PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

ШАБЛОНЫ КОНФИГУРАЦИЙ С JINJA

ШАБЛОНЫ КОНФИГУРАЦИЙ С JINJA

Jinja2 это язык шаблонов, который используется в Python.

Jinja2 используется для генерации документов на основе одного или нескольких шаблонов.

Примеры использования:

- шаблоны для генерации HTML-страниц
- шаблоны для генерации конфигурационных файлов в Unix/Linux
- шаблоны для генерации конфигурационных файлов сетевых устройств

Установить Jinja2 можно с помощью pip:

```
pip install jinja2
```

ШАБЛОНЫ КОНФИГУРАЦИЙ С JINJA

Идея Jinja очень проста: разделение данных и шаблона. Это позволяет использовать один и тот же шаблон, но подставлять в него разные данные.

В самом простом случае, шаблон это просто текстовый файл, в котором указаны места подстановки значений, с помощью переменных Jinja.

ШАБЛОНЫ КОНФИГУРАЦИЙ С JINJA

Пример шаблона Jinja:

```
hostname {{name}}
!
interface Loopback255
 description Management loopback
 ip address 10.255.
{{id}}.1 255.255.255.255
!
interface GigabitEthernet0/0
 description LAN to
{{name}} sw1 {{int}}
 ip address
{{ip}} 255.255.255.0
!
router ospf 10
 router-id 10.255.
{{id}}.1
 auto-cost reference-bandwidth 10000
 network 10.0.0.0 0.255.255.255 area 0
```

ШАБЛОНЫ КОНФИГУРАЦИЙ С JINJA

Пример скрипта с генерацией файла на основе шаблона Jinja (файл basic_generator.py):

```
from jinja2 import Template

template = Template("""
hostname {{name}}
!
interface Loopback255
 description Management loopback
 ip address 10.255.{{id}}.1 255.255.255.255
!
interface GigabitEthernet0/0
 description LAN to {{name}} sw1 {{int}}
 ip address {{ip}} 255.255.255.0
!
router ospf 10
 router-id 10.255.{{id}}.1
 auto-cost reference-bandwidth 10000
 network 10.0.0.0 0.255.255.255 area 0
""")

liverpool = {'id':'11', 'name':'Liverpool', 'int':'Gi1/0/17', 'ip':'10.1.1.10'}

print(template.render(liverpool))
```

ШАБЛОНЫ КОНФИГУРАЦИЙ С JINJA

Если запустить скрипт basic_generator.py, то вывод будет таким:

```
$ python basic_generator.py

hostname Liverpool
!
interface Loopback255
 description Management loopback
 ip address 10.255.11.1 255.255.255.255
!
interface GigabitEthernet0/0
 description LAN to Liverpool sw1 Gi1/0/17
 ip address 10.1.1.10 255.255.255.0
!
router ospf 10
 router-id 10.255.11.1
 auto-cost reference-bandwidth 10000
 network 10.0.0.0 0.255.255.255 area 0
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

В этом примере логика разнесена в 3 разных файла (все файлы находятся в каталоге 1_example):

- router_template.py - шаблон
- routers_info.yml - в этом файле, в виде списка словарей (в формате YAML), находится информация о маршрутизаторах, для которых нужно сгенерировать конфигурационный файл
- router_config_generator.py - в этом скрипте импортируется файл с шаблоном и считывается информация из файла в формате YAML, а затем генерируются конфигурационные файлы маршрутизаторов

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Файл router_template.py

```
# -*- coding: utf-8 -*-
from jinja2 import Template

template_r1 = Template("""
hostname {{name}}
!
interface Loopback10
 description MPLS loopback
 ip address 10.10.{{id}}.1 255.255.255.255
 !
interface GigabitEthernet0/0
 description WAN to {{name}} sw1 G0/1
!
interface GigabitEthernet0/0.1{{id}}1
 description MPLS to {{to_name}}
 encapsulation dot1Q 1{{id}}1
 ip address 10.{{id}}.1.2 255.255.255.252
 ip ospf network point-to-point
 ip ospf hello-interval 1
 ip ospf cost 10
!
interface GigabitEthernet0/1
 description LAN {{name}} to sw1 G0/2 !
interface GigabitEthernet0/1.{{IT}}
 description PW IT {{name}} - {{to_name}}
 encapsulation dot1Q {{IT}}
 xconnect 10.10.{{to_id}}.1 {{id}}11 encapsulation mpls
 backup peer 10.10.{{to_id}}.2 {{id}}21
 backup delay 1 1
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Файл routers_info.yml

```yaml
- id: 11
  name: Liverpool
  to_name: LONDON
  IT: 791
  BS: 1550
  to_id: 1

- id: 12
  name: Bristol
  to_name: LONDON
  IT: 793
  BS: 1510
  to_id: 1

- id: 14
  name: Coventry
  to_name: Manchester
  IT: 892
  BS: 1650
  to_id: 2
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Файл router_config_generator.py

```
# -*- coding: utf-8 -*-
import yaml
from jinja2 import Template
from router_template import template_r1

routers = yaml.load(open('routers_info.yml'))

for router in routers:
    r1_conf = router['name']+'_r1.txt'
    with open(r1_conf,'w') as f:
        f.write(template_r1.render(router))
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Файл router_config_generator.py:

- импортирует шаблон template_r1
- из файла routers_info.yml список параметров считывается в переменную routers

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Затем в цикле перебираются объекты (словари) в списке routers:

- название файла, в который записывается итоговая конфигурация, состоит из поля name в словаре и строки _r1.txt
  - например, Liverpool_r1.txt
- файл с таким именем открывается в режиме для записи
- в файл записывается результат рендеринга шаблона с использованием текущего словаря
- конструкция with сама закрывает файл
- управление возвращается в начало цикла (пока не переберутся все словари)

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Запускаем файл router_config_generator.py:

```
$ python router_config_generator.py
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA2

Liverpool_r1.txt:

```
hostname Liverpool
!
interface Loopback10
 description MPLS loopback
 ip address 10.10.11.1 255.255.255.255
!
interface GigabitEthernet0/0
 description WAN to Liverpool sw1 G0/1
!
interface GigabitEthernet0/0.1111
 description MPLS to LONDON
 encapsulation dot1Q 1111
 ip address 10.11.1.2 255.255.255.252
 ip ospf network point-to-point
 ip ospf hello-interval 1
 ip ospf cost 10
!
interface GigabitEthernet0/1
 description LAN Liverpool to sw1 G0/2
!
interface GigabitEthernet0/1.791
 description PW IT Liverpool - LONDON
 encapsulation dot1Q 791
 xconnect 10.10.1.1 1111 encapsulation mpls
  backup peer 10.10.1.2 1121
  backup delay 1 1
!
interface GigabitEthernet0/1.1550
 description PW BS Liverpool - LONDON
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA С КОРРЕКТНЫМ ИСПОЛЬЗОВАНИЕМ ПРОГРАММНОГО ИНТЕРФЕЙСА

ПРИМЕР ИСПОЛЬЗОВАНИЯ JINJA С КОРРЕКТНЫМ ИСПОЛЬЗОВАНИЕМ ПРОГРАММНОГО ИНТЕРФЕЙСА

Термин "программный интерфейс" относится к способу работы Jinja с вводными данными и шаблоном, для генерации итоговых файлов.

Переделанный пример предыдущего скрипта, шаблона и файла с данными (все файлы находятся в каталоге 2_example):

Шаблон templates/router_template.txt это обычный текстовый файл:

```
hostname {{name}}
!
interface Loopback10
 description MPLS loopback
 ip address 10.10.{{id}}.1 255.255.255.255
 !
interface GigabitEthernet0/0
 description WAN to {{name}} sw1 G0/1
!
interface GigabitEthernet0/0.1{{id}}1
 description MPLS to {{to_name}}
 encapsulation dot1Q 1{{id}}1
 ip address 10.{{id}}.1.2 255.255.255.252
 ip ospf network point-to-point
 ip ospf hello-interval 1
 ip ospf cost 10
!
interface GigabitEthernet0/1
 description LAN {{name}} to sw1 G0/2 !
interface GigabitEthernet0/1.{{IT}}
 description PW IT {{name}} - {{to_name}}
 encapsulation dot1Q {{IT}}
 xconnect 10.10.{{to_id}}.1 {{id}}11 encapsulation mpls
 backup peer 10.10.{{to_id}}.2 {{id}}21
  backup delay 1 1
!
interface GigabitEthernet0/1.{{BS}}
 description PW BS {{name}} - {{to_name}}
 encapsulation dot1Q {{BS}}
```

Файл с данными routers_info.yml

```yaml
- id: 11
  name: Liverpool
  to_name: LONDON
  IT: 791
  BS: 1550
  to_id: 1

- id: 12
  name: Bristol
  to_name: LONDON
  IT: 793
  BS: 1510
  to_id: 1

- id: 14
  name: Coventry
  to_name: Manchester
  IT: 892
  BS: 1650
  to_id: 2
```

Скрипт для генерации конфигураций router_config_generator_ver2.py

```
# -*- coding: utf-8 -*-
from jinja2 import Environment, FileSystemLoader
import yaml

env = Environment(loader=FileSystemLoader('templates'))
template = env.get_template('router_template.txt')

routers = yaml.load(open('routers_info.yml'))

for router in routers:
    r1_conf = router['name']+'_r1.txt'
    with open(r1_conf,'w') as f:
        f.write(template.render(router))
```

Файл router_config_generator.py импортирует из модуля jinja2:

- **FileSystemLoader** - загрузчик, который позволяет работать с файловой системой
  - тут указывается путь к каталогу, где находятся шаблоны
  - в данном случае, шаблон находится в каталоге templates
- **Environment** - класс для описания параметров окружения:
  - в данном случае, указан только загрузчик
  - но в нем можно указывать методы обрабатки шаблона

Обратите внимание, что шаблон теперь находится в каталоге **templates**.

Если шаблоны находятся в текущем каталоге, надо добавить пару строк и изменить значение в загручике:

```
import os

curr_dir = os.path.dirname(os.path.abspath(__file__))
env = Environment(loader = FileSystemLoader(curr_dir))
```

Метод **get_template()** используется для того, чтобы получить шаблон. В скобках указывается имя файла.

СИНТАКСИС ШАБЛОНОВ JINJA2

СИНТАКСИС ШАБЛОНОВ JINJA2

До сих пор, в примерах шаблонов Jinja2 использовалась только подстановка переменных. Это самый простой и понятный пример использования шаблонов. Но синтаксис шаблонов Jinja на этом не ограничивается.

СИНТАКСИС ШАБЛОНОВ JINJA2

В шаблонах Jinja2 можно использовать:

- переменные
- условия (if/else)
- циклы (for)
- фильтры - специальные встроенные методы, которые позволяют делать преобразования переменных
- тесты - используются для проверки соответствует ли переменная какому-то условию

Кроме того, Jinja поддерживает наследование между шаблонами. А также позволяет добавлять содержимое одного шаблона в другой.

СИНТАКСИС ШАБЛОНОВ JINJA2

Для генерации шаблонов будет использовать скрипт cfg_gen.py

```
# -*- coding: utf-8 -*-
from jinja2 import Environment, FileSystemLoader
import yaml
import sys

TEMPLATE_DIR, template = sys.argv[1].split('/')
VARS_FILE = sys.argv[2]

env = Environment(loader=FileSystemLoader(TEMPLATE_DIR),
                  trim_blocks=True, lstrip_blocks=True)
template = env.get_template(template)

vars_dict = yaml.load(open(VARS_FILE))

print(template.render(vars_dict))
```

СИНТАКСИС ШАБЛОНОВ JINJA2

В строке

```
env = Environment(loader = FileSystemLoader(TEMPLATE_DIR), trim_blocks=True)
```

Параметр trim_blocks=True - удаляет первую пустую строку после блока конструкции, если установлено в True (по умолчанию False).

Также можно добавлять параметр lstrip_blocks=True - если установлено в True, пробелы и табы в начале строки удаляются (по умолчанию False).

СИНТАКСИС ШАБЛОНОВ JINJA2

Для того, чтобы посмотреть на результат, нужно вызвать скрипт и передать ему два аргумента:

- шаблон
- файл с переменными, в формате YAML

Результат будет выведен на стандартный поток вывода.

СИНТАКСИС ШАБЛОНОВ JINJA2

Пример запуска скрипта:

```
$ python cfg_gen.py templates/variables.txt data_files/vars.yml
```

КОНТРОЛЬ СИМВОЛОВ WHITESPACE

TRIM_BLOCKS

Параметр trim_blocks удаляет первую пустую строку после блока конструкции, если его значение равно True (по умолчанию False).

Посмотрим на эффект применения флага на примере шаблона templates/env_flags.txt:

```
router bgp {{ bgp.local_as }}
 {% for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}
```

TRIM_BLOCKS

Если скрипт cfg_gen.py запускается без флагов trim_blocks, lstrip_blocks:

```
env = Environment(loader = FileSystemLoader(TEMPLATE_DIR))
```

Вывод будет таким:

```
$ python cfg_gen.py templates/env_flags.txt data_files/router.yml
router bgp 100

 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100

 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

TRIM_BLOCKS

При добавлении флага trim_blocks таким образом:

```
env = Environment(loader = FileSystemLoader(TEMPLATE_DIR), trim_blocks=True)
```

Результат выполнения будет таким:

```
$ python cfg_gen.py templates/env_flags.txt data_files/router.yml
router bgp 100
  neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
  neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

LSTRIP_BLOCKS

Но перед строками neighbor ... remote-as появились два пробела. Так получилось из-за того, что перед блоком {% for ibgp in bgp.ibgp_neighbors %} стоит пробел. После того, как был отключен лишний перевод строки, пробелы и табы перед блоком добавляются к первой строке блока.

Но это не влияет на следующие строки. Поэтому строки с neighbor ... update-source отображаются с одним пробелом.

LSTRIP_BLOCKS

Параметр lstrip_blocks контролирует то, будут ли удаляться пробелы и табы от начала строки до начала блока (до открывающейся фигурной скобки).

Если добавить аргумент lstrip_blocks=True таким образом:

```
env = Environment(loader = FileSystemLoader(TEMPLATE_DIR), trim_blocks=True, lstrip_blocks=True)
```

LSTRIP_BLOCKS

Результат выполнения будет таким:

```
$ python cfg_gen.py templates/env_flags.txt data_files/router.yml
router bgp 100
 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

ОТКЛЮЧЕНИЕ LSTRIP_BLOCKS ДЛЯ БЛОКА

Иногда, нужно отключить функциональность lstrip_blocks для блока.

Например, если параметр lstrip_blocks установлен равным True в окружении, но нужно отключить его для второго блока в шаблоне (файл templates/env_flags2.txt):

```
router bgp {{ bgp.local_as }}
 {% for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}

router bgp {{ bgp.local_as }}
 {%+ for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}
```

ОТКЛЮЧЕНИЕ LSTRIP_BLOCKS ДЛЯ БЛОКА

Результат будет таким:

```
$ python cfg_gen.py templates/env_flags2.txt data_files/router.yml
router bgp 100
 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100

router bgp 100
  neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

ОТКЛЮЧЕНИЕ LSTRIP_BLOCKS ДЛЯ БЛОКА

Плюс после знака процента отключает lstrip_blocks для блока. В данном случае, только для начала блока.

Если сделать таким образом (плюс добавлен в выражении для завершения блока):

```
router bgp {{ bgp.local_as }}
 {% for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}

router bgp {{ bgp.local_as }}
 {%+ for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {%+ endfor %}
```

ОТКЛЮЧЕНИЕ LSTRIP_BLOCKS ДЛЯ БЛОКА

Он будет отключен и для конца блока:

```
$ python cfg_gen.py templates/env_flags2.txt data_files/router.yml
router bgp 100
 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100

router bgp 100
  neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
  neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

УДАЛЕНИЕ WHITESPACE В БЛОКЕ

Аналогичным образом можно контролировать удаление whitespace для блока.

Для этого примера в окружении не выставлены флаги:

```
env = Environment(loader = FileSystemLoader(TEMPLATE_DIR))
```

УДАЛЕНИЕ WHITESPACE В БЛОКЕ

Шаблон templates/env_flags3.txt:

```
router bgp {{ bgp.local_as }}
 {% for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}

router bgp {{ bgp.local_as }}
 {%- for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}
```

Обратите внимание на минус в начале второго блока. Минут удаляет все whitespace символы. В данном случае, в начале блока.

УДАЛЕНИЕ WHITESPACE В БЛОКЕ

Результат будет таким:

```
$ python cfg_gen.py templates/env_flags3.txt data_files/router.yml
router bgp 100

 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100

 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100


router bgp 100
 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100

 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

УДАЛЕНИЕ WHITESPACE В БЛОКЕ

Если добавить минут в конец блока:

```
router bgp {{ bgp.local_as }}
 {% for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {% endfor %}

router bgp {{ bgp.local_as }}
 {%- for ibgp in bgp.ibgp_neighbors %}
 neighbor {{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor {{ ibgp }} update-source {{ bgp.loopback }}
 {%- endfor %}
```

УДАЛЕНИЕ WHITESPACE В БЛОКЕ

Удалится пустая строка и в конце блока:

```
$ python cfg_gen.py templates/env_flags3.txt data_files/router.yml
router bgp 100

 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100

 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100


router bgp 100
 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
```

ПЕРЕМЕННЫЕ

ПЕРЕМЕННЫЕ

Переменные в шаблоне указываются в двойных фигурных скобках:

```
hostname {{ name }}

interface Loopback0
 ip address 10.0.0.{{ id }} 255.255.255.255
```

ПЕРЕМЕННЫЕ

Значения переменных подставляются на основе словаря, который передается шаблону.

Переменная, которая передается в словаре, может быть не только числом или строкой, но и, например, списком или словарем. Внутри шаблона можно, соответственно, обращаться к элементу по номеру или по ключу.

ПЕРЕМЕННЫЕ

Пример шаблона templates/variables.txt, с использованием разных вариантов переменных:

```
hostname {{ name }}

interface Loopback0
 ip address 10.0.0.{{ id }} 255.255.255.255

vlan {{ vlans[0] }}

router ospf 1
 router-id 10.0.0.{{ id }}
 auto-cost reference-bandwidth 10000
 network {{ ospf.network }} area {{ ospf['area'] }}
```

ПЕРЕМЕННЫЕ

И соответствующий файл data_files/vars.yml с переменными:

```yaml
id: 3
name: R3
vlans:
  - 10
  - 20
  - 30
ospf:
  network: 10.0.1.0 0.0.0.255
  area: 0
```

ПЕРЕМЕННЫЕ

Обратите внимание на использование переменной vlans в шаблоне:

- так как переменная vlans это список, можно указывать какой именно элемент из списка нам нужен

Если передается словарь (как в случае с переменной ospf), то внутри шаблона можно обращаться к объектам словаря, используя один из вариантов:

- ospf.network или ospf['network']

ПЕРЕМЕННЫЕ

Результат запуска скрипта будет таким:

```
$ python cfg_gen.py templates/variables.txt data_files/vars.yml
hostname R3

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

vlan 10

router ospf 1
 router-id 10.0.0.3
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
```

ЦИКЛ FOR

ЦИКЛ FOR

Цикл for позволяет проходится по элементам последовательности.

Цикл for должен находится внутри символов {% %}. Кроме того, нужно явно указывать завершение цикла:

```
{% for vlan in vlans %}
  vlan
{{ vlan }}
{% endfor %}
```

ЦИКЛ FOR

Пример шаблона templates/for.txt с использованием цикла:

```
hostname {{ name }}

interface Loopback0
 ip address 10.0.0.{{ id }} 255.255.255.255

{% for vlan, name in vlans.items() %}
vlan {{ vlan }}
 name {{ name }}
{% endfor %}

router ospf 1
 router-id 10.0.0.{{ id }}
 auto-cost reference-bandwidth 10000
 {% for networks in ospf %}
 network {{ networks.network }} area {{ networks.area }}
 {% endfor %}
```

ЦИКЛ FOR

Файл data_files/for.yml с переменными:

```
id: 3
name: R3
vlans:
  10: Marketing
  20: Voice
  30: Management
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

ЦИКЛ FOR

В цикле for можно проходится как по элементам списка (например, список ospf), так и по словарю (словарь vlans). И, аналогичным образом, по любой последовательности.

ЦИКЛ FOR

Результат выполнения будет таким:

```
$ python cfg_gen.py templates/for.txt data_files/for.yml
hostname R3

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

vlan 10
 name Marketing
vlan 20
 name Voice
vlan 30
 name Management

router ospf 1
 router-id 10.0.0.3
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

IF/ELIF/ELSE

IF/ELIF/ELSE

if позволяет добавлять условие в шаблон. Например, можно использовать if чтобы добавлять какие-то части шаблона, в зависимости от наличия переменных в словаре с данными.

IF/ELIF/ELSE

Конструкция if также должна находиться внутри {% %}. И нужно явно указывать окончание условия:

```
{% if ospf %}
router ospf 1
 router-id 10.0.0.
{{ id }}
 auto-cost reference-bandwidth 10000
{% endif %}
```

IF/ELIF/ELSE

Пример шаблона templates/if.txt:

```
hostname {{ name }}

interface Loopback0
 ip address 10.0.0.
{{ id }} 255.255.255.255

{% for vlan, name in vlans.items() %}
vlan
{{ vlan }}
 name
{{ name }}
{% endfor %}

{% if ospf %}
router ospf 1
 router-id 10.0.0.
{{ id }}
 auto-cost reference-bandwidth 10000

{% for networks in ospf %}
 network
{{ networks.network }} area {{ networks.area }}

{% endfor %}
{% endif %}
```

IF/ELIF/ELSE

Выражение if ospf работает так же, как в Python: если переменная существует и не пустая, результат будет True. Если переменной нет, или она пустая, результат будет False.

То есть, в этом шаблоне конфигурация OSPF генерируется только в том случае, если переменная ospf существует и не пустая.

IF/ELIF/ELSE

Конфигурация будет генерироваться с двумя вариантами данных.

Сначала, с файлом data_files/if.yml, в котором нет переменной ospf:

```
id: 3
name: R3
vlans:
  10: Marketing
  20: Voice
  30: Management
```

IF/ELIF/ELSE

Результат будет таким:

```
$ python cfg_gen.py templates/if.txt data_files/if.yml

hostname R3

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

vlan 10
 name Marketing
vlan 20
 name Voice
vlan 30
 name Management
```

IF/ELIF/ELSE

Теперь аналогичный шаблон, но с файлом data_files/if_ospf.yml:

```
id: 3
name: R3
vlans:
  10: Marketing
  20: Voice
  30: Management
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

IF/ELIF/ELSE

Теперь результат выполнения будет таким:

```
hostname R3

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

vlan 10
 name Marketing
vlan 20
 name Voice
vlan 30
 name Management

router ospf 1
 router-id 10.0.0.3
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

IF/ELIF/ELSE

Как и в Python, в Jinja можно делать ответвления в условии.

Пример шаблона templates/if_vlans.txt:

```
{% for intf, params in trunks.items() %}
interface
{{ intf }}

{% if params.action == 'add' %}
 switchport trunk allowed vlan add
{{ params.vlans }}

{% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove
{{ params.vlans }}

{% else %}
 switchport trunk allowed vlan
{{ params.vlans }}

{% endif %}
{% endfor %}
```

IF/ELIF/ELSE

Файл data_files/if_vlans.yml с данными:

```
trunks:
  Fa0/1:
    action: add
    vlans: 10,20
  Fa0/2:
    action: only
    vlans: 10,30
  Fa0/3:
    action: delete
    vlans: 10
```

В данном примере, в зависимости от значения параметра action, генерируются разные команды.

IF/ELIF/ELSE

В шаблоне можно было использовать и такой вариант обращения к вложенным словарям:

```
{% for intf in trunks %}
interface {{ intf }}
 {% if trunks[intf]['action'] == 'add' %}
 switchport trunk allowed vlan add {{ trunks[intf]['vlans'] }}
 {% elif trunks[intf]['action'] == 'delete' %}
 switchport trunk allowed vlan remove {{ trunks[intf]['vlans'] }}
 {% else %}
 switchport trunk allowed vlan {{ trunks[intf]['vlans'] }}
 {% endif %}
{% endfor %}
```

IF/ELIF/ELSE

В итоге, будет сгенерирована такая конфигурация:

```
$ python cfg_gen.py templates/if_vlans.txt data_files/if_vlans.yml
interface Fa0/1
 switchport trunk allowed vlan add 10,20
interface Fa0/3
  switchport trunk allowed vlan remove 10
interface Fa0/2
  switchport trunk allowed vlan 10,30
```

IF/ELIF/ELSE

Также, с помощью if, можно фильтровать по каким элементам последовательности пройдется цикл for.

Пример шаблона templates/if_for.txt с фильтром, в цикле for:

```
{% for vlan, name in vlans.items() if vlan > 15 %}
vlan
{{ vlan }}
 name
{{ name }}
{% endfor %}
```

IF/ELIF/ELSE

Файл с данными (data_files/if_for.yml):

```
vlans:
  10: Marketing
  20: Voice
  30: Management
```

IF/ELIF/ELSE

Результат выполнения:

```
$ python cfg_gen.py templates/if_for.txt data_files/if_for.yml
vlan 20
 name Voice
vlan 30
 name Management
```

ФИЛЬТРЫ

ФИЛЬТРЫ

В Jinja переменные можно изменять с помощью фильтров. Фильтры отделяются от переменной вертикальной чертой (pipe |) и могут содержать дополнительные аргументы.

Кроме того, к переменной могут быть применены несколько фильтров. В таком случае, фильтры просто пишутся последовательно, и каждый из них отделен вертикальной чертой.

ФИЛЬТРЫ

Jinja поддерживает большое количество встроенных фильтров. Мы рассмотрим лишь несколько из них. Остальные фильтры можно найти в документации.

Также, достаточно легко, можно создавать и свои собственные фильтры. Мы не будем рассматривать эту возможность, но это хорошо описано в документации .

DEFAULT

Фильтр default позволяет указать для переменной значение по умолчанию. Если переменная определена, будет выводиться переменная, если переменная не определена, будет выводиться значение, которое указано в фильтре default.

DEFAULT

Пример шаблона templates/filter_default.txt:

```
router ospf 1
 auto-cost reference-bandwidth {{ ref_bw | default(10000) }}
 {% for networks in ospf %}
 network {{ networks.network }} area {{ networks.area }}
 {% endfor %}
```

Если переменная ref_bw определена в словаре, будет подставлено её значение. Если же переменной нет, будет подставлено значение 10000.

DEFAULT

Файл с данными (data_files/filter_default.yml):

```
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

DEFAULT

Результат выполнения:

```
$ python cfg_gen.py templates/filter_default.txt data_files/filter_default.yml
router ospf 1
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

DEFAULT

По умолчанию, если переменная определена и её значение пустой объект, будет считаться, что переменная и её значение есть.

Если нужно сделать так, чтобы значение по умолчанию подставлялось и в том случае, когда переменная пустая (то есть, обрабатывается как False в Python), надо указать дополнительный параметр boolean=true.

DEFAULT

Например, если файл данных был бы таким:

```
ref_bw: ''
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

DEFAULT

То в итоге сгенерировался такой результат:

```
$ python cfg_gen.py templates/filter_default.txt data_files/filter_default.yml
router ospf 1
 auto-cost reference-bandwidth
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

DEFAULT

Если же, при таком же файле данных, изменить шаблон таким образом:

```
router ospf 1
 auto-cost reference-bandwidth {{ ref_bw | default(10000, boolean=true) }}
{% for networks in ospf %}
 network {{ networks.network }} area {{ networks.area }}
{% endfor %}
```

Вместо default(10000, boolean=true), можно написать default(10000, true)

DEFAULT

Результат уже будет таким (значение по умолчанию подставится):

```
$ python cfg_gen.py templates/filter_default.txt data_files/filter_default.yml
router ospf 1
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

DICTSORT

Фильтр dictsort позволяет сортировать словарь. По умолчанию, сортировка выполняется по ключам. Но, изменив параметры фильтра, можно выполнять сортировку по значениям.

Синтаксис фильтра:

```
dictsort(value, case_sensitive=False, by='key')
```

После того, как dictsort отсортировал словарь, он возвращает список кортежей, а не словарь.

DICTSORT

Пример шаблона templates/filter_dictsort.txt с использованием фильтра dictsort:

```
{% for intf, params in trunks | dictsort %}
interface
{{ intf }}

{% if params.action == 'add' %}
 switchport trunk allowed vlan add
{{ params.vlans }}

{% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove
{{ params.vlans }}

{% else %}
 switchport trunk allowed vlan
{{ params.vlans }}

{% endif %}
{% endfor %}
```

Обратите внимание, что фильтр ожидает словарь, а не список кортежей или итератор.

DICTSORT

Файл с данными (data_files/filter_dictsort.yml):

```
trunks:
  Fa0/1:
    action: add
    vlans: 10,20
  Fa0/2:
    action: only
    vlans: 10,30
  Fa0/3:
    action: delete
    vlans: 10
```

DICTSORT

Результат выполнения будет таким (интерфейсы упорядочены):

```
$ python cfg_gen.py templates/filter_dictsort.txt data_files/filter_dictsort.yml
interface Fa0/1
  switchport trunk allowed vlan add 10,20
interface Fa0/2
  switchport trunk allowed vlan 10,30
interface Fa0/3
  switchport trunk allowed vlan remove 10
```

JOIN

Фильтр join работает так же, как и метод join в Python.

С помощью фильтра join можно объединять элементы последовательности в строку, с опциональным разделителем между элементами.

JOIN

Пример шаблона templates/filter_join.txt с использованием фильтра join:

```
{% for intf, params in trunks | dictsort %}
interface
{{ intf }}

{% if params.action == 'add' %}
 switchport trunk allowed vlan add
{{ params.vlans | join(',') }}

{% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove
{{ params.vlans | join(',') }}

{% else %}
 switchport trunk allowed vlan
{{ params.vlans | join(',') }}

{% endif %}
{% endfor %}
```

JOIN

Файл с данными (data_files/filter_join.yml):

```
trunks:
  Fa0/1:
    action: add
    vlans:
      - 10
      - 20
  Fa0/2:
    action: only
    vlans:
      - 10
      - 30
  Fa0/3:
    action: delete
    vlans:
      - 10
```

JOIN

Результат выполнения:

```
$ python cfg_gen.py templates/filter_join.txt data_files/filter_join.yml
interface Fa0/1
  switchport trunk allowed vlan add 10,20
interface Fa0/2
  switchport trunk allowed vlan 10,30
interface Fa0/3
  switchport trunk allowed vlan remove 10
```

ТЕСТЫ

ТЕСТЫ

Кроме фильтров, Jinja также поддерживает тесты. Тесты позволяют проверять переменные на какое-то условие.

Jinja поддерживает большое количество встроенных тестов. Мы рассмотрим лишь несколько из них. Остальные тесты вы можете найти в документации.

Тесты, как и фильтры, можно создавать самостоятельно.

DEFINED

Тест defined позволяет проверить есть ли переменная в словаре данных.

Пример шаблона templates/test_defined.txt:

```
router ospf 1
{% if ref_bw is defined %}
 auto-cost reference-bandwidth
{{ ref_bw }}
{% else %}
 auto-cost reference-bandwidth 10000
{% endif %}
{% for networks in ospf %}
 network
{{ networks.network }} area {{ networks.area }}
{% endfor %}
```

DEFINED

Этот пример более громоздкий, чем вариант с использованием фильтра default, но этот тест может быть полезен в том случае, если, в зависимости от того, определена переменная или нет, нужно выполнять разные команды.

Файл с данными (data_files/test_defined.yml):

```
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

DEFINED

Результат выполнения:

```
$ python cfg_gen.py templates/test_defined.txt data_files/test_defined.yml
router ospf 1
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

ITERABLE

Тест iterable проверяет является ли объект итератором.

Благодаря таким проверкам, можно делать ответвления в шаблоне, которые будут учитывать тип переменной.

ITERABLE

Шаблон templates/test_iterable.txt (сделаны отступы, чтобы былы понятней ответвления):

```
{% for intf, params in trunks | dictsort %}
interface
{{ intf }}

{% if params.vlans is iterable %}

{% if params.action == 'add' %}
 switchport trunk allowed vlan add
{{ params.vlans | join(',') }}

{% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove
{{ params.vlans | join(',') }}

{% else %}
 switchport trunk allowed vlan
{{ params.vlans | join(',') }}

{% endif %}

{% else %}

{% if params.action == 'add' %}
 switchport trunk allowed vlan add
{{ params.vlans }}

{% elif params.action == 'delete' %}
```

ITERABLE

Файл с данными (data_files/test_iterable.yml):

```
trunks:
  Fa0/1:
    action: add
    vlans:
      - 10
      - 20
  Fa0/2:
    action: only
    vlans:
      - 10
      - 30
  Fa0/3:
    action: delete
    vlans: 10
```

ITERABLE

Обратите внимание на последнюю строку: vlans: 10. В данном случае, 10 уже не находится в списке и фильтр join в таком случае не работает. Но, засчет теста is iterable (в этом случае результат будет false), в этом случае шаблон уходит в ветку else.

ITERABLE

Результат выполнения:

```
$ python cfg_gen.py templates/test_iterable.txt data_files/test_iterable.yml
  interface Fa0/1
            switchport trunk allowed vlan add 10,20
        interface Fa0/2
            switchport trunk allowed vlan 10,30
        interface Fa0/3
            switchport trunk allowed vlan remove 10
```

Такие отступы получились из-за того, что в шаблоне используются отступы, но не установлено lstrip_blocks=True (он удалет пробелы и табы в начале строки).

SET

SET

Внутри шаблона можно присваивать значения переменным. Это могут быть новые переменные, а могут быть измененные значения переменных, которые были переданы шаблону.

Таким образом можно запомнить значение, которое, например, было получено в результате применения нескольких фильтров. И в дальнейшем использовать имя переменной, а не повторять снова все фильтры.

SET

Пример шаблона templates/set.txt, в котором выражение set используется чтобы задать более короткие имена параметрам:

```
{% for intf, params in trunks | dictsort %}

{% set vlans = params.vlans %}

{% set action = params.action %}

interface
{{ intf }}

{% if vlans is iterable %}

{% if action == 'add' %}
 switchport trunk allowed vlan add
{{ vlans | join(',') }}

{% elif action == 'delete' %}
 switchport trunk allowed vlan remove
{{ vlans | join(',') }}

{% else %}
 switchport trunk allowed vlan
{{ vlans | join(',') }}

{% endif %}

{% else %}
```

SET

Обратите внимание на вторую и третью строки:

```
{% set vlans = params.vlans %}
{% set action = params.action %}
```

Таким образом созданы новые переменные и дальше используются уже эти новые значения. Так шаблон выглядит понятней.

SET

Файл с данными (data_files/set.yml):

```
trunks:
  Fa0/1:
    action: add
    vlans:
      - 10
      - 20
  Fa0/2:
    action: only
    vlans:
      - 10
      - 30
  Fa0/3:
    action: delete
    vlans: 10
```

SET

Результат выполнения:

```
$ python cfg_gen.py templates/set.txt data_files/set.yml

interface Fa0/1
 switchport trunk allowed vlan add 10,20

interface Fa0/2
 switchport trunk allowed vlan 10,30

interface Fa0/3
 switchport trunk allowed vlan remove 10
```

INCLUDE

INCLUDE

Выражение include позволяет добавить один шаблон в другой.

Переменные, которые передаются как данные, должны содержать все данные и для основного шаблона, и для того, который добавлен через include.

INCLUDE

Шаблон templates/vlans.txt:

```
{% for vlan, name in vlans.items() %}
vlan
{{ vlan }}
 name
{{ name }}
{% endfor %}
```

INCLUDE

Шаблон templates/ospf.txt:

```
router ospf 1
 auto-cost reference-bandwidth 10000
{% for networks in ospf %}
 network
{{ networks.network }} area {{ networks.area }}
{% endfor %}
```

INCLUDE

Шаблон templates/bgp.txt:

```
router bgp {{ bgp.local_as }}
{% for ibgp in bgp.ibgp_neighbors %}
 neighbor
{{ ibgp }} remote-as {{ bgp.local_as }}
 neighbor
{{ ibgp }} update-source {{ bgp.loopback }}
{% endfor %}
{% for ebgp in bgp.ebgp_neighbors %}
 neighbor
{{ ebgp }} remote-as {{ bgp.ebgp_neighbors[ebgp] }}
{% endfor %}
```

INCLUDE

Шаблон templates/switch.txt использует созданные шаблоны ospf и vlans:

```
{% include 'vlans.txt' %}

{% include 'ospf.txt' %}
```

INCLUDE

Файл с данными, для генерации конфигурации (data_files/switch.yml):

```
vlans:
  10: Marketing
  20: Voice
  30: Management
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

INCLUDE

Результат выполнения скрипта:

```
$ python cfg_gen.py templates/switch.txt data_files/switch.yml
vlan 10
 name Marketing
vlan 20
 name Voice
vlan 30
 name Management

router ospf 1
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
```

Итоговая конфигурация получилась такой, как-будто строки из шаблонов ospf.txt и vlans.txt, находились в шаблоне switch.txt.

INCLUDE

Шаблон templates/router.txt:

```
{% include 'ospf.txt' %}

{% include 'bgp.txt' %}

logging
{{ log_server }}
```

В данном случае, кроме include, добавлена ещё одна строка в шаблон, чтобы показать, что выражения include могут идти вперемешку с обычным шаблоном.

INCLUDE

Файл с данными (data_files/router.yml):

```
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
bgp:
  local_as: 100
  loopback: lo100
  ibgp_neighbors:
    - 10.0.0.2
    - 10.0.0.3
  ebgp_neighbors:
    90.1.1.1: 500
    80.1.1.1: 600
log_server: 10.1.1.1
```

INCLUDE

Результат выполнения скрипта будет таким:

```
$ python cfg_gen.py templates/router.txt data_files/router.yml
router ospf 1
 auto-cost reference-bandwidth 10000
 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0

router bgp 100
 neighbor 10.0.0.2 remote-as 100
 neighbor 10.0.0.2 update-source lo100
 neighbor 10.0.0.3 remote-as 100
 neighbor 10.0.0.3 update-source lo100
 neighbor 90.1.1.1 remote-as 500
 neighbor 80.1.1.1 remote-as 600

logging 10.1.1.1
```

НАСЛЕДОВАНИЕ ШАБЛОНОВ

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Наследование шаблонов это очень мощный функционал, который позволяет избежать повторения одного и того же в разных шаблонов.

При использовании наследования различают:

- **базовый шаблон** - это шаблон, в котором описывается каркас шаблона.
  - в этом шаблоне могут находится любые обычные выражения или текст. Но, кроме того, в этом шаблоне определяются специальные **блоки** (**block**).
- **дочерний шаблон** - шаблон, который расширяет базовый шаблон, заполняя обозначенные блоки.
  - дочерние шаблоны могут переписывать или дополнять блоки, определенные в базовом шаблоне.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Пример базового шаблона templates/base_router.txt:

```
!
{% block services %}
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
{% endblock %}
!
no ip domain lookup
!
ip ssh version 2
!
{% block ospf %}
router ospf 1
 auto-cost reference-bandwidth 10000
{% endblock %}
!
{% block bgp %}
{% endblock %}
!
{% block alias %}
{% endblock %}
!
line con 0
 logging synchronous
 history size 100
line vty 0 4
 logging synchronous
 history size 100
```

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Обратите внимание на четыре блока, которые созданы в шаблоне:

```
{% block services %}
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
{% endblock %}
!
{% block ospf %}
router ospf 1
 auto-cost reference-bandwidth 10000
{% endblock %}
!
{% block bgp %}
{% endblock %}
!
{% block alias %}
{% endblock %}
```

Это заготовки для соответствующих разделов конфигурации. Дочерний шаблон, который будет использовать этот базовый шаблон как основу, может заполнять все блоки или только какие-то из них.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Дочерний шаблон templates/hq_router.txt:

```
{% extends "base_router.txt" %}

{% block ospf %}
{{ super() }}
{% for networks in ospf %}
 network {{ networks.network }} area {{ networks.area }}
{% endfor %}
{% endblock %}

{% block alias %}
alias configure sh do sh
alias exec ospf sh run | s ^router ospf
alias exec bri show ip int bri | exc unass
alias exec id show int desc
alias exec top sh proc cpu sorted | excl 0.00%__0.00%__0.00%
alias exec c conf t
alias exec diff sh archive config differences nvram:startup-config system:running-config
alias exec desc sh int desc | ex down
{% endblock %}
```

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Первая строка в шаблоне templates/hq_router.txt очень важна:

```
{% extends "base_router.txt" %}
```

Именно она говорит о том, что шаблон hq_router.txt будет построен на основе шаблона base_router.txt.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Внутри дочернего шаблона всё происходит внутри блоков. Засчет блоков, которые были определены в базовом шаблоне, дочерний шаблон может расширять родительский шаблон.

Обратите внимание, что те строки, которые описаны в дочернем шаблоне за пределами блоков, игнорируются.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

В базовом шаблоне три блока: ospf, bgp, alias. В дочернем шаблоне заполнены только два из них: ospf и alias.

В этом удобство наследования. Не обязательно заполнять все блоки в каждом дочернем шаблоне.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

При этом, блоки ospf и alias используются по-разному. В базовом шаблоне, в блоке ospf уже была часть конфигурации:

```
{% block ospf %}
router ospf 1
 auto-cost reference-bandwidth 10000
{% endblock %}
```

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Поэтому, в дочернем шаблоне есть выбор: использовать эту конфигурацию и дополнить её, или полностью переписать всё в дочернем шаблоне.

В данном случае, конфигурация дополняется.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Именно поэтому в дочернем шаблоне templates/hq_router.txt блок ospf начинается с выражения {{ super() }}:

```
{% block ospf %}
{{ super() }}

{% for networks in ospf %}
 network
{{ networks.network }} area {{ networks.area }}

{% endfor %}
{% endblock %}
```

НАСЛЕДОВАНИЕ ШАБЛОНОВ

{{ super() }} переносит в дочерний шаблон содержимое этого блока из родительского шаблона. Засчет этого, в дочерний шаблон перенесутся строки из родительского.

Выражение super не обязательно должно находится в самом начале блока. Оно может быть в любом месте блока. Содержимое базового шаблона, перенесется в то место, где находится выражение super.

НАСЛЕДОВАНИЕ ШАБЛОНОВ

В блоке alias просто описаны нужные alias. И, даже если бы в родительском шаблоне были какие-то настроки, они были бы затерты содержимым дочернего шаблона.

Файл с данными для генерации конфигурации по шаблону (data_files/hq_router.yml):

```
ospf:
  - network: 10.0.1.0 0.0.0.255
    area: 0
  - network: 10.0.2.0 0.0.0.255
    area: 2
  - network: 10.1.1.0 0.0.0.255
    area: 0
```

НАСЛЕДОВАНИЕ ШАБЛОНОВ

Результат выполнения будет таким:

```
$ python cfg_gen.py templates/hq_router.txt data_files/hq_router.yml
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
router ospf 1
 auto-cost reference-bandwidth 10000

 network 10.0.1.0 0.0.0.255 area 0
 network 10.0.2.0 0.0.0.255 area 2
 network 10.1.1.0 0.0.0.255 area 0
!
!
alias configure sh do sh
alias exec ospf sh run | s ^router ospf
alias exec bri show ip int bri | exc unass
alias exec id show int desc
alias exec top sh proc cpu sorted | excl 0.00%__0.00%__0.00%
alias exec c conf t
alias exec diff sh archive config differences nvram:startup-config system:running-config
alias exec desc sh int desc | ex down
!
line con 0
```