ttp

Release 0.0.1

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 and capable of representing their configuration or state data in the form 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 sets, prepare them and map to the groups for parsing
- output system - allows to format parsing results in certain way and return or save to them certain destinations

CHAPTER 2

Installation

Using pip:

```
pip install ttp
```

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

```
python setup.py install
```

CHAPTER 3

Quick start

After installing ttp, to use it as a module:

```
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
results = parser.result(format='json')[0]
print(results)
[
    [
        {
            "description": "Router-id-loopback",
```

(continues on next page)

(continued from previous page)

```
            "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
```

In the template each variable that we want to extract must be placed within `{{ }}` brackets, name of match variable will become dictionary key with value equal to extracted data.

Data can be an OS path to file or directory with files to parse, template can be sourced from text file as well, for instance:

```
parser = ttp(data="/path/to/data/file.txt", template="/path/to/template.txt")
```

Data and templates can be loaded to the parser object after instantiation:

```
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, add data, add template and run parsing:
parser = ttp()
parser.add_input(data_to_parse)
parser.add_template(ttp_template)
parser.parse()
```

(continues on next page)

(continued from previous page)

```
results = parser.result(format='pprint')[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'}]]
```

CHAPTER 4

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"
        }
    ]
]
```

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"/>
```

Note: 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 is `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
```

4.1 Parsing 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
 !
```

(continues on next page)

(continued from previous page)

```
 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") }}
    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 }}
  !
  <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.

4.2 Parse text tables

TBD

4.3 Parse show commands output

TBD

4.4 Filtering with TTP

TBD

4.5 Outputting results

TBD

CHAPTER 5

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

or using short keywords

ttp -d "/path/to/data/" -t "path/to/template.txt" -o 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
- `-t, --template` path to text file with template content or name of the template in templates.py
- `-o, --outputter` format results data using yaml, json, raw or pprint formatter and prints them to terminal
- `-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
- `--one` forcefully run parsing using single process
- `--multi` forcefully run parsing in multiple processes

CHAPTER 6

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', groups=['all']*)

Method to load additional data to be parsed. This data will be used to fill in template input with input_name and parse that data against a list of provided groups.

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

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 to this lookup table to reference in templates
- `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=None*)
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

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()
Method to delete all input data for all templates, can be used prior to adding new set of data to parse with same templates, instead of re-initializing ttp object.

get_input_commands_dict()
Method to iterate over all templates and inputs to get a list of commands that needs to be present in text data, that text data will be consumed by inputs, each input will extract output for its commands and will run groups parsing for it.

Returns

Dictionary of {"input_name": [input commands list]}

get_input_commands_list()
Method to iterate over all templates and inputs to get a list of commands that needs to be present in text data, that text data will be consumed by inputs, each input will extract output for its commands and will run groups parsing for it.

Returns

List of unique commands - [command1, command2, . . . , commandN]

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'*, *returner='self'*, ***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
- `returner` (str) returner to use to return data - self, file, terminal
- `structure` (str) structure type, valid values - `list` or `dictionary`

kwargs - can contain any attributes supported by output tags, for instance:

- `format` (str) formatter name - yaml, json, raw, pprint, csv, table, tabulate
- `functions` (str) reference 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
}
```

set_input(*data, input_name='Default_Input', groups=['all']*)

Method to replace existing templates data with new set of data. This method run clear_input first and add_input method after that.

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

CHAPTER 7

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 with all function pre-cached by Python, meaning that dataset of 3,262,464 lines can be parsed in under 16 seconds best case and under 22 seconds worst case.

7.1 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
- global variables are not shared between processes and have per-process significance (name space), this is due to the fact that global vars not shared between processes

CHAPTER 8

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.

8.1 Group reference

8.1.1 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
default	Value to supply for variables by default

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"
```

- inputN (optional) - string that contains name of the input tag 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* inputs: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>
```

(continues on next page)

(continued from previous page)

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

Template:

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

(continues on next page)

(continued from previous page)

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

Result:

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

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"
            }
        ]
    }
]
```


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.

8.1.2 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
group functions	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

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 tag names that should be used to run group results through. The sequence of macroses provided *preserved* and macroses executed in specified order, in other words macro named name2 will run after macro name1.

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

Macro function must accept only one attribute to hold group results, for groups data supplied to macro function is a dictionary of data matched by this group.

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:

```
[
    {
        "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
            }
        ]
    }
]
```

group functions

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

- functionN - name of the group function together with it's attributes

The main advantage of using string of functions against defining functions directly in the group tag is the fact that it allows to define sequence of functions to run group results through and that order will be honored. 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 'l' symbol must be used to separate function names, not comma

Example

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)
    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:

```
[
    {
```

(continues on next page)

(continued from previous page)

```
        "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"
            }
        ]
    }
]
```

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'">
```

(continues on next page)

(continued from previous page)

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

exclude

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

TBD

excludeall

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

TBD

del

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

TBD

sformat

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

TBD

itemize

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

TBD

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

If below data parsed with that template:

```
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
```

This result will be produced:

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

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

If below data parsed with that template:

```
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
```

This result will be produced:

```
[
    {
        "interfaces": [              <----this is the start of nested list
            {
                "vlan": {
                    "L3": [          <----this is the start of another nested list
                        {
```

(continues on next page)

(continued from previous page)

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

Note: containsall group function in above template just to demonstrate filtering capabilities and not related to path formatters

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
  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 }}
  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>
```

(continues on next page)

(continued from previous page)

```
<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
```

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"
                }
            }
        ]
    }
]
```

No name attribute

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.

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"
        }
    },
    {
        "description": "Management",
        "interface": "Vlan777",
        "ips": {
            "ip": "192.168.0.1",
            "mask": "24"
        },
```

(continues on next page)

(continued from previous page)

```
        "vrf": "MGMT"
    }
]
```

CHAPTER 9

Match Variables

Match variables used to denote names of pieces of information that needs to be extracted from text data. For instance in this template:

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

Match variables must be placed between `{{` and `}}` double curly brackets, in above example match variables are `interface` and `trunk_vlans` will store matching results extracted from this text data:

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

In other words, if above data will be parsed with given template, this results will be produced:

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

In addition, match variables can be accompanied with various function to process data during parsing or 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 combined.

9.1 Match Variables reference

9.1.1 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
*_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

exact

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

By default all digits in template replaced with 'd+' pattern, if _exact_ present, digits will stay unchanged and will be used for parsing.

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"
            },
            {
                "limit": "300",
                "warning": "80"
```

(continues on next page)

(continued from previous page)

```
            }
        ],
        "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"
    }
}
```

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
 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("regular_expression") }}`

- regular_expression (optional) - regex to use to substitute portion of the string, default is "S+", meaning any non-space character one or more times.

Primary use case of this indicator is to ignore changing data in text we need to parse, for example consider below output:

```
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,
```

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, 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"
        }
    ]
]
```


9.1.2 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 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
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
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

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
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
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"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

record

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

- var_name (mandatory) - a string containing variable name where to record match results

Records match results in template variable with given name after all functions run finished for match result. That recorded variable can be referenced within other functions such as *set*

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
- 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 append to match result

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"
}
```

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

Template:

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

<group name="interfacesset">
```

(continues on next page)

(continued from previous page)

```
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>
```

(continues on next page)

(continued from previous page)

```
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"
        }
    ]
}
```

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']
```

(continues on next page)

(continued from previous page)

```
</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

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', 'add_field') }}
```

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

Lookup function takes match value and perform lookup on that value in lookup table. Lookup 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 option on what to do with looked up 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

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>
```

(continues on next page)

(continued from previous page)

```
<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"
            }
        ]
    }
}
```

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>
```

(continues on next page)

(continued from previous page)

```
<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"
            }
        ]
    }
}
```

startswith_re

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

- pattern(mandatory) - string pattern to check

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

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

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

contains

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

- pattern(mandatory) - string pattern to check

This function evaluates if match value contains given string pattern, 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

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

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

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

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

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

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) }}
```

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.

Example

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"
        }
    ]
}]
```

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
```

(continues on next page)

(continued from previous page)

```
!
</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
 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"
        },
```

(continues on next page)

(continued from previous page)

```
        "interfaces_rdnsv4_google_server": {
            "interface": "GigabitEthernet3/11",
            "ip": "dns.google",
            "mask": "255.255.255.255"
        }
    }
]
```

sformat

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

TBD

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",
```

(continues on next page)

(continued from previous page)

```
                "hours": "10",
                "mins": "46",
                "secs": "10",
                "weeks": "27"
            }
        }
    }
]
```

9.1.3 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 see if any of variables match 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>
```

(continues on next page)

(continued from previous page)

```
<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+)) *(?=\n)'
```

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

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
 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 }}
<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 10

TTP tags

TTP includes a number of additional tags to structure templates or provide additional data to use during parsing.

10.1 Template

TTP templates support <template> tag to define several templates within single template, each template processes separately, no data shared across templates.

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 organize such a work flow when results produced by one template can be leveraged by 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 and that can 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_ }}
```

(continues on next page)

(continued from previous page)

```
    {{ intf_list | ROW }}
</group>
</group>
"""

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()
```

10.1.1 Template tag attributes

There are a number of attributes can be used with template tag, these attributes help to define template processing behavior.

Attribute	Description
name	Uniquely identifies template
base_path	Fully qualified OS path to data
results	Identifies the way how results should be grouped
pathchar	Character to use for group name-path processing

name

TBD

base_path

TBD

results

TBD

pathchar

TBD

10.2 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
!
interface GigabitEthernet3/3
 switchport access vlan 600
!

[sw-2.txt]
```

(continues on next page)

(continued from previous page)

```
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:

```
[
    {
        "interfaces1": [
            {
                "access_vlan": "500",
                "interface": "GigabitEthernet3/2"
            },
            {
```

(continues on next page)

(continued from previous page)

```
                "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"
            }
        ]
    }
]
```

10.2.1 Input tag attributes

There are a number of attributes can be specified in input tag, these attributes help to define input behavior and how data should be loaded and parsed.

Attribute	Description
name	Uniquely identifies input within template
groups	Specifies group(s) that should be used to parse input data
load	Identifies loader that should be used to load text data for input tag itself
preference	specify preference on how to handle inputs' groups and groups' input

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.

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"
                }
            ]
        }
    }
]
```

preference

`preference="merge|group_inputs|input_groups"`

TBD

10.2.2 Input tag functions

Input tag support functions to pre-process data.

Attribute	Description
functions	String with functions defined int it
commands	Comma separated list of commands output to extract from text data
test	Test function to verify input function handling

functions

```
functions="function1('attr1', 'attr2') | function2"
```

TBD

commands

```
commands="command1, command2, ... , commandN"
```

TBD

test

```
test=""
```

TBD

10.2.3 Input parameters

Apart from input attributes specified in <input> tag, text payload of <input> tag can be used to pass additional parameters. These parameters is a key value pairs and serve to provide information that should be used during input data loading. Input tag *load* attribute can be used to specify which loader to use to parse data in tag's text, e.g. if data structured in yaml format, yaml loader can be used to convert it in Python data structure.

Parameter	Description
url	Single url string or list of urls of input data location
extensions	Extensions of files to load input data from, e.g. "txt" or "log" or "conf"
filters	Regular expression or list of regexes to use to filter input data files based on their names

url

`url="url-1"` or `url=["url-1", "url-2", ... , "url-N"]`

- url-N - string or list of strings that contains absolute or relative (if base path provided) OS path to file or to directory of file(s) that needs to be parsed.

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.

10.3 Variables

ttp supports definition of custom variables using dedicated xml tags <v>, <vars> or <variables>. Withing this tags variables can be defined in various formats and loaded using one of supported loaders. Variables can also be defined in external text files using *include* attribute.

Custom variables can be used in a number of places within the templates, primarily in match variable functions, to store data off the groups definitions.

Data can also be recorded in variables during parsing and referenced later to construct dynamic path or within variables functions.

10.3.1 Variable tag 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

10.3.2 Variable getters

TTP template variables also support a number of getters - functions targeted to get some information and assign it to variable.

Function	Description
gethostname	this function tries to extract hostname out of source data prompts
getfilename	returns a name of the source data
gettime	not implemented yet

10.3.3 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-->
```

(continues on next page)

(continued from previous page)

```
<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

10.3.4 include

`include="path"`

- path - absolute OS path to text file with variables data.

10.3.5 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
```

(continues on next page)

(continued from previous page)

```
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

10.3.6 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.

10.3.7 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">
```

(continues on next page)

(continued from previous page)

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

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

Result:

```
[
    {
        "interfaces": {
            "description": "input_1_data",
            "interface": "GigabitEthernet3/11"
        },
        "vars": {
            "hostname_var": "switch1"
        }
    }
]
```

10.3.8 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".

10.3.9 gettime

`var_name="gettime"`

TBD

10.4 Lookups

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

10.4.1 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.

10.4.2 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.

10.4.3 include

`include="path"`

- path - absolute OS path to text file with lookup table data.

10.4.4 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.

10.4.5 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>
```

(continues on next page)

(continued from previous page)

```
<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"
        }
    }
]
```

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"
    }
}
```

10.4.6 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"
                }
            ]
        }
    }
]
```

10.4.7 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
  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"
        }
    }
]
```

10.5 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.

10.5.1 Outputs reference

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.

Output attributes

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
functions	pipe separated list of functions to run results through

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

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.

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.

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

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
macro	passes results through macro function

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"
>
[
    {
        "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

TBD

dict_to_list

TBD

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"
```

(continues on next page)

(continued from previous page)

```
        }
    ]
]
```

Formatters

TTP supports *raw*, *yaml*, *json*, *csv*, jinja2, *pprint*, tabulate, *table*, *excel* formatters. Formatters have a number of attributes that can be used 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 string representation of results, except for *raw* output formatter, which just returns input data without modifying it.

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 needs 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.

Example

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="pprint" returner="terminal"/>

<output format="table" returner="terminal"/>
```

Results:

```
First output will print to terminal, after passing results through pprint function:
[   [   {'interface': 'Loopback0', 'ip': '192.168.0.113', 'mask': '24'},
        {'interface': 'Vlan778', 'ip': '2002::fd37', 'mask': '124'}],
    [   {'interface': 'Loopback10', 'ip': '192.168.0.10', 'mask': '24'},
        {'interface': 'Vlan710', 'ip': '2002::fd10', 'mask': '124'}]]

Above data will serve as an input to second outputter, that outputter
will format data in table list of lists:
[['interface', 'ip', 'mask'],
['Loopback0', '192.168.0.113', '24'],
['Vlan778', '2002::fd37', '124'],
['Loopback10', '192.168.0.10', '24'],
['Vlan710', '2002::fd10', '124']]
```

Note: csv and tabulate outputters use table outputter to construct a list of lists, after that they use it to represent data in certain format. Meaning all the attributes supported by table outputter, inherently supported by csv and tabulate outputters.

csv

This outputter takes parsing result as an input, transforms it in list of lists using table outputter and emits csv structured table.

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 needs to be installed on the system.

Tabulate outputter uses python tabulate module to transform and emit results in a plain-text table.

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="tabulate" returner="terminal"/>
```

Results:

```
interface    ip               mask
-----------  -------------  ------
Loopback0    192.168.0.113      24
Vlan778      2002::fd37        124
```

jinja2

Prerequisites: Jinja2 module needs to be installed on the system

This outputters allow to render parsing results with jinja2 template. Jinja2 template can be enclosed in output tag text data. Jinja2 templates can help to produce any text output out of parsing results. There are lots of use cases for it, to name a few:

- vendor configuration translator - parse vendor A configuration, emit configuration for vendor B
- markdown - use Jinja2 template to produce markdown report etc.

Within jinja2, the whole parsing results data passed into the renderer within *_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
 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 needs 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.

Example

Template:

```
<input load="text">
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Vlan778
 ip address 2002::fd37/124
 ip vrf CPE1
!
</input>

<group name="interfaces_1">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<group name="interfaces_2">
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<output
format="excel"
```

(continues on next page)

(continued from previous page)

```
returner="file"
filename="excel_out_%Y-%m-%d_%H-%M-%S"
url="C:/result/"
load="yaml"
>
table:
  - headers: interface, ip, mask, vrf, description
    path: interfaces_1
    tab_name: tab-1
  - path: interfaces_2
    tab_name: tab-2
</output>
```

TTP will produce excel table with two tabs using results from different groups. Table will be saved under *C:/result/* path in *excel_out_%Y-%m-%d_%H-%M-%S.xslx* file.

Formatter attributes

Formatter	Attribute	Description
table, csv, tabulate, excel	*path*	dot separated string that denotes path to data within results tree
tabulate	*format_attributes*	string of **args*, ***kwargs* to pass to formatter
table, csv, tabulate, excel	*headers*	comma separated string of table headers
csv	*sep*	character to separate items, by default it is comma
table, csv, tabulate, excel	*missing*	string to replace missing items based on provided headers
table, csv, tabulate, excel	*key*	string to use while flattening dictionary of data results

path

```
path="path_to_data"
```

- path_to_data - dot separated string of path items within results tree, used to specify location of data to work with.

In the case when results data is a nested structure and we want to output only part of it in a certain format, path attribute can be used to identify the portion of results to work with.

Supported by: table, csv, tabulate output formatters

Example

In this example we want to emit BGP peers in a table format, however, list of peer dictionaries is nested within results tree behind *bgp_config* and *peers* sections. We can set *path* to *bgp_config.peers* value to reference required data and pass it through output formatter, in this case csv.

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
```

(continues on next page)

(continued from previous page)

```
</input>

<group name="bgp_config">
router bgp {{ bgp_as }}
 <group name="peers">
  neighbor {{ peer }}
    description {{ description  }}
 </group>
</group>

<output name="out1" format="pprint" returner="terminal"/>

<output name="out2" path="bgp_config.peers" format="csv" returner="terminal"/>
```

Results:

```
[   {   'bgp_config': {   'bgp_as': '65100',
                          'peers': [   {   'description': 'vic-mel-core1',
                                           'peer': '10.145.1.9'},
                                       {   'description': 'qld-bri-core1',
                                           'peer': '192.168.101.1'}]}}]

description,peer
vic-mel-core1,10.145.1.9
qld-bri-core1,192.168.101.1
```

Outputter *out1* will emit data in native python format but structured by pprint for ease of read, while outputter *out2* will format peers data in a table using tabulate formatter. Returner *terminal* will print results to command line screen.

format_attributes

`format_attributes="**args, **kwargs"`

- args - list of attribute values e.g. *value1, value2, value3*, to pass to formatter
- kwargs - list of attribute name-value pairs e.g. *key1=value1, key2-value2*, to pass to formatter

Supported by: tabulate output formatter

Some outputters can be invoked with a number of additional arguments to modify their behavior, this arguments can be passed to them using *format_attributes* attribute.

Example

Tabulate outputter supports a number of table formates that can be specified using *tablefmt* argument, in below template data will be formatted using tabulate formatter with tabulate table format set to *fancy_grid* and results will be printer to terminal screen.

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>
```

(continues on next page)

(continued from previous page)

```
<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="csv"
returner="terminal" format_attributes="tablefmt='fancy_grid'"/>
```

Results:

```
| description   | peer          |
| vic-mel-core1 | 10.145.1.9    |
| qld-bri-core1 | 192.168.101.1 |
```

headers

```
headers="header1, header2, ... headerN"
```

- headers - comma separated string of table headers

Table formatter will identify the list of headers automatically, however, their order will be undefined and can change. To solve that problem, predefined list of headers can be supplied to formatter. Headers have to match key names of the results dictionaries and they are case sensitive.

Supported by: table, csv, tabulate output formatters

Example

Template:

```
<input load="text">
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
!
</input>

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

(continues on next page)

(continued from previous page)

```
<output
format="tabulate"
returner="terminal"
headers="interface, description, vrf, ip, mask"
/>
```

Results:

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

sep

`sep="char"`

- char - separator character to use for csv formatter, default value is comma ","

Supported by: csv output formatter

missing

`missing="value"`

- value - string to use to substitute empty cells in table, default is empty - ""

Supported by: table, csv, tabulate output formatters

Example

Template:

```
<input load="text">
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Vlan778
 ip address 2002::fd37/124
 ip vrf CPE1
!
</input>

<group>
interface {{ interface }}
 ip address {{ ip }}/{{ mask }}
 description {{ description }}
 ip vrf {{ vrf }}
</group>

<output
format="tabulate"
returner="terminal"
```

(continues on next page)

(continued from previous page)

```
missing="UNDEFINED"
/>
```

Results:

```
description          interface    ip               mask  vrf
-------------------  -----------  -------------  ------  ---------
Router-id-loopback   Loopback0    192.168.0.113      24  UNDEFINED
UNDEFINED            Vlan778      2002::fd37        124  CPE1
```

key

`key="name"`

- name - name of the key to use in a dictionary to associate data value

This attribute helps to solve specific problem when results data is a dictionary of dictionaries similar to this:

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

If `key` will be set to "intf_name", above data will be transformed into list of dictionaries such as:

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

With that list of lists table formatter will be able to create below list of lists and emit it in desirable format (csv, tabulate):

```
[
['description', 'intf_name', 'ip', 'mask', 'vrf'],
['Router-id-loopback', 'Loopback0', '192.168.0.113', '24', ''],
['', 'Vlan778', '2002::fd37', '124', 'CPE1']
]
```

Example

Template:

```
<input load="text">
interface Loopback0
 description Router-id-loopback
 ip address 192.168.0.113/24
!
interface Vlan778
 ip address 2002::fd37/124
 ip vrf CPE1
!
</input>

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

<output
format="tabulate"
returner="terminal"
key="intf_name"
/>
```

Results:

```
description         intf_name    ip              mask  vrf
------------------  -----------  -------------  ------  -----
Router-id-loopback  Loopback0    192.168.0.113      24
                    Vlan778      2002::fd37        124  CPE1
```

Returners

TTP has *file*, *terminal* and *self* returners. The purpose of returner is to return or emit or save data to certain destination.

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.

terminal

Results will be printed to terminal window.

Returner attributes

Returner	Attribute	Description
file	*url*	OS path to folder there to save results
file	*filename*	name of the file to save data in

url

If returner is file - url attribute helps to specify full OS path to folder where file should be stored.

filename

If returner is file - filename specifies the name of the file to save data in. Filename attribute support a number of formatters.

Time filename formatters:

```
* ``%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.

10.6 Macro

One of the core features of TTP is to allow data processing on the go, as a result it 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 TTP_TAGS/Outputs:macro

Python Module Index

t

Index