loopback10
 ip address 192.168.0.10  subnet mask 24
!
interface Vlan710
 ip address 2002::fd10 subnet mask 124
!
</input>

<group name="interfaces_with_funcs" functions="to_ip('ip', 'mask')">
interface {{ interface }}
 ip address {{ ip }}  subnet mask {{ mask }}
</group>

<group name="interfaces_with_to_ip_args" to_ip = "'ip', 'mask'">
interface {{ interface }}
 ip address {{ ip }}  subnet mask {{ mask }}
</group>

<group name="interfaces_with_to_ip_kwargs" to_ip = "ip_key='ip', mask_key='mask'">
interface {{ interface }}
 ip address {{ ip }}  subnet mask {{ mask }}
</group>
```

Results:

```
[   {   'interfaces_with_funcs': [   {   'interface': 'Loopback10',
                                         'ip': IPv4Interface('192.168.0.10/24'),
                                         'mask': '24'},
                                     {   'interface': 'Vlan710',
                                         'ip': IPv6Interface('2002::fd10/124'),
                                         'mask': '124'}],
        'interfaces_with_to_ip_args': [   {   'interface': 'Loopback10',
                                              'ip': IPv4Interface('192.168.0.10/24'),
                                              'mask': '24'},
                                          {   'interface': 'Vlan710',
                                              'ip': IPv6Interface('2002::fd10/124'),
                                              'mask': '124'}],
        'interfaces_with_to_ip_kwargs': [   {   'interface': 'Loopback10',
                                                'ip': IPv4Interface('192.168.0.10/24
↪'),
                                                'mask': '24'},
                                            {   'interface': 'Vlan710',
```

(continues on next page)

(continued from previous page)

```
                                                    'ip': IPv6Interface('2002::fd10/124'),
                                                    'mask': '124'}]}]
```

exclude

`exclude="variable1, variable2, ..., variableN"`

- variableN - name of the variable on presence of which to invalidate/exclude group results

This function allows to invalidate group match results based on the fact that **any** of the given variable names/keys are present.

Example

Here groups with either `ip` or `description` variables matches, will be excluded from results.

Template:

```
<input load="text">
interface Vlan778
 description some description 1
 ip address 2002:fd37::91/124
!
interface Vlan779
 description some description 2
!
interface Vlan780
 switchport port-security mac 4
!
</input>

<group name="interfaces" exclude="ip, description">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description | ORPHRASE }}
 switchport port-security mac {{ sec_mac }}
</group>
```

Results:

```
[
    {
        "interfaces": {
            "interface": "Vlan780",
            "sec_mac": "4"
        }
    }
]
```

excludeall

`excludeall="variable1, variable2, ..., variableN"`

- variable - name of the variable on presence of which to invalidate/exclude group results

excludeall allows to invalidate group results based on the fact that **all** of the given variable names/keys are present in match results.

del

`del="variable1, variable2, ..., variableN"`

- variableN - name of the variable to delete results for

Example

Template:

```
<input load="text">
interface Vlan778
 description some description 1
 ip address 2002:fd37::91/124
!
interface Vlan779
 description some description 2
!
interface Vlan780
 switchport port-security mac 4
!
</input>

<group name="interfaces-test1-31" del="description, ip">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description | ORPHRASE }}
 switchport port-security mac {{ sec_mac }}
</group>
```

Results:

```
[
    {
        "interfaces-test1-31": [
            {
                "interface": "Vlan778",
                "mask": "124"
            },
            {
                "interface": "Vlan779"
            },
            {
                "interface": "Vlan780",
                "sec_mac": "4"
            }
        ]
    }
]
```

sformat

`sformat="string='text', add_field='name'"` or `sformat="'text', 'name'"`

- string - mandatory, string to format
- add_field - mandatory, name of new field with value produced by sformat to add to group results

sformat (string format) method used to form string in certain way using template variables and group match results. The order of variables to use for formatting is:

> 1 global variables produced by *record* function 2 template variables as specified in <vars> tag 3 group match results

Next variables in above list override the previous one.

Example

Template:

```
<vars>
domain = "com"
</vars>

<input load="text">
switch-1 uptime is 27 weeks, 3 days, 10 hours, 46 minutes, 10 seconds
</input>

<input load="text">
Default domain is lab.local
</input>

<group name="uptime">
{{ hostname | record("hostname")}} uptime is {{ uptime | PHRASE }}
</group>

<group name="fqdn_dets_1" sformat="string='{hostname}.{fqdn},{domain}', add_field=
↪'fqdn'">
Default domain is {{ fqdn }}
</group>
```

Results:

```
[
    {
        "uptime": {
            "hostname": "switch-1",
            "uptime": "27 weeks, 3 days, 10 hours, 46 minutes, 10 seconds"
        }
    },
    {
        "fqdn_dets_1": {
            "fqdn": "switch-1.lab.local,com"
        }
    }
]
```

string `{hostname}.{fqdn},{domain}` formatted using `hostname` variable from globally recorded vars, `fqdn` variable from group match results and `domain` variable defined in template vars. In this example `add_field` was set to `fqdn` to override fqdn match variable matched values

itemize

`itemize="key='name', path='path.to.result'"` or `functions="itemize(key='name', path='path.to.result')"`

- key - mandatory, name of the key to use create a list of items from

- path - optional, by default path taken from group name attribute, dot separated string of there to save a list of items within results tree

This function allows to take single item result from group match results and place it into the list at path provided. Motivation behind this function is to be able to provide create a list of items out of match results produced by group. For instance produce a list of all IPs configured on device or VRFs or OSPF processes etc. without the need to iterate over parsing results to extract items in question.

Example

Let's say we need to extract a list of all interfaces configured on device.

Template:

```
<input load="text">
interface Vlan778
 description some description 1
 ip address 2002:fd37::91/124
!
interface Vlan779
 description some description 2
!
interface Vlan780
 switchport port-security mac 4
 ip address 192.168.1.1/124
!
</input>

<group name="interfaces_list" itemize="interface">
interface {{ interface }}
 ip address {{ ip }}
</group>
```

Results:

```
[
    {
        "interfaces_list": [
            "Vlan778",
            "Vlan779",
            "Vlan780"
        ]
    }
]
```

cerberus

`cerberus="schema='var_name', log_errors=False, allow_unknown=True, add_errors=False"`

- `schema` - string, mandatory, name of template variable that contains Cerberus schema structure
- `log_errors` - bool, default is False, if set to True will log Cerberus validation errors with WARNING level
- `allow_unknown` - bool, default is True, if set to False, Cerberus will invalidate match results with keys that are not defined in schema
- `add_errors` - bool, default is False, if set to True, Cerberus validation errors will be added to results under "validation_errors" key

Prerequisites Cerberus library need to be installed on the system.

This function uses Cerberus validation engine to validate group results, returning `True` if validation succeeded and `False` otherwise.

Cerberus Validation schema must be defined in one of template variables.

Example

Let's say we want to extract information only for interfaces that satisfy these set of criteria:

- has "Gigabit" in the name
- contains "Customer" in description
- dot1q vlan id is in 200-300 range
- interface belongs to one of VRFs - "Management" or "Data"

Template:

```
<input load="text">
interface GigabitEthernet1/3.251
 description Customer #32148
 encapsulation dot1q 251
 vrf forwarding Management
 ipv6 address 2002:fd37::91/124
!
interface GigabitEthernet1/3.321
 description Customer #151678
 encapsulation dot1q 321
 vrf forwarding Voice
 ip address 172.16.32.10 255.255.255.128
!
interface Vlan779
 description South Bank Customer #78295
 vrf forwarding Data
 ip address 192.168.23.53 255.255.255.0
!
interface TenGigabitEthernet3/1.298
 description PDSENS Customer #783290
 encapsulation dot1q 298
 vrf forwarding Data
 ipv6 address 2001:ad56::1273/64
!
</input>

<vars>
my_schema = {
    "interface": {
        "regex": ".*Gigabit.*"
    },
    "vrf": {
        "allowed": ["Data", "Management"]
    },
    "description": {
        "regex": ".*Customer.*"
    },
    "vid": {
        "min": 200,
        "max": 300
```

(continues on next page)

(continued from previous page)

```
    }
}
</vars>

<group name="filtered_interfaces*" cerberus="my_schema">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 encapsulation dot1q {{ vid | to_int }}
 vrf forwarding {{ vrf }}
 ip address {{ ip }} {{ mask }}
 ipv6 address {{ ipv6 }}/{{ maskv6 }}
</group>
```

Result:

```
[
    [
        {
            "filtered_interfaces": [
                {
                    "description": "Customer #32148",
                    "interface": "GigabitEthernet1/3.251",
                    "ipv6": "2002:fd37::91",
                    "maskv6": "124",
                    "vid": 251,
                    "vrf": "Management"
                },
                {
                    "description": "PDSENS Customer #783290",
                    "interface": "TenGigabitEthernet3/1.298",
                    "ipv6": "2001:ad56::1273",
                    "maskv6": "64",
                    "vid": 298,
                    "vrf": "Data"
                }
            ]
        }
    ]
]
```

By default only results that passed validation criteria will be returned by TTP, however, if `add_errors` set to True:

```
<group name="filtered_interfaces*" cerberus="schema='my_schema', add_errors=True">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 encapsulation dot1q {{ vid | to_int }}
 vrf forwarding {{ vrf }}
 ip address {[ ip }} {{ mask }}
 ipv6 address {{ ipv6 }}/{{ maskv6 }}
</group>
```

None of the results will be filtered, but validation errors information will be included:

```
[
    [
        {
            "filtered_interfaces": [
```

(continues on next page)

(continued from previous page)

```
                {
                    "description": "Customer #32148",
                    "interface": "GigabitEthernet1/3.251",
                    "ipv6": "2002:fd37::91",
                    "maskv6": "124",
                    "vid": 251,
                    "vrf": "Management"
                },
                {
                    "description": "Customer #151678",
                    "interface": "GigabitEthernet1/3.321",
                    "validation_errors": {
                        "vid": [
                            "max value is 300"
                        ],
                        "vrf": [
                            "unallowed value Voice"
                        ]
                    },
                    "vid": 321,
                    "vrf": "Voice"
                },
                {
                    "description": "South Bank Customer #78295",
                    "interface": "Vlan779",
                    "validation_errors": {
                        "interface": [
                            "value does not match regex '.*Gigabit.*'"
                        ]
                    },
                    "vrf": "Data"
                },
                {
                    "description": "PDSENS Customer #783290",
                    "interface": "TenGigabitEthernet3/1.298",
                    "ipv6": "2001:ad56::1273",
                    "maskv6": "64",
                    "vid": 298,
                    "vrf": "Data"
                }
            ]
        }
    ]
]
```

void

`void=""` or `functions="void"`

The purpose of this function is to return False on group results validation, effectively allowing to skip results for this group.

str_to_unicode

`str_to_unicode=""` or `functions="str_to_unicode"`

If python2 used to run TTP, this function iterates over group results and converts strings of type `str` into `unicode` type strings. For python3 this function does nothing.

equal

`equal="key, value"`

- key - name of the key to verify value for
- value - value to verify equality against

This functions check if value of certain key is equal to value provided and returns True is so and False otherwise.

Example

Template:

```
<input load="text">
interface FastEthernet1/0/1
 description Foo
!
interface FastEthernet1/0/2
 description wlap2
!
</input>

<group name="interfaces" equal="description, Foo">
interface {{ interface }}
 description {{ description }}
</group>
```

Results:

```
[
    [
        {
            "interfaces": {
                "description": "Foo",
                "interface": "FastEthernet1/0/1"
            }
        }
    ]
]
```

to_int

`to_int=""` or `to_int="key1, key2, keyN"`

- keyN - name of keys to run conversion for, if omitted, all group match results items will be attempted to convert into integer.

This function tries to convert string representation of digit into integer using python int() function, if fails it next tries to convert to integer using python float() function. If either int() or float() conversion was successful, string converted to digit will replace match result, on failure nothing will be done with match results.

Example

Template:

```
<input load="text">
Subscription ID = 1
Version = 1
Num Subpackets = 1
Subpacket[0]
   Subpacket ID = PDCP PDU with Ciphering (0xC3)
   Subpacket Version = 26.1
   Subpacket Size = 60,5 bytes
   SRB Cipher Algo = LTE AES
   DRB Cipher Algo = LTE AES
   Num PDUs = 1
</input>

<group name="all_to_int" to_int="">
Subscription ID = {{ Subscription_ID }}
Version = {{ version }}
Num Subpackets = {{ Num_Subpackets }}
   Subpacket ID = {{ Subpacket_ID | PHRASE }}
   Subpacket Version = {{ Subpacket_Version }}
   Subpacket Size = {{ Subpacket_Size | PHRASE }}
   SRB Cipher Algo = {{ SRB_Cipher_Algo | PHRASE }}
   DRB Cipher Algo = {{ DRB_Cipher_Algo | PHRASE }}
   Num PDUs = {{ Num_PDUs }}
</group>

<group name="some_to_int" to_int="version, Subpacket_Version">
Subscription ID = {{ Subscription_ID }}
Version = {{ version }}
Num Subpackets = {{ Num_Subpackets }}
   Subpacket ID = {{ Subpacket_ID | PHRASE }}
   Subpacket Version = {{ Subpacket_Version }}
   Subpacket Size = {{ Subpacket_Size | PHRASE }}
   SRB Cipher Algo = {{ SRB_Cipher_Algo | PHRASE }}
   DRB Cipher Algo = {{ DRB_Cipher_Algo | PHRASE }}
   Num PDUs = {{ Num_PDUs }}
</group>
```

Results:


```
[
    [
        {
            "all_to_int": {
                "DRB_Cipher_Algo": "LTE AES",
                "Num_PDUs": 1,
                "Num_Subpackets": 1,
                "SRB_Cipher_Algo": "LTE AES",
                "Subpacket_ID": "PDCP PDU with Ciphering (0xC3)",
                "Subpacket_Size": "60,5 bytes",
                "Subpacket_Version": 26.1,
                "Subscription_ID": 1,
                "version": 1
            },
            "some_to_int": {
                "DRB_Cipher_Algo": "LTE AES",
                "Num_PDUs": "1",
                "Num_Subpackets": "1",
```


(continues on next page)

(continued from previous page)

```
                "SRB_Cipher_Algo": "LTE AES",
                "Subpacket_ID": "PDCP PDU with Ciphering (0xC3)",
                "Subpacket_Size": "60,5 bytes",
                "Subpacket_Version": 26.1,
                "Subscription_ID": "1",
                "version": 1
            }
        }
    ]
]
```

contains_val

`contains_val="key, value"`

- `key` - name of key to check value for
- `value` - value to check against

This function checks if value for certain key in group results contains value provided, returning None if so and False otherwise. Value can be checked as is, or can be a reference to variable from `<vars>` tag. Function evaluates to None if no such key found in group results.

Example-1

Template:

```
<input load="text">
interface Vlan779
 ip address 2.2.2.2/24
!
interface Vlan780
 ip address 2.2.2.3/24
!
</input>

<group name="interfaces" contains_val="'ip', '2.2.2.2/24'">
interface {{ interface }}
 ip address {{ ip }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "interface": "Vlan779",
            "ip": "2.2.2.2/24"
        }
    }
]
```

Example-2

In this example, value to check for defined as a variable. This can be useful if veriables need to be set dynamically.

Template:

```
<input load="text">
interface Lo0
ip address 124.171.238.50 32
!
interface Lo1
ip address 1.1.1.1 32
</input>

<vars>
ip_in_question="1.1.1.1"
</vars>

<group contains_val="ip, ip_in_question">
interface {{ interface }}
ip address {{ ip }} {{ mask }}
</group>
```

Results:

```
[
    [
        {
            "interface": "Lo1",
            "ip": "1.1.1.1",
            "mask": "32"
        }
    ]
]
```

ip_in_question - name of thevariable from <vars> tag.

exclude_val

`exclude_val="key, value"`

- `key` - name of key to check value for
- `value` - value to check against

This function checks if certain key in group results equal to value provided, returning False if so and True otherwise. Value can be compared as is, or can be a reference to variable from `<vars>` tag.

Example-2

In this example, value to check for defined as a variable. This can be useful if veriables need to be set dynamically.

Template:

```
<input load="text">
interface Lo0
ip address 124.171.238.50 32
!
interface Lo1
ip address 1.1.1.1 32
</input>

<vars>
ip_in_question="1.1.1.1"
```

(continues on next page)

(continued from previous page)

```
</vars>

<group exclude_val="ip, ip_in_question">
interface {{ interface }}
ip address {{ ip }} {{ mask }}
</group>
```

Results:

```
[
    [
        {
            "interface": "Lo0",
            "ip": "124.171.238.50",
            "mask": "32"
        }
    ]
]
```

record

`record="source, target"`

- `source` - name of variable to source value from
- `target` - optional, name of variable to assign value to

Depending on requirements match variable `record` might not be enough due to the fact that it can only record values during parsing phase, group `record` function on the other hand can record variable values during results processing phase. Group *set* function can make use of this recorded variables adding them to produced results.

Group `record` function saved variable value in two dictionaries that represent different scopes of access:

1. Per-input scope - this dictionary available during processing of all groups for this particular input; `_ttp_["results_object"].vars` dictionary
2. Global scope - this dictionary available across all templates, inputs and groups; `_ttp_["global_vars"]` dictionary

Example-0

In this example match variable `record` function used to save match values, however, due to the way how data structured, only last match value got recorded, overriding previous matches, i.e. "VRF1" vrf was matched first and recorded by match variable `record` function, following with "VRF2" being matched and recorded as well, overriding previous value of "VRF1"

Template:

```
<input load="text">
router bgp 65123
 !
 address-family ipv4 vrf VRF1
  neighbor 10.1.100.212 activate
 exit-address-family
 !
 address-family ipv4 vrf VRF2
  neighbor 10.6.254.67 activate
 exit-address-family
```

(continues on next page)

(continued from previous page)

```
</input>

<group name="bgp_config">
router bgp {{ bgp_asn }}

<group name="VRFs" record="vrf">
 address-family {{ afi }} vrf {{ vrf | record(vrf) }}
  <group name="neighbors**.{{ neighbor }}**" method="table">
  neighbor {{ neighbor | let("afi_activated", True) }} activate
  {{ vrf | set(vrf) }}
  </group>
 exit-address-family {{ _end_ }}
</group>

</group>
```

Result:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.1.100.212": {
                                "afi_activated": true,
                                "vrf": "VRF2"
                            }
                        },
                        "vrf": "VRF1"
                    },
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.6.254.67": {
                                "afi_activated": true,
                                "vrf": "VRF2"
                            }
                        },
                        "vrf": "VRF2"
                    }
                ],
                "bgp_asn": "65123"
            }
        }
    ]
]
```

Example-1

In this example same data was parsed by same template, using group `record` function to record match results. To keep it simple same name "vrf" used as a source and target name for variables.

Template:

```
<input load="text">
router bgp 65123
 !
 address-family ipv4 vrf VRF2
  neighbor 10.100.100.212 activate
  neighbor 10.227.147.122 activate
 exit-address-family
 !
 address-family ipv4 vrf VRF1
  neighbor 10.61.254.67 activate
  neighbor 10.61.254.68 activate
 exit-address-family
</input>

<group name="bgp_config">
router bgp {{ bgp_asn }}

<group name="VRFs" record="vrf">
 address-family {{ afi }} vrf {{ vrf }}
  <group name="neighbors**.{{ neighbor }}**" method="table" set="vrf">
  neighbor {{ neighbor | let("afi_activated", True) }} activate
  </group>
 exit-address-family {{ _end_ }}
</group>

</group>
```

Results:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.100.100.212": {
                                "afi_activated": true,
                                "vrf": "VRF2"
                            },
                            "10.227.147.122": {
                                "afi_activated": true,
                                "vrf": "VRF2"
                            }
                        },
                        "vrf": "VRF2"
                    },
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.61.254.67": {
                                "afi_activated": true,
                                "vrf": "VRF1"
                            },
                            "10.61.254.68": {
                                "afi_activated": true,
```

(continues on next page)

(continued from previous page)

```
                                "vrf": "VRF1"
                            }
                        },
                        "vrf": "VRF1"
                    }
                ],
                "bgp_asn": "65123"
            }
        }
    ]
]
```

Example-3

In this example source and target name of variables being changed.

Template:

```
<input load="text">
router bgp 65123
 !
 address-family ipv4 vrf VRF2
  neighbor 10.100.100.212 activate
  neighbor 10.227.147.122 activate
 exit-address-family
 !
 address-family ipv4 vrf VRF1
  neighbor 10.61.254.67 activate
  neighbor 10.61.254.68 activate
 exit-address-family
</input>

<group name="bgp_config">
router bgp {{ bgp_asn }}

<group name="VRFs" record="vrf, vrf_name">
 address-family {{ afi }} vrf {{ vrf }}
  <group name="neighbors**.{{ neighbor }}**" method="table" set="vrf_name, peer_vrf">
  neighbor {{ neighbor | let("afi_activated", True) }} activate
  </group>
 exit-address-family {{ _end_ }}
</group>

</group>
```

Results:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.100.100.212": {
                                "afi_activated": true,
```

(continues on next page)

(continued from previous page)

```
                                "peer_vrf": "VRF2"
                            },
                            "10.227.147.122": {
                                "afi_activated": true,
                                "peer_vrf": "VRF2"
                            }
                        },
                        "vrf": "VRF2"
                    },
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.61.254.67": {
                                "afi_activated": true,
                                "peer_vrf": "VRF1"
                            },
                            "10.61.254.68": {
                                "afi_activated": true,
                                "peer_vrf": "VRF1"
                            }
                        },
                        "vrf": "VRF1"
                    }
                ],
                "bgp_asn": "65123"
            }
        }
    ]
]
```

set

`set="source, target, default"`

- `source` - name of variable to get value from
- `target` - optional, name of variable to assign value to
- `default` - optional, default value to assign to target variable if no source variable found

This function uses `_ttp_["results_object"].vars` dictionary to retrieve values and assign them to variable with name provided. Reference group *record* function for examples.

Example

This example demonstrates how to use set function default value. In particular, we specify default vrf value as a 'global', as a result groups that does not have vrf match, will use this default value.

Template:

```
<input load="text">
router bgp 65123
 !
 address-family ipv4
  neighbor 10.100.100.212 activate
  neighbor 10.227.147.122 activate
 exit-address-family
```

(continues on next page)

(continued from previous page)

```
 !
 address-family ipv4 vrf VRF1
  neighbor 10.61.254.67 activate
  neighbor 10.61.254.68 activate
 exit-address-family
</input>

<group name="bgp_config">
router bgp {{ bgp_asn }}

<group name="VRFs" record="vrf">
 address-family {{ afi }} vrf {{ vrf }}
 address-family {{ afi | _start_ }}
  <group name="neighbors**.{{ neighbor }}**" method="table" set="vrf, default='global'
↪">
  neighbor {{ neighbor | let("afi_activated", True) }} activate
  </group>
 exit-address-family {{ _end_ }}
</group>

</group>
```

Results:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.100.100.212": {
                                "afi_activated": true,
                                "vrf": "global"
                            },
                            "10.227.147.122": {
                                "afi_activated": true,
                                "vrf": "global"
                            }
                        }
                    },
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.61.254.67": {
                                "afi_activated": true,
                                "vrf": "VRF1"
                            },
                            "10.61.254.68": {
                                "afi_activated": true,
                                "vrf": "VRF1"
                            }
                        },
                        "vrf": "VRF1"
                    }
                ],
```

(continues on next page)

(continued from previous page)

```
                "bgp_asn": "65123"
            }
        }
    ]
]
```

Warning: default value will not be used as long as variable with given name found in `_ttp_["results_object"].vars` dictionary.

For instance, reordering text data above as:

```
router bgp 65123
 !
 address-family ipv4 vrf VRF1
  neighbor 10.61.254.67 activate
  neighbor 10.61.254.68 activate
 exit-address-family
 !
 address-family ipv4
  neighbor 10.100.100.212 activate
  neighbor 10.227.147.122 activate
 exit-address-family
```

will lead to improper results:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.61.254.67": {
                                "afi_activated": true,
                                "vrf": "VRF1"
                            },
                            "10.61.254.68": {
                                "afi_activated": true,
                                "vrf": "VRF1"
                            }
                        },
                        "vrf": "VRF1"
                    },
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.100.100.212": {
                                "afi_activated": true,
                                "vrf": "VRF1"
                            },
                            "10.227.147.122": {
                                "afi_activated": true,
                                "vrf": "VRF1"
```

(continues on next page)

(continued from previous page)

```
                                }
                            }
                        }
                    ],
                    "bgp_asn": "65123"
                }
            }
    ]
]
```

expand

`expand=""`

This function can be used to expand dot separated match variable names to nested dictionary within this particular group.

Warning: match variables can be expanded up to the same level only, meaning all except last item in match variable name should be the same, non-deterministic results will be produced otherwise.

Example

In this template target.x match variables will be expanded/transformed to nested dictionary

Template:

```
<input load="text">
switch-1#show cdp neighbors detail
-------------------------
Device ID: switch-2
Entry address(es):
  IP address: 10.13.1.7
Platform: cisco WS-C6509,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet4/6,  Port ID (outgoing port): GigabitEthernet1/5

-------------------------
Device ID: switch-3
Entry address(es):
  IP address: 10.17.14.1
Platform: cisco WS-C3560-48TS,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/1,  Port ID (outgoing port): GigabitEthernet0/1
</input>

<group name="cdp*" expand="">
Device ID: {{ target.id }}
  IP address: {{ target.top_label }}
Platform: {{ target.bottom_label | ORPHRASE }},  Capabilities: {{ ignore(ORPHRASE) }}
Interface: {{ src_label | resuball(IfsNormalize) }},  Port ID (outgoing port): {{␣
↪trgt_label | ORPHRASE | resuball(IfsNormalize) }}
</group>
```

Result:

```
[
    [
        {
            "cdp": [
                {
                    "src_label": "GigabitEthernet4/6",
                    "target": {
                        "bottom_label": "cisco WS-C6509",
                        "id": "switch-2",
                        "top_label": "10.13.1.7"
                    },
                    "trgt_label": "GigabitEthernet1/5"
                },
                {
                    "src_label": "GigabitEthernet1/1",
                    "target": {
                        "bottom_label": "cisco WS-C3560-48TS",
                        "id": "switch-3",
                        "top_label": "10.17.14.1"
                    },
                    "trgt_label": "GigabitEthernet0/1"
                }
            ]
        }
    ]
]
```


validate

`validate="schema, result='valid', info='', errors='', allow_unknown=True"`

Function to add validation results produced by Cerberus library to parsing results. Primary usecase - compliance validation and testing.

Supported parameters

- `schema` name of template variable that contains Cerberus Schema structure
- `result` field name to store boolean `True|False` validation results
- `errors` field name to store validation errors
- `info` user defined string containing test description, if provided, rendered with *sformat* function

Example

Consider simple usecase - put table together with checks that interfaces have description defined

Template:

```
<input load="text">
device-1#
interface Lo0
!
interface Lo1
 description this interface has description
</input>

<input load="text">
```

(continues on next page)

(continued from previous page)

```
device-2#
interface Lo10
!
interface Lo11
 description another interface with description
</input>

<vars>
intf_description_validate = {
    'description': {'required': True, 'type': 'string'}
}
hostname="gethostname"
</vars>

<group validate="intf_description_validate, info='{interface} has description',␣
↪result='validation_result', errors='err_details'">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 {{ hostname | set(hostname) }}
</group>

<output>
format = "tabulate"
headers = "hostname, info, validation_result, err_details"
format_attributes = "tablefmt='fancy_grid'"
returner = "terminal"
colour = ""
</output>
```

Results printed to screen:

hostname	info	validation_result	err_details
device-1	Lo0 has description	False	{'description': ['required field']}
device-1	Lo1 has description	True	{}
device-2	Lo10 has description	False	{'description': ['required field']}
device-2	Lo11 has description	True	{}

CHAPTER 6

Forming Results Structure

6.1 Group Name Attribute

Group attribute *name* used to uniquely identify group and its results within results structure. This attribute is a dot separated string, there is every dot represents a next level in hierarchy. This string is split into **path items** using dot character and converted into nested hierarchy of dictionaries and/or lists.

Consider a group with this name attribute value:

```
<group name="interfaces.vlan.L3.vrf-enabled">
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

If below data parsed with that template:

```
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
```

This result will be produced:

```
[
    {
        "interfaces": {
            "SVIs": {
                "L3": {
                    "vrf-enabled": {
                        "description": "Management",
                        "interface": "Vlan777",
                        "ip": "192.168.0.1",
```

(continues on next page)

(continued from previous page)

```
                        "mask": "24",
                        "vrf": "MGMT"
                    }
                }
            }
        }
    }
]
```

Name attribute allows to from arbitrary (from practical perspective) depth structure in deterministic fashion, enabling further programmatic consumption of produced results.

6.2 Path formatters

By default ttp assumes that all the *path items* must be joined into a dictionary structure, in other words group name "item1.item2.item3" will be transformed into nested dictionary:

```
{"item1":
     {"item2":
      {"item3": {}
    }
  }
}
```

That structure will be populated with results as parsing progresses, but in case if for "item3" more than single result datum needs to be saved, ttp will transform "item3" child to list and save further results by appending them to that list. That process happens automatically but can be influenced using *path formatters*.

Supported path formatters * and ** for group *name* attribute can be used following below rules:

- If single start character * used as a suffix (appended to the end) of path item, next level (child) of this path item always will be a list
- If double start character ** used as a suffix (appended to the end) of path item, next level (child) of this path item always will be a dictionary

Example

Consider this group with name attribute formed in such a way that interfaces item child will be a list and child of L3 path item also will be a list.:

```
<group name="interfaces*.vlan.L3*.vrf-enabled">
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

If below data parsed with that template:

```
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
```

This result will be produced:

```
[
    {
        "interfaces": [              <----this is the start of nested list
            {
                "vlan": {
                    "L3": [          <----this is the start of another nested list
                        {
                            "vrf-enabled": {
                                "description": "Management",
                                "interface": "Vlan777",
                                "ip": "192.168.0.1",
                                "mask": "24",
                                "vrf": "MGMT"
                            }
                        }
                    ]
                }
            }
        ]
    }
]
```

6.3 Dynamic Path

Above are examples of static path, where all the path items are known and predefined beforehand, however, ttp supports dynamic path formation using match variable results for certain match variable names, i.e we have match variable name set to *interface* and correspondent match result would be Gi0/1, it is possible to use Gi0/1 as a path item.

Search for dynamic path item value happens using below sequence:

- *First* - group match results searched for path item value,
- *Second* - upper group results cache (latest values) used,
- *Third* - template variables searched for path item value,
- *Last* - group results discarded as invalid

Dynamic path items specified in group *name* attribute using "*{{ item_name }}*" format, there "*{{ item_name }}*" dynamically replaced with value found using above sequence.

Example-1

In this example interface variable match values will be used to substitute {{ interface }} dynamic path items.

Data:

```
interface Port-Chanel11
  description Storage
!
interface Loopback0
  description RID
  ip address 10.0.0.3/24
!
interface Vlan777
  description Management
```

(continues on next page)

(continued from previous page)

```
  ip address 192.168.0.1/24
  vrf MGMT
```

Template:

```
<group name="interfaces.{{ interface }}">
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "Loopback0": {
                "description": "RID",
                "ip": "10.0.0.3",
                "mask": "24"
            },
            "Port-Chanel11": {
                "description": "Storage"
            },
            "Vlan777": {
                "description": "Management",
                "ip": "192.168.0.1",
                "mask": "24",
                "vrf": "MGMT"
            }
        }
    }
]
```

Because each path item is a string, and each item produced by spilling name attributes using '.' dot character, it is possible to produce dynamic path there portions of path item will be dynamically substituted.

Data:

```
interface Port-Chanel11
  description Storage
!
interface Loopback0
  description RID
  ip address 10.0.0.3/24
!
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
```

Template:

```
<group name="interfaces.cool_{{ interface }}_interface">
interface {{ interface }}
  description {{ description }}
```

(continues on next page)

(continued from previous page)

```
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "cool_Loopback0_interface": {
                "description": "RID",
                "ip": "10.0.0.3",
                "mask": "24"
            },
            "cool_Port-Chanel11_interface": {
                "description": "Storage"
            },
            "cool_Vlan777_interface": {
                "description": "Management",
                "ip": "192.168.0.1",
                "mask": "24",
                "vrf": "MGMT"
            }
        }
    }
]
```

Note: Substitution of dynamic path items happens using re.sub method without the limit set on the count of such a substitutions, e.g. if path item "cool_{{ interface }}_interface_{{ interface }}" and if interface value is "Gi0/1" resulted path item will be "cool_Gi0/1_interface_Gi0/1"

Nested hierarchies also supported with dynamic path, as if no variable found in the group match results ttp will try to find variable in the dynamic path cache or template variables.

Example-3

Data:

```
ucs-core-switch-1#show run | section bgp
router bgp 65100
  vrf CUST-1
    neighbor 59.100.71.193
      remote-as 65101
      description peer-1
      address-family ipv4 unicast
        route-map RPL-1-IMPORT-v4 in
        route-map RPL-1-EXPORT-V4 out
      address-family ipv6 unicast
        route-map RPL-1-IMPORT-V6 in
        route-map RPL-1-EXPORT-V6 out
    neighbor 59.100.71.209
      remote-as 65102
      description peer-2
      address-family ipv4 unicast
        route-map AAPTVRF-LB-BGP-IMPORT-V4 in
        route-map AAPTVRF-LB-BGP-EXPORT-V4 out
```

Template:

```
<vars>
hostname = "gethostname"
</vars>

<group name="{{ hostname }}.router.bgp.BGP_AS_{{ asn }}">
router bgp {{ asn }}
  <group name="vrfs.{{ vrf_name }}">
  vrf {{ vrf_name }}
    <group name="peers.{{ peer_ip }}">
    neighbor {{ peer_ip }}
      remote-as {{ peer_asn }}
      description {{ peer_description }}
      <group name="afi.{{ afi }}.unicast">
      address-family {{ afi }} unicast
        route-map {{ rpl_in }} in
        route-map {{ rpl_out }} out
      </group>
    </group>
   </group>
</group>
```

Result:

```
- ucs-core-switch-1:
    router:
      bgp:
        BGP_AS_65100:
          vrfs:
            CUST-1:
              peers:
                59.100.71.193:
                  afi:
                    ipv4:
                      unicast:
                        rpl_in: RPL-1-IMPORT-v4
                        rpl_out: RPL-1-EXPORT-V4
                    ipv6:
                      unicast:
                        rpl_in: RPL-1-IMPORT-V6
                        rpl_out: RPL-1-EXPORT-V6
                  peer_asn: '65101'
                  peer_description: peer-1
                59.100.71.209:
                  afi:
                    ipv4:
                      unicast:
                        rpl_in: RPL-2-IMPORT-V6
                        rpl_out: RPL-2-EXPORT-V6
                  peer_asn: '65102'
                  peer_description: peer-2
```

6.4 Dynamic path with path formatters

Dynamic path with path formatters is also supported. In example below child for *interfaces* will be a list.

Example

Data:

```
interface Port-Chanel11
  description Storage
!
interface Loopback0
  description RID
  ip address 10.0.0.3/24
!
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
```

Template:

```
<group name="interfaces*.{{ interface }}">
interface {{ interface }}
  description {{ description }}
  ip address {{ ip }}/{{ mask }}
  vrf {{ vrf }}
</group>
```

Result:

```
[
    {
        "interfaces": [
            {
                "Loopback0": {
                    "description": "RID",
                    "ip": "10.0.0.3",
                    "mask": "24"
                },
                "Port-Chanel11": {
                    "description": "Storage"
                },
                "Vlan777": {
                    "description": "Management",
                    "ip": "192.168.0.1",
                    "mask": "24",
                    "vrf": "MGMT"
                }
            }
        ]
    }
]
```

6.5 Anonymous group

If no nested functionality required or results structure needs to be kept as flat as possible, templates without <group> tag can be used - so called *non hierarchical templates*.

There is a notion of *top* <group> tag exists, that at the tag that located in the top of xml document hierarchy, that tag can be lacking name attribute as well.

In both cases above, ttp will automatically reconstruct <group> tag and name attribute for it, setting name to "_anonymous_" value. At the end _anonymous_ path will be stripped of results tree to flatten it.

Note: <group> tag without name attribute does have support for all the other group attributes as well as nested groups, however, nested groups *must* have name attribute set on them otherwise nested hierarchy will not be preserved leading to unpredictable results.

Warning: Template variables name attribute ignored if groups with "_anonymous_" path used, as a result template variables will not be save into results.

Example

Example for <group> without *name* attribute.

Data:

```
interface Port-Chanel11
  description Storage
!
interface Loopback0
  description RID
  ip address 10.0.0.3/24
!
interface Vlan777
  description Management
  ip address 192.168.0.1/24
  vrf MGMT
!
```

Template:

```
<group>
interface {{ interface }}
  description {{ description }}
<group name = "ips">
  ip address {{ ip }}/{{ mask }}
</group>
  vrf {{ vrf }}
!{{_end_}}
</group>
```

Result:

```
[
    {
        "description": "Storage",
        "interface": "Port-Chanel11"
    },
    {
        "description": "RID",
        "interface": "Loopback0",
        "ips": {
            "ip": "10.0.0.3",
            "mask": "24"
        }
```

(continues on next page)

(continued from previous page)

```
    },
    {
        "description": "Management",
        "interface": "Vlan777",
        "ips": {
            "ip": "192.168.0.1",
            "mask": "24"
        },
        "vrf": "MGMT"
    }
]
```

6.6 Null path name attribute

It is possible to specify null path as a name, null path looks like `name="_"` or null path can be used as a first item in the path - `name="_.nextlevel"`.

Special handling implemented for null path - TTP will merge results with parent for group with null path, as a result null path _ will not appear in results.

One of the usecases for this feature is to create a group that will behave like a normal group in terms of results forming and processing, but will merge with parent in the process of saving into overall results.

Example

In this example peer_software used together with _line_ indicator to extract results, however, for _line_ to behave properly it was defined within separate group with explicit _stat_ and _end_ indicators. First, this is how template would look like without null path:

```
<input load="text">
Device ID: switch-2.net
IP address: 10.251.1.49

Version :
Cisco Internetwork Operating System Software
IOS (tm) s72033_rp Software (s72033_rp-PK9SV-M), Version 12.2(17d)SXB11a, RELEASE␣
↪SOFTWARE (fc1)

advertisement version: 2
</input>

<group>
Device ID: {{ peer_hostname }}
IP address: {{ peer_ip }}

<group name="peer_software">
Version : {{ _start_ }}
{{ peer_software | _line_ }}
{{ _end_ }}
</group>

</group>
```

And result would be:

```
[
    [
        {
            "peer_hostname": "switch-2.net",
            "peer_ip": "10.251.1.49",
            "peer_software": {
                "peer_software": "Cisco Internetwork Operating System Software \nIOS␣
↪(tm) s72033_rp Software (s72033_rp-PK9SV-M), Version 12.2(17d)SXB11a, RELEASE␣
↪SOFTWARE (fc1)"
            }
        }
    ]
]
```

Above results have a bit of redundancy in them as they have unnecessary hierarchy to store peer_software details, to avoid that, null path can be used:

```
<input load="text">
Device ID: switch-2.net
IP address: 10.251.1.49

Version :
Cisco Internetwork Operating System Software
IOS (tm) s72033_rp Software (s72033_rp-PK9SV-M), Version 12.2(17d)SXB11a, RELEASE␣
↪SOFTWARE (fc1)

advertisement version: 2
</input>

<group>
Device ID: {{ peer_hostname }}
IP address: {{ peer_ip }}

<group name="_">
Version : {{ _start_ }}
{{ peer_software | _line_ }}
{{ _end_ }}
</group>

</group>
```

Results with new template:

```
[
    [
        {
            "peer_hostname": "switch-2.net",
            "peer_ip": "10.251.1.49",
            "peer_software": "Cisco Internetwork Operating System Software \nIOS (tm)␣
↪s72033_rp Software (s72033_rp-PK9SV-M), Version 12.2(17d)SXB11a, RELEASE SOFTWARE␣
↪(fc1)"
        }
    ]
]
```

Even though peer_software match variable was defined in separate group, because of null path, it was merged with parent group, flattening results structure.

6.7 Absolute path

By default TTP treats name attribute as a relative path, relative to parent groups, expanding path to full (absolute) path for each and every group.

For instance for below template:

```
<input load="text">
router bgp 65123
 !
 address-family ipv4 vrf VRF1
  neighbor 10.100.100.212 route-policy DENY_ALL in
  neighbor 10.227.147.122 route-policy DENY_ALL in
 exit-address-family
 !
 address-family ipv4 vrf VRF2
  neighbor 10.61.254.67 route-policy DENY_ALL in
  neighbor 10.61.254.68 route-policy DENY_ALL in
 exit-address-family
</input>

<group name="bgp_config">
router bgp {{ bgp_asn }}

<group name="VRFs">
 address-family {{ afi }} vrf {{ vrf }}
  <group name="neighbors**.{{ neighbor }}**" method="table">
  neighbor {{ neighbor }} route-policy {{ ingreass_rpl }} in
  </group>
</group>

</group>
```

Paths for child groups will be expanded to the list of absolute path items:

Name attribute	Path
bgp_config	[bgp_config]
VRFs	[bgp_config, VRFs]
neighbors**.{{ neighbor }}**	[bgp_config, VRFs, neighbors, {{ neighbor }}]

Results structure for above template will look like:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.100.100.212": {
                                "ingreass_rpl": "DENY_ALL"
                            },
                            "10.227.147.122": {
                                "ingreass_rpl": "DENY_ALL"
                            }
```

(continues on next page)

(continued from previous page)

```
                        },
                        "vrf": "VRF1"
                    },
                    {
                        "afi": "ipv4",
                        "neighbors": {
                            "10.61.254.67": {
                                "ingreass_rpl": "DENY_ALL"
                            },
                            "10.61.254.68": {
                                "ingreass_rpl": "DENY_ALL"
                            }
                        },
                        "vrf": "VRF2"
                    }
                ],
                "bgp_asn": "65123"
            }
        }
    ]
]
```

However, sometimes it might be beneficial to have a capability to flatten hierarchical structure by specifying absolute path for child groups.

To instruct TTP to treat name attribute as an absolute path, it should be prepended (started) with forward slash / character.

Example Template:

```
<input load="text">
router bgp 65123
 !
 address-family ipv4 vrf VRF1
  neighbor 10.100.100.212 route-policy DENY_ALL in
  neighbor 10.227.147.122 route-policy DENY_ALL in
 exit-address-family
 !
 address-family ipv4 vrf VRF2
  neighbor 10.61.254.67 route-policy DENY_ALL in
  neighbor 10.61.254.68 route-policy DENY_ALL in
 exit-address-family
</input>

<group name="bgp_config">
router bgp {{ bgp_asn }}

<group name="VRFs">
 address-family {{ afi }} vrf {{ vrf }}
  <group name="/neighbors**.{{ neighbor }}**" method="table">
  neighbor {{ neighbor }} route-policy {{ ingreass_rpl }} in
  </group>
</group>

</group>
```

In above template, note the name of this child group - *name="/neighbors**.{{ neighbor }}**"* - it is prepended with forward slash character and treated as absolute path. Result structure for above template will be:

```
[
    [
        {
            "bgp_config": {
                "VRFs": [
                    {
                        "afi": "ipv4",
                        "vrf": "VRF1"
                    },
                    {
                        "afi": "ipv4",
                        "vrf": "VRF2"
                    }
                ],
                "bgp_asn": "65123"
            },
            "neighbors": {
                "10.100.100.212": {
                    "ingreass_rpl": "DENY_ALL"
                },
                "10.227.147.122": {
                    "ingreass_rpl": "DENY_ALL"
                },
                "10.61.254.67": {
                    "ingreass_rpl": "DENY_ALL"
                },
                "10.61.254.68": {
                    "ingreass_rpl": "DENY_ALL"
                }
            }
        }
    ]
]
```


This is because path attribute will not be expanded for *neighbors* child group and will be treated as is, effectively shortening the hierarchy of results structure and flattening it.

6.8 Template results mode

TBD

6.9 TTP object results structure

TBD

6.10 Expanding Match Variables

Match variables can have name with dot characters in it. . .

CHAPTER 7

Inputs

Inputs can be used to specify data location and how it should be loaded or filtered. Inputs can be attached to groups for parsing, for instance this particular input data should be parsed by this set of groups only. That can help to increase the overall performance as only data belonging to particular group will be parsed.

Note: Order of inputs preserved as internally they represented using OrderedDict object, that can be useful if data produced by first input needs to bused by other inputs.

Assuming we have this folders structure to store data that needs to be parsed:

```
/my/base/path/
    Data/
      Inputs/
         data-1/
            sw-1.conf
            sw-1.txt
         data-2/
            sw-2.txt
            sw3.txt
```

Where content:

```
[sw-1.conf]
interface GigabitEthernet3/7
 switchport access vlan 700
!
interface GigabitEthernet3/8
 switchport access vlan 800
!

[sw-1.txt]
interface GigabitEthernet3/2
 switchport access vlan 500
```

(continues on next page)

(continued from previous page)

```
!
interface GigabitEthernet3/3
 switchport access vlan 600
!

[sw-2.txt]
interface Vlan221
  ip address 10.8.14.130/25

interface Vlan223
  ip address 10.10.15.130/25

[sw3.txt]
interface Vlan220
  ip address 10.9.14.130/24

interface Vlan230
  ip address 10.11.15.130/25
```

Template below uses inputs in such a way that for "data-1" folder only files that have ".txt" extension will be parsed by group "interfaces1", for input named "dataset-2" only files with names matching "sw-d.*" regular expression will be parsed by "interfaces2" group. In addition, base path provided that will be appended to each url within *url* input parameter. Tag text for input "dataset-1" structured using YAML representation, while "dataset-2" uses python language definition.

As a result of inputs filtering, only "sw-1.txt" will be processed by "dataset-1" input because it is the only file that has ".txt" extension, only "sw-2.txt" will be processed by input "dataset-2" because "sw3.txt" not matched by "sw-d.*" regular expression.

Template:

```
<template base_path="/my/base/path/">
<input name="dataset-1" load="yaml" groups="interfaces1">
url: "/Data/Inputs/data-1/"
extensions: ["txt"]
</input>

<input name="dataset-2" load="python" groups="interfaces2">
url = ["/Data/Inputs/data-2/"]
filters = ["sw\-\d.*"]
</input>

<group name="interfaces1">
interface {{ interface }}
 switchport access vlan {{ access_vlan }}
</group>

<group name="interfaces2">
interface {{ interface }}
  ip address {{ ip  }}/{{ mask }}
</group>
</template>
```

And result would be:

```
[
    {
```

(continues on next page)

(continued from previous page)

```
        "interfaces1": [
            {
                "access_vlan": "500",
                "interface": "GigabitEthernet3/2"
            },
            {
                "access_vlan": "600",
                "interface": "GigabitEthernet3/3"
            }
        ]
    },
    {
        "interfaces2": [
            {
                "interface": "Vlan221",
                "ip": "10.8.14.130",
                "mask": "25"
            },
            {
                "interface": "Vlan223",
                "ip": "10.10.15.130",
                "mask": "25"
            }
        ]
    }
]
```

7.1 Inputs reference

7.1.1 Attributes

There are a number of attributes supported by input tag, these attributes help to define input behavior and how data should be loaded and parsed.

Additionally input tag text payload can contain structured data, that data can be retrieved using `get_input_load` method. Input tag `load` attribute instructs how to load that data. For instance, if tag text structured in yaml format, yaml loader can be used to load it in Python data structure.

Attributes in input tag and attributes loaded from input tag text are combined in single structure if both are dictionaries, as a result, most of the attributes can be specified in either way.

Attribute	Description
name	Uniquely identifies input within template
groups	comma-separated list of group(s) that should be used to parse input data
load	loader name that should be used to load text data from input tag
url	single or list of urls of data location
extensions	single or list of file extensions to load, e.g. “txt” or “log” or “conf”
filters	single or list of regular expression to filter file names

name

```
name="string"
```

- string (optional) - name of the input to reference in group *input* attribute. Default value is "Default_Input" and used internally to store set of data that should be parsed by all groups.

groups

```
groups="group1, group2, ... , groupN"
```

- groupN (optional) - Default value is "all", comma separated string of group names that should be used to parse given input data. Default value is "all" - input data will be parsed by each group.

Each group will be used only once to parse input data, for instance if `groups="group1, group1"`, group1 will be parse that input data only once, as TTP makes a list of unique (non repeating values, internally, that achieved by converting list to set and back to sorted list) groups for each input.

Note: Group tag *input* attribute can be used to reference inputs' names or OS path to files, it is considered to be more specific, for example when several groups in the template have identical *name* attribute, referencing these groups by name in input tag *groups* attribute will result in input data to be parsed by all the groups with that name, on the other hand, if input name referenced in group tag *input* attribute, data of this input will only be parsed by this group even if several group have the same name.

---

load

```
load="loader_name"
```

- loader_name - name of the loader that should be used to load input tag text data, supported values are `python`, `yaml`, `json` or `text`, if text used as a loader, text data within input tag itself used as an input data and parsed by a set of given groups or by all groups.

Example

Below template contains input with text data that should be parsed, that is useful for testing purposes or for small data sets.

Template:

```
<input name="test1" load="text" groups="interfaces.trunks">
interface GigabitEthernet3/3
 switchport trunk allowed vlan add 138,166-173
!
interface GigabitEthernet3/4
 switchport trunk allowed vlan add 100-105
!
interface GigabitEthernet3/5
 switchport trunk allowed vlan add 459,531,704-707
</input>

<group name="interfaces.trunks">
interface {{ interface }}
 switchport trunk allowed vlan add {{ trunk_vlans }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "trunks": [
                {
                    "interface": "GigabitEthernet3/3",
                    "trunk_vlans": "138,166-173"
                },
                {
                    "interface": "GigabitEthernet3/4",
                    "trunk_vlans": "100-105"
                },
                {
                    "interface": "GigabitEthernet3/5",
                    "trunk_vlans": "459,531,704-707"
                }
            ]
        }
    }
]
```

url

`url="url-1"` or `url=["url-1", "url-2", ... , "url-N"]`

- url-N - string or list of strings that contains absolute or relative OS path to file or to directory of file(s) that needs to be parsed.

Few notes on relative path:

- if template tag `base_path` attribute provide, base_path value used to extend relative path - appended to relative path of each url
- if no template tag `base_path` attribute provided, in case if url parameter contains relative path, this path will be extended in relation to the folder where TTP invoked

TTP uses Python built-in OS module to load input files. Examples of relative path: `./relative/path/` or `../relative/path/` or `relative/path/` - any path that OS module considers as a relative path.

Example-1

Template tag contains `base_path` attribute.

Template:

```
<template base_path="C:/base/path/to/">
<input load="yaml">
url: "./Data/Inputs/dataset_1/"
</input>

<group name="interfaces">
interface {{ interface }}
  ip address {{ ip  }}/{{ mask }}
</group>
</template>
```

After combining base path and provided url, TTP will use `C:/base/path/to/Data/Inputs/dataset_1/` to load input data files.

Example-2

No `base_path` attribute.

Template:

```
<input load="yaml">
url: "./Data/Inputs/dataset_1/"
</input>

<group name="interfaces">
interface {{ interface }}
  ip address {{ ip  }}/{{ mask }}
</group>
```

In this case TTP will search for data files using relative path `./Data/Inputs/dataset_1/`, extending it in relation to current directory, directory where TTP was executed.

extensions

`extensions="extension-1"` or `extensions=["extension-1", "extension-2", ... , "extension-N"]`

- extension-N - string or list of strings that contains file extensions that needs to be parsed e.g. txt, log, conf etc. In case if *url* is OS path to directory and not single file, ttp will use this strings to check if file names ends with one of given extensions, if so, file will be loaded and skipped otherwise.

filters

`filters="regex-1"` or `filters=["regex-1", "regex-2", ... , "regex-N"]`

- regex-N - string or list of strings that contains regular expressions. If *url* is OS path to directory and not single file, ttp will use this strings to run re search against file names to load only files with names that matched by at least one regex.

7.1.2 Functions

Input tag support functions to pre-process data.

Attribute	Description
functions attribute	pipe-separated list of functions
macro	comma-separated list of macro functions to run input data through
extract_commands	comma-separated list of commands output to extract from text data
test	Test function to verify input function handling

functions attribute

`functions="function1('attributes') | function2('attributes') | ... | functionN('attributes')"`

- functionN - name of the input function together with it's attributes

This attribute allow to define a sequence of function, the main advantage of using string of functions against defining functions directly in the input tag is the fact that functions order will be honored, otherwise functionality is the same.

Warning: pipe 'l' symbol must be used to separate function names, not comma

macro

```
macro="name1, name2, ... , nameN"
```

- nameN - comma separated string of macro functions names that should be used to run input data through. The sequence is *preserved* and macros executed in specified order, in other words macro named name2 will run after macro name1.

Macro brings Python language capabilities to input data processing and validation during TTP module execution, as it allows to run custom python functions. Macro functions referenced by their name in input tag macro definitions.

Macro function must accept only one attribute to hold input data text.

Depending on data returned by macro function, TTP will behave differently according to these rules:

- If macro returns True or False - original data unchanged, macro handled as condition functions, stopping further functions execution on False and keeps processing input data on True
- If macro returns None - data processing continues, no additional logic associated
- If macro returns single item - that item replaces original data supplied to macro and processed further by other input tag functions

extract_commands

```
extract_commands="command1, command2, ... , commandN"
```

Purpose of this function is for each network device command string TTP can extract associated data from input text, so that input groups will only process data they designed to parse

..note:: to be able to successfully extract show commands output, text data should contain device hostname together with command itself. `gethostname` function will be called on data to extract hostname

Example

In below template, only "show interfaces" command output will be processed, as only that command specified in input `extract_commands` attribute.

Template:

```
<input load="text" extract_commands="show interfaces">
cpe1#show int
GigabitEthernet33 is up, line protocol is up
  Hardware is CSR vNIC, address is 0800.2779.9999 (bia 0800.2779.9999)
cpe1#show interfaces
GigabitEthernet44 is up, line protocol is up
  Hardware is CSR vNIC, address is 0800.2779.e896 (bia 0800.2779.e896)
cpe1#show interf
GigabitEthernet55 is up, line protocol is up
  Hardware is CSR vNIC, address is 0800.2779.e888 (bia 0800.2779.e888)
</input>
```

(continues on next page)

(continued from previous page)

```
<group name="interfaces_status">
{{ interface }} is up, line protocol is up
  Hardware is CSR vNIC, address is {{ mac }} (bia {{ bia_mac }})
</group>
```

Result:

```
[
    [
        {
            "interfaces_status": {
                "bia_mac": "0800.2779.e896",
                "interface": "GigabitEthernet44",
                "mac": "0800.2779.e896"
            }
        }
    ]
]
```

test

`test=""`

Test function to verify input function call, test simply prints informational message to the screen, indicating that input test function was called.

7.1.3 Sources

Inputs can uses various sources to retrieve data for parsing.

Source	Description
Netmiko	uses Netmiko library to retrieve data from devices over SSH or Telnet
Nornir	uses Nornir library with Netmiko plugin to retrieve data from devices

Netmiko

Prerequisites: Netmiko library need to be installed on the system

This source allows to retrieve configuration or state data from network devices using SSH or Telnet by connecting to devices one by one (serial execution).

Supported attributes

- `commands` list of commands to retrieve from devices
- `devices` list of devices to retrieve commands from
- `username` device username, if value is `get_user_input` prompts user for input
- `password` device password, if value is `get_user_pass` prompts user for input
- `Netmiko kwargs` - any other arguments to pass on to Netmiko ConnectHandler method

Example

Template:

```
<vars>
hostname="gethostname"
</vars>

<input source="netmiko" name="arp">
devices = ["192.168.217.10", "192.168.217.7"]
device_type = "cisco_ios"
username = "cisco"
password = "cisco"
commands = ["show ip arp"]
</input>

<group name="arp" input="arp">
Internet  {{ ip }}  {{ age }}   {{ mac }} ARPA   {{ interface }}
{{ hostname | set(hostname) }}
</group>


<input source="netmiko" name="interfaces">
host = "192.168.217.10"
device_type = "cisco_ios"
username = "get_user_input"
password = "get_user_pass"
commands = ["show run"]
</input>

<group name="interfaces" input="interfaces">
interface {{ interface }}
 description {{ description | ORPHRASE }}
 encapsulation dot1Q {{ dot1q }}
 ip address {{ ip }} {{ mask }}
{{ hostname | set(hostname) }}
</group>
```

Nornir

Prerequisites: Nornir library need to be installed on the system

This source allows to retrieve configuration or state data from network devices using Nornir library. Nornir runs connection to devices asynchronously (in parallel), allowing significantly reduce the time required to retrieve data.

This source uses netmiko_send_command task plugin to send commands to devices.

Supported attributes

- `hosts` Nornir hosts inventory data with devices' details
- `commands` list of commands to execute on devices
- `username` devices username, if value is `get_user_input` prompts user for input
- `password` devices password, if value is `get_user_pass` prompts user for input
- `num_workers` default is 100, maximum number of worker threads to instantiate for tasks execution
- `netmiko_kwargs` arguments to pass on to netmiko_send_command task plugin, default values:

```
strip_prompt = False
strip_command = False
```

Nornir normally uses inventory data to get username and password values, TTP allows to specify these attributes separately and share them with each host in inventory. Username and password provided within hosts inventory considered to be more specific and not overridden.

Example

Template:

```
<input source="nornir" name="arp">
hosts = {
    "R1": {
            "hostname": "192.168.1.151",
            "platform": "cisco_ios"
        },
    "R2": {
            "hostname": "192.168.1.153",
            "username": "cisco",
            "password": "cisco",
            "platform": "cisco_ios"
        }
}
username = "get_user_input"
password = "get_user_pass"
commands = ["show ip arp"]
netmiko_kwargs = {
    "strip_prompt": False,
    "strip_command": False
}
</input>

<group name="arp" input="arp">
Internet  {{ ip }}  {{ age }}   {{ mac }} ARPA   {{ interface }}
{{ hostname | set(hostname) }}
</group>
```

CHAPTER 8

Outputs

Outputs system allows to process parsing results, format them in certain way and return results to various destination. For instance, using yaml formatter results can take a form of YAML syntax and using file returner these results can be saved into file.

Outputs can be chained, say results after passing through first outputter will serve as an input for next outputter. That allows to implement complex processing logic of results produced by ttp.

The opposite way would be that each output defined in template will work with parsing results, transform them in different way and return to different destinations. An example of such a behavior might be the case when first outputter form csv table and saves it onto the file, while second outputter will render results with Jinja2 template and print them to the screen.

In addition two types of outputter exists - template specific and group specific. Template specific outputs will process template overall results, while group-specific will work with results of this particular group only.

There is a set of function available in outputs to process/modify results further.

Note: If several outputs provided - they run sequentially in the order defined in template. Within single output, processing order is - functions run first, after that formatters, followed by returners.

8.1 Outputs reference

8.1.1 Attributes

There are a number of attributes that outputs system can use. Some attributes can be specific to output itself (name, description), others can be used by formatters or returners.

Name	Description
name	name of the output, can be referenced in group *output* attribute
description	attribute to contain description of outputter
load	name of the loader to use to load output tag text
returner	returner to use to return data e.g. self, file, terminal
format	formatter to use to format results

name

```
name="output_name"
```

Name of the output, optional attribute, can be used to reference it in groups *output* attribute, in that case that output will become group specific and will only process results for this group.

description

```
name="descrition_string"
```

descrition_string, optional string that contains output description or notes, can serve documentation purposes.

load

```
load="loader_name"
```

Name of the loader to use to render supplied output tag text data, default is python.

Supported loaders:

- python - uses python exec method to load data structured in native Python formats
- yaml - relies on PyYAML to load YAML structured data
- json - used to load JSON formatted variables data
- ini - configparser Python standard module used to read variables from ini structured file
- csv - csv formatted data loaded with Python *csv* standard library module

returner

```
returner=returner_name"
```

Name of the returner to use to return results.

format

```
format=formatter_name"
```

Name of the formatter to use to format results.

8.1.2 Functions

Output system provides support for a number of functions. Functions help to process overall parsing results with intention to modify, check or filter them in certain way.

Name	Description
is_equal	checks if results equal to structure loaded from the output tag text
set_data	insert arbitrary data to results at given path, replacing any existing results
dict_to_list	transforms dictionary to list of dictionaries at given path
traverse	returns data at given path location of results tree
macro	passes results through macro function
output functions	pipe separated list of functions to run results through
deepdiff	function to compare result structures

is_equal

```
functions="is_equal"
```

Function is_equal load output tag text data into python structure (list, dictionary etc.) using given loader and performs comparison with parsing results. is equal returns a dictionary of three elements:

```
{
    "is_equal": true|false,
    "output_description": "output description as set in description attribute",
    "output_name": "name of the output"
}
```

This function use-cases are various tests or compliance checks, one can construct a set of template groups to produce results, these results can be compared with predefined structures to check if they are matching, based on comparison a conclusion can be made such as whether or not source data satisfies certain criteria.

Example

Template:

```
<input load="text">
interface Loopback0
 ip address 192.168.0.113/24
!
interface Vlan778
 ip address 2002::fd37/124
!
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>

<output
name="test output 1"
load="json"
description="test results equality"
functions="is_equal"
>
[
```

(continues on next page)

(continued from previous page)

```
    {
        "interfaces": [
            {
                "interface": "Loopback0",
                "ip": "192.168.0.113",
                "mask": "24"
            },
            {
                "interface": "Vlan778",
                "ip": "2002::fd37",
                "mask": "124"
            }
        ]
    }
]
</output>
```

Results:

```
{
    "is_equal": true,
    "output_description": "test results equality",
    "output_name": "test output 1"
}
```

set_data

This function not yet tested and not available for use, listed here as a placeholder.

dict_to_list

`dict_to_list="key_name='key', path='dot.separated.path'"`

- key_name - string, name of the key to use to assign previous key as a value
- path - string, dot separated path to data that need to be transformed

This functions help to flatten dictionary data by converting it to list e.g. if data is:

```
{"Fa0" : {"admin": "administratively down"}, "Ge0/1": {"access_vlan": "24"}}
```

and key_name="interface", dit_to_list function will return this list:

```
[ {"admin": "administratively down", "interface": "Fa0"},
  {"access_vlan": "24", "interface": "Ge0/1"} ]
```

Primary usecase is to produce list data out of dictionary, this function used internally by table output formatter for that purpose.

Example

Template:

```
<input load="text">
some.user@router-fw-host> show configuration interfaces | display set
```

(continues on next page)

(continued from previous page)

```
set interfaces ge-0/0/11 unit 0 description "SomeDescription glob1"
set interfaces ge-0/0/11 unit 0 family inet address 10.0.40.121/31
set interfaces lo0 unit 0 description "Routing Loopback"
set interfaces lo0 unit 0 family inet address 10.6.4.4/32
</input>

<group name="{{ interface }}{{ unit }}**" method="table">
set interfaces {{ interface }} unit {{ unit }} family inet address {{ ip }}
set interfaces {{ interface }} unit {{ unit }} description "{{ description | ORPHRASE␣
↪}}"
</group>

<output dict_to_list="key_name='interface'"/>
```

Result:

```
[
    [
        [
            {
                "description": "SomeDescription glob1",
                "interface": "ge-0/0/110",
                "ip": "10.0.40.121/31"
            },
            {
                "description": "Routing Loopback",
                "interface": "lo00",
                "ip": "10.6.4.4/32"
            }
        ]
    ]
]
```

As a comparison example, here is how results would look like without running them through dict_to_list function:

```
[
    [
        {
            "ge-0/0/110": {
                "description": "SomeDescription glob1",
                "ip": "10.0.40.121/31"
            },
            "lo00": {
                "description": "Routing Loopback",
                "ip": "10.6.4.4/32"
            }
        }
    ]
]
```

traverse

`traverse="path='dot.separated.path'"`

- path - string, dot separated path to data that need to be transformed

traverse function walks results tree up to the level of given path and return data at that location.

Example

Template:

```
<input load="text">
some.user@router-fw-host> show configuration interfaces | display set
set interfaces ge-0/0/11 unit 0 description "SomeDescription glob1"
set interfaces ge-0/0/11 unit 0 family inet address 10.0.40.121/31
set interfaces lo0 unit 0 description "Routing Loopback"
set interfaces lo0 unit 0 family inet address 10.6.4.4/32
</input>

<group name="my.long.path.{{ interface }}{{ unit }}**" method="table">
set interfaces {{ interface }} unit {{ unit }} family inet address {{ ip }}
set interfaces {{ interface }} unit {{ unit }} description "{{ description | ORPHRASE
↪}}"
</group>

<output traverse="path='my.long.path'"/>
```

Result:

```
[
    [
        {
            "ge-0/0/110": {
                "description": "SomeDescription glob1",
                "ip": "10.0.40.121/31"
            },
            "lo00": {
                "description": "Routing Loopback",
                "ip": "10.6.4.4/32"
            }
        }
    ]
]
```

For comparison, without traverse TTP would return these results:

```
[
    [
        {
            "my": {
                "long": {
                    "path": {
                        "ge-0/0/110": {
                            "description": "SomeDescription glob1",
                            "ip": "10.0.40.121/31"
                        },
                        "lo00": {
                            "description": "Routing Loopback",
                            "ip": "10.6.4.4/32"
                        }
                    }
                }
            }
        }
    ]
]
```

macro

`macro="func_name"` or `functions="macro('func_name1') | macro('func_name2')"`

Output macro function allows to process whole results using custom function(s) defined within <macro> tag.

Example

Template:

```
<input load="text">
interface Vlan778
 ip address 2002::fd37::91/124
!
interface Loopback991
 ip address 192.168.0.1/32
!
</input>

<macro>
def check_svi(data):
    # data is a list of lists:
    # [[{'interface': 'Vlan778', 'ip': '2002::fd37::91', 'mask': '124'},
    #   {'interface': 'Loopback991', 'ip': '192.168.0.1', 'mask': '32'}]]
    for item in data[0]:
        if "Vlan" in item["interface"]:
            item["is_svi"] = True
        else:
            item["is_svi"] = False
</macro>

<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>

<output macro="check_svi"/>
```

Results:

```
[
    [
        {
            "interface": "Vlan778",
            "ip": "2002::fd37::91",
            "is_svi": true,
            "mask": "124"
        },
        {
            "interface": "Loopback991",
            "ip": "192.168.0.1",
            "is_svi": false,
            "mask": "32"
        }
    ]
]
```

output functions

```
functions="function1('attributes') | function2('attributes') | ... |
functionN('attributes')"
```

- functionN - name of the output function together with it's attributes

String, that contains pipe separated list of output functions with functions' attributes

deepdiff

```
deepdiff="input_before, input_after, template_before, mode=bulk,
add_field=difference, **kwargs
```

- `input_before` - string, name of input, which results should be used to compare with
- `input_after` - string, name of input, which results should be used for comparing
- `template_before` - string, name of template tag, results of which to use to compare with
- `add_field` - string, name of field to add compare results, by default is False, hence compare results will replace results data
- `mode` - string, `bulk` (default) or `iterate` modes supported to modify comparison behavior
- `kwargs` - any arguments supported by deepdiff DeepDiff object, such as ignore_order or verbose_level

Prerequisites: Python deepdiff library need to be installed.

This function takes parsing results for specified inputs and compares one against another using DeepDiff library deepdiff object.

The usecase for this function might be having two folders on the hard drive, one folder with data before and second folder with data after changes were done to network devices, TTP can be used to parse this data and run results comparison using deepdiff function, showing the differences between Python structures content, as opposed to comparing text data itself.

Few words about **mode**. In `bulk` mode overall `input_before` results compared with overall `input_after` results, in `iterate` mode **first** item in results for `input_before` compared (iterated) against each item in results for `input_after`.

Example-1

In this example, results of inputs with names `input_before` and `input_after` will be compared against each other using default 'bulk' comparison mode.

Template:

```
<input name="input_before" load="text">
interface FastEthernet1/0/1
 description Foo
!
</input>

<input name="one_more" load="text">
interface FastEthernet1/0/1
 description FooBar
!
</input>

<input name="input_after" load="text">
```

(continues on next page)

(continued from previous page)

```
interface FastEthernet1/0/1
 description Bar
!
</input>

<group
name="interfaces*">
interface {{ interface }}
 description {{ description }}
</group>

<output deepdiff="input_before, input_after, add_field=difference, ignore_order=False,
↪ verbose_level=2"/>
```

Results:

```
[   [   {   'interfaces': [   {   'description': 'Foo',
                                  'interface': 'FastEthernet1/0/1'}]},
        {   'interfaces': [   {   'description': 'FooBar',
                                  'interface': 'FastEthernet1/0/1'}]},
        {   'interfaces': [   {   'description': 'Bar',
                                  'interface': 'FastEthernet1/0/1'}]},
        {   'difference': {   'values_changed': {   "root['interfaces'][0][
↪'description']": {   'new_value': 'Bar',
                                                                                ␣
↪       'old_value': 'Foo'}}}}]]
```

As you can see comparison results were appended to overall results as a dictionary with top key set to `add_field` value `difference` in this case, if `add_field` would be omitted, parsing results will be replaced with comparison outcome and TTP will produce this output:

```
[   {   'values_changed': {   "root['interfaces'][0]['description']": {   'new_value
↪': 'Bar',
                                                                          'old_value
↪': 'Foo'}}}]
```

Example-2

This example uses `iterate` mode to produce a list of compare results for each item in `input_after` results

Template:

```
<input name="input_before" load="text">
interface FastEthernet1/0/1
 description Foo
!
</input>

<input name="input_after" load="text">
interface FastEthernet1/0/1
 description FooBar
!
</input>

<input name="input_after" load="text">
interface FastEthernet1/0/2
 description Bar
```

(continues on next page)

(continued from previous page)

```
!
</input>

<group
name="interfaces*">
interface {{ interface }}
 description {{ description }}
</group>

<output deepdiff="input_before, input_after, add_field=difference, mode=iterate,␣
↪ignore_order=False, verbose_level=2"/>
```

Results:

```
[   [   {   'interfaces': [   {   'description': 'Foo',
                                  'interface': 'FastEthernet1/0/1'}]},
        {   'interfaces': [   {   'description': 'FooBar',
                                  'interface': 'FastEthernet1/0/1'}]},
        {   'interfaces': [   {   'description': 'Bar',
                                  'interface': 'FastEthernet1/0/2'}]},
        {   'difference': [   {   'values_changed': {   "root['interfaces'][0][
↪'description']": {   'new_value': 'FooBar',
                                                                                ␣
↪               'old_value': 'Foo'}}},
                              {   'values_changed': {   "root['interfaces'][0][
↪'description']": {   'new_value': 'Bar',
                                                                                ␣
↪               'old_value': 'Foo'},
                                                        "root['interfaces'][0][
↪'interface']": {   'new_value': 'FastEthernet1/0/2',
                                                                                ␣
↪             'old_value': 'FastEthernet1/0/1'}}}]}]]
```

Each item input_after compared against input_before, producing difference results accordingly.

Example-3

In this example we going to demonstrate how to use another template results to run deepdiff comparison with.

Template:

```
<template name="data_before" results="per_template">
<input load="text">
switch-1#show run int
interface Vlan778
 ip address 1.1.1.1/24
</input>

<input load="text">
switch-2#show run int
interface Vlan779
 ip address 2.2.2.1/24
</input>

<vars>
hostname="gethostname"
</vars>
```

(continues on next page)

(continued from previous page)

```
<group name="{{ hostname }}.interfaces.{{ interface }}">
interface {{ interface }}
 ip address {{ ip }}
</group>
</template>

<template name="data_after" results="per_template">
<input load="text">
switch-1#show run int
interface Vlan778
 ip address 1.1.1.2/24
</input>

<input load="text">
switch-2#show run int
interface Vlan779
 ip address 2.2.2.2/24
</input>

<vars>
hostname="gethostname"
</vars>

<group name="{{ hostname }}.interfaces.{{ interface }}">
interface {{ interface }}
 ip address {{ ip }}
</group>

<output deepdiff="template_before=data_before, add_field=difference"/>
</template>
```

Results:

```
[   [   {   'switch-1': {'interfaces': {'Vlan778': {'ip': '1.1.1.1/24'}}},
            'switch-2': {'interfaces': {'Vlan779': {'ip': '2.2.2.1/24'}}}}],
    [   {   'switch-1': {'interfaces': {'Vlan778': {'ip': '1.1.1.2/24'}}},
            'switch-2': {'interfaces': {'Vlan779': {'ip': '2.2.2.2/24'}}}},
        {   'difference': {   'values_changed': {   "root[0]['switch-1']['interfaces
↪']['Vlan778']['ip']": {   'new_value': '1.1.1.2/24',
                                                                                    ⌋
↪                          'old_value': '1.1.1.1/24'},
                                                  "root[0]['switch-2']['interfaces
↪']['Vlan779']['ip']": {   'new_value': '2.2.2.2/24',
                                                                                    ⌋
↪                          'old_value': '2.2.2.1/24'}}}}]]
```

Above output contains results for both templates, in addition to that second template results contain item with **difference** dictionary, that outline values changed between inputs of two different templates

8.1.3 Formatters

TTP supports a number of output formatters.

Name	Description
raw	default formatter, results returned as is
yaml	results transformed in a YAML structured, multi-line text
json	results transformed in a JSON structured, multi-line text
pprint	results transformed in a string using python pprint module
table	results transformed in a list of lists, each list representing table row
csv	uses table formatter results to emit CSV spreadsheet
tabulate	uses table formatter results to emit text table using tabulate module
excel	uses table formatter results to emit Excel table using openpyxl module
jinja2	renders Jinja2 template with parsing results
N2G	produces xml structured diagram using N2G module

Formatters can accept various attributes to supply additional information or modify behavior.

In general case formatters take python structured data - dictionary, list, list of dictionaries etc. - as an input, format that data in certain way and return new representation of results.

raw

If format is raw, no formatting will be applied and native python structure will be returned, results will not be converted to string.

yaml

Prerequisites: Python PyYAML library need to be installed

This formatter will run results through PyYAML module to produce YAML structured results.

JSON

This formatter will run results through Python built-in JSON module `dumps` method to produce *JSON (JavaScript Object Notation) <http://json.org>* structured results.

Note: json.dumps() will have these additional attributes set `sort_keys=True, indent=4, separators=(',', ': ')`

---

pprint

As the name implies, python built-in pprint module will be used to structure python data in a more readable.

table

This formatter will transform results into a list of lists, where first list item will represent table headers, all the rest of items will represent table rows.

For table formatter to work correctly, results data should have certain structure, namely:

- list of flat dictionaries
- single flat dictionary

- dictionary of flat dictionaries if `key` attribute provided

Flat dictionary - such a dictionary where all values are strings. It is not a limitation and in fact dictionary values can be of any structure, but they will be placed in table as is.

Supported formatter arguments

- `path` dot separated string to results that table formatter should use
- `headers` comma separated string of tab table headers, headers put randomly otherwise
- `missing` value to use to substitute empty cells in table, default is empty string - “”
- `key` key name to transform dictionary data to list of dictionaries

---

Note: csv, excel and tabulate formatter use table formatter to construct a table structure. As a result all attributes supported by table formatter, inherently supported by csv, excel and tabulate formatters.

Example-1

Template:

```
<input load="text">
interface Loopback0
 ip address 192.168.0.113/24
!
interface Vlan778
 ip address 2002::fd37/124
!
</input>

<input load="text">
interface Loopback10
 ip address 192.168.0.10/24
!
interface Vlan710
 ip address 2002::fd10/124
!
</input>

<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>

<output format="table"/>
```

Results:

```
[[['interface', 'ip', 'mask'],
  ['Loopback0', '192.168.0.113', '24'],
  ['Vlan778', '2002::fd37', '124'],
  ['Loopback10', '192.168.0.10', '24'],
  ['Vlan710', '2002::fd10', '124']]]
```

Example-2

This example is to demonstrate usage of `key` and other attributes

Template:

```
<input load="text">
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Loopback1
 description Router-id-loopback
 ip address 192.168.0.1/24
!
interface Vlan778
 ip address 2002::fd37/124
 ip vrf CPE1
!
interface Vlan779
 ip address 2002::bbcd/124
 ip vrf CPE2
!
</input>
<group name="interfaces**.{{ interface }}">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<output
path="interfaces"
format="table"
headers="intf, ip, mask, vrf, description, switchport"
key="intf"
missing="Undefined"
/>
```

Results:

```
[[['intf', 'ip', 'mask', 'vrf', 'description', 'switchport'],
  ['Loopback0', '192.168.0.113', '24', 'Undefined', 'Router-id-loopback', 'Undefined
↪'],
  ['Loopback1', '192.168.0.1', '24', 'Undefined', 'Router-id-loopback', 'Undefined'],
  ['Vlan778', '2002::fd37', '124', 'CPE1', 'Undefined', 'Undefined'],
  ['Vlan779', '2002::bbcd', '124', 'CPE2', 'Undefined', 'Undefined']]]
```

Above template produces this structure:

```
[[{'interfaces': {'Loopback0': {'description': 'Router-id-loopback',
                                'ip': '192.168.0.113',
                                'mask': '24'},
                  'Loopback1': {'description': 'Router-id-loopback',
                                'ip': '192.168.0.1',
                                'mask': '24'},
                  'Vlan778': {'ip': '2002::fd37', 'mask': '124', 'vrf': 'CPE1'},
                  'Vlan779': {'ip': '2002::bbcd', 'mask': '124', 'vrf': 'CPE2'}}}]]
```

`key` attribute instructs TTP to use *intf* as a name for *interfaces* dictionary keys while transforming it to a list of dictionaries.

csv

This formatter takes parsing result as an input, transforms it in list of lists using table formatter and emits csv structured table.

Supported formatter arguments

- `sep` separator character to use for csv formatter, default value is comma `,`
- `quote` quote character to use for csv formatter, default value is double quote `"`
- `path` dot separated string to results that csv formatter should use
- `headers` comma separated string of tab table headers, headers put randomly otherwise
- `missing` value to use to substitute empty cells in table, default is empty string - `""`
- `key` key name to transform dictionary data to list of dictionaries

Example

Template:

```
<input load="text">
interface Loopback0
 ip address 192.168.0.113/24
!
interface Vlan778
 ip address 2002::fd37/124
!
</input>

<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>

<output format="csv" returner="terminal"/>
```

Results:

```
interface,ip,mask
Loopback0,192.168.0.113,24
Vlan778,2002::fd37,124
```

tabulate

Prerequisites: tabulate module need to be installed on the system.

Tabulate formatter uses python tabulate module to transform and emit results in a plain-text table.

Supported formatter arguments

- `path` dot separated string to results that tabulate formatter should use
- `headers` comma separated string of tab table headers, headers put randomly otherwise
- `missing` value to use to substitute empty cells in table, default is empty string - “”
- `key` key name to transform dictionary data to list of dictionaries
- `format_attributes` ***args, ***kwargs* to pass on to tabulate object

Example

Template:

```
<input load="text">
router bgp 65100
  neighbor 10.145.1.9
    description vic-mel-core1
  !
  neighbor 192.168.101.1
    description qld-bri-core1
</input>

<group name="bgp_config">
router bgp {{ bgp_as }}
  <group name="peers">
  neighbor {{ peer }}
    description {{ description  }}
  </group>
</group>

<output
name="out2"
path="bgp_config.peers"
format="tabulate"
returner="terminal"
format_attributes="tablefmt='fancy_grid'"
/>
```

Results printed to terminal screen:

```
╒═══════════════╤═══════════════╕
│ description   │ peer          │
╞═══════════════╪═══════════════╡
│ vic-mel-core1 │ 10.145.1.9    │
├───────────────┼───────────────┤
│ qld-bri-core1 │ 192.168.101.1 │
╘═══════════════╧═══════════════╛
```

jinja2

Prerequisites: Jinja2 module need to be installed on the system

This formatter allow to render parsing results with jinja2 template. Jinja2 template should be enclosed in output tag text data. Jinja2 templates can help to produce any text output out of parsing results.

Within jinja2, the whole parsing results passed in *_data_* variable, that variable can be referenced in template accordingly.

Example

Template:

```
<input load="text">
interface Loopback0
 ip address 192.168.0.113/24
!
interface Vlan778
```

(continues on next page)

(continued from previous page)

```
 ip address 2002::fd37/124
!
</input>

<input load="text">
interface Loopback10
 ip address 192.168.0.10/24
!
interface Vlan710
 ip address 2002::fd10/124
!
</input>

<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>

<output format="jinja2" returner="terminal">
{% for input_result in _data_ %}
{% for item in input_result %}
if_cfg id {{ item['interface'] }}
    ip address {{ item['ip'] }}
    subnet mask {{ item['mask'] }}
#
{% endfor %}
{% endfor %}
</output>
```

Results:

```
if_cfg id Loopback0
    ip address 192.168.0.113
    subnet mask 24
#
if_cfg id Vlan778
    ip address 2002::fd37
    subnet mask 124
#
if_cfg id Loopback10
    ip address 192.168.0.10
    subnet mask 24
#
if_cfg id Vlan710
    ip address 2002::fd10
    subnet mask 124
#
```

excel

Prerequisites: openpyxl module need to be installed on the system

This formatter takes table structure defined in output tag text and transforms parsing results into table on a per tab basis using *table* formatter, as a results all attributes supported by table formatter can be used in excel formatter as well.

Supported formatter arguments

- `table` list of dictionaries describing excel table structure

Each dictionary item in `table` structure can have these attributes:

- `path` dot separated string to results that excel formatter should use
- `tab_name` name of this tab in excel spreadsheet, by default tab names are "Sheet<number>"
- `headers` comma separated string of tab table headers, headers put randomly otherwise
- `missing` value to use to substitute empty cells in table, default is empty string - ""
- `key` key name to transform dictionary data to list of dictionaries

Example

Template:

```
<input load="text">
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Loopback1
 description Router-id-loopback
 ip address 192.168.0.1/24
!
interface Vlan778
 ip address 2002::fd37/124
 ip vrf CPE1
!
interface Vlan779
 ip address 2002::bbcd/124
 ip vrf CPE2
!
</input>

<group name="loopbacks**.{{ interface }}">
interface {{ interface | contains("Loop") }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<group name="vlans*">
interface {{ interface | contains("Vlan") }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<output
format="excel"
returner="file"
filename="excel_out_%Y-%m-%d_%H-%M-%S.xslx"
url="./Output/"
load="yaml"
>
table:
  - headers: interface, ip, mask, vrf, description
    path: loopbacks
```

(continues on next page)

(continued from previous page)

```
    key: interface
    tab_name: loopbacks
  - path: vlans
</output>
```

TTP will produce excel table with two tabs using results from different groups. Table will be saved under *./Output/* path in *excel_out_%Y-%m-%d_%H-%M-%S.xslx* file.

N2G

Prerequisites: N2G module need to be installed on the system

N2G takes structured data and transforms it into xml format supported by a number of diagram editors.

Supported formatter arguments

- `path` dot separated string to results that N2G formatter should use to produce XML diagram.
- `module` name of N2G diagramming module to use - `yed` or `drawio`
- `node_dups` what to do with node duplicates - `skip` (default), `log`, `update`
- `link_dups` what to do with link duplicates - `skip` (default), `log`, `update`
- `method` name of N2G method to load data - `from_list` (default), `from_dict`, `from_csv`
- `method_kwargs` keyword arguments dictionary to pass to `method`
- `algo` name of layout algorithm to use for diagram

Example

In this example data from `show cdp neighbors detail` command output parsed in a list of dictionaries and fed into N2G to produce diagram in yED graphml format.

Template:

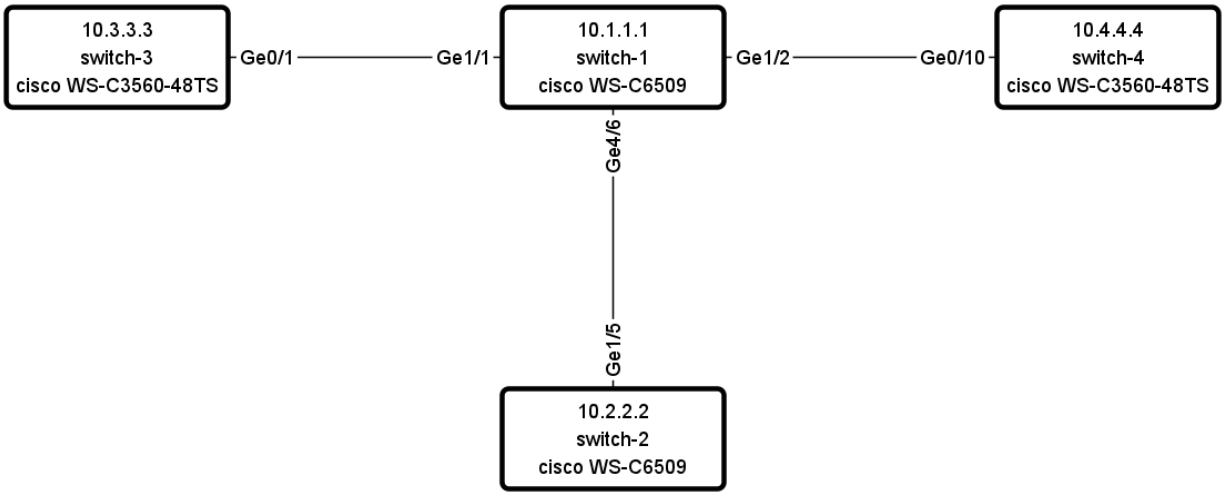
```
<input load="text">
switch-1#show cdp neighbors detail
-------------------------
Device ID: switch-2
Entry address(es):
  IP address: 10.2.2.2
Platform: cisco WS-C6509,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet4/6,  Port ID (outgoing port): GigabitEthernet1/5

-------------------------
Device ID: switch-3
Entry address(es):
  IP address: 10.3.3.3
Platform: cisco WS-C3560-48TS,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/1,  Port ID (outgoing port): GigabitEthernet0/1

-------------------------
Device ID: switch-4
Entry address(es):
  IP address: 10.4.4.4
Platform: cisco WS-C3560-48TS,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/2,  Port ID (outgoing port): GigabitEthernet0/10
</input>
```

(continues on next page)

(continued from previous page)

```
<input load="text">
switch-2#show cdp neighbors detail
-------------------------
Device ID: switch-1
Entry address(es):
  IP address: 10.1.1.1
Platform: cisco WS-C6509,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/5,  Port ID (outgoing port): GigabitEthernet4/6
</input>

<vars>
hostname='gethostname'
IfsNormalize = {
    'Ge':['^GigabitEthernet']
}
</vars>

<group name="cdp*" expand="">
Device ID: {{ target.id }}
  IP address: {{ target.top_label }}
Platform: {{ target.bottom_label | ORPHRASE }},  Capabilities: {{ ignore(ORPHRASE) }}
Interface: {{ src_label | resuball(IfsNormalize) }},  Port ID (outgoing port): {{␣
↪trgt_label | ORPHRASE | resuball(IfsNormalize) }}
{{ source | set("hostname") }}
</group>

<output format="n2g" load="python">
path = "cdp"
module = "yed"
node_duplicates = "update"
method = "from_list"
algo = "kk"
</output>

<out returner="file" url="./Output/" filename="cdp_diagram.graphml"/>
```

Results will be saved in *./Output/cdp_diagram.graphml* file and after editing diagram might look like this:



8.1.4 Returners

TTP has *file*, *terminal* and *self* returners. The purpose of returner is to return or emit or save data to certain destination.

Returner	Description
self	return result to calling function
file	save results to file
terminal	print results to terminal screen
syslog	send results over UDP to Syslog server

self

Default returner, data processed by output returned back to ttp for further processing, that way outputs can be chained to produce required results. Another use case is when ttp used as a module, results can be formatted retrieved out of ttp object.

file

Results will be saved to text file on local file system. One file will be produced per template to contain all the results for all the inputs and groups of this template.

Supported returner attributes

- `url` OS path to folder where file should be stored
- `filename` name of the file, can contain these time formatter:

```
* ``%m``  Month as a decimal number [01,12].
* ``%d``  Day of the month as a decimal number [01,31].
* ``%H``  Hour (24-hour clock) as a decimal number [00,23].
* ``%M``  Minute as a decimal number [00,59].
* ``%S``  Second as a decimal number [00,61].
* ``%z``  Time zone offset from UTC.
* ``%a``  Locale's abbreviated weekday name.
* ``%A``  Locale's full weekday name.
* ``%b``  Locale's abbreviated month name.
* ``%B``  Locale's full month name.
* ``%c``  Locale's appropriate date and time representation.
* ``%I``  Hour (12-hour clock) as a decimal number [01,12].
* ``%p``  Locale's equivalent of either AM or PM.
```

For instance, filename="OUT_%Y-%m-%d_%H-%M-%S_results.txt" will be rendered to "OUT_2019-09-09_18-19-58_results.txt" filename. By default filename is set to "output_<ctime>.txt", where "ctime" is a string produced after rendering "%Y-%m-%d_%H-%M-%S" by python time.strftime() function.

terminal

Results will be printed to terminal window. Terminal returner support colouring output using colorama module

Supported returner attributes

- `colour` if present with any value, colorama module will be initiated to colour certain words in output
- `red_words` comma separated list of patterns to colour in red, default is *False,No,Failed,Error,Failure,Fail,false,no,failed,error,failure,fail*

- `green_words` comma separated list of patterns to colour in green, default is *True,Yes,Success,Ok,true,yes,success,ok*
- `yeallow_words` comma separated list of patterns to colour in yellow, default is *Warning,warning*

Example

Template:

```
<input load="text">
interface Port-Channel11
  description Storage Management
interface Loopback0
  description RID
interface Vlan777
  description Management
</input>

<group>
interface {{ interface | contains("Port-Channel") }}
  description {{ description }}
  {{ is_lag | set(True) }}
  {{ is_loopback| set(False) }}
</group>

<group>
interface {{ interface | contains("Loop") }}
  description {{ description }}
  {{ is_lag | set(False) }}
  {{ is_loopback| set(True) }}
</group>

<output
returner="terminal"
colour=""
red="false,False"
green="true,True"
format="json"
/>
```

Results printed to screen:

syslog

This returner send result to remote Syslog servers over UDP using syslog handler from Python built-in loggin library.

Supported returner attributes

- `servers` list of servers to send logs to
- `port` UDP port servers listening on, default 514
- `facility` syslog facility number, default 77
- `path` path to parsing results emit to syslog
- `iterate` if set to True and parsing result is a list, iterates and send each item individually, default is *True*

Sample Template:

```
<input load="text">
router-2-lab#show ip arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.1.13.4              -   0050.5685.14d6  ARPA   GigabitEthernet3.13
Internet  10.1.13.5              -   0050.5685.14d7  ARPA   GigabitEthernet4.14
</input>

<input load="text">
router-3-lab#show ip arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.1.13.1             98   0050.5685.5cd1  ARPA   GigabitEthernet1.11
Internet  10.1.13.3              -   0050.5685.14d5  ARPA   GigabitEthernet2.12
</input>

<vars>hostname="gethostname"</vars>

<group name="arp_table*" method="table">
Internet  {{ ip }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface }}
Internet  {{ ip }}  -                   {{ mac }}  ARPA   {{ interface }}
{{ hostname | set(hostname) }}
</group>

<output returner="syslog" load="python">
servers="192.168.1.175"
port="10514"
path="arp_table"
iterate=True
facility=77
</output>
```

CHAPTER 9

Template Tag

TTP templates support <template> tag to define several templates within single file. Each template processed separately, no data shared between templates, but results of one template can be used by lookup functions in another template.

Only two levels of hierarchy supported - top template tag and a number of child template tags within it, further template tags nested within children are ignored.

First use case for this functionality stems from the fact that templates executed in sequence, meaning it is possible to use results produced by one template in next template(s), for instance first template can produce lookup table text file and other template will rely on.

Another use case is templates grouping under single definition to simplify loading - instead of adding each template to TTP object, all of them can be loaded in one go.

For instance:

```
from ttp import ttp

template1="""
<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>
"""

template2="""
<group name="vrfs">
VRF {{ vrf }}; default RD {{ rd }}
<group name="interfaces">
  Interfaces: {{ _start_ }}
    {{ intf_list | ROW }}
</group>
</group>
"""
```

(continues on next page)

(continued from previous page)

```
parser = ttp()
parser.add_data(some_data)
parser.add_template(template1)
parser.add_template(template2)
parser.parse()
```

Above code will produce same results as this code:

```
from ttp import ttp

template="""
<template>
<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
</group>
</template>

<template>
<group name="vrfs">
VRF {{ vrf }}; default RD {{ rd }}
<group name="interfaces">
  Interfaces: {{ _start_ }}
    {{ intf_list | ROW }}
</group>
</group>
</template>
"""

parser = ttp()
parser.add_data(some_data)
parser.add_template(template)
parser.parse()
```

9.1 Template tag attributes

There are a number of attributes supported by template tag. These attributes help to define template processing behavior.

Attribute	Description
name	Unique template identifier
base_path	Fully qualified OS path to data
results	Identifies the way results grouping method
pathchar	Character to use for group name-path processing

9.1.1 name

`name="template_name"`

Template name attribute is a string that indicates the unique name of the template. This attribute required if final results structure should be dictionary and not list (default behavior) as can be indicated in `ttp.result` method using `structure` argument, e.g.

Example

In below example results produced by TTP will be formed into dictionary structure using template names attributes as top level keys.

Consider this code:

```
from ttp import ttp
import json

template="""
<template name="template-1">
<input load="text">
interface Vlan778
 ip address 2002:fd37::91/124
</input>
<group name="interfaces-1">
interface {{ interface }}
 ip address {{ ip }}
</group>
</template>

<template name="template-2">
<input load="text">
interface Vlan778
 description V6 Management vlan
</input>
<group name="interfaces-2">
interface {{ interface }}
 description {{ description | ORPHRASE }}
</group>
</template>
"""

parser=ttp(template=template)
parser.parse()
results = parser.result(structure="dictionary")
print(json.dumps(results, sort_keys=True, indent=4, separators=(',', ': ')))
```

Results would be:

```
{
    "template-1": [
        {
            "interfaces-1": {
                "interface": "Vlan778",
                "ip": "2002:fd37::91/124"
            }
        }
    ],
    "template-2": [
        {
            "interfaces-2": {
                "description": "V6 Management vlan",
                "interface": "Vlan778"
            }
        }
    ]
}
```

9.1.2 base_path

`base_path="/os/base/path/to/data/"`

This attributes allows to specify base OS file system path to the location of data folders, folders with actual data can be detailed further using relative path in inputs' url attribute.

Example

In below template base_path attribute set to `/path/to/Data/`, as a result all urls for all inputs within this template will be extended to absolute path in such a way that:

- Input dataset-1 url `/data-1/` will become `/path/to/Data/data-1/`
- Input dataset-2 url `/data-2/` will become `/path/to/Data/data-2/`

Absolute path will be used to load data for each input.

Template:

```
<template base_path="/path/to/Data/">

<input name="dataset-1">
url = "/data-1/"
</input>

<input name="dataset-2">
url = "/data-2/"
</input>

<group name="interfaces1" input="dataset-1">
interface {{ interface }}
 switchport access vlan {{ access_vlan }}
</group>

<group name="interfaces2" input="dataset-2">
interface {{ interface }}
  ip address {{ ip  }}/{{ mask }}
</group>

</template>
```

9.1.3 results

`results="per_template|per_input"`

Template results attribute allows to influence the logic used to combine template results, options are:

- per_input - default, allows to combine results on a per input basis. For instance, if we have two text files with data that needs to be parsed, first file will be parsed by a set of groups associated with this template, combining results in a structure, that will be appended to the list of overall template results. Same will happen with next file. As a result, for this particular template two result items will be produced, one for each file.
- per_template - allows to combine results on a per template basis. For instance, if we have two text files with data that needs to be parsed, first file will be parsed by a set of groups associated with this template, combining results in a structure, that structure will be used by TTP to merge with results produced by next file. As a result, for this particular template single results item will be produced, that item will contain merged results for all inputed files/datum.

Main usecase for per_template behavior is to combine results across all the inputs and produce structure that will be more flat and might be easier to work with in certain situations.

Example

In this template we have two templates defined, with same set of inputs/data and groups, but first template has per_input (default) logic, while second template was configured to use per_template behavior.

Template:

```
<template>
<input load="text">
interface Vlan778
 ip address 2002:fd37::91/124
interface Vlan800
 ip address 172.16.10.1/24
</input>

<input load="text">
interface Vlan779
 ip address 192.168.1.1/24
interface Vlan90
 ip address 192.168.90.1/24
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip }}
</group>
</template>


<template results="per_template">
<input load="text">
interface Vlan778
 ip address 2002:fd37::91/124
interface Vlan800
 ip address 172.16.10.1/24
</input>

<input load="text">
interface Vlan779
 ip address 192.168.1.1/24
interface Vlan90
 ip address 192.168.90.1/24
</input>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip }}
</group>
</template>
```

Results:

```
[
    [ <-------------------------------------------------first template results:
        {
            "interfaces": [
```

(continues on next page)

(continued from previous page)

```
                {
                    "interface": "Vlan778",
                    "ip": "2002:fd37::91/124"
                },
                {
                    "interface": "Vlan800",
                    "ip": "172.16.10.1/24"
                }
            ]
        },
        {
            "interfaces": [
                {
                    "interface": "Vlan779",
                    "ip": "192.168.1.1/24"
                },
                {
                    "interface": "Vlan90",
                    "ip": "192.168.90.1/24"
                }
            ]
        }
    ],
    [ <-----------------------------------------------second template results:
        {
            "interfaces": [
                {
                    "interface": "Vlan778",
                    "ip": "2002:fd37::91/124"
                },
                {
                    "interface": "Vlan800",
                    "ip": "172.16.10.1/24"
                },
                {
                    "interface": "Vlan779",
                    "ip": "192.168.1.1/24"
                },
                {
                    "interface": "Vlan90",
                    "ip": "192.168.90.1/24"
                }
            ]
        }
    ]
]
```

9.1.4 pathchar

`pathchar="."`

At the moment this argument behavior is not fully implemented/tested, hence refrain from using it.

pathchar allows to specify character to use to separate path items for groups name attribute, by default it is dot character.

CHAPTER 10

Template Variables

TTP supports definition of arbitrary variables using dedicated xml tags <v>, <vars> or <variables>. Withing these tags variables can be defined in various formats and loaded using one of supported loaders. Variables can also be defined in external text files and loaded using *include* attribute.

Various values can be recorded in template variables before, during or after parsing. That additional data can be added to results, used for dynamic path constructions.

10.1 Inputs reference

10.1.1 Attributes

Attribute	Description
name	String of dot-separated path items
load	Indicates which loader to use to read tag data, default is *python*
include	Specifies location of the file with variables data to load
key	If csv loader used, *key* specifies column name to use as a key

load

`load="loader_name"`

- loader_name (optional) - name of the loader to use to render supplied variables data, default is python.

Supported loaders:

- python - uses python *exec* method to load data structured in native Python formats
- yaml - relies on PyYAML to load YAML structured data
- json - used to load json formatted variables data
- ini - *configparser* Python standart module used to read variables from ini structured file

- csv - csv formatted data loaded with Python *csv* standart library module

Example

Template

```
<input load="text">
interface GigabitEthernet1/1
 ip address 192.168.123.1 255.255.255.0
!
</input>

<!--Python formatted variables data-->
<vars name="vars">
python_domains = ['.lab.local', '.static.on.net', '.abc']
</vars>

<!--YAML formatted variables data-->
<vars load="yaml" name="vars">
yaml_domains:
  - '.lab.local'
  - '.static.on.net'
  - '.abc'
</vars>

<!--Json formatted variables data-->
<vars load="json" name="vars">
{
    "json_domains": [
        ".lab.local",
        ".static.on.net",
        ".abc"
    ]
}
</vars>

<!--INI formatted variables data-->
<variables load="ini" name="vars">
[ini_domains]
1: '.lab.local'
2: '.static.on.net'
3: '.abc'
</variables>

<!--CSV formatted variables data-->
<variables load="csv" name="vars.csv">
id, domain
1,  .lab.local
2,  .static.on.net
3,  .abc
</variables>

<group name="interfaces">
interface {{ interface }}
 ip address {{ ip }} {{ mask }}
</group>
```

Result as displayed by Python pprint outputter

YAML, JSON and Python formats are suitalble for encoding any arbitrary data and loaded as is.

INI structured data loaded into python nested dictionary, where top level keys represent ini section names each with nested dictionary of variables.

CSV data also transformed into dictionary using first column values to fill in dictionary keys, unless specified otherwise using *key* attribute

include

`include="path"`

- path - absolute OS path to text file with variables data.

name

`name="variables_tag_name"`

- variables_tag_name - dot separated string that specifies path in results structure where variables should be saved, by default it is empty, meaning variables will not be saved in results. Path string follows all the same rules as for group name attribute, for instance *{{ var_name }}* can be used to dynamically form path or "*" and "**" can indicate what type of structure to use for child - list or dictionary.

Example

Template

```
<vars name="vars.info**.{{ hostname }}">
# path will be formaed dynamically
hostname='switch-1'
serial='AS4FCVG456'
model='WS-3560-PS'
</vars>

<vars name="vars.ip*">
# variables that will be saved under {'vars': {'ip': []}} path
IP="Undefined"
MASK="255.255.255.255"
</vars>

<vars load="yaml">
# set of vars in yaml format that will not be included in results
intf_mode: "layer3"
</vars>

<input load="text">
interface Vlan777
 description Management
 ip address 192.168.0.1 24
 vrf MGMT
!
</input>

<group name="interfaces">
interface {{ interface }}
 description {{ description }}
 ip address {{ ip | record("IP") }} {{ mask }}
 vrf {{ vrf }}
 {{ mode | set("intf_mode") }}
</group>
```

Result

key

`key="column_name"`

- column_name - optional string attribute that can be used by csv loader to use given column values as a key for dictionary constructed out of csv data.

10.1.2 Getters

TTP template variables also support a number of getters - functions targeted to get some information and assign it to variable. Getters called for each input datum.

Function	Description
gethostname	this function tries to extract hostname out of source data prompts
getfilename	returns a name of the source data
get_time	returns current time
get_date	returns current date
get_timestamp	returns combination of current date and time
get_timestamp_ms	returns combination of current date and time with milliseconds
get_timestamp_iso	returns timestamp in ISO format in UTC timezone
get_time_ns	returns current time in nanoseconds since Epoch

gethostname

`var_name="gethostname"`

Using this getter function TTP tries to extract device's hostname out of it prompt.

Supported prompts are:

- Juniper such as `some.user@hostname>`
- Huawei such as `<hostname>`
- Cisco IOS Exec such as `hostname>`
- Cisco IOS XR such as `RP/0/4/CPU0:hostname#`
- Cisco IOS Priviledged such as `hostname#`
- Fortigate such as `hostname (context) #`

Example

Template:

```
<input load="text">
switch1#show run int
interface GigabitEthernet3/11
 description input_1_data
</input>

<vars name="vars">
hostname_var = "gethostname"
```

(continues on next page)

(continued from previous page)

```
</vars>

<group name="interfaces">
interface {{ interface }}
 description {{ description }}
</group>
```

Result:

```
[
    {
        "interfaces": {
            "description": "input_1_data",
            "interface": "GigabitEthernet3/11"
        },
        "vars": {
            "hostname_var": "switch1"
        }
    }
]
```

getfilename

`var_name="getfilename"`

This function returns the name of input data file if data was loaded from file, if data was loaded from text it will return "text_data".

get_time

`var_name="get_time"`

Returns current time in `%H:%M:%S` format.

get_date

`var_name="get_date"`

Returns current date in `%Y-%m-%d` format.

get_timestamp

`var_name="get_timestamp"`

Returns current timestamp in `%Y-%m-%d %H:%M:%S` format.

get_timestamp_ms

`var_name="get_timestamp_ms"`

Returns current timestamp but with milliseconds precision in a format of `%Y-%m-%d %H:%M:%S.%ms`

get_timestamp_iso

```
var_name="get_timestamp_iso"
```

Returns current timestamp in ISO format with UTC timezone e.g. `2020-06-30T11:07:01.212349+00:00`. Uses python datetime function to produce timestamp.

get_time_ns

```
var_name="get_time_ns"
```

This function uses time.time_ns method to return current time in nanoseconds since Epoch

CHAPTER 11

Lookup Tables

Lookups tag allows to define a lookup table that will be transformed into lookup dictionary, dictionary that can be used to lookup values to include them into parsing results. Lookup table can be called from match variable using *lookup* function.

Table 1: lookup tag attributes

Name	Description
name	name of the lookup table to reference in match variable *lookup* function
load	name of the loader to use to load lookup text
include	specifies location of the file to load lookup table from
key	If csv loader used, *key* specifies column name to use as a key
database	Name of database loader touse to load lookup data

11.1 name

```
name="lookup_table_name"
```

- lookup_table_name(mandatory) - string to use as a name for lookup table, that is required attribute without it lookup data will not be loaded.

11.2 load

```
load="loader_name"
```

- loader_name (optional) - name of the loader to use to render supplied variables data, default is python.

Supported loaders:

- python - uses python *exec* method to load data structured in native Python formats
- yaml - relies on PyYAML to load YAML structured data

- json - used to load json formatted variables data
- ini - *configparser* Python standard module used to read variables from ini structured file
- csv - csv formatted data loaded with Python *csv* standard library module

If load is csv, first column by default will be used to create lookup dictionary, it is possible to supply *key* with column name that should be used as a keys for row data. If any other type of load provided e.g. python or yaml, that data must have a dictionary structure, there keys will be compared against match result and on success data associated with given key will be included in results.

11.3 include

```
include="path"
```

- path - absolute OS path to text file with lookup table data.

11.4 key

```
key="column_name"
```

- column_name - optional string attribute that can be used by csv loader to use given column values as a key for dictionary constructed out of csv data.

11.5 CSV Example

Template:

```
<lookup name="aux_csv" load="csv">
ASN,as_name,as_description,prefix_num
65100,Subs,Private ASN,734
65200,Privs,Undef ASN,121
</lookup>

<input load="text">
router bgp 65100
</input>

<group name="bgp_config">
router bgp {{ bgp_as | lookup("aux_csv", add_field="as_details") }}
</group>
```

Result:

```
[
    {
        "bgp_config": {
            "as_details": {
                "as_description": "Private ASN",
                "as_name": "Subs",
                "prefix_num": "734"
            },
            "bgp_as": "65100"
```

(continues on next page)

(continued from previous page)

```
        }
    }
]
```

Because no *key* attribute provided, csv data was loaded in python dictionary using first column - ASN - as a key. This is the resulted lookup dictionary:

```
{
  "65100": {
        "as_name": "Subs",
        "as_description" : "Private ASN",
        "prefix_num": "734"
    },
  "65200": {
        "as_name": "Privs",
        "as_description" : "Undef ASN",
        "prefix_num": "121"
    }
}
```

If *key* will be set to "as_name", lookup dictionary will become:

```
{
  "Subs": {
        "ASN": "65100",
        "as_description" : "Private ASN",
        "prefix_num": "734"
    },
  "Privs": {
        "ASN": "65200",
        "as_description" : "Undef ASN",
        "prefix_num": "121"
    }
}
```

11.6 INI Example

If table provided in INI format, data will be transformed into dictionary with top key equal to lookup table names, next level of keys will correspond to INI sections which will nest a dictionary of actual key-value pairs. For instance in below template with lookup name "location", INI data will be loaded into this python dictionary structure:

```
{ "locations":
    { "cities": {
        "-mel-": "7 Name St, Suburb A, Melbourne, Postal Code",
        "-bri-" : "8 Name St, Suburb B, Brisbane, Postal Code"
    }
}}
```

As a result dictionary data to use for lookup can be referenced using "locations.cities" string in lookup/rlookup match variables function.

Template:

```
<input load="text">
router bgp 65100
  neighbor 10.145.1.9
    description vic-mel-core1
  !
  neighbor 192.168.101.1
    description qld-bri-core1
</input>

<lookup name="locations" load="ini">
[cities]
-mel- : 7 Name St, Suburb A, Melbourne, Postal Code
-bri- : 8 Name St, Suburb B, Brisbane, Postal Code
</lookup>

<group name="bgp_config">
router bgp {{ bgp_as }}
 <group name="peers">
  neighbor {{ peer }}
    description {{ description | rlookup('locations.cities', add_field='location') }}
 </group>
</group>
```

Result:

```
[
    {
        "bgp_config": {
            "bgp_as": "65100",
            "peers": [
                {
                    "description": "vic-mel-core1",
                    "location": "7 Name St, Suburb A, Melbourne, Postal Code",
                    "peer": "10.145.1.9"
                },
                {
                    "description": "qld-bri-core1",
                    "location": "8 Name St, Suburb B, Brisbane, Postal Code",
                    "peer": "192.168.101.1"
                }
            ]
        }
    }
]
```

11.7 YAML Example

YAML data must be structured as a dictionary, once loaded it will correspond to python dictionary that will be used to lookup values.

Template:

```
<lookup name="yaml_look" load="yaml">
'65100':
  as_description: Private ASN
```

(continues on next page)

(continued from previous page)

```
  as_name: Subs
  prefix_num: '734'
'65101':
  as_description: Cust-1 ASN
  as_name: Cust1
  prefix_num: '156'
</lookup>

<input load="text">
router bgp 65100
</input>

<group name="bgp_config">
router bgp {{ bgp_as | lookup("yaml_look", add_field="as_details") }}
</group>
```

Result:

```
[
    {
        "bgp_config": {
            "as_details": {
                "as_description": "Private ASN",
                "as_name": "Subs",
                "prefix_num": "734"
            },
            "bgp_as": "65100"
        }
    }
]
```

11.8 database

`database="db name"`

Name of database to use to populate lookup data.

Below is a list of supported databases

11.8.1 geoip2 database

Loads GeoIP2 .mmdb files to use with match variable "geoip_lookup" function. Supports City, ASN and Country databases. Databases can be found at MaxMind website.

Example

Sample template lookup tag to define geoip2 .mmdb files location:

```
<lookup name="geoip2_test" database="geoip2">
citY    = 'C:/path/to/GeoLite2-City.mmdb'
AsN     = 'C:/path/to/GeoLite2-ASN.mmdb'
Country = 'C:/path/to/GeoLite2-Country.mmdb'
</lookup>
```

To correctly load databases TTP expects “City”, “ASN”, “Country” arguments to be defined within lookup tag data, argument names are not case sensetive, each argument should contain OS patch to respective databse file.

Above example contains Python formatted data, but it can be YAML or JSON as well, for instance YAML formatted data:

```
<lookup name="geoip2_test" database="geoip2" load="YAML">
citY: 'C:/path/to/GeoLite2-City.mmdb'
AsN: 'C:/path/to/GeoLite2-ASN.mmdb'
Country: 'C:/path/to/GeoLite2-Country.mmdb'
</lookup>
```

CHAPTER 12

Macro Tag

TTP has a number of built-in function for various systems - function for groups, functions for outputs, functions for variables and functions for match variables. To extend this functionality even further, TTP allows to define custom functions using <macro> tags.

Macro is a python code within <macro> tag text. This code can contain a number of function definitions, these functions can be referenced within TTP templates.

Warning: Python exec function used to load macro code, as a result it is unsafe to use templates from untrusted sources, as code within macro tag will be executed on template load.

For further details check:

- Match variables *macro*
- Groups *macro*
- Outputs *macro*
- Inputs *macro*

TTP internally uses `_ttp_` dictionary to contain reference to all groups, inputs, outputs, match variables and getter functions. That dictionary injected in global space of macro function and can be used to call TTP functions.

CHAPTER 13

Doc Tag

TBD

CHAPTER 14

Writing templates

Writing templates is simple.

To create template, take data that needs to be parsed and replace portions of it with match variables:

```
# Data we want to parse
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Vlan778
 description CPE_Acces_Vlan
 ip address 2002::fd37/124
 ip vrf CPE1
!

# TTP template
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
```

Above data and template can be saved in two files, and ttp CLI tool can be used to parse it with command:

```
ttp -d "/path/to/data/file.txt" -t "/path/to/template.txt" --outputter json
```

And get these results:

```
[
    [
        {
            "description": "Router-id-loopback",
            "interface": "Loopback0",
            "ip": "192.168.0.113",
            "mask": "24"
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "description": "CPE_Acces_Vlan",
            "interface": "Vlan778",
            "ip": "2002::fd37",
            "mask": "124",
            "vrf": "CPE1"
        }
    ]
]
```

Warning: TTP match variables names used as regular expressions group names, hence they must be valid Python identifiers.

Above process is very similar to writing Jinja2 templates but in reverse direction - we have text and we need to transform it into structured data, as opposed to having structured data, that needs to be rendered with Jinja2 template to produce text.

Warning: Indentation is important. Trailing spaces and tabs are ignored by TTP.

TTP use leading spaces and tabs to produce better match results, exact number of leading spaces and tabs used to form regular expressions. There is a way to ignore indentation by the use of *ignore* indicator coupled with `[\s\t]*` or `\s+` or `\s{1,3}` or `\t+` etc. regular expressions.

TTP supports various output formats, for instance, if we need to emit data not in json but csv format we can use outputter and write this template:

```
<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<output format="csv" returner="terminal"/>
```

Run ttp CLI tool without -o option to print only results produced by outputter defined within template:

```
ttp -d "/path/to/data/file.txt" -t "/path/to/template.txt"
```

We told TTP that `returner="terminal"`, because of that results will be printed to terminal screen:

```
description,interface,ip,mask,vrf
Router-id-loopback,Loopback0,192.168.0.113,24,
CPE_Acces_Vlan,Vlan778,2002::fd37,124,CPE1
```

14.1 XML Primer

TBD

14.2 HOW TOs

14.2.1 How to parse hierarchical configuration data

TTP can use simple templates that does not contain much hierarchy (same as the data that parsed by them), but what to do if we want to extract information from below text:

```
router bgp 12.34
 address-family ipv4 unicast
  router-id 1.1.1.1
 !
 vrf CT2S2
  rd 102:103
  !
  neighbor 10.1.102.102
   remote-as 102.103
   address-family ipv4 unicast
    send-community-ebgp
    route-policy vCE102-link1.102 in
    route-policy vCE102-link1.102 out
   !
  !
  neighbor 10.2.102.102
   remote-as 102.103
   address-family ipv4 unicast
    route-policy vCE102-link2.102 in
    route-policy vCE102-link2.102 out
   !
  !
 vrf AS65000
  rd 102:104
  !
  neighbor 10.1.37.7
   remote-as 65000
   address-family ipv4 labeled-unicast
    route-policy PASS-ALL in
    route-policy PASS-ALL out
```

In such a case we have to use ttp groups to define nested, hierarchical structure, sample template might look like this:

```
<group name="bgp_cfg">
router bgp {{ ASN }}
 <group name="ipv4_afi">
 address-family ipv4 unicast {{ _start_ }}
  router-id {{ bgp_rid }}
 </group>

 <group name="vrfs">
 vrf {{ vrf }}
  rd {{ rd }}

  <group name="neighbors">
  neighbor {{ neighbor }}
   remote-as {{ neighbor_asn }}
   <group name="ipv4_afi">
   address-family ipv4 unicast {{ _start_ }}
    send-community-ebgp {{ send_community_ebgp | set("Enabled") }}
```

(continues on next page)

(continued from previous page)

```
    route-policy {{ RPL_IN }} in
    route-policy {{ RPL_OUT }} out
   </group>
  </group>
 </group>
</group>
```

Above data and template can be saved in two files and run using ttp CLI tool with command:

```
ttp -d "/path/to/data/file.txt" -t "/path/to/template.txt" --outputter yaml
```

These results will be printed to screen:

```
- bgp_cfg:
    ASN: '12.34'
    ipv4_afi:
      bgp_rid: 1.1.1.1
    vrfs:
    - neighbors:
      - ipv4_afi:
          RPL_IN: vCE102-link1.102
          RPL_OUT: vCE102-link1.102
          send_community_ebgp: Enabled
        neighbor: 10.1.102.102
        neighbor_asn: '102.103'
      - ipv4_afi:
          RPL_IN: vCE102-link2.102
          RPL_OUT: vCE102-link2.102
        neighbor: 10.2.102.102
        neighbor_asn: '102.103'
      rd: 102:103
      vrf: CT2S2
    - neighbors:
      - ipv4_afi:
          RPL_IN: PASS-ALL
          RPL_OUT: PASS-ALL
      - neighbor: 10.1.37.7
        neighbor_asn: '65000'
      rd: 102:104
      vrf: AS65000
```

Not too bad, but let's say we want VRFs to be represented as a dictionary with VRF names as keys, same goes for neighbors - we want them to be a dictionary with neighbor IPs as a key, we can use TTP dynamic path feature together with path formatters to accomplish exactly that, here is the template:

```
<group name="bgp_cfg">
router bgp {{ ASN }}
 <group name="ipv4_afi">
 address-family ipv4 unicast {{ _start_ }}
  router-id {{ bgp_rid }}
 </group>
 !
 <group name="vrfs.{{ vrf }}">
 vrf {{ vrf }}
  rd {{ rd }}
  !
```

(continues on next page)

(continued from previous page)

```
  <group name="peers.{{ neighbor }}**">
  neighbor {{ neighbor }}
   remote-as {{ neighbor_asn }}
   <group name="ipv4_afi">
   address-family ipv4 unicast {{ _start_ }}
    send-community-ebgp {{ send_community_ebgp | set("Enabled") }}
    route-policy {{ RPL_IN }} in
    route-policy {{ RPL_OUT }} out
   </group>
  </group>
 </group>
</group>
```

After parsing TTP will print these structure:

```
- bgp_cfg:
    ASN: '12.34'
    ipv4_afi:
      bgp_rid: 1.1.1.1
    vrfs:
      AS65000:
        peers:
          10.1.37.7:
            ipv4_afi:
              RPL_IN: PASS-ALL
              RPL_OUT: PASS-ALL
            neighbor_asn: '65000'
        rd: 102:104
      CT2S2:
        peers:
          10.1.102.102:
            ipv4_afi:
              RPL_IN: vCE102-link1.102
              RPL_OUT: vCE102-link1.102
              send_community_ebgp: Enabled
            neighbor_asn: '102.103'
          10.2.102.102:
            ipv4_afi:
              RPL_IN: vCE102-link2.102
              RPL_OUT: vCE102-link2.102
            neighbor_asn: '102.103'
        rd: 102:103
```

That's better, but what actually changed to have such a different results, well, not to much by the look of it, but quite a lot in fact.

TTP group's name attribute actually used as a path where to save group parsing results within results tree, to denote different levels dot symbol can be used, that is how we get new *vrf* and *peers* keys in the output.

In addition we used TTP dynamic path feature by introducing `{{ vrf }}` and `{{ neighbor }}` in the name of the group, that will be dynamically substituted with matching results.

Moreover, we also have to use double star `**` path formatter to tell TTP that `{{ neighbor }}` child content should be kept as a dictionary and not transformed into list (default behavior) whenever we add new data to that portion of results tree.

14.2.2 How to parse text tables

Parsing text tables is fairly simple as long as they are regular - meaning there are repetitive patterns can be found in text. For instance this text:

```
Protocol  Address      Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.1         98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.13.3        131   0150.7685.14d5  ARPA   GigabitEthernet2.13
Internet  10.12.13.4        198   0950.5C8A.5c41  ARPA   GigabitEthernet2.17
```

is a table and is easy to parse with TTP using this single pattern:

```
Internet  {{ ip | IP }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface }}
```

`IP` and `DIGIT` are regular expression formatters, indicating that special regexes need to be use to match ip and age variables. If we add additional entries in above text, that are different from existing ones, we will have to add more patterns in template and combine them in a group. For instance this text:

```
Protocol  Address      Age (min)  Hardware Addr   Type   Interface
Internet  10.12.13.1         98   0950.5785.5cd1  ARPA   FastEthernet2.13
Internet  10.12.13.3        131   0150.7685.14d5  ARPA   GigabitEthernet2.13
Internet  10.12.13.4        198   0950.5C8A.5c41  ARPA   GigabitEthernet2.17
Internet  10.12.14.5          -   0950.5C8A.5d42  ARPA   GigabitEthernet3
Internet  10.12.15.6        164   0950.5C8A.5e43  ARPA   GigabitEthernet4.21  *
```

would require two additional patterns to match all the lines:

```
<group name="table_data">
Internet  {{ ip | IP | _start_ }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface␣
↪}}
Internet  {{ ip | IP | _start_ }}  -                    {{ mac }}  ARPA   {{ interface␣
↪}}
Internet  {{ ip | IP | _start_ }}  {{ age | DIGIT }}   {{ mac }}  ARPA   {{ interface␣
↪}}  *
</group>
```

We also have to use _start_ indicator, as each line is a complete match and on each subsequent match we need to save previous matches in results. However, above template can be simplified a bit:

```
<group name="table_data" method="table">
Internet  {{ ip | IP }}  {{ age }}   {{ mac }}  ARPA   {{ interface }}
Internet  {{ ip | IP }}  {{ age }}   {{ mac }}  ARPA   {{ interface }}  *
</group>
```

Excluding DIGIT regex formatters will still allow to match all digits but will match hyphen symbol as well, in addition to that, TTP groups tag has `method` attribute, this attribute makes every pattern in a group to be group start regex without the need to specify _start_ explicitly. Parsing text table data with above template will produce these results:

```
[   [   {   'table_data': [   {   'age': '98',
                                  'interface': 'FastEthernet2.13',
                                  'ip': '10.12.13.1',
                                  'mac': '0950.5785.5cd1'},
                              {   'age': '131',
                                  'interface': 'GigabitEthernet2.13',
                                  'ip': '10.12.13.3',
                                  'mac': '0150.7685.14d5'},
                              {   'age': '198',
```

(continues on next page)

(continued from previous page)

```
                                    'interface': 'GigabitEthernet2.17',
                                    'ip': '10.12.13.4',
                                    'mac': '0950.5C8A.5c41'},
                                {   'age': '-',
                                    'interface': 'GigabitEthernet3',
                                    'ip': '10.12.14.5',
                                    'mac': '0950.5C8A.5d42'},
                                {   'age': '164',
                                    'interface': 'GigabitEthernet4.21',
                                    'ip': '10.12.15.6',
                                    'mac': '0950.5C8A.5e43'}]}]]
```

TTP can help parsing text tables data for one more specific usecase, for example this data:

```
VRF VRF-CUST-1 (VRF Id = 4); default RD 12345:241;
  Old CLI format, supports IPv4 only
  Flags: 0xC
  Interfaces:
    Te0/3/0.401             Te0/3/0.302             Te0/3/0.315
    Te0/3/0.316             Te0/3/0.327
```

has text table embedded into it, and if we want to extract all the interfaces that belongs to this particular VRF, we can use this template:

```
<group name="vrf.{{ vrf_name }}">
VRF {{ vrf_name }} (VRF Id = {{ vrf_id}}); default RD {{ vrf_rd }};
<group name="interfaces">
  Interfaces: {{ _start_ }}
    {{ interfaces | ROW | joinmatches(",") }}
</group>
</group>
```

In above temple `ROW` regex formatter will help to match all lines with words separated by 2 or more spaces between them, producing this results:

```
[
    [
        {
            "vrf": {
                "VRF-CUST-1": {
                    "interfaces": {
                        "interfaces": "Te0/3/0.401             Te0/3/0.302
↪   Te0/3/0.315 Te0/3/0.316             Te0/3/0.327"
                    },
                    "vrf_id": "4",
                    "vrf_rd": "12345:241"
                }
            }
        }
    ]
]
```

While TTP extracted all interfaces, they are combined in a single string, below template can be used to produce list of interfaces instead:

```
<group name="vrf.{{ vrf_name }}">
VRF {{ vrf_name }} (VRF Id = {{ vrf_id}}); default RD {{ vrf_rd }};
<group name="interfaces">
  Interfaces: {{ _start_ }}
    {{ interfaces | ROW | resub(" +", ",", 20) | split(',') | joinmatches }}
</group>
</group>
```

In this template same match result processed inline using `resub` function to replace all consequential occurrence of spaces with singe comma character, after substitution, results processing continues through `split` function, that split string into a list of items using comma character, finally, `joinmatches` function tells TTP to join all matches in single list, producing these results:

```
[
    [
        {
            "vrf": {
                "VRF-CUST-1": {
                    "interfaces": {
                        "interfaces": [
                            "Te0/3/0.401",
                            "Te0/3/0.302",
                            "Te0/3/0.315",
                            "Te0/3/0.316",
                            "Te0/3/0.327"
                        ]
                    },
                    "vrf_id": "4",
                    "vrf_rd": "12345:241"
                }
            }
        }
    ]
]
```

14.2.3 How to parse show commands output

Show commands output parsing with TTP is the same as parsing any text data that contains repetitive patterns and require a certain level of familiarity with tools built into TTP to not only parse but also process match results.

As a usecase let's consider parsing "show cdp neighbors detail" command output of Cisco IOS device, source data:

```
my_switch_1#show cdp neighbors detail
-------------------------
Device ID: switch-2.net
Entry address(es):
IP address: 10.251.1.49
Platform: cisco WS-C6509,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet4/6,  Port ID (outgoing port): GigabitEthernet1/5
Holdtime : 130 sec

Version :
Cisco Internetwork Operating System Software
IOS (tm) s72033_rp Software (s72033_rp-PK9SV-M), Version 12.2(17d)SXB11a, RELEASE␣
↪SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
```

(continues on next page)

(continued from previous page)

```
Copyright (c) 1986-2006 by cisco Systems, Inc.
Compiled Thu 13-Apr-06 04:50 by kehsiao

advertisement version: 2
VTP Management Domain: ''
Duplex: full
Unidirectional Mode: off

--------------------------
Device ID: switch-2
Entry address(es):
IP address: 10.151.28.7
Platform: cisco WS-C3560-48TS,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/1,  Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 165 sec

Version :
Cisco IOS Software, C3560 Software (C3560-IPBASE-M), Version 12.2(25)SEB2, RELEASE␣
↪SOFTWARE (fc1)
Copyright (c) 1986-2005 by Cisco Systems, Inc.
Compiled Tue 07-Jun-05 23:34 by yenanh

advertisement version: 2
Protocol Hello:  OUI=0x00000C, Protocol ID=0x0112; payload len=27,␣
↪value=00000000FFFFFFFF010221FF0000000000000000152BC02D80FF0000
VTP Management Domain: ''
Native VLAN: 500
Duplex: full
Unidirectional Mode: off
```

The goal is to get this results structure:

```
{
        local_hostname: str,
        local_interface: str,
        peer_hostname: str,
        peer_interface: str,
        peer_ip: str,
        peer_platform: str,
        peer_capabilities: [cap1, cap2],
        peer_software: str
}
```

Template to achieve this:

```
<vars>
hostname="gethostname"
</vars>

<group name="cdp_peers">
-------------------------- {{ _start_ }}
Device ID: {{ peer_hostname }}
IP address: {{ peer_ip }}
Platform: {{ peer_platform | ORPHRASE }},  Capabilities: {{ peer_capabilities |␣
↪ORPHRASE | split(" ") }}
Interface: {{ local_interface }},  Port ID (outgoing port): {{ peer_interface }}
{{ local_hostname | set("hostname") }}
```

(continues on next page)

(continued from previous page)

```
<group name="_">
Version : {{ _start_ }}
{{ peer_software | _line_ }}
{{ _end_ }}
</group>

</group>
```

Results:

```
[[[
    {
        "local_hostname": "my_switch_1",
        "local_interface": "GigabitEthernet4/6",
        "peer_capabilities": [
            "Router",
            "Switch",
            "IGMP"
        ],
        "peer_hostname": "switch-2.net",
        "peer_interface": "GigabitEthernet1/5",
        "peer_ip": "10.251.1.49",
        "peer_platform": "cisco WS-C6509",
        "peer_software": "Cisco Internetwork Operating System Software \nIOS (tm)␣
↪s72033_rp Software (s72033_rp-PK9SV-M), Version 12.2(17d)SXB11a, RELEASE SOFTWARE␣
↪(fc1)\nTechnical Support: http://www.cisco.com/techsupport\nCopyright (c) 1986-2006␣
↪by cisco Systems, Inc.\nCompiled Thu 13-Apr-06 04:50 by kehsiao"
    },
    {
        "local_hostname": "my_switch_1",
        "local_interface": "GigabitEthernet1/1",
        "peer_capabilities": [
            "Switch",
            "IGMP"
        ],
        "peer_hostname": "switch-2",
        "peer_interface": "GigabitEthernet0/1",
        "peer_ip": "10.151.28.7",
        "peer_platform": "cisco WS-C3560-48TS",
        "peer_software": "Cisco IOS Software, C3560 Software (C3560-IPBASE-M),␣
↪Version 12.2(25)SEB2, RELEASE SOFTWARE (fc1)\nCopyright (c) 1986-2005 by Cisco␣
↪Systems, Inc.\nCompiled Tue 07-Jun-05 23:34 by yenanh"
    }
]]]
```

Special attention should be paid to this aspects of above template:

- use of explicit _start_ indicator to define start of the group
- ORPHRASE regex formatter to match a single word and collection of words
- _line_ indicator used within separate group to combine software version description, that group has special null path - "_" - indicating that result for this group should be merged with parent group
- explicit use of _end_ indicator to make sure that only relevant information matched
- special handling of peer_capabilities match result by converting into list by splitting match result using space character

14.2.4 How to filter with TTP

14.2.5 How to produce time series data with TTP

Time stamped data is very easy to produce with TTP, as it has built-in time related functions, allowing to add timestamp to match results. For example, interface counters can be parsed with TTP every X number of seconds, marked with timestamp, producing simple time series data.

Consider this source data:

```
GigabitEthernet1 is up, line protocol is up
     297 packets input, 25963 bytes, 0 no buffer
     160 packets output, 26812 bytes, 0 underruns
GigabitEthernet2 is up, line protocol is up
     150 packets input, 2341 bytes, 0 no buffer
     351 output errors, 3459 collisions, 0 interface resets
```

And the goal is to get this result:

```
{
    timestamp: {
        interface: {
            in_pkts: int,
            out_pkts: int
            }
        }
    }
```

Template to produce above structure is:

```
<vars>
timestamp = "get_timestamp_ms"
</vars>

<group name = "{{ timestamp }}.{{ interface }}">
{{ interface }} is up, line protocol is up
     {{ in_pkts}} packets input, 25963 bytes, 0 no buffer
     {{ out_pkts }} packets output, 26812 bytes, 0 underruns
</group>
```

Results after parsing above data with template:

```
[
    [
        {
            "2019-11-10 16:18:32.523": {
                "GigabitEthernet1": {
                    "in_pkts": "297",
                    "out_pkts": "160"
                },
                "GigabitEthernet2": {
                    "in_pkts": "150"
                }
            }
        }
    ]
]
```

Attention should be paid to the fact, that timestamps produced using local time of the system that happens to parse text data, as a result get_time_ns function can be used to produce time in nanoseconds since the epoch (midnight, 1st of January, 1970) in UTC.

CHAPTER 15

CLI tool

TTP comes with simple CLI tool that takes path to data, path to template and produces parsing results. Results can be represented in one of formats supported by CLI tool - yaml, json, raw or pprint, results will be printer to screen. Alternatively, format can be specified using template output tags and printed to screen or returned to file using returners.

Sample usage:

```
ttp --data "/path/to/data/" --template "path/to/template.txt" --outputter json

results will be printed to screen in JSON format.
```

Available options

- `-d, --data` Path to data file or directory with files to process
- `-dp, --data-prefix` OS base path to folder with data separated across additional folders as specified in TTP input tags
- `-t, --template` Path to text file with template content
- `-tn, --template-name` Name of template within file referenced by -t option if file has python (.py) extension
- `-o, --outputter` Format results using yaml, json, raw or pprint formatter and prints them to terminal
- `-ot, --out-template` Name of template to output results for
- `-l, --logging` Logging level - “DEBUG”, “INFO”, “WARNING”, “ERROR”, “CRITICAL”
- `-lf, --log-file` OS path to file where to write logs instead of printing them to terminal
- `-T, --Timing` Print simple timing information to screen about time spent on parsing data
- `s, --structure` Final results structure - ‘list’ or ‘dictionary’
- `-v, --vars` Json string containing variables to add to TTP object
- `--one` Forcefully run parsing using single process
- `--multi` Forcefully run parsing in multiple processes

CHAPTER 16

API reference

API reference for TTP module.

class ttp.**ttp**(*data=''*, *template=''*, *log_level='WARNING'*, *log_file=None*, *base_path=''*, *vars={}*)

Template Text Parser main class to load data, templates, lookups, variables and dispatch data to parser object to parse in single or multiple processes, construct final results and run outputs.

Parameters

- `data` file object or OS path to text file or directory with text files with data to parse
- `template` file object or OS path to text file with template
- `base_path` (str) base OS path prefix to load data from for template's inputs
- `log_level` (str) level of logging "DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"
- `log_file` (str) path where to save log file
- `vars` dictionary of variables to make available to ttp parser

Example:

```
from ttp import ttp
parser = ttp(data="/os/path/to/data/dir/", template="/os/path/to/template.txt")
parser.parse()
result = parser.result(format="json")
print(result[0])
```

add_input(*data, input_name='Default_Input', template_name='_root_template_', groups=['all']*)

Method to load additional data to be parsed. Data associated with certain input of `input_name` and template of `template_name`.

> **Warning:** `add_input` should be called only after templates added

Parameters

- `data` file object or OS path to text file or directory with text files with data to parse
- `input_name` (str) name of the input to put data in, default is *Default_Input*
- `groups` (list) list of group names to use to parse this input data
- `template_name` (str) name of the template to add input for

add_lookup(*name*, *text_data=''*, *include=None*, *load='python'*, *key=None*)
Method to add lookup table data to all templates loaded so far. Lookup is a text representation of structure that can be loaded into python dictionary using one of the available loaders - python, csv, ini, yaml, json.

Parameters

- `name` (str) name to assign this lookup table for referencing
- `text_data` (str) text to load lookup table/dictionary from
- `include` (str) absolute or relative /os/path/to/lookup/table/file.txt
- `load` (str) name of TTP loader to use to load table data
- `key` (str) specify key column for csv loader to construct dictionary

`include` can accept relative OS path - relative to the directory where TTP will be invoked either using CLI tool or as a module

add_template(*template*, *template_name='_root_template_'*, *filter=[]*)
Method to load TTP templates into the parser.

Parameters

- `template` file object or OS path to text file with template
- `template_name` (str) name of the template
- `filter` (list) list of templates' names to load,

`filter` attribute allow to filter the list of template names that should be loaded. Checks done against child templates as well. For templates specified in filter list, groups/macro/inputs/etc. will not be loaded and no results produced.

add_vars(*vars*)
Method to add variables to ttp and its templates to reference during parsing

Parameters

- `vars` dictionary of variables to make available to ttp parser

clear_input(*template_name='_all_'*)
Method to delete all input data for all or some templates, can be used prior to adding new set of data to parse with same templates, instead of re-initializing ttp object.

Parameters

- `template_name` (str) name of the template to clear input for, clears for all templates by default

clear_result(*templates=[]*)
Method to clear parsing results for templates.

Parameters

- `templates` (list or str) - name of template(s) to clear results for, if not provided will clear results for all templates.

get_input_load()
Method to retrieve input tag text load. Using input `load` attribute, text data can be loaded into python structure using one of the supported loaders, for instance if text data structured using YAML, YAML loader can be used to produce python native structure, that structure will be returned by this method.

Primary use case is to specify parameters within TTP input that can be used by other applications/scrips.

Returns

Dictionary of {"template_name": {"input_name": "input load data"}} across all templates, where input_name set to input name attribute value, by default it is "Default_Input", and template_name set to name of the template, by default it is "_root_template_"

> **Warning:** inputs load can override one another if combination of template_name and input_name is not unique.

parse(*one=False*, *multi=False*)
Method to parse data with templates.

Parameters

- `one` (boolean) if set to True will run parsing in single process
- `multi` (boolean) if set to True will run parsing in multiprocess

By default one and multi set to False and TTP will run parsing following below rules:

1. if one or multi set to True run in one or multi process
2. if overall data size is less then 5Mbyte, use single process
3. if overall data size is more then 5Mbytes, use multiprocess

In addition to 3 TTP will check if number of input data items more then 1, if so multiple processes will be used and one process otherwise.

result(*templates=[]*, *structure='list'*, ***kwargs*)
Method to get parsing results, supports basic filtering based on templates' names, results can be formatted and returned to specified returner.

Parameters

- `templates` (list or str) names of the templates to return results for
- `structure` (str) structure type, valid values - `list`, `dictionary` or `flat_list`

kwargs - can contain any attributes supported by output tags, for instance:

- `format` (str) output formatter name - yaml, json, raw, pprint, csv, table, tabulate
- `functions` (str) reference output functions to run results through

Example:

```
from ttp import ttp
parser = ttp(data="/os/path/to/data/dir/", template="/os/path/to/template.txt
↪")
parser.parse()
json_result = parser.result(format="json")[0]
yaml_result = parser.result(format="yaml")[0]
print(json_result)
print(yaml_result)
```

Returns

By default template results set to *per_input* and structure set to *list*, returns list such as:

```
[
   [ template_1_input_1_results,
     template_1_input_2_results,
     ...
     template_1_input_N_results ],
   [ template_2_input_1_results,
     template_2_input_2_results,
     ...
]
```

If template results set to *per_template* and structure set to *list*, returns list such as:

```
[
   [ template_1_input_1_2...N_joined_results ],
   [ template_2_input_1_2...N_joined_results ]
]
```

If template results set to *per_input* and structure set to *dictionary*, returns dictionary such as:

```
{
   template_1_name: [
     input_1_results,
     input_2_results,
     ...
     input_N_results
    ],
   template_2_name: [
     input_1_results,
     input_2_results
    ],
     ...
}
```

If template results set to *per_template* and structure set to *dictionary*, returns dictionary such as:

```
{
   template_1_name: input_1_2...N_joined_results,
   template_2_name: input_1_2...N_joined_results
}
```

If structure set to *flat_list*, results will be combined across all templates in a list of dictionaries. For instance, with structure set to *list* result might look like this:

```
[[[{'interface': 'Lo0', 'ip': '192.168.0.1', 'mask': '32'},
   {'interface': 'Lo1', 'ip': '1.1.1.1', 'mask': '32'}],
  [{'interface': 'Lo2', 'ip': '2.2.2.2', 'mask': '32'},
   {'interface': 'Lo3', 'ip': '3.3.3.3', 'mask': '32'}]]]
```

But with structure set to *flat_list* it will be flattened to this:

```
[{'interface': 'Lo0', 'ip': '192.168.0.1', 'mask': '32'},
 {'interface': 'Lo1', 'ip': '1.1.1.1', 'mask': '32'},
 {'interface': 'Lo2', 'ip': '2.2.2.2', 'mask': '32'},
 {'interface': 'Lo3', 'ip': '3.3.3.3', 'mask': '32'}]
```

set_input(*data, input_name='Default_Input', template_name='_root_template_', groups=['all']*)
Method to replace existing templates inputs data with new set of data. This method is alias to `clear_input` and `add_input` methods.

> **Warning:** `set_input` should be called only after templates added

Parameters

- `data` file object or OS path to text file or directory with text files with data to parse
- `input_name` (str) name of the input to put data in, default is *Default_Input*
- `groups` (list) list of group names to use to parse this input data
- `template_name` (str) name of the template to set input for

16.1 _ttp_ dictionary reference

TBD

CHAPTER 17

Performance

TTP has performance of approximately 211 lines per millisecond on Intel Core i5-3320M CPU @ 2.6GHz (CPU End-of-Life July 2014) if running in multiprocess mode, dataset of 3,262,464 lines can be parsed in under 16 seconds best case and under 22 seconds worst case. Multiprocessing mode approximately 30-40% faster compared to running in single process, the difference is more significant the more data has to be parsed.

When TTP ready to parse data it goes through decision logic to determine parsing mode following below rules:

- run in single process if `one=True` was set for TTP parse method
- run in multiprocess if `multi=True` was set for TTP parse method
- run in single process if overall size of loaded data less then 5MByte
- run in multiprocess if overall size of loaded data more then 5MByte and at least two datums loaded

In multiprocessing mode, TTP starts one process per each CPU core on the system and forms a queue of work, there each item contains data for single input datum. For instance we have a folder with 100 files to process, TTP forms queue of 100 chunks of work, each chunk containing text data from single file, in multiprocessing mode that work distributed across several cores in such a way that as long as chunk of work finished by the process it picks up another chunk, without waiting for other processes to finish.

17.1 Multiprocessing mode restrictions

While multiprocessing mode has obvious processing speed increase benefits, it comes with several restrictions.

- per_template results mode not supported with multiprocessing as no results shared between processes, only per_input mode supported with multiprocessing
- startup time for multiprocessing is slower compared to single process, as each process takes time to initiate
- global variables space not shared between processes, as a result a number of functions will not be able to operate properly, such as:
  - match variable count function - `globvar` will not have access to global variables
  - match variable record function - record cannot save variables in global namespace

- match variable lookup function - will not work if reference group that parse different inputs due to `_ttp_['template_obj']` not shared between processes

17.2 General performance considerations

Keep data processing out of TTP if you are after best performance, the more processing/functions TTP has to run, the more time it will take to finish parsing.

During parsing, avoid use of broad match regular expressions, such as `.*` unless no other options left, one such expression used for `_line_` indicator internally. As a result of excessive matches, processing time can increase significantly. Strongly consider using `_end_` indicator together with any broad match regexes to limit the scope of text processed.

Consider providing TTP with as clean data as possible - data the contains only data that will be matched by TTP. That will help to save CPU cycles by not processing unrelated data, also that will grantee that no false positive matches exist. For instance, input `commands` function can be used to pre-process data and present only required commands output to certain groups.

Python Module Index

t

Index