Python for network engineers

Release 1.0

Jan 01, 2021

Contents

1 Introduction **3**

About book . . . 3
Who is this book for? . . . 3
OS and Python requirements . . . 3
Examples . . . 4
Tasks . . . 4
Book formats . . . 4
Frequently Asked Questions (FAQ) . . . 4
Will there be a printed version of the book? . . . 4
Why is there no topic X in the book? . . . 4
How does this differ from the regular Python introductory book? . . . 4
Why is this book specifically for network engineers? . . . 5
Why Python? . . . 5
Acknowledgments . . . 6

2 Book resources **7**

Preparing the working environment . . . 7
Exercises . . . 7
Tests . . . 8
Further reading . . . 9

3 I. Python basics **11**

1. Preparing for work . . . 12
Working environment . . . 13
OS and editor . . . 15
Package management system Pip . . . 16
Virtual environment . . . 17
Python interpreter . . . 21
Further reading . . . 21
Exercises . . . 23

2. Using Git and Github . . . 24
Git fundamentals . . . 24
Displaying repository status in invitation . . . 25
Working with Git . . . 26
Additional features . . . 30
Github authentication . . . 34
Working with own repository . . . 35
Working with repository of tasks and examples . . . 39
Further reading . . . 41
Tasks . . . 42
3. Getting started with Python . . . 43
Python syntax . . . 43
Python interpreter. IPython . . . 45
IPython special commands . . . 50
Variables . . . 52
Tasks . . . 55
4. Python data types . . . 56
Numbers . . . 56
Strings . . . 59
List . . . 72
Dictionary . . . 77
Tuple . . . 86
Set . . . 87
Boolean values . . . 90
Types conversion . . . 91
Types checking . . . 93
Method chaining . . . 95
Sorting basics . . . 96
Further reading . . . 97
Tasks . . . 98
5. Basic scripts . . . 101
Executable file . . . 101
Passing arguments to the script (sys.argv) . . . 102
User input . . . 103
Tasks . . . 105
6. Control structures . . . 114
if/elif/else . . . 114
for . . . 120
while . . . 127
break, continue, pass . . . 128
for/else, while/else . . . 132
Working with try/except/else/finally . . . 134
Further reading . . . 139
Tasks . . . 141
7. Working with files . . . 144

File opening . . . 144
File reading . . . 145
File writing . . . 149
File closing . . . 153
with statement . . . 154
Examples of working with files . . . 156
Further reading . . . 162
Tasks . . . 163
8. Python basic examples . . . 167
Formatting lines with f-strings . . . 167
Variable unpacking . . . 172
List, dict, set comprehensions . . . 178
Further reading . . . 186

4 II. Code reuse 187
9. Functions . . . 188
Creation of functions . . . 188
Namespace. Scope of variables . . . 192
Function parameters and arguments . . . 194
Example of using variable length keyword arguments and unpacking arguments . . . 205
Further reading . . . 208
Tasks . . . 209
10. Useful functions . . . 217
print . . . 217
range . . . 220
sorted . . . 223
enumerate . . . 226
zip . . . 228
all . . . 231
any . . . 231
Anonymous function (lambda expression) . . . 232
map . . . 233
filter . . . 235
11. Modules . . . 237
Module import . . . 237
Create your own modules . . . 240
`if __name__ == "__main__"` . . . 242
Further reading . . . 244
Tasks . . . 246
12. Useful modules . . . 251
subprocess . . . 251
os . . . 256
ipaddress . . . 258
tabulate . . . 263
pprint . . . 267

Tasks . . . 272
13. Iterators, iterable and generators . . . 274
Iterable . . . 274
Iterators . . . 274
Generator . . . 277
Further reading . . . 278

5 III. Regular expressions 279
14. Regular expression (regex) syntax . . . 280
Regular expression syntax . . . 280
Character sets . . . 282
Repeating characters . . . 283
Special symbols . . . 287
Greedy qualifiers . . . 292
Grouping . . . 293
Parsing the output of 'show ip dhcp snooping' command using named groups . . . 295
Non-capturing group . . . 298
Repeating the captured result . . . 299
15. Module re . . . 301
Match object . . . 301
Search function . . . 307
Match function . . . 313
Finditer function . . . 314
Findall function . . . 319
Compile function . . . 321
Flags . . . 324
Function re.split . . . 328
Function re.sub . . . 329
Further reading . . . 331
Tasks . . . 332

6 IV. Data writing and transmission 337
16. Unicode . . . 338
Unicode standard . . . 338
Unicode in Python 3 . . . 339
Conversion between bytes and strings . . . 342
Examples of converting between bytes and strings . . . 343
Converting errors . . . 347
Further reading . . . 350
17. Working with CSV, JSON, YAML files . . . 351
Work with CSV files . . . 351
Work with JSON files . . . 357
Work with YAML files . . . 364
Further reading . . . 368
Tasks . . . 370

7 V. Working with network devices **377**
18. Connection to network devices . . . 378
Password input . . . 379
Module pexpect . . . 380
Example of pexpect use . . . 384
Module telnetlib . . . 388
Module paramiko . . . 397
Module netmiko . . . 402
Module scrapli . . . 410
Further reading . . . 422
Tasks . . . 424
19. Concurent connections to multiple devices . . . 432
Measure script execution time . . . 432
Processes and threads in Python (CPython) . . . 433
Number of threads . . . 434
Thread safety . . . 436
Module logging . . . 437
Module concurrent.futures . . . 439
Further reading . . . 455
Tasks . . . 457
20. Jinja2 configuration templates . . . 464
Getting started with Jinja2 . . . 464
Example of using Jinja . . . 465
Jinja2 template syntax . . . 467
Template inheritance . . . 491
Further reading . . . 496
Tasks . . . 497
21. Parsing command output with TextFSM . . . 502
Getting started with TextFSM . . . 502
TextFSM template syntax . . . 504
State rules . . . 506
Examples of TextFSM usage . . . 508
TextFSM CLI Table . . . 524
Further reading . . . 529
Tasks . . . 531

8 VI. Basics of object-oriented programming **535**
22. OOP basics . . . 536
OOP basics . . . 536
Class creation . . . 538
Method creation . . . 539
Parameter self . . . 541
Method `__init__` . . . 543
Class example . . . 544
Class namespace . . . 546

Class variables . . . 546
Tasks . . . 549
23. Special methods . . . 558
Underscore in names . . . 558
Methods `__str__`, `__repr__` . . . 562
Arithmetic operator support . . . 564
Protocols . . . 567
Tasks . . . 580
24. Inheritance . . . 585
Inheritance basics . . . 585
Tasks . . . 590

9 VII. Working with databases 595
25. Database operations . . . 596
SQL . . . 596
SQLite . . . 597
SQL basics (in sqlite3 CLI) . . . 599
Sqlite3 module . . . 619
Further reading . . . 644
Tasks . . . 645

10 VIII. Additional information 659
Testing tasks with the pyneng utility . . . 660
Where to solve tasks . . . 660
Installing the pyneng script . . . 660
pyneng utility . . . 660
Checking tasks with tests . . . 661
How to get answers . . . 662
pyneng output . . . 662
Tasks checking with tests . . . 665
Pytest basics . . . 665
Specifics of using pytest to check tasks . . . 670
argparse . . . 674
Nested parsers . . . 679
String formatting with `%` operator . . . 686
Naming convention . . . 687
Variable names . . . 687
Module and package names . . . 688
Function names . . . 688
Class names . . . 688
Underscore in names . . . 688
Underscore in name . . . 689
Two underscores . . . 691
Two underscores before name . . . 691
Two underscores before and after name . . . 692

Python 2.7 and Python 3.6 distinctions . . . 693
Unicode . . . 693
print function . . . 694
input instead of raw_input . . . 694
range instead of xrange . . . 695
Dictionary methods . . . 695
Variables unpacking . . . 696
Iterator instead of list . . . 697
subprocess.run . . . 697
Jinja2 . . . 697
Modules pexpect, telnetlib, paramiko . . . 698
Trivia . . . 698
Additional information . . . 698
Preparing Windows . . . 699
Installing Python 3.7 . . . 699
Cmder . . . 699
Installing Mu . . . 699
Tips for completing tasks on windows . . . 699
Preparing Linux . . . 702
Installing Python 3.7 on Debian 9 . . . 702
Virtual environment . . . 702
List of modules that need to be installed to complete tasks . . . 703

11 What's next **705**
You have ideas for the scripts you want to write . . . 705
Python for network equipment automation . . . 706
General Python . . . 707
Books . . . 707
Courses . . . 708
Coding challenges . . . 708
Podcasts . . . 708
Documentation . . . 708

12 Download PDF/Epub **709**

The book covers Python basics with examples and tasks built around networking topics.

On the one hand, this book is basic enough to be mastered by anyone, and on the other hand, covers all the main topics that will allow you to further grow on your own. This book is not intended to be an in-depth look at Python. The purpose of this book is to explain the basics of Python in clear language and provide an understanding of the necessary tools for practical use. Everything in the book is focused on network equipment and interaction with it. This immediately makes it possible to use the knowledge gained in the daily work of network engineers. All examples are shown using Cisco equipment as an example, but of course they apply to any other equipment.

Note: This book covers Python 3.7.

The book was written by Natasha Samoylenko. Translated from Russian by Aidar Khairullin.

1

Introduction

About book

From the one hand, book is basic enough so everyone can handle it, from the other hand, book covers all main topics which allow you to develop skill independently in the future. Python deep dive is not a goal of this book. The goal is to explain Python basics in plain language and provide understanding of necessary tools for practical usage. Everything in this book is focused on network equipment and interaction with it. It right away gives opportunity to use knowledge gained at the course in network engineers daily work. All shown examples are based on Cisco equipment but, of course, they could be applied to any other equipment.

Who is this book for?

For network engineers with or without programming experience. All examples and homework will be formed with a focus on network equipment. This book will be useful for network engineers who want to automate their daily basis routine tasks and want start coding but don't know how to approach this. Still haven't decided whether it worth reading this book? Read feedbacks.

OS and Python requirements

All examples and terminal outputs in the book are shown on Debian Linux. Python 3.7 is used in this book but for the majority of examples Python 3.x will be enough. Only some examples requires Python version higher than 3.6. It always explicitly indicated and generally concerns some additional features.

Examples

All examples from the book resides in repository. All examples have educational purpose. It means they not necessarily show the best solution since they are based on information which was covered in previous chapters. Moreover, often enough the examples in chapters are developing in tasks. In other words, in tasks you have to create better, more universal and, in general, more proper version of code. It's better to write code from the book on your own or at least download examples and try to modify them. So the information will be better remembered. If you don't have this possibility, for example when you read book on road, it's better to repeat examples later on your own. In any case, it's necessary to do tasks manually.

Tasks

All tasks and auxiliary files can be downloaded from the same repository, where code examples are located.

Book formats

Book is available in PDF and Epub formats. Both of them are being updated automatically.

Frequently Asked Questions (FAQ)

Will there be a printed version of the book?

No, there will be no printed version. The book has existed in some form since late 2015 (in Russian). All this time the book has been changing. I love this opportunity to change the book, write something differently.

Why is there no topic X in the book?

There are still a huge number of useful topics and it is simply impossible to fit all of them into one book. Of course, each reader has priorities and it seems that this particular module is very necessary for everyone, but there are a lot of such topics/modules. Globally, nothing will change in the book, new topics will not be added.

How does this differ from the regular Python introductory book?

The main differences are three:

- The basics part is rather brief
- Implies a certain domain of knowledge (network-based equipment)
- All examples are focused on network equipment, as far as possible

Why is this book specifically for network engineers?

There are several reasons:

- Network engineers already have experience in IT and some of concepts are familiar to them and it is likely that some programming basics will be familiar to most. This means that it will be much easier to deal with Python
- Working in CLI and writing scripts is unlikely to frighten them
- Network engineers have a familiar knowledge domain on which to build examples and tasks

If you tell on abstract examples “about cats and bunnies”, it is one thing. But when you have the ability to use ideas from subject area in examples, things get easier, you get concrete ideas about how to improve a program, a script. And when a person tries to improve it, they start to deal with something new - it’s a very powerful way to move forward.

Why Python?

The reasons are as follows:

- In the context of network equipment, Python is often used now
- Some equipment has Python embedded or has an API that supports Python
- Python is simple enough to learn (of course, it is relatively, and another language may seem simpler but it is rather to be because of experience with the language than because Python is complex)
- With Python you will not quickly reach the limits of language capabilities
- Python can be used not only to write scripts but also to develop applications. Of course, this is not the task of this book but at least you will spend your time on a language that will allow you to go further than simple scripts
- For example GNS3 is written on Python

And one more point - in the context of book, Python should not be seen as the only correct variant nor as the “correct” language. No, Python is just a tool like a screwdriver and we learn to use it for specific tasks. That is, there is no ideological background here, no “only Python” and no worship especially. It is strange to worship a screwdriver :-) Everything is simple - there is a good and convenient tool that will approach different tasks. He’s not the best language at all and he’s not

the only language at all. Start with it and then you can choose something else if you want to - that knowledge will still be there.

Acknowledgments

Thank you to all who expressed interest in the first announcement of the course - your interest confirmed that someone would need it.

Pavel Pasynok, thank you for agreeing to course. It's been interesting working with you and it's given me an incentive to finish the course and I'm particularly glad that the knowledge that you've learned from course has found practical application.

Alexey Kirillov, thank you very much :-) I have always been able to discuss with you any question on course. You helped me maintain my motivation and not get in a muddle. Communicating with you has inspired me to continue, especially in difficult moments. Thank you for your inspiration, positive emotions and support!

Thanks to all those who wrote comments on book - thanks to you now book not only has fewer typographical errors and typos, but also the contents of book have improved.

Thanks to all participants of online course - thanks to your questions book has become much better.

Slava Skorokhod, thank you so much for volunteering to be an editor - number of errors is now going to zero :-)

2 Book resources

Resources that will come in handy during the learning process:

- Repository with examples and tasks
- answers
- Write about errors/inaccuracies in the book
- Write about errors/inaccuracies in the tasks

Preparing the working environment

To complete tasks, you can use several options:

- take a prepared virtual machine vmware or vagrant (virtualbox)
- prepare a virtual machine yourself
- use one of the cloud services
- work without creating a virtual machine

More about these options *Working environment*.

Exercises

Repository with examples and tasks

All tasks and auxiliary files can be downloaded from the repository. Tasks are duplicated in the book solely for a convenient overview of all tasks in the section. Since all supporting files, code, and tests are in the repository, it is best to do tasks in a copy of the repository. How to make a copy is described in section 2. Using Git and GitHub.

Sometimes, a certain section can be especially difficult, in this case, you can stop at a minimum of tasks. This will allow you to move on and not abandon your studies. Later you can come back and finish the tasks. In general, of course, it is better to do all the tasks, since practice is the only way to properly learn the topic, but sometimes it is better to do fewer tasks and continue studying than to get stuck on one topic and abandon everything.

Chapter	Minimum	All tasks
04_data_structures	4.1, 4.2, 4.3, 4.6	4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8
05_basic_scripts	5.1, 5.1a, 5.2, 5.2a	5.1, 5.1a, 5.1b, 5.1c, 5.1d, 5.2, 5.2a, 5.3, 5.3a
06_control_structures	6.1, 6.2, 6.3	6.1, 6.2, 6.2a, 6.2b, 6.3
07_files	7.1, 7.2, 7.3	7.1, 7.2, 7.2a, 7.2b, 7.2c, 7.3, 7.3a, 7.3b
09_functions	9.1, 9.1a, 9.2, 9.2a, 9.3	9.1, 9.1a, 9.2, 9.2a, 9.3, 9.3a, 9.4
11_modules	11.1, 11.2	11.1, 11.2
12_useful_modules	12.1, 12.2	12.1, 12.2, 12.3
15_module_re	15.1, 15.2, 15.3, 15.4	15.1, 15.1a, 15.1b, 15.2, 15.2a, 15.3, 15.4, 15.5
17_serialization	17.1, 17.2, 17.3	17.1, 17.2, 17.3, 17.3a, 17.3b, 17.4
18_ssh_telnet	18.1, 18.1a, 18.2, 18.2a, 18.2b, 18.3	18.1, 18.1a, 18.1b, 18.2, 18.2a, 18.2b, 18.2c, 18.3
19_concurrent_connections	19.1, 19.2, 19.3	19.1, 19.2, 19.3, 19.3a, 19.4
20_jinja2	20.1, 20.2, 20.3	20.1, 20.2, 20.3, 20.4, 20.5, 20.5a
21_textfsm	21.1, 21.1a, 21.2, 21.3, 21.4	21.1, 21.1a, 21.2, 21.3, 21.4, 21.5
22_oop_basics	22.1, 22.1a, 22.1b, 22.2, 22.2a	22.1, 22.1a, 22.1b, 22.1c, 22.1d, 22.2, 22.2a, 22.2b, 22.2c
23_oop_special_methods	23.1, 23.1a, 23.2	23.1, 23.1a, 23.2, 23.3, 23.3a
24_oop_inheritance	24.1, 24.2, 24.2a	24.1, 24.1a, 24.2, 24.2a, 24.2b, 24.2c, 24.2d
25_db	25.1, 25.2, 25.3	25.1, 25.2, 25.3, 25.4, 25.5, 25.5a, 25.6

Tests

Repository with examples and tasks

There are automated tests in the repository for checking assignments. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pyneng.

Further reading

Almost every book chapter has subchapter “Further reading” which includes useful materials and references on the subject, plus references to official documentations. Moreover, I prepared a collection of resources on “Python for network engineers” topic where you can find a lot of useful articles, books, video courses and podcasts.

3

I. Python basics

First part of the book is dedicated to Python basics. It covers:

- Python data types
- How to create basic scripts
- Control structures
- Working with files

1. Preparing for work

In order to start working with Python, you need to decide on a few things:

- operating system
- editor
- Python version

This book uses Debian Linux (on other OS the output may differ slightly) and Python 3.7.

Working environment

To complete tasks, you can use several options:

- prepare a virtual machine
- use one of the cloud services
- work without creating a virtual machine

Preparing the virtual machine/host

- tips for *Preparing Windows*
- tips for *Preparing Linux*

List of modules to be installed:

```
pip install pytest pytest-clarity pyyaml tabulate jinja2 textfsm pexpect netmiko␣
↪graphviz
```

You also need to install graphviz (example for debian):

```
apt-get install graphviz
```

Cloud service

Another option is to use one of the following services:

- repl.it – this service provides an online Python interpreter as well as a graphics editor.
- PythonAnywhere - a separate virtual machine. In the free version you can work only from the command line, that is, there is no graphical text editor

Network equipment

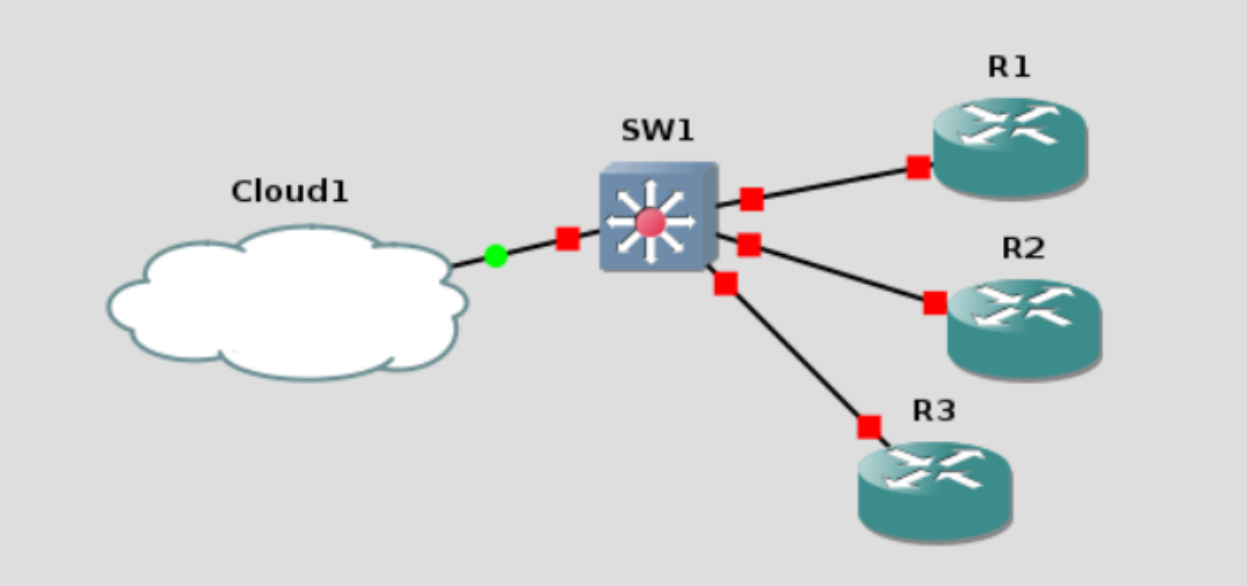
For the 18th section of the book, you need to prepare virtual or real network equipment.

All examples and tasks in which network equipment is used use the same number of devices: three routers with the following basic settings:

- user: cisco
- password: cisco
- password for enable mode: cisco
- SSH version 2 (version 2 is required), Telnet

- Router IPs: 192.168.100.1, 192.168.100.2, 192.168.100.3
- IP addresses must be accessible from the virtual machine on which you perform tasks and can be assigned on physical/logical/loopback interfaces

The topology can be arbitrary. Example topology:



Basic config:

```
hostname R1
!
no ip domain lookup
ip domain name pyneng
!
crypto key generate rsa modulus 1024
ip ssh version 2
!
username cisco password cisco
enable secret cisco
!
line vty 0 4
 logging synchronous
 login local
 transport input telnet ssh
```

On some interface, you need to configure an IP address

```
interface ...
 ip address 192.168.100.1 255.255.255.0
```

Aliases (optional)

```
!
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
alias exec bgp sh run | s ^router bgp
```

Optionally, you can configure the EEM applet to display the commands that the user enters:

```
!
event manager applet COMM_ACC
 event cli pattern ".*" sync no skip no occurs 1
 action 1 syslog msg "User $_cli_username entered $_cli_msg on device $_cli_host "
!
```

OS and editor

You can choose any OS and any editor but it is better to use Python version 3.7 because book uses this version. All examples in book were run on Debian, other operating systems may have a slightly different output. You can use Linux, macOS or Windows to perform tasks from a book.

You can select any text editor or IDE that supports Python. Generally, working with Python requires minimal editor settings and often the editor recognizes Python by default.

Mu editor

Mu editor is a simple Python editor for beginner programmers.

Mu has clean and simple user interface. It has important features such as checking code against PEP 8 and debugger. Plus, Mu runs on different operating systems (macOS, Windows, Linux).

Note: Mu tutorials

IDE PyCharm

PyCharm — is an integrated development environment for Python. For beginners it may be difficult option due to the large number of settings but it depends on personal preferences. Pycharm supports a huge number of features, even in free version.

Pycharm is a great IDE but I think it's a little difficult for beginners. I wouldn't recommend using it if you're not familiar with it and you're just starting to learn Python. You can always switch to it after book but for now it's better to try something else.

Geany

Geany - is a text editor that supports different programming languages, including Python. It is also a cross-platform editor and supports Linux, macOS, and Windows.

Note: Editor variants above are given for example, they can be replaced by any text editor that supports Python.

Package management system Pip

Pip will be used to install Python packages. It is a package management system used to install packages from Python Package Index (Pypi). Most likely, if you already have Python installed, pip is installed.

Check pip version:

```
$ pip --version
pip 19.1.1 from /home/vagrant/venv/pyneng-py3-7/lib/python3.7/site-packages/pip␣
↪(python 3.7)
```

If command failed, pip is not installed. Pip installation is described in documentation

Module installation

Command to install modules `pip install`:

```
$ pip install tabulate
```

To delete package:

```
$ pip uninstall tabulate
```

In addition, it is sometimes necessary to update package:

```
$ pip install --upgrade tabulate
```

pip or pip3

Depending on how Python is installed and configured in system it may be necessary to use pip3 instead of pip. To check which option is used, you should execute command `pip --version`.

A version where pip corresponds to Python 2.7:

```
$ pip --version
pip 9.0.1 from /usr/local/lib/python2.7/dist-packages (python 2.7)
```

A version where pip3 corresponds to 3.7:

```
$ pip3 --version
pip 19.1.1 from /home/vagrant/venv/pyneng-py3-7/lib/python3.7/site-packages/pip␣
↪(python 3.7)
```

If system uses pip3, then every time a Python module is installed in book it will be necessary to replace pip with pip3.

Alternatively, call pip:

```
$ python3.7 -m pip install tabulate
```

Thus, it is always clear for which version of Python the package is installed.

Virtual environment

Virtual environments:

- Allow different projects to be isolated from each other
- Packages that are needed by different projects are in different places - for example, if one project requires 1.0 package and another project requires the same package but version 3.1, they will not interfere with each other
- Packages that are installed in virtual environments do not impact on global packages

---

Note: Python has several options for creating virtual environments. You can use any of them. To start with, you can use virtualenvwrapper and then eventually you can figure out which option you prefer.

---

virtualenvwrapper

Virtual environments are created with virtualenvwrapper.

Installing virtualenvwrapper with pip:

```
$ sudo pip3.7 install virtualenvwrapper
```

After installation, in . bashrc file in current user's home folder, you need to add several lines:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.7
export WORKON_HOME=~/venv
. /usr/local/bin/virtualenvwrapper.sh
```

If you are using a command interpreter other than bash, see if it is supported in virtualenvwrapper documentation. Environment variable VIRTUALENVWRAPPER_PYTHON points to Python command line binary file, WORKON_HOME – points to location of virtual environments. The third line indicates location of script installed with virtualenvwrapper package. To start virtualenvwrapper.sh script work with virtual environments, bash must be restarted.

Restart command interpreter:

```
$ exec bash
```

This may not always be the right option. More on Stack Overflow.

Working with virtual environments

Creating a new virtual environment in which Python 3.7 is used by default:

```
$ mkvirtualenv --python=/usr/local/bin/python3.7 pyneng
New python executable in PyNEng/bin/python
Installing distribute.....................done.
Installing pip..............done.
(pyneng)$
```

The name of virtual environment is shown in parentheses before standard invitation. That means you're inside it. Virtualenvwrapper uses Tab to autocomplete name of virtual environment. This is particularly useful when there are many virtual environments. Now "pyneng" directory was created in directory specified in environment variable WORKON_HOME:

```
(pyneng)$ ls -ls venv
total 52
....
4 -rwxr-xr-x 1 nata nata   99 Sep 30 16:41 preactivate
```

(continues on next page)

(continued from previous page)

```
4 -rw-r--r-- 1 nata nata   76 Sep 30 16:41 predeactivate
4 -rwxr-xr-x 1 nata nata   91 Sep 30 16:41 premkproject
4 -rwxr-xr-x 1 nata nata  130 Sep 30 16:41 premkvirtualenv
4 -rwxr-xr-x 1 nata nata  111 Sep 30 16:41 prermvirtualenv
4 drwxr-xr-x 6 nata nata 4096 Sep 30 16:42 pyneng
```

Exit virtual environment:

```
(pyneng)$ deactivate
$
```

To move to created virtual environment, you should run “workon” command:

```
$ workon pyneng
(pyneng)$
```

If you want to go from one virtual environment to another, you don’t need to do deactivate, you can go directly through “workon”:

```
$ workon Test
(Test)$ workon pyneng
(pyneng)$
```

If you want to remove virtual environment, you should use “rmvirtualenv”:

```
$ rmvirtualenv Test
Removing Test...
$
```

See which packages are installed in a virtual environment using “lssitepackages”:

```
(pyneng)$ lssitepackages
ANSI.py                                pexpect-3.3-py2.7.egg-info
ANSI.pyc                               pickleshare-0.5-py2.7.egg-info
decorator-4.0.4-py2.7.egg-info         pickleshare.py
decorator.py                           pickleshare.pyc
decorator.pyc                          pip-1.1-py2.7.egg
distribute-0.6.24-py2.7.egg            pxssh.py
easy-install.pth                       pxssh.pyc
fdpexpect.py                           requests
fdpexpect.pyc                          requests-2.7.0-py2.7.egg-info
FSM.py                                 screen.py
FSM.pyc                                screen.pyc
IPython                                setuptools.pth
```

(continues on next page)

(continued from previous page)

```
ipython-4.0.0-py2.7.egg-info           simplegeneric-0.8.1-py2.7.egg-info
ipython_genutils                       simplegeneric.py
ipython_genutils-0.1.0-py2.7.egg-info  simplegeneric.pyc
path.py                                test_path.py
path.py-8.1.1-py2.7.egg-info           test_path.pyc
path.pyc                               traitlets
pexpect                                traitlets-4.0.0-py2.7.egg-info
```

Built-in venv module

Starting from version 3.5, it is recommended that Python use venv to create virtual environments:

```
$ python3.7 -m venv new/pyneng
```

Python or python3 can be used instead of python 3.7, depending on how Python 3.7 is installed. This command creates specified directory and all necessary subdirectories within it if they have not been created.

Command creates the following directory structure:

```
$ ls -ls new/pyneng
total 16
4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 21 14:50 bin
4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 21 14:50 include
4 drwxr-xr-x 3 vagrant vagrant 4096 Aug 21 14:50 lib
4 -rw-r--r-- 1 vagrant vagrant   75 Aug 21 14:50 pyvenv.cfg
```

To move to a virtual environment, you should execute command:

```
$ source new/pyneng/bin/activate
```

To exit virtual environment, use command “deactivate”:

```
$ deactivate
```

More about the venv module in documentation.

Package installation

For example, let’s install simplejson package in virtual environment.

```
(pyneng)$ pip install simplejson
...
```

(continues on next page)

(continued from previous page)

```
Successfully installed simplejson
Cleaning up...
```

If you open Python interpreter and import simplejson, it is available and there are no errors:

```
(pyneng)$ python
>>> import simplejson
>>> simplejson
<module 'simplejson' from '/home/vagrant/venv/pyneng-py3-7/lib/python3.7/site-
↪packages/simplejson/__init__.py'>
>>>
```

But if you exit from virtual environment and try to do the same thing, there is no such module:

```
(pyneng)$ deactivate

$ python
>>> import simplejson
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'simplejson'
>>>
```

Python interpreter

Before you start, check that when you call Python interpreter, the output is:

```
$ python
Python 3.7.3 (default, May 13 2019, 15:44:23)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Output shows that Python 3.7 is set. Invitation >>>, this is a standard invitation from Python interpreter. Interpreter call is executed by python command and to exit you need to type `quit()`, or press Ctrl+D.

Note: Book will use ipython instead of standard Python interpreter

---

Further reading

Documentation:

- Python Setup and Usage
- pip
- venv
- virtualenvwrapper

Editors and IDE:

- PythonEditors
- IntegratedDevelopmentEnvironments
- VIM and Python - a Match Made in Heaven

Exercises

Task 1.1

The only task in this section is preparation for work.

To do that:

- Decide which OS you want to use:
- Install Python 3.7. Verify that Python and pip are installed
- Create a virtual environment
- Choose the editor

2. Using Git and Github

Warning: This is a difficult section to understand. Especially because this section is at the very beginning of the book. You can skip it, but I would highly recommend trying to understand the basics of git/Github and use it to store tasks.

There are a lot of tasks in book and you have to store them somewhere. One option is to use Git and Github to do this. Of course, there are other ways to do this but Github can be used for other things in future. Tasks and examples from book are in a separate repository on Github. They can be downloaded as a zip archive but it is better to work with repository using Git, then you can see changes made and easily update repository. If this is the first time working with Git and especially if this is the first version control system you work with, there are a lot of information, so this chapter focuses on practical side of question and it says:

- How to start using Git and Github
- How to perform basic setup
- How to view information and/or changes

There will be no much theory in this subsection but references to useful resources are listed. Try doing all basic settings for tasks and then continue reading the book. And at the end, when basic work with Git and Github is already routine, read more about them. What could Git be useful for:

- to store configurations and all configuration changes
- to store documentation and all its versions
- to store schemes and all its versions
- to store code and its versions

Github allows you to centrally store all above items, but it should be taken into account that these resources will be available to others as well. Github also has private repositories, but even these probably should not contain information such as passwords. Of course, main use of Github is to place code of various projects. In addition, Github can be also used to:

- host for your website (GitHub Pages)
- Hosting for online presentations and a tool to create them (GitPitch)
- together with GitBook, it is also a platform for publishing books, documentation, etc

Git fundamentals

Git is a distributed version control system (Version Control System, VCS) that is widely used and released under GNU GPL v2 license. It can:

- track changes in files;
- store multiple versions of the same file;
- cancel changes made;
- record who made changes and when.

Git stores changes as a snapshot of entire repository. This snapshot is created after each “commit” command.

Git installation:

```
$ sudo apt-get install git
```

Git initial setup

To start working with Git you need to specify user name and e-mail that will be used to synchronize local repository with Github repository:

```
$ git config --global user.name "username"
$ git config --global user.email "username.user@example.com"
```

See Git settings:

```
$ git config --list
```

Repository initialization

Repository is initialized using “git init” command:

```
[~/tools/first_repo]
$ git init
Initialized empty Git repository in /home/vagrant/tools/first_repo/.git/
```

After executing this command, current directory creates .git folder containing service files needed for Git.

Displaying repository status in invitation

This is an additional functionality that is not required to work with Git but is very helpful in this regard. When working with Git it is very convenient when you can immediately determine whether you are in a regular directory or in a Git repository. In addition, it would be good to understand status of current repository. To do this, you need to install a special utility that will show status of repository. To install utility, copy its repository to user’s home directory under which you work:

```
cd ~
git clone https://github.com/magicmonty/bash-git-prompt.git .bash-git-prompt --
↪depth=1
```

And then add to the end of ~/.bashrc file such lines:

```
GIT_PROMPT_ONLY_IN_REPO=1
source ~/.bash-git-prompt/gitprompt.sh
```

To apply changes, restart bash:

```
exec bash
```

In my configuration command line invitation is spread over several lines, so you will have a different one. Please note that additional information appears when you move to repository.

Now, if you're in a regular catalog, invitation is like this:

```
[~]
vagrant@jessie-i386:
$
```

If you go to Git repository:

```
[~]
vagrant@jessie-i386:
$ cd tools/first_repo/

[~/tools/first_repo]
vagrant@jessie-i386:  [master L|✔]
```

Working with Git

There are a few basic commands you need to know to work with Git.

git status

When working with Git it is important to understand current status of repository. For this purpose Git has a `git status` command

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|✔]
13:02 $ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Git reports that we are in master branch (this branch is auto-created and used by default) and that it has nothing to commit. Git also offers to create or copy files and then use `git add` command to start Git tracking them.

Create README file and add "test" line to it

```
$ vi README
$ echo "test" >> README
```

After that, invitation looks like this

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|…2]
```

Invitation shows that there are two untracked files:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|…2]
13:14 $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .README.un~
        README

nothing added to commit but untracked files present (use "git add" to track)
```

Two files came out because I have undo-files configured for Vim. These are special files that allow you to undo changes not only in current file session but also in the previous sessions. Note that Git reports there are files that it does not track and tells you using which command you can start tracking.

File .gitignore

Undo-file .README.un~ is a special file that does not need to be added to repository. Git has option to specify which files or directories to ignore. To do this, you need to create appropriate templates in `.gitignore` file in repository directory.

To make Git ignore Vim undo-files you can add such a line to .gitignore file

```
*.un~
```

This means that Git must ignore all files that end with ".un~".

After that, git status shows

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|…2]
13:33 $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        README

nothing added to commit but untracked files present (use "git add" to track)
```

Note that there is no .README.un~ file in the output. Once a file was added to repository .gitignore, files that are listed in it are being ignored.

git add

Command `git add` is used to start Git tracking files.

You can specify that you want to track a particular file

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|…2]
13:33 $ git add README
```

Or all files

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|●1…1]
13:36 $ git add .
```

Git status output

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|●2]
13:36 $ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   README
```

Now files are in a section called “Changes to be committed”.

git commit

After all necessary files have been added in staging, you can commit changes. Staging is a collection of files that will be added to the next commit. Command `git commit` has only one mandatory parameter - flag “-m”. It allows you to specify a message for this commit.

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|●2]
13:37 $ git commit -m "First commit. Add .gitignore and README files"
[master (root-commit) ef84733] First commit. Add .gitignore and README files
 2 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 README
```

After that, git status displays

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|✔]
13:47 $ git status
On branch master
nothing to commit, working directory clean
```

Phrase “nothing to commit, working directory clean” indicates that there are no changes to add to Git or to commit.

Additional features

git diff

Command `git diff` allows you to see the difference between different states. For example, README and .gitignore files have been changed in repository.

Command `git status` shows that both files have been changed

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|+ 2]
13:53 $ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .gitignore
        modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
```

Command `git diff` command shows what changes have been made since last commit

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|+ 2]
13:53 $ git diff
diff --git a/.gitignore b/.gitignore
index 8eee101..07aab05 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,2 @@
 *.un~
-
+*.pyc
diff --git a/README b/README
index 2e7479e..79a508e 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
 First try
+
+Additional comment
```

If you add changes made to staging via `git add` command and run `git diff` again, it will show nothing

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|+ 2]
13:54 $ git add .

[~/tools/first_repo]
vagrant@jessie-i386: [master L|● 2]
13:57 $ git diff
```

To show the difference between staging and last commit, add parameter `--staged`

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|● 2]
13:57 $ git diff --staged
diff --git a/.gitignore b/.gitignore
index 8eee101..07aab05 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,2 @@
 *.un~
-
+*.pyc
diff --git a/README b/README
index 2e7479e..79a508e 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
 First try
+
+Additional comment
```

Commit changes

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|● 2]
13:59 $ git commit -m "Update .gitignore and README"
[master 58bb8ce] Update .gitignore and README
 2 files changed, 3 insertions(+), 1 deletion(-)
```

git log

Command `git log` command shows when last changes were made

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|✔]
14:00 $ git log
commit 58bb8cecbc08a8be76288e96b06d6a875f91a9b1
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 14:00:53 2017 +0000

    Update .gitignore and README

commit ef8473307e0a119496ef154e0bcaff703b1f8a71
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 13:47:30 2017 +0000

    First commit. Add .gitignore and README files
```

By default, command displays all commits starting from the nearest time. With help of additional parameters it is possible not only to look at information about commits but also what changes have been made.

Flag `-p` allows you to display the differences that have been made by each commit

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|✔]
14:02 $ git log -p
commit 58bb8cecbc08a8be76288e96b06d6a875f91a9b1
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 14:00:53 2017 +0000

    Update .gitignore and README

diff --git a/.gitignore b/.gitignore
index 8eee101..07aab05 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,2 @@
 *.un~
-
+*.pyc
diff --git a/README b/README
index 2e7479e..79a508e 100644
--- a/README
+++ b/README
@@ -1 +1,3 @@
 First try
+
+Additional comment

commit ef8473307e0a119496ef154e0bcaff703b1f8a71
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 13:47:30 2017 +0000

    First commit. Add .gitignore and README files

diff --git a/.gitignore b/.gitignore
new file mode 100644
index 0000000..8eee101
--- /dev/null
+++ b/.gitignore
@@ -0,0 +1,2 @@
+*.un~
+
diff --git a/README b/README
new file mode 100644
```

Shorter output option can be displayed with flag `--stat`

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L|✔]
14:05 $ git log --stat
commit 58bb8cecbc08a8be76288e96b06d6a875f91a9b1
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 14:00:53 2017 +0000

    Update .gitignore and README

 .gitignore | 2 +-
 README     | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)

commit ef8473307e0a119496ef154e0bcaff703b1f8a71
Author: pyneng <pyneng.course@gmail.com>
Date:   Fri May 26 13:47:30 2017 +0000

    First commit. Add .gitignore and README files

 .gitignore | 2 ++
 README     | 1 +
 2 files changed, 3 insertions(+)
```

Github authentication

In order to start working with GitHub, you need to register on it. It is better to use SSH key authentication to work safely with Github.

Generation of a new SSH key (use e-mail that is linked to Github):

```
$ ssh-keygen -t rsa -b 4096 -C "github_email@gmail.com"
```

On all questions, just press Enter. It is more secure to use passphrase but if you press Enter when asked then passphrase will not be requested from you permanently during operations with repository.

Start SSH agent:

```
$ eval "$(ssh-agent -s)"
```

Add key to SSH agent:

```
$ ssh-add ~/.ssh/id_rsa
```

Add SSH key to Github

To add a key you have to copy it. For example, you can display key with cat to copy it:

```
$ cat ~/.ssh/id_rsa.pub
```

After copying, go to Github. When you are on any Github page, in upper right-hand corner click on picture of your profile and select “Settings” in drop down list. In list on the left, select field “SSH and GPG keys”. Then press “New SSH key” and in “Title” field write key name (for example “Home”) and in field “Key” insert the content that was copied from file ~/. ssh/id_rsa.pub.

Note: If Github requests a password, enter your account password on Github.

To check if everything has been successful, try executing command `ssh -T git@github.com`.

The output should be as follows:

```
$ ssh -T git@github.com
Hi username! You've successfully authenticated, but GitHub does not provide shell␣
↪access.
```

Now you are ready to work with Git and Github.

Working with own repository

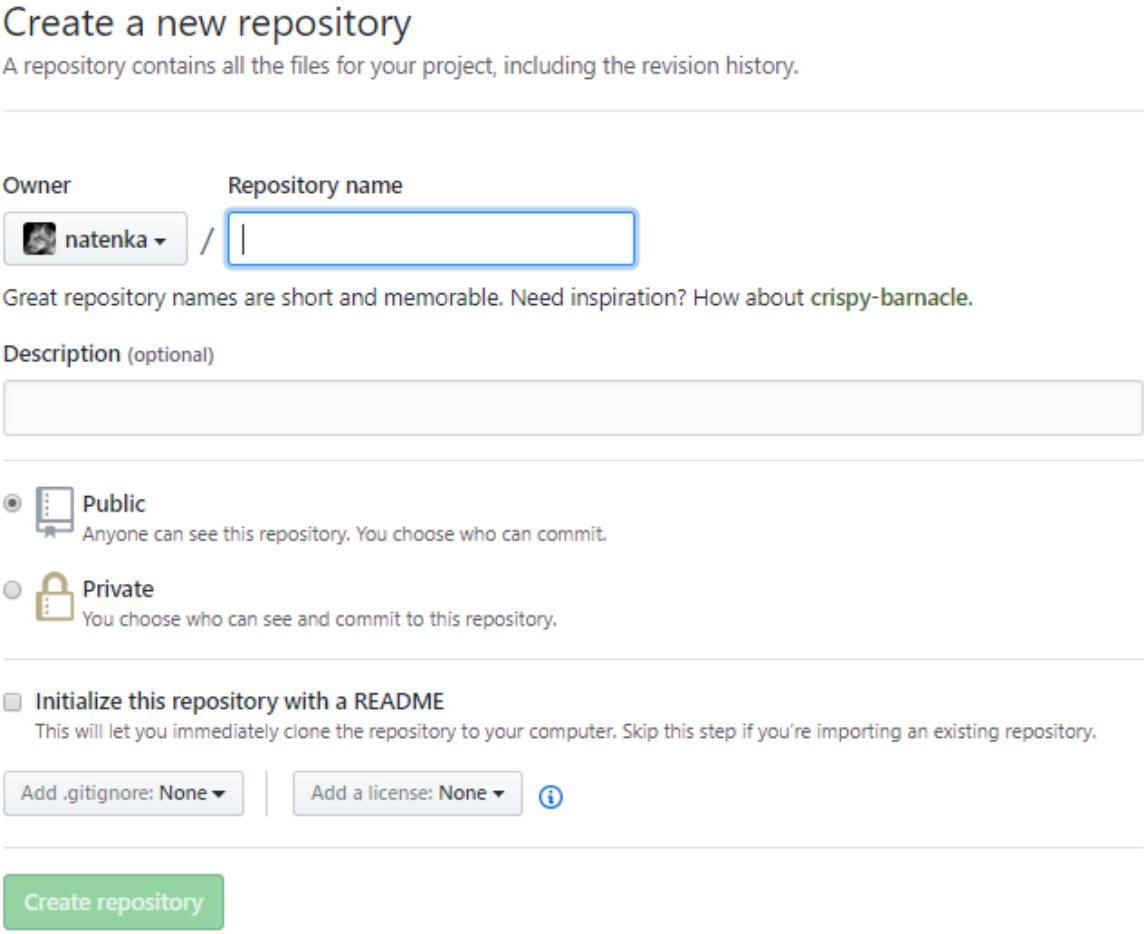
This chapter covers how to work with repository on your local machine.

Creating a Github repository

To create a Github repository you need:

- log in to GitHub
- In upper right corner press plus and select “New repository” to create a new repository
- Name of repository should be entered in window that appears

You can put “Initialize this repository with a README”. This will create a README.md file that only contains repository name.



Cloning a Github repository

To work locally with repository, it should be cloned.

Use `git clone` command to clone repository:

```
$ git clone ssh://git@github.com/pyneng/online-2-natasha-samoylenko.git
Cloning into 'online-2-natasha-samoylenko'...
remote: Counting objects: 241, done.
remote: Compressing objects: 100% (191/191), done.
remote: Total 241 (delta 43), reused 239 (delta 41), pack-reused 0
Receiving objects: 100% (241/241), 119.60 KiB | 0 bytes/s, done.
Resolving deltas: 100% (43/43), done.
Checking connectivity... done.
```

Compared to this command, you need to change:

- `pyneng user name` for your Github user name

- `online-2-natasha-samoylenko` repository name for your Github repository

As a result, in current directory in which `git clone` was executed, a directory with name of repository will appear, in my case - "online-2-natasha-samoylenko". This directory now contains the contents of Github repository.

Working with repository

The previous command not only copied repository to use it locally, but also configured Git accordingly:

- Folder .git was created
- All repository data is downloaded
- Downloaded all changes that were in repository
- Github repository is configured as a remote for local repository

Now you have a complete local Git repository where you can work. Typically, sequence of steps will be as follows:

- Before starting, synchronize local content with Github using `git pull` command
- Modifying repository files
- Adding modified files to staging with "git add" command
- Commit changes using `git commit` command
- Transferring local changes to Github repository with `git push` command

When working with tasks at work and at home, it is necessary to pay special attention to first and last step:

- The first step is to update local repository
- The last step - load changes to Github

Synchronizing local repository with remote repository

All commands are executed inside repository directory (in example above - online-2-natasha-samoylenko).

If contents of local repository are the same as those of remote repository, output will be:

```
$ git pull
Already up-to-date.
```

If there were changes, output would be something like this:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 5 (delta 4), reused 5 (delta 4), pack-reused 0
Unpacking objects: 100% (5/5), done.
From ssh://github.com/pyneng/online-2-natasha-samoylenko
   89c04b6..fc4c721  master     -> origin/master
Updating 89c04b6..fc4c721
Fast-forward
 exercises/03_data_structures/task_3_3.py | 2 ++
 1 file changed, 2 insertions(+)
```

Adding new files or changes to existing files

If you want to add a specific file (in this case, README.md), you need to enter `git add README.md` command. All files of current directory are added by `git add .` command.

Commit

You should specify message when you are running a commit. It is better if message is with meaning, rather than just "update" or similar. Commit could be done by a command similar to `git commit -m "Tasks 4.1-4.3 are completed"`.

Push on GitHub

Command "git push" is used to load all local changes to Github:

```
$ git push origin master
Counting objects: 5, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 426 bytes | 0 bytes/s, done.
Total 5 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To ssh://git@github.com/pyneng/online-2-natasha-samoylenko.git
   fc4c721..edcf417  master -> master
```

Before executing `git push` you can run `git log -p/origin..` - it will show what changes you are going to add to your repository on Github.

Working with repository of tasks and examples

All examples and tasks of book are published in a separate repository.

Copying repository from Github

Examples and tasks are sometimes updated, so it will be more convenient to clone this repository to your machine and periodically update it.

To copy a repository from Github, run `git clone`:

```
$ git clone https://github.com/natenka/pyneng-examples-exercises
Cloning into 'pyneng-examples-exercises'...
remote: Counting objects: 1263, done.
remote: Compressing objects: 100% (504/504), done.
remote: Total 1263 (delta 735), reused 1263 (delta 735), pack-reused 0
Receiving objects: 100% (1263/1263), 267.10 KiB | 444.00 KiB/s, done.
Resolving deltas: 100% (735/735), done.
Checking connectivity... done.
```

Updating local copy of repository

If you need to update local repository to synchronize it with Github version, you need to perform `git pull` from inside the created pyneng-examples-exercises directory.

If there were no updates, output would be:

```
$ git pull
Already up-to-date.
```

If there were updates, output would be something like this:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/natenka/pyneng-examples-exercises
   49e9f1b..1eb82ad  master     -> origin/master
Updating 49e9f1b..1eb82ad
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Please note that only README.md file has been changed.

View changes

If you want to see what changes have been made, you can use `git log`:

```
$ git log -p -1
commit 98e393c27e7aae4b41878d9d979c7587bfeb24b4
Author: Nataliya Samoylenko <nataliya.samoylenko@gmail.com>
Date:   Fri Aug 18 17:32:07 2017 +0300

    Update task_24_4.md

diff --git a/exercises/24_ansible_for_network/task_24_4.md b/exercises/24_ansible_
↪for_network/task_24_4.md
index c4307fa..137a221 100644
--- a/exercises/24_ansible_for_network/task_24_4.md
+++ b/exercises/24_ansible_for_network/task_24_4.md
@@ -13,11 +13,12 @@
 * apply ACL to interface

 ACL should be like:
+
 ip access-list extended INET-to-LAN
  permit tcp 10.0.1.0 0.0.0.255 any eq www
  permit tcp 10.0.1.0 0.0.0.255 any eq 22
  permit icmp any any
-
+

 Check playbook execution on R1 router.
```

In this command `-p` flag indicates that the output of Linux diff utility should be displayed for changes, not just commit comment. In turn, `-1` indicates that only the latest commit should be shown.

View changes that will be synchronized

The previous version of `git log` relies on number of commands but this is not always convenient. Before executing `git pull` you can see what changes have been made since last synchronization.

The following command shall be used:

```
$ git log -p ..origin/master
commit 4c1821030d20b3682b67caf362fd777d098d9126
Author: Nataliya Samoylenko <nataliya.samoylenko@gmail.com>
Date:   Mon May 29 07:53:45 2017 +0300
```

(continues on next page)

(continued from previous page)

```
Update README.md

diff --git a/tools/README.md b/tools/README.md
index 2b6f380..4f8d4af 100644
--- a/tools/README.md
+++ b/tools/README.md
@@ -1 +1,4 @@
+
+Here you can find PDF versions of configuration manuals of tools that are used␣
↪on course.
```

In this case, changes were only in one file. This command will be very useful to see what changes have been made to tasks and which tasks. This will make it easier to navigate and to understand whether it is related to tasks you have already done and, if so, whether they should be changed.

Note: "..origin/master" in `git log -p ..origin/master` means to show all commits that are present in origin/master (in this case, it's GitHub) but that are not in local copy of repository

---

If changes were in tasks you haven't yet done, this output will tell you which files should be copied from course repository to your personal repository (and maybe the entire section if you haven't yet done tasks from this section).

Further reading

Documentation:

- Informative git prompt for bash and fish
- Authenticating to GitHub
- Connecting to GitHub with SSH

About Git/GitHub:

- git/github guide. a minimal tutorial - minimum knowledge required to work with Git and GitHub
- Pro Git book
- Version control system (GIT) (course on Hexlet)

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 2.1

Create your repository based on repository template with tasks and examples. To do this, press "Use this template".

Created repository will be a copy of pyneng-examples-exercises-en repository, but is not tied to it. It's better to perform tasks in prepared files in exercises directory as tests for tasks depend on created directory structure.

3. Getting started with Python

This section covers:

- Python syntax
- Work in interactive mode
- Python variables

Python syntax

The first thing that tends to catch your eye when it comes to Python syntax is that indentation matters:

- It determines which code is inside the block
- When a block of code starts and ends

Example of Python code:

```
a = 10
b = 5

if a > b:
    print("A greater than B")
    print(a - b)
else:
    print("B is greater than or equal to A")
    print(b - a)

print("End")

def open_file(filename):
    print("Reading File", filename)
    with open(filename) as f:
        return f.read()
        print("Ready")
```

Note: This code is shown for syntax demonstration. Although if/else statement has not yet been covered, it is likely that the meaning of code will be clear in general.

Python understands which lines refer to “if” on indentation basis. Execution of a block `if a > b` ends when another string with the same indent as string `if a > b` appears. Similarly to block `else`.

The second feature of Python is that some expressions must be followed by colon (for example, after `if a > b` and after `else`).

Several rules and recommendations on indentation:

- Tabs or spaces can be used as indents (it is better to use spaces or more precisely to configure editor so that Tab is 4 spaces - then when using Tab key, 4 spaces will be placed instead of 1 tab sign).
- Number of spaces must be the same in one block (it is better to have the same number of spaces in whole code - popular option is to use 2-4 spaces, for example, this book uses 4 spaces).

Another feature of code above is empty lines. It makes reading code easier. Other syntax features will be shown during process of familiarization with data structures in Python.

Note: Python has a special document that describes how best to write Python code PEP 8 - Style Guide for Python Code.

Comments

When writing code you often need to leave a comment, for example, to describe features of code.

Comments in Python can be one-line:

```
# A very important comment
a = 10
b = 5 # A much needed comment
```

One-line comments start with hash sign. Note that comment can be in line where code itself is or in a separate line.

If it is necessary to write several lines with comments in order to not put hash sign before each line, you can make a multi-line comment:

```
"""
Very important
and long comment
"""
a = 10
b = 5
```

Three double or three single quotes may be used for a multi-line comment. Comments can be used both to comment on what happens in code and to exclude execution of a particular line or block of code (i.e., to comment it).

Python interpreter. IPython

Interpreter makes it possible to receive an instant response to executed actions. You can say that interpreter works as CLI (Command Line Interface) of network devices: each command will be executed immediately after pressing Enter. However, there is an exception: more complex objects (such as cycles or functions) are executed only after twice pressing Enter.

In previous section, a standard interpreter was called to verify Python installation. There is also an improved interpreter IPython. IPython allows much more than standard interpreter called by "python" command. Some examples (IPython features are much broader):

- Autocomplete Tab commands or hints if there are more than one command version
- More structured and understandable output of commands
- Automatic indentation in loops and other objects
- You can either walk through the command execution history or watch it with %history 'magic' command

You can install IPython using pip (installation will be done in a virtual environment if configured):

```
pip install ipython
```

After that, you can move to IPython as follows:

```
$ ipython
Python 3.7.3 (default, May 13 2019, 15:44:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Command `quit` is used to exit. The following is how IPython will be used.

To get acquainted with interpreter, you can use it as a calculator:

```
In [1]: 1 + 2
Out[1]: 3

In [2]: 22*45
Out[2]: 990

In [3]: 2**3
Out[3]: 8
```

In IPython, input and output are marked:

- In - user input data

- Out - result that command returns (if any)
- Numbers after In or Out are sequential numbers of executed commands in current IPython session

Example of string output by function `print`:

```
In [4]: print('Hello!')
Hello!
```

When a loop is created in interpreter, for example, invitation changes to ellipsis inside loop. To complete loop and exit this shortcut, double press Enter:

```
In [5]: for i in range(5):
   ...:     print(i)
   ...:
0
1
2
3
4
```

help()

In IPython you can view help for an arbitrary object, function or method using `help`:

```
In [1]: help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from given object. If encoding or
 |  errors is specified, then object must expose a data buffer
 |  that will be decoded using given encoding and error handler.
...

In [2]: help(str.strip)
Help on method_descriptor:

strip(...)
    S.strip([chars]) -> str

```

(continues on next page)

(continued from previous page)

```
    Return a copy of string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars instead.
```

The second option is:

```
In [3]: ?str
Init signature: str(self, /, *args, **kwargs)
Docstring:
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from given object. If encoding or
errors is specified, then object must expose a data buffer
that will be decoded using given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
Type:           type

In [4]: ?str.strip
Docstring:
S.strip([chars]) -> str

Return a copy of string S with leading and trailing
whitespace removed.
If chars is given and not None, remove characters in chars instead.
Type:      method_descriptor
```

print

Function `print` displays information on a standard output (current terminal screen). If you want to get a string, you should place it in quotes(double or single). If you want to get, for example, a computation result or just a number, quotes are not needed:

```
In [6]: print('Hello!')
Hello!

In [7]: print(5*5)
25
```

If you want to get several values in a row through a space, you have to enumerate them through a

comma:

```
In [8]: print(1*5, 2*5, 3*5, 4*5)
5 10 15 20

In [9]: print('one', 'two', 'three')
one two three
```

By default, at the end of each expression passed to `print`, there will be a new line character. If it is necessary that after the output of each expression there would be no new line, an additional "end" argument should be specified as the last expression in `print`.

See also:

Additional parameters of print function *print*

dir()

Function `dir` can be used to see what attributes (variables tied to object) and methods (functions tied to object) are available.

For example, for number the output will be (pay attention on various methods that allow arithmetic operations):

```
In [10]: dir(5)
Out[10]:
['__abs__',
 '__add__',
 '__and__',
 ...
 'bit_length',
 'conjugate',
 'denominator',
 'imag',
 'numerator',
 'real']
```

The same for string:

```
In [11]: dir('hello')
Out[11]:
['__add__',
 '__class__',
 '__contains__',
 ...
```

(continues on next page)

(continued from previous page)

```
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
 'zfill']
```

If you call `dir` with no value, it shows existing methods, attributes, and variables defined in current session of interpreter:

```
In [12]: dir()
Out[12]:
['__builtin__',
 '__builtins__',
 '__doc__',
 '__name__',
 '_dh',
 ...
 '_oh',
 '_sh',
 'exit',
 'get_ipython',
 'i',
 'quit']
```

For example, after creating variable a and test():

```
In [13]: a = 'hello'

In [14]: def test():
   ....:     print('test')
   ....:

In [15]: dir()
Out[15]:
 ...
 'a',
 'exit',
 'get_ipython',
 'i',
 'quit',
 'test']
```

IPython special commands

IPython has special commands that make work with interpreter easier. All of them are started with percent sign.

%history

For example, `%history` command allows to look at history of commands entered by user in current IPython session.

```
In [1]: a = 10

In [2]: b = 5

In [3]: if a > b:
   ...:     print("A is bigger")
   ...:
A is bigger

In [4]: %history
a = 10
b = 5
if a > b:
    print("A is bigger")
%history
```

With `%history` you can copy needed block of code.

%time

Command `%time` shows how many seconds it took to execute expression.

```
In [5]: import subprocess

In [6]: def ping_ip(ip_address):
    ..:     reply = subprocess.run(['ping', '-c', '3', '-n', ip_address],
    ..:                            stdout=subprocess.PIPE,
    ..:                            stderr=subprocess.PIPE,
    ..:                            encoding='utf-8')
    ..:     if reply.returncode == 0:
    ..:         return True
    ..:     else:
    ..:         return False
```

(continues on next page)

(continued from previous page)

```
    ..:

In [7]: %time ping_ip('8.8.8.8')
CPU times: user 0 ns, sys: 4 ms, total: 4 ms
Wall time: 2.03 s
Out[7]: True

In [8]: %time ping_ip('8.8.8')
CPU times: user 0 ns, sys: 8 ms, total: 8 ms
Wall time: 12 s
Out[8]: False

In [9]: items = [1, 3, 5, 7, 9, 1, 2, 3, 55, 77, 33]

In [10]: %time sorted(items)
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 8.11 µs
Out[10]: [1, 1, 2, 3, 3, 5, 7, 9, 33, 55, 77]
```

More about IPython you can find in IPython documentation.

Briefly, information can be viewed in IPython via %quickref command:

```
IPython -- An enhanced Interactive Python - Quick Reference Card
============================================================

obj?, obj??      : Get help, or more help for object (also works as
                   ?obj, ??obj).
?foo.*abc*       : List names in 'foo' containing 'abc' in them.
%magic           : Information about IPython's 'magic' % functions.

Magic functions are prefixed by % or %%, and typically take their arguments
without brackets, quotes or even commas for convenience.  Line magics take a
single % and cell magics are prefixed with two %%.

Example magic function calls:

%alias d ls -F   : 'd' is now an alias for 'ls -F'
alias d ls -F    : Works if 'alias' not a python name
alist = %alias   : Get list of aliases to 'alist'
cd /usr/share    : Obvious. cd -<tab> to choose from visited dirs.
%cd??            : See help AND source for magic %cd
%timeit x=10     : time 'x=10' statement with high precision.
```

(continues on next page)

(continued from previous page)

```
%%timeit x=2**100
x**100           : time 'x**100' with a setup of 'x=2**100'; setup code is not
                   counted.  This is an example of a cell magic.

System commands:

!cp a.txt b/     : System command escape, calls os.system()
cp a.txt b/      : after %rehashx, most system commands work without !
cp ${f}.txt $bar : Variable expansion in magics and system commands
files = !ls /usr : Capture sytem command output
files.s, files.l, files.n: "a b c", ['a','b','c'], 'a\nb\nc'

History:

_i, _ii, _iii    : Previous, next previous, next next previous input
_i4, _ih[2:5]    : Input history line 4, lines 2-4
exec _i81        : Execute input history line #81 again
%rep 81          : Edit input history line #81
_, __, ___       : previous, next previous, next next previous output
_dh              : Directory history
_oh              : Output history
%hist            : Command history of current session.
%hist -g foo     : Search command history of (almost) all sessions for 'foo'.
%hist -g         : Command history of (almost) all sessions.
%hist 1/2-8      : Command history containing lines 2-8 of session 1.
%hist 1/ ~2/     : Command history of session 1 and 2 sessions before current.
```

Variables

Variables in Python do not require variable type declaration (since Python is a language with dynamic typing) and they are references to a memory area. Variable naming rules:

- Name of variable can consist only of letters, digits and an underscore
- Name cannot start with a digit
- Name cannot contain special characters @, $, %

An example of creating variables in Python:

```
In [1]: a = 3

In [2]: b = 'Hello'
```

(continues on next page)

(continued from previous page)

```
In [3]: c, d = 9, 'Test'

In [4]: print(a,b,c,d)
3 Hello 9 Test
```

Note that Python does not need to specify that “a” is a number, and “b” is a string.

Variables are references to memory area. This can be demonstrated by using `id()` which shows object ID:

```
In [5]: a = b = c = 33

In [6]: id(a)
Out[6]: 31671480

In [7]: id(b)
Out[7]: 31671480

In [8]: id(c)
Out[8]: 31671480
```

In this example you can see that all three names refer to the same identifier, so it is the same object to which three references “a”, “b” and “c” point. Python numbers has one feature that can be slightly misleading: numbers from -5 to 256 are pre-created and stored in an array (list). Therefore, when you create a number from this range you actually create a reference to number in generated array.

Note: This feature is specific to implementation of CPython which is discussed in book

---

This can be verified as follows:

```
In [9]: a = 3

In [10]: b = 3

In [11]: id(a)
Out[11]: 4400936168

In [12]: id(b)
Out[12]: 4400936168

In [13]: id(3)
Out[13]: 4400936168
```

Note that a, b and number 3 have identical identifiers. They are all references to an existing number in the list.

If you do the same with number more than 256, all identifiers will be different:

```
In [14]: a = 500

In [15]: b = 500

In [16]: id(a)
Out[16]: 140239990503056

In [17]: id(b)
Out[17]: 140239990503032

In [18]: id(500)
Out[18]: 140239990502960
```

However, if you assign variables to each other, identifiers are all the same (in this version a, b and c are referring to the same object):

```
In [19]: a = b = c = 500

In [20]: id(a)
Out[20]: 140239990503080

In [21]: id(b)
Out[21]: 140239990503080

In [22]: id(c)
Out[22]: 140239990503080
```

Variable names

Variable names should not overlap with names of operators and modules or other reserved words. Python has recommendations for naming functions, classes and variables:

- variable names are usually written in lowercase or in uppercase (e.g., DB_NAME, db_name)
- function names are written in lowercase, with underline between words (for example get_names)
- class names are given with capital letters without spaces, it is called CamelCase (for example, CiscoSwitch)

Tasks

All tasks and additional files can be downloaded from repository.

> **Warning:** Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 3.1

Install IPython in a virtual environment or globally in OS. After installation, running ipython command should open IPython interpreter (the output may differ slightly):

```
$ ipython
Python 3.7.3 (default, May 13 2019, 15:44:23)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

4. Python data types

Python has several data types:

- Numbers
- Strings
- Lists
- Dictionaries
- Tuples
- Sets
- Boolean

These data types, in turn, can be classified by several criteria:

- mutable (lists, dictionaries and sets)
- immutable (integers, strings and tuples)
- ordered (lists, tuples, strings and dictionaries)
- unordered (sets)

Numbers

With numbers it is possible to perform various mathematical operations.

```
In [1]: 1 + 2
Out[1]: 3

In [2]: 1.0 + 2
Out[2]: 3.0

In [3]: 10 - 4
Out[3]: 6

In [4]: 2**3
Out[4]: 8
```

Division int and float:

```
In [5]: 10/3
Out[5]: 3.3333333333333335

```

(continues on next page)

(continued from previous page)

```
In [6]: 10/3.0
Out[6]: 3.3333333333333335
```

The round() function - round a number to a given precision in decimal digits:

```
In [9]: round(10/3.0, 2)
Out[9]: 3.33

In [10]: round(10/3.0, 4)
Out[10]: 3.3333
```

Remainder of division:

```
In [11]: 10 % 3
Out[11]: 1
```

Comparison operators

```
In [12]: 10 > 3.0
Out[12]: True

In [13]: 10 < 3
Out[13]: False

In [14]: 10 == 3
Out[14]: False

In [15]: 10 == 10
Out[15]: True

In [16]: 10 <= 10
Out[16]: True

In [17]: 10.0 == 10
Out[17]: True
```

The int() function allows converting to int type. The second argument can specify number system:

```
In [18]: a = '11'

In [19]: int(a)
Out[19]: 11
```

If you specify that string should be read as a binary number, the result is:

```
In [20]: int(a, 2)
Out[20]: 3
```

Convert to int from float:

```
In [21]: int(3.333)
Out[21]: 3

In [22]: int(3.9)
Out[22]: 3
```

The bin() function produces a binary representation of a number (note that the result is a string):

```
In [23]: bin(8)
Out[23]: '0b1000'

In [24]: bin(255)
Out[24]: '0b11111111'
```

Similarly, function hex() produces a hexadecimal value:

```
In [25]: hex(10)
Out[25]: '0xa'
```

And, of course, you can do several changes at the same time:

```
In [26]: int('ff', 16)
Out[26]: 255

In [27]: bin(int('ff', 16))
Out[27]: '0b11111111'
```

For more complex mathematical functions, Python has a `math` module:

```
In [28]: import  math

In [29]: math.sqrt(9)
Out[29]: 3.0

In [30]: math.sqrt(10)
Out[30]: 3.1622776601683795

In [31]: math.factorial(3)
Out[31]: 6
```

(continues on next page)

(continued from previous page)

```
In [32]: math.pi
Out[32]: 3.141592653589793
```

Strings

String in Python is:

- sequence of characters enclosed in quotes
- immutable ordered data type

Examples of strings:

```
In [9]: 'Hello'
Out[9]: 'Hello'
In [10]: "Hello"
Out[10]: 'Hello'

In [11]: tunnel = """
   ....: interface Tunnel0
   ....:  ip address 10.10.10.1 255.255.255.0
   ....:  ip mtu 1416
   ....:  ip ospf hello-interval 5
   ....:  tunnel source FastEthernet1/0
   ....:  tunnel protection ipsec profile DMVPN
   ....: """

In [12]: tunnel
Out[12]: '\ninterface Tunnel0\n ip address 10.10.10.1 255.255.255.0\n ip mtu␣
↪1416\n ip ospf hello-interval 5\n tunnel source FastEthernet1/0\n tunnel␣
↪protection ipsec profile DMVPN\n'

In [13]: print(tunnel)

interface Tunnel0
 ip address 10.10.10.1 255.255.255.0
 ip mtu 1416
 ip ospf hello-interval 5
 tunnel source FastEthernet1/0
 tunnel protection ipsec profile DMVPN
```

Strings can be summed. Then they merge into one string:

```
In [14]: intf = 'interface'

In [15]: tun = 'Tunnel0'

In [16]: intf + tun
Out[16]: 'interfaceTunnel0'

In [17]: intf + ' ' + tun
Out[17]: 'interface Tunnel0'
```

You can multiply a string by a number. In this case, string repeats specified number of times:

```
In [18]: intf * 5
Out[18]: 'interfaceinterfaceinterfaceinterfaceinterface'

In [19]: '#' * 40
Out[19]: '########################################'
```

The fact that strings are an ordered data type allows to refer to characters in a string by a number starting from zero:

```
In [20]: string1 = 'interface FastEthernet1/0'

In [21]: string1[0]
Out[21]: 'i'
```

All characters in a string are numbered from zero. But if you need to refer to a character from the end, you can specify negative values (this time with 1).

```
In [22]: string1[1]
Out[22]: 'n'

In [23]: string1[-1]
Out[23]: '0'
```

In addition to referring to a specific character you can make string slices by specifying a number range. Slicing starts with first number (included) and ends at second number (excluded):

```
In [24]: string1[0:9]
Out[24]: 'interface'

In [25]: string1[10:22]
Out[25]: 'FastEthernet'
```

If no second number is specified, slice is until the end of string:

```
In [26]:  string1[10:]
Out[26]: 'FastEthernet1/0'
```

Slice last three character of string:

```
In [27]: string1[-3:]
Out[27]: '1/0'
```

You can also specify a step in slice. For example, you can get odd numbers:

```
In [28]: a = '0123456789'

In [29]: a[1::2]
Out[29]: '13579'
```

Or you can get all even numbers of string a:

```
In [31]: a[::2]
Out[31]: '02468'
```

Slices can also be used to get a string in reverse order:

```
In [28]: a = '0123456789'

In [29]: a[::]
Out[29]: '0123456789'

In [30]: a[::-1]
Out[30]: '9876543210'
```

Note: Entries `a[::]` and `a[:]` give the same result but double colon makes it possible to indicate that not every element should be taken, but for example every second element.

The `len` function allows you to get number of characters in a string:

```
In [1]: line = 'interface Gi0/1'

In [2]: len(line)
Out[2]: 15
```

Note: Function and method differ in that method is tied to a particular type of object and function is generally more universal and can be applied to objects of different types. For example, `len` function

can be applied to strings, lists, dictionaries and so on, but `startswith` method only applies to strings.

String methods

When automating, very often it will be necessary to work with strings, since config file, command output and commands sent - are strings. Knowledge of various methods (actions) that can be applied to strings helps to work with them more efficiently.

Strings are immutable data type, so all methods that convert string returns a new string and the original string remains unchanged.

Methods upper, lower, swapcase, capitalize

Methods `upper`, `lower`, `swapcase`, `capitalize` perform string case conversion:

```
In [25]: string1 = 'FastEthernet'

In [26]: string1.upper()
Out[26]: 'FASTETHERNET'

In [27]: string1.lower()
Out[27]: 'fastethernet'

In [28]: string1.swapcase()
Out[28]: 'fASTeTHERNET'

In [29]: string2 = 'tunnel 0'

In [30]: string2.capitalize()
Out[30]: 'Tunnel 0'
```

It is very important to pay attention to the fact that methods often return the converted string. And, therefore, we must not forget to assign it to some variable (you can use the same).

```
In [31]: string1 = string1.upper()

In [32]: print(string1)
FASTETHERNET
```

Method count

Method `count` used to count how many times a character or substring occurs in a string:

```
In [33]: string1 = 'Hello, hello, hello, hello'

In [34]: string1.count('hello')
Out[34]: 3

In [35]: string1.count('ello')
Out[35]: 4

In [36]: string1.count('l')
Out[36]: 8
```

Method find

You can pass a substring or character to `find` and it will return the lowest index where first character of the substring is (for the first match):

```
In [37]: string1 = 'interface FastEthernet0/1'

In [38]: string1.find('Fast')
Out[38]: 10

In [39]: string1[string1.find('Fast')::]
Out[39]: 'FastEthernet0/1'
```

If no match is found, `find` method returns `-1`.

Methods startswith, endswith

Checking if a string starts or ends with certain symbols (methods `startswith`, `endswith`):

```
In [40]: string1 = 'FastEthernet0/1'

In [41]: string1.startswith('Fast')
Out[41]: True

In [42]: string1.startswith('fast')
Out[42]: False

In [43]: string1.endswith('0/1')
Out[43]: True
```

(continues on next page)

(continued from previous page)

```
In [44]: string1.endswith('0/2')
Out[44]: False
```

Method replace

Replacing a sequence of characters in a string with another sequence (method `replace`):

```
In [45]: string1 = 'FastEthernet0/1'

In [46]: string1.replace('Fast', 'Gigabit')
Out[46]: 'GigabitEthernet0/1'
```

Method strip

Often when a file is processed, the file is opened line by line. But at the end of each line, there are usually some special characters (and may be at the beginning). For example, new line character.

To get rid of them, it is very convenient to use method `strip`:

```
In [47]: string1 = '\n\tinterface FastEthernet0/1\n'

In [48]: print(string1)

    interface FastEthernet0/1


In [49]: string1
Out[49]: '\n\tinterface FastEthernet0/1\n'

In [50]: string1.strip()
Out[50]: 'interface FastEthernet0/1'
```

By default, `strip` method removes blank characters. This character set includes: `\t\n\r\f\v`

Method `strip` can be passed as an argument of any characters. Then at the beginning and at the end of the line all characters that were specified in the line will be removed:

```
In [51]: ad_metric = '[110/1045]'

In [52]: ad_metric.strip('[]')
Out[52]: '110/1045'
```

Method `strip` removes special characters at the beginning and at the end of the line. If you want to remove characters only on the left or only on the right, you can use `lstrip` and `rstrip`.

Method split

Method `split` splits the string using a symbol (or symbols) as separator and returns a list of strings:

```
In [53]: string1 = 'switchport trunk allowed vlan 10,20,30,100-200'

In [54]: commands = string1.split()

In [55]: print(commands)
['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']
```

In example above, `string1.split` splits the string by spaces and returns a list of strings. The list is saved to `commands` variable.

By default, separator is a space symbol (spaces, tabs, new line), but you can specify any separator in parentheses:

```
In [56]: vlans = commands[-1].split(',')

In [57]: print(vlans)
['10', '20', '30', '100-200']
```

In `commands` list, the last element is a string with vlans, so the index -1 is used. Then string is split into parts using `split` `commands[-1].split(',')`. Since separator is a comma, this list is received `['10', '20', '30', '100-200']`.

A useful feature of `split` method with default separator is that the string is not only split into a list of strings by whitespace characters, but the whitespace characters are also removed at the beginning and at the end of the line:

```
In [58]: string1 = '  switchport trunk allowed vlan 10,20,30,100-200\n\n'

In [59]: string1.split()
Out[59]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']
```

Method `split` has another good feature: by default, method splits a string not by one whitespace character, but by any number. For example, this will be very useful when processing show commands:

```
In [60]: sh_ip_int_br = "FastEthernet0/0       15.0.15.1    YES manual up
↪up"
```

(continues on next page)

(continued from previous page)

```
In [61]: sh_ip_int_br.split()
Out[61]: ['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up']
```

And this is the same string when one space is used as the separator:

```
In [62]: sh_ip_int_br.split(' ')
Out[62]:
['FastEthernet0/0', '', '', '', '', '', '', '', '', '', '', '', '15.0.15.1', '', '
↪', '', '', '', '', 'YES', 'manual', 'up', '', '', '', '', '', '', '', '', '', '
↪', '', '', '', '', '', '', '', '', '', 'up']
```

String formatting

When working with strings, there are often situations where different data needs to be substituted in string template. This can be done by combining string parts and data, but Python has a more convenient way - strings formatting.

String formatting can help, for example, in such situations:

- need to set values to a string by a certain template
- need to format output by columns
- need to convert numbers to binary format

There are several options for string formatting:

- with operator `%` — older option
- method `format` — relatively new option
- f-строки — new option that appeared in Python 3.6

Although `format` is recommended, string formatting can often be found through `%`.

String formatting with `format` method

Example of `format` method use:

```
In [1]: "interface FastEthernet0/{}".format('1')
Out[1]: 'interface FastEthernet0/1'
```

A special symbol {} indicates that the value that is passed to `format` method is placed here. Each pair of curly braces represents one place for the substitution.

Values that are placed in curly braces may be of different types. For example, it can be a string, number or list:

```
In [3]: print('{}'.format('10.1.1.1'))
10.1.1.1

In [4]: print('{}'.format(100))
100

In [5]: print('{}'.format([10, 1, 1,1]))
[10, 1, 1, 1]
```

You can align result in columns by formatting strings. In string formatting, you can specify how many characters are selected for the data. If number of characters in the data is less than number of characters selected, the missing characters are filled with blanks.

For example, you can allign data in columns of equal width of 15 characters with right side alignment:

```
In [3]: vlan, mac, intf = ['100', 'aabb.cc80.7000', 'Gi0/1']

In [4]: print("{:>15} {:>15} {:>15}".format(vlan, mac, intf))
            100  aabb.cc80.7000           Gi0/1
```

Alignment to the left:

```
In [5]: print("{:15} {:15} {:15}".format(vlan, mac, intf))
100             aabb.cc80.7000  Gi0/1
```

Output template can also be multi-string:

```
In [6]: ip_template = '''
   ...: IP address:
   ...: {}
   ...: '''

In [7]: print(ip_template.format('10.1.1.1'))

IP address:
10.1.1.1
```

You can also use string formatting to change the display format of numbers.

For example, you can specify how many digits after the comma to show:

```
In [9]: print("{:.3f}".format(10.0/3))
3.333
```

Using string formatting, you can convert numbers to binary format:

```
In [11]: '{:b} {:b} {:b} {:b}'.format(192, 100, 1, 1)
Out[11]: '11000000 1100100 1 1'
```

You can still specify additional parameters such as column width:

```
In [12]: '{:8b} {:8b} {:8b} {:8b}'.format(192, 100, 1, 1)
Out[12]: '11000000  1100100        1        1'
```

You can also specify that numbers should be supplemented with zeros instead of spaces:

```
In [13]: '{:08b} {:08b} {:08b} {:08b}'.format(192, 100, 1, 1)
Out[13]: '11000000 01100100 00000001 00000001'
```

You can enter names in curly braces. This makes it possible to pass arguments in any order and also makes template more understandable:

```
In [15]: '{ip}/{mask}'.format(mask=24, ip='10.1.1.1')
Out[15]: '10.1.1.1/24'
```

Another useful feature of string formatting is argument number specification:

```
In [16]: '{1}/{0}'.format(24, '10.1.1.1')
Out[16]: '10.1.1.1/24'
```

For example this can prevent repetitive transmission of the same values:

```
In [19]: ip_template = '''
    ...: IP address:
    ...: {:<8} {:<8} {:<8} {:<8}
    ...: {:08b} {:08b} {:08b} {:08b}
    ...: '''

In [20]: print(ip_template.format(192, 100, 1, 1, 192, 100, 1, 1))

IP address:
192      100      1        1
11000000 01100100 00000001 00000001
```

In example above the octet address has to be passed twice - one for decimal format and other for binary.

By specifying value indexes that are passed to `format` method, it is possible to avoid duplication:

```
In [21]: ip_template = '''
    ...: IP address:
```

(continues on next page)

(continued from previous page)

```
    ...: {0:<8} {1:<8} {2:<8} {3:<8}
    ...: {0:08b} {1:08b} {2:08b} {3:08b}
    ...: '''

In [22]: print(ip_template.format(192, 100, 1, 1))

IP address:
192      100      1        1
11000000 01100100 00000001 00000001
```

Strings formatting with f-Strings

Python 3.6 added a new version of string formatting - f-strings or interpolation of strings. The f-strings allow not only to set values to template, but also to perform calls to functions, methods, etc.

In many situations f-strings are easier to use than format, and f-strings work faster than format and other methods of string formatting.

Syntax

F-string is a literal line with a letter f in front of it. Inside f-string, in curly braces there are names of variables that will be substituted:

```
In [1]: ip = '10.1.1.1'

In [2]: mask = 24

In [3]: f"IP: {ip}, mask: {mask}"
Out[3]: 'IP: 10.1.1.1, mask: 24'
```

The same result with `format` method you can achieve by: `"IP: {ip}, mask: {mask}".format(ip=ip, mask=mask)`.

A very important difference between f-strings and `format`: f-strings are expressions that are processed, not just strings. That is, in case of ipython, as soon as we wrote the expression and pressed Enter, it was performed and instead of expressions `{ip}` and `{mask}` the values of variables were substituted.

Therefore, for example, you cannot first write a template and then define variables that are used in template:

```
In [1]: f"IP: {ip}, mask: {mask}"
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-e6f8e01ac9c4> in <module>()
----> 1 f"IP: {ip}, mask: {mask}"

NameError: name 'ip' is not defined
```

In addition to substituting variable values you can write expressions in curly braces:

```
In [5]: first_name = 'William'

In [6]: second_name = 'Shakespeare'

In [7]: f"{first_name.upper()} {second_name.upper()}"
Out[7]: 'WILLIAM SHAKESPEARE'
```

After colon in f-strings you can specify the same values as in `format`:

```
In [9]: oct1, oct2, oct3, oct4 = [10, 1, 1, 1]

In [10]: print(f'''
    ...: IP address:
    ...: {oct1:<8} {oct2:<8} {oct3:<8} {oct4:<8}
    ...: {oct1:08b} {oct2:08b} {oct3:08b} {oct4:08b}''')

IP address:
10       1        1        1
00001010 00000001 00000001 00000001
```

Warning: Since for full explanation of f-strings it is necessary to show examples with loops and work with objects that have not yet been covered, this topic is also in the section *Formatting lines with f-strings* with additional examples and explanations.

Literal strings concatenation

Python has very convenient functionality — literal strings concatenation

```
In [1]: s = ('Test' 'String')

In [2]: s
Out[2]: 'TestString'
```

(continues on next page)

(continued from previous page)

```
In [3]: s = 'Test' 'String'

In [4]: s
Out[4]: 'TestString'
```

You can even wrap parts of a line on different lines, but only if they are in parentheses:

```
In [5]: s = ('Test'
   ...: 'String')

In [6]: s
Out[6]: 'TestString'
```

This is very convenient to use in regex:

```
regex = (
    '(\S+) +(\S+) +'
    '\w+ +\w+ +'
    '(up|down|administratively down) +'
    '(\w+)'
)
```

This way, the regex can be split and made easier to understand. Plus you can add explanatory comments in strings.

```
regex = (
    '(\S+) +(\S+) +' # interface and IP
    '\w+ +\w+ +'
    '(up|down|administratively down) +' # Status
    '(\w+)' # Protocol
)
```

It is also convenient to use this technique when writing a long message:

```
In [7]: message = ('During command execution "{}" '
   ...: 'such error occured "{}".\n'
   ...: 'Exclude this command from the list? [y/n]')

In [8]: message
Out[8]: 'During command execution "{}" such error occured "{}".\nExclude this␣
↪command from the list? [y/n]'
```

List

List in Python is:

- sequence of elements separated by comma and enclosed in square brackets
- mutable ordered data type

Examples of lists:

```
In [1]: list1 = [10,20,30,77]
In [2]: list2 = ['one', 'dog', 'seven']
In [3]: list3 = [1, 20, 4.0, 'word']
```

Creating a list using a literal:

```
In [1]: vlans = [10, 20, 30, 50]
```

Note: Literal is an expression that creates an object.

Create a list using `list` function:

```
In [2]: list1 = list('router')

In [3]: print(list1)
['r', 'o', 'u', 't', 'e', 'r']
```

Since a list is an ordered data type just like a string, in lists you can refer to an item by number, make slices:

```
In [4]: list3 = [1, 20, 4.0, 'word']

In [5]: list3[1]
Out[5]: 20

In [6]: list3[1::]
Out[6]: [20, 4.0, 'word']

In [7]: list3[-1]
Out[7]: 'word'

In [8]: list3[::-1]
Out[8]: ['word', 4.0, 20, 1]
```

You can reverse list by `reverse` method:

```
In [10]: vlans = ['10', '15', '20', '30', '100-200']

In [11]: vlans.reverse()

In [12]: vlans
Out[12]: ['100-200', '30', '20', '15', '10']
```

Since lists are mutable, list elements can be changed:

```
In [13]: list3
Out[13]: [1, 20, 4.0, 'word']

In [14]: list3[0] = 'test'

In [15]: list3
Out[15]: ['test', 20, 4.0, 'word']
```

You can also create a list of lists. As in a regular list you can refer to items in nested lists:

```
In [16]: interfaces = [['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up
↪'],
   ....: ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'],
   ....: ['FastEthernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]

In [17]: interfaces[0][0]
Out[17]: 'FastEthernet0/0'

In [18]: interfaces[2][0]
Out[18]: 'FastEthernet0/2'

In [19]: interfaces[2][1]
Out[19]: '10.0.2.1'
```

The `len` function returns number of items in list:

```
In [1]: items = [1, 2, 3]

In [2]: len(items)
Out[2]: 3
```

And `sorted` function sorts list items in ascending order and returns a new list with sorted items:

```
In [1]: names = ['John', 'Michael', 'Antony']

```

(continues on next page)

(continued from previous page)

```
In [2]: sorted(names)
Out[2]: ['Antony', 'John', 'Michael']
```

List methods

List is a mutable data type, so it is important to note that most list methods change a list in place without returning anything.

join

Method `join` collects a list of strings into one string with separator specified before join:

```
In [16]: vlans = ['10', '20', '30']

In [17]: ','.join(vlans)
Out[17]: '10,20,30'
```

Note: Method `join` actually string method but since the value must be passed to it as a list, it is covered here.

---

append

Method append adds specified item to the end of list:

```
In [18]: vlans = ['10', '20', '30', '100-200']

In [19]: vlans.append('300')

In [20]: vlans
Out[20]: ['10', '20', '30', '100-200', '300']
```

Method append changes list on spot and does not return anything.

extend

If you want to combine two lists you can use one of two methods: `extend` method or addition operation. These methods have an important difference: `extend` changes list to which method is applied and addition returns a new list that consists of two.

Method extend:

```
In [21]: vlans = ['10', '20', '30', '100-200']

In [22]: vlans2 = ['300', '400', '500']

In [23]: vlans.extend(vlans2)

In [24]: vlans
Out[24]: ['10', '20', '30', '100-200', '300', '400', '500']
```

Addition operation:

```
In [27]: vlans = ['10', '20', '30', '100-200']

In [28]: vlans2 = ['300', '400', '500']

In [29]: vlans + vlans2
Out[29]: ['10', '20', '30', '100-200', '300', '400', '500']
```

Note that when adding lists in IPython the 'Out' line appeared. This means that the result of summation can be assigned to variable:

```
In [30]: result = vlans + vlans2

In [31]: result
Out[31]: ['10', '20', '30', '100-200', '300', '400', '500']
```

pop

Method pop removes item that corresponds to specified number. Method returns this item:

```
In [28]: vlans = ['10', '20', '30', '100-200']

In [29]: vlans.pop(-1)
Out[29]: '100-200'

In [30]: vlans
Out[30]: ['10', '20', '30']
```

Without number specified the last item in list is deleted.

remove

Method remove removes specified item (remove does not return deleted item):

```
In [31]: vlans = ['10', '20', '30', '100-200']

In [32]: vlans.remove('20')

In [33]: vlans
Out[33]: ['10', '30', '100-200']
```

In remove you must specify item to be deleted, not its index. If item number is specified, error occurs:

```
In [34]: vlans.remove(-1)
---------------------------------------------------
ValueError      Traceback (most recent call last)
<ipython-input-32-f4ee38810cb7> in <module>()
----> 1 vlans.remove(-1)

ValueError: list.remove(x): x not in list
```

index

Method index - returns the first index of the passed value:

```
In [35]: vlans = ['10', '20', '30', '100-200']

In [36]: vlans.index('30')
Out[36]: 2
```

insert

Method insert allows to insert an item into a specific place in list:

```
In [37]: vlans = ['10', '20', '30', '100-200']

In [38]: vlans.insert(1, '15')

In [39]: vlans
Out[39]: ['10', '15', '20', '30', '100-200']
```

sort

Method `sort` sorts list in place:

```
In [40]: vlans = [1, 50, 10, 15]

In [41]: vlans.sort()

In [42]: vlans
Out[42]: [1, 10, 15, 50]
```

Dictionary

Dictionaries are mutable ordered data type:

- data in dictionary are pairs `key: value`
- values are accessible by key, not by number as in lists
- entries in dictionary stored in order they were added
- since dictionaries are mutable, dictionary items can be changed, added, removed
- key must be an immutable object: number, string, tuple
- value can be data of any type

Note: In other programming languages a similar dictionary can be called an associative array, hash, or hash table.

Example of dictionary:

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}
```

You can write it down like this:

```
london = {
    'id': 1,
    'name': 'London',
    'it_vlan': 320,
    'user_vlan': 1010,
    'mngmt_vlan': 99,
    'to_name': None,
    'to_id': None,
```

(continues on next page)

(continued from previous page)

```
    'port': 'G1/0/11'
}
```

In order to get a value from dictionary you have to refer to key in the same way as in lists, only key will be used instead of number:

```
In [1]: london = {'name': 'London1', 'location': 'London Str'}

In [2]: london['name']
Out[2]: 'London1'

In [3]: london['location']
Out[3]: 'London Str'
```

Similarly, a new key-value pair could be added:

```
In [4]: london['vendor'] = 'Cisco'

In [5]: print(london)
{'vendor': 'Cisco', 'name': 'London1', 'location': 'London Str'}
```

Or rewritten:

```
In [6]: london['vendor'] = 'cisco ios'

In [7]: print(london)
{'vendor': 'cisco ios', 'name': 'London1', 'location': 'London Str'}
```

In dictionary you can use a dictionary as a value:

```
london_co = {
    'r1': {
        'hostname': 'london_r1',
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2': {
        'hostname': 'london_r2',
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
```

(continues on next page)

(continued from previous page)

```
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    'sw1': {
        'hostname': 'london_sw1',
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '3850',
        'ios': '3.6.XE',
        'ip': '10.255.0.101'
    }
}
```

You can get values from nested dictionary by:

```
In [7]: london_co['r1']['ios']
Out[7]: '15.4'

In [8]: london_co['r1']['model']
Out[8]: '4451'

In [9]: london_co['sw1']['ip']
Out[9]: '10.255.0.101'
```

Function `sorted` sorts dictionary keys in ascending order and returns a new list with sorted keys:

```
In [1]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [2]: sorted(london)
Out[2]: ['location', 'name', 'vendor']
```

Dictionary methods

clear

Method `clear` allows to clear dictionary:

```
In [1]: london = {'name': 'London1', 'location': 'London Str'}

In [2]: london.clear()

```

(continues on next page)

(continued from previous page)

```
In [3]: london
Out[3]: {}
```

copy

Method copy allows to create a full copy of dictionary.

If one dictionary is equal to other:

```
In [4]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [5]: london2 = london

In [6]: id(london)
Out[6]: 25489072

In [7]: id(london2)
Out[7]: 25489072

In [8]: london['vendor'] = 'Juniper'

In [9]: london2['vendor']
Out[9]: 'Juniper'
```

In this case `london2` is another name that refers to dictionary `london`. And when you change `london` dictionary, `london2` dictionary changes as well because it's a link to the same object.

Therefore, if you want to make a copy of dictionary, use copy method:

```
In [10]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [11]: london2 = london.copy()

In [12]: id(london)
Out[12]: 25524512

In [13]: id(london2)
Out[13]: 25563296

In [14]: london['vendor'] = 'Juniper'

In [15]: london2['vendor']
Out[15]: 'Cisco'
```

get

If you query a key that is not present in dictionary, an error occurs:

```
In [16]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [17]: london['ios']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-17-b4fae8480b21> in <module>()
----> 1 london['ios']

KeyError: 'ios'
```

Method get queries for key and if there is no key, returns None instead.

```
In [18]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [19]: print(london.get('ios'))
None
```

Method get() also allows you to specify another value instead of None:

```
In [20]: print(london.get('ios', 'Ooops'))
Ooops
```

setdefault

Method setdefault searches for key and if there is no key, instead of error it creates a key with None value.

```
In [21]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [22]: ios = london.setdefault('ios')

In [23]: print(ios)
None

In [24]: london
Out[24]: {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'ios':␣
↪None}
```

If key is present, setdefault returns value that corresponds to it:

```
In [25]: london.setdefault('name')
Out[25]: 'London1'
```

The second argument allows to specify which value should correspond to key:

```
In [26]: model = london.setdefault('model', 'Cisco3580')

In [27]: print(model)
Cisco3580

In [28]: london
Out[28]:
{'name': 'London1',
 'location': 'London Str',
 'vendor': 'Cisco',
 'ios': None,
 'model': 'Cisco3580'}
```

Method `setdefault` replaces this expression:

```
In [30]: if key in london:
    ...:     value = london[key]
    ...: else:
    ...:     london[key] = 'somevalue'
    ...:     value = london[key]
    ...:
```

keys, values, items

Methods `keys`, `values`, `items`:

```
In [24]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [25]: london.keys()
Out[25]: dict_keys(['name', 'location', 'vendor'])

In [26]: london.values()
Out[26]: dict_values(['London1', 'London Str', 'Cisco'])

In [27]: london.items()
Out[27]: dict_items([('name', 'London1'), ('location', 'London Str'), ('vendor',
↪'Cisco')])
```

All three methods return special view objects that contains keys, values, and key-value pairs of dictionary, respectively.

A very important feature of view is that they change together with dictionary. And in fact, they just give you a way to look at objects, but they don't make a copy of them.

Example of uning keys:

```
In [28]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [29]: keys = london.keys()

In [30]: print(keys)
dict_keys(['name', 'location', 'vendor'])
```

Now keys variable corresponds to view `dict_keys`, in which three keys: name, location and vendor.

But if we add another key-value pair to dictionary, keys object will also change:

```
In [31]: london['ip'] = '10.1.1.1'

In [32]: keys
Out[32]: dict_keys(['name', 'location', 'vendor', 'ip'])
```

If you want to get a simple list of keys that will not be changed with dictionary changes, it is enough to convert view to list:

```
In [33]: list_keys = list(london.keys())

In [34]: list_keys
Out[34]: ['name', 'location', 'vendor', 'ip']
```

del

Remove key and value:

```
In [35]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco'}

In [36]: del london['name']

In [37]: london
Out[37]: {'location': 'London Str', 'vendor': 'Cisco'}
```

update

Method update allows you to add contents of one dictionary to another dictionary:

```
In [38]: r1 = {'name': 'London1', 'location': 'London Str'}

In [39]: r1.update({'vendor': 'Cisco', 'ios':'15.2'})

In [40]: r1
Out[40]: {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'ios':
↪'15.2'}
```

Values can be updated in the same way:

```
In [41]: r1.update({'name': 'london-r1', 'ios':'15.4'})

In [42]: r1
Out[42]:
{'name': 'london-r1',
 'location': 'London Str',
 'vendor': 'Cisco',
 'ios': '15.4'}
```

Dictionary creation options

Literal

A dictionary can be created with help of a literal:

```
In [1]: r1 = {'model': '4451', 'ios': '15.4'}
```

dict

Construction dict allows you to create a dictionary in several ways.

If you use strings as keys you can use this option to create a dictionary:

```
In [2]: r1 = dict(model='4451', ios='15.4')

In [3]: r1
Out[3]: {'model': '4451', 'ios': '15.4'}
```

The second option of creating a dictionary with dict():

```
In [4]: r1 = dict([('model','4451'), ('ios','15.4')])

In [5]: r1
Out[5]: {'model': '4451', 'ios': '15.4'}
```

dict.fromkeys

In a situation where you need to create a dictionary with known keys but so far empty values (or identical values), `fromkeys` method is very convenient:

```
In [5]: d_keys = ['hostname', 'location', 'vendor', 'model', 'ios', 'ip']

In [6]: r1 = dict.fromkeys(d_keys)

In [7]: r1
Out[7]:
{'hostname': None,
 'location': None,
 'vendor': None,
 'model': None,
 'ios': None,
 'ip': None}
```

By default `fromkeys` sets None value. But you can also pass your own value:

```
In [8]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [9]: models_count = dict.fromkeys(router_models, 0)

In [10]: models_count
Out[10]: {'ISR2811': 0, 'ISR2911': 0, 'ISR2921': 0, 'ASR9002': 0}
```

This option of creating a dictionary is not suitable for all cases. For example, if you use a mutable data type in value, a reference to the same object will be created:

```
In [10]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [11]: routers = dict.fromkeys(router_models, [])
    ...:

In [12]: routers
Out[12]: {'ISR2811': [], 'ISR2911': [], 'ISR2921': [], 'ASR9002': []}
```

(continues on next page)

(continued from previous page)

```
In [13]: routers['ASR9002'].append('london_r1')

In [14]: routers
Out[14]:
{'ISR2811': ['london_r1'],
 'ISR2911': ['london_r1'],
 'ISR2921': ['london_r1'],
 'ASR9002': ['london_r1']}
```

In this case, each key refers to the same list. Therefore, when a value is added to one of lists, others are updated.

Note: A dictionary comprehension is better for this task. See section *List, dict, set comprehensions*

Tuple

Tuple in Python is:

- a sequence of elements separated by a comma and enclosed in parentheses
- immutable ordered data type

Roughly speaking, a tuple is a list that can't be changed. We can say that the tuple has read-only permissions. It could be a defense against accidental change.

Create an empty tuple:

```
In [1]: tuple1 = tuple()

In [2]: print(tuple1)
()
```

Tuple with one element (note the comma):

```
In [3]: tuple2 = ('password',)
```

Tuple from list:

```
In [4]: list_keys = ['hostname', 'location', 'vendor', 'model', 'ios', 'ip']

In [5]: tuple_keys = tuple(list_keys)
```

(continues on next page)

(continued from previous page)

```
In [6]: tuple_keys
Out[6]: ('hostname', 'location', 'vendor', 'model', 'ios', 'ip')
```

Objects in tuple can be accessed as well as objects in list, by order number:

```
In [7]: tuple_keys[0]
Out[7]: 'hostname'
```

But since tuple is immutable you cannot assign a new value:

```
In [8]: tuple_keys[1] = 'test'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-1c7162cdefa3> in <module>()
----> 1 tuple_keys[1] = 'test'

TypeError: 'tuple' object does not support item assignment
```

Function `sorted` sorts tuple elements in ascending order and returns a new list with sorted elements:

```
In [2]: tuple_keys = ('hostname', 'location', 'vendor', 'model', 'ios', 'ip')

In [3]: sorted(tuple_keys)
Out[3]: ['hostname', 'ios', 'ip', 'location', 'model', 'vendor']
```

Set

Set is a mutable unordered data type. Set always contains only unique elements. Set in Python is a sequence of elements that are separated by a comma and placed in curly braces.

Set can easily remove repetitive elements:

```
In [1]: vlans = [10, 20, 30, 40, 100, 10]

In [2]: set(vlans)
Out[2]: {10, 20, 30, 40, 100}

In [3]: set1 = set(vlans)

In [4]: print(set1)
{40, 100, 10, 20, 30}
```

Set methods

add

Method add adds an item to set:

```
In [1]: set1 = {10,20,30,40}

In [2]: set1.add(50)

In [3]: set1
Out[3]: {10, 20, 30, 40, 50}
```

discard

Method `discard` allows deleting elements without showing an error if there is no element in set:

```
In [3]: set1
Out[3]: {10, 20, 30, 40, 50}

In [4]: set1.discard(55)

In [5]: set1
Out[5]: {10, 20, 30, 40, 50}

In [6]: set1.discard(50)

In [7]: set1
Out[7]: {10, 20, 30, 40}
```

clear

Method `clear` empties set:

```
In [8]: set1 = {10,20,30,40}

In [9]: set1.clear()

In [10]: set1
Out[10]: set()
```

Operations with sets

Sets are useful in performing different operations such as finding union of sets, intersection and so on.

Union of sets can be obtained by `union` or operator `|`:

```
In [1]: vlans1 = {10,20,30,50,100}
In [2]: vlans2 = {100,101,102,102,200}

In [3]: vlans1.union(vlans2)
Out[3]: {10, 20, 30, 50, 100, 101, 102, 200}

In [4]: vlans1 | vlans2
Out[4]: {10, 20, 30, 50, 100, 101, 102, 200}
```

Intersection of sets can be obtained by `intersection` or operator &:

```
In [5]: vlans1 = {10,20,30,50,100}
In [6]: vlans2 = {100,101,102,102,200}

In [7]: vlans1.intersection(vlans2)
Out[7]: {100}

In [8]: vlans1 & vlans2
Out[8]: {100}
```

Options for set creation

You cannot create an empty set using a literal set (in this case it will not be a set but a dictionary):

```
In [1]: set1 = {}

In [2]: type(set1)
Out[2]: dict
```

But an empty set can be created in this way:

```
In [3]: set2 = set()

In [4]: type(set2)
Out[4]: set
```

Set from string:

```
In [5]: set('long long long long string')
Out[5]: {' ', 'g', 'i', 'l', 'n', 'o', 'r', 's', 't'}
```

Set from list:

```
In [6]: set([10, 20, 30, 10, 10, 30])
Out[6]: {10, 20, 30}
```

Boolean values

Boolean values in Python are two constants `True` and `False`.

In Python, not only True and False are considered True and False values.

- True value:
  - any non-zero number
  - any non-empty string
  - any non-empty object
- False value:
  - 0
  - None
  - empty string
  - empty object

Other True and False values tend to follow the condition logically.

To check boolean value of object you can use `bool`:

```
In [2]: items = [1, 2, 3]

In [3]: empty_list = []

In [4]: bool(empty_list)
Out[4]: False

In [5]: bool(items)
Out[5]: True

In [6]: bool(0)
Out[6]: False
```

(continues on next page)

(continued from previous page)

```
In [7]: bool(1)
Out[7]: True
```

Types conversion

Python has several useful built-in features that allow data to be converted from one type to another.

int

`int` converts a string to int:

```
In [1]: int("10")
Out[1]: 10
```

Using `int` function you can convert a binary number into a decimal number (binary number must be written as a string)

```
In [2]: int("11111111", 2)
Out[2]: 255
```

bin

You can convert a decimal number to binary format with `bin`:

```
In [3]: bin(10)
Out[3]: '0b1010'

In [4]: bin(255)
Out[4]: '0b11111111'
```

hex

A similar function exists for conversion to hexadecimal format:

```
In [5]: hex(10)
Out[5]: '0xa'

In [6]: hex(255)
Out[6]: '0xff'
```

list

Function `list` converts an argument to a list:

```
In [7]: list("string")
Out[7]: ['s', 't', 'r', 'i', 'n', 'g']

In [8]: list({1, 2, 3})
Out[8]: [1, 2, 3]

In [9]: list((1, 2, 3, 4))
Out[9]: [1, 2, 3, 4]
```

set

Function `set` converts an argument into a set:

```
In [10]: set([1, 2, 3, 3, 4, 4, 4, 4])
Out[10]: {1, 2, 3, 4}

In [11]: set((1, 2, 3, 3, 4, 4, 4, 4))
Out[11]: {1, 2, 3, 4}

In [12]: set("string string")
Out[12]: {' ', 'g', 'i', 'n', 'r', 's', 't'}
```

This function is very useful when you need to get unique elements in a sequence.

tuple

Function `tuple` converts argument into a tuple:

```
In [13]: tuple([1, 2, 3, 4])
Out[13]: (1, 2, 3, 4)

In [14]: tuple({1, 2, 3, 4})
Out[14]: (1, 2, 3, 4)

In [15]: tuple("string")
Out[15]: ('s', 't', 'r', 'i', 'n', 'g')
```

This can be useful if you want an immutable object.

str

Function `str` converts an argument into a string:

```
In [16]: str(10)
Out[16]: '10'
```

Types checking

This type of error can occur when converting data types:

```
In [1]: int('a')
---------------------------------------------------------
ValueError           Traceback (most recent call last)
<ipython-input-42-b3c3f4515dd4> in <module>()
----> 1 int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

Error is perfectly logical. We're trying to convert string 'a' into decimal format. For example, this can be useful when you want to go through a list of strings and convert to a number the strings that contain numbers, you can get that error. To avoid error, it would be nice to be able to check what we're working with.

isdigit

Python has such methods. For example, `isdigit` method can be used to check whether a string consists only of digits:

```
In [2]: "a".isdigit()
Out[2]: False

In [3]: "a10".isdigit()
Out[3]: False

In [4]: "10".isdigit()
Out[4]: True
```

isalpha

Method `isalpha` makes it possible to check whether a string consists only of letters:

```
In [7]: "a".isalpha()
Out[7]: True

In [8]: "a100".isalpha()
Out[8]: False

In [9]: "a--  ".isalpha()
Out[9]: False

In [10]: "a ".isalpha()
Out[10]: False
```

isalnum

Method `isalnum` makes it possible to check whether a string consists of letters or numbers:

```
In [11]: "a".isalnum()
Out[1]: True

In [12]: "a10".isalnum()
Out[12]: True
```

type

Sometimes, depending on the result, a library or function can return different types of objects. For example, if there is one object, string is returned. If several, tuple is returned. We have to construct the program in different ways, depending on whether a string or a tuple has been returned.

Method `type` function can help:

```
In [13]: type("string")
Out[13]: str

In [14]: type("string") == str
Out[14]: True
```

Similar to tuple (and other data types):

```
In [15]: type((1, 2, 3))
Out[15]: tuple

In [16]: type((1, 2, 3)) == tuple
```

(continues on next page)

(continued from previous page)

```
Out[16]: True

In [17]: type((1, 2, 3)) == list
Out[17]: False
```

Method chaining

Often, you need to perform several operations with data, for example:

```
In [1]: line = "switchport trunk allowed vlan 10,20,30"

In [2]: words = line.split()

In [3]: words
Out[3]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30']

In [4]: vlans_str = words[-1]

In [5]: vlans_str
Out[5]: '10,20,30'

In [6]: vlans = vlans_str.split(",")

In [7]: vlans
Out[7]: ['10', '20', '30']
```

In the script:

```
line = "switchport trunk allowed vlan 10,20,30"
words = line.split()
vlans_str = words[-1]
vlans = vlans_str.split(",")
print(vlans)
```

In this case, variables are used to store the intermediate result and subsequent methods/actions are performed with the variable. This is a completely normal version of the code, especially at first when it's hard perceive more complex expressions.

However, in Python, there are often expressions in which actions or methods are applied one after the other in one expression. For example, the previous code could be written like this:

```
line = "switchport trunk allowed vlan 10,20,30"
vlans = line.split()[-1].split(",")
```

(continues on next page)

(continued from previous page)

```
print(vlans)
```

Since there are no expressions in parentheses that would indicate the priority of execution, everything is executed from left to right.

First, `line.split()` is executed - we get the list, then to the resulting list applies `[-1]` - we get the last element of the list, the line `10,20,30`. The method `split(",")` is applied to this line and as a result we get the list `['10', '20', '30']`.

The main nuance when writing such chains, the previous method/action should return something what the next method/action is waiting for. And it is imperative that something is returned, otherwise there will be an error.

Sorting basics

When sorting data like list of lists or list of tuples, `sorted` sorts by the first element of nested lists (tuples), and if the first element is the same, on the second:

```
In [1]: data = [[1, 100, 1000], [2, 2, 2], [1, 2, 3], [4, 100, 3]]

In [2]: sorted(data)
Out[2]: [[1, 2, 3], [1, 100, 1000], [2, 2, 2], [4, 100, 3]]
```

If the sort is done for a list of numbers that are written as strings, the sort will be lexicographic, not natural, and the order will be:

```
In [7]: vlans = ['1', '30', '11', '3', '10', '20', '30', '100']

In [8]: sorted(vlans)
Out[8]: ['1', '10', '100', '11', '20', '3', '30', '30']
```

For the sorting to be “correct” it is necessary to convert vlans to numbers.

The same problem appears, for example, with IP addresses:

```
In [2]: ip_list = ["10.1.1.1", "10.1.10.1", "10.1.2.1", "10.1.11.1"]

In [3]: sorted(ip_list)
Out[3]: ['10.1.1.1', '10.1.10.1', '10.1.11.1', '10.1.2.1']
```

How to solve the problem with sorting IP addresses is discussed in section “10. Useful functions”.

Further reading

Documentation:

- Strings. String Methods
- Lists basics. More on lists
- Tuples. More on tuples
- Sets basics. More on sets
- Dict basics. More on dicts
- Common Sequence Operations

String formatting:

- String formating usage examples
- Documentaion on string formating
- Python 3's f-Strings: An Improved String Formatting Syntax (Guide)
- Python String Formatting Best Practices

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Note: In section 4, the tests can be easily "tricked" into making the correct output without getting results from initial data using Python. This does not mean that the task was done correctly, it is just that at this stage it is difficult otherwise test the result.

Task 4.1

Using the prepared nat string, get a new string where the FastEthernet interface is replaced with GigabitEthernet. Print the resulting new string to the standard output (stdout) using print.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
nat = "ip nat inside source list ACL interface FastEthernet0/1 overload"
```

Task 4.2

Convert string in mac variable from XXXX:XXXX:XXXX format to XXXX.XXXX.XXXX format. Print the resulting new string to the standard output (stdout) using print.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
mac = "AAAA:BBBB:CCCC"
```

Task 4.3

Get the following list of VLANs from the config string: `["1", "3", "10", "20", "30", "100"]`

Write the resulting list to the result variable. (this is the variable that will be checked in the test)

Print the resulting list to the standard output (stdout) using print.

Here is a very important point that you need to get exactly the list (data type), and not, for example, a string that looks like the list shown.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
config = "switchport trunk allowed vlan 1,3,10,20,30,100"
```

Task 4.4

Vlans list is a list of VLANs collected from all devices on the network, therefore there are duplicate VLAN numbers in the list.

Get a new list of unique VLAN numbers from the vlans list, sorted in ascending order of numbers. To get the final list, you cannot delete specific vlans manually.

Write the resulting list to the result variable. (this is the variable that will be checked in the test)

Print the resulting list to the standard output (stdout) using print.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
vlans = [10, 20, 30, 1, 2, 100, 10, 30, 3, 4, 10]
```

Task 4.5

From the strings command1 and command2, get a list of VLANs that exist in both command1 and command2 (intersection).

In this case, the result should be a list: `['1', '3', '8']`.

Write the resulting list to the result variable. (this is the variable that will be checked in the test)

Print the resulting list to the standard output (stdout) using print.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
command1 = "switchport trunk allowed vlan 1,2,3,5,8"
command2 = "switchport trunk allowed vlan 1,3,8,9"
```

Task 4.6

Process the ospf_route string and print the information to the stdout as follows:

```
Prefix                10.0.24.0/24
AD/Metric             110/41
Next-Hop              10.0.13.3
```

(continues on next page)

(continued from previous page)

```
Last update           3d18h
Outbound Interface    FastEthernet0/0
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
ospf_route = "       10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0"
```

Task 4.7

Convert MAC address in mac string to binary string like this: 101010101010101010111011101110111100110011001100

Print the resulting new string to the standard output (stdout) using print.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
mac = "AAAA:BBBB:CCCC"
```

Task 4.8

Convert the IP address in the ip variable to binary and print output in columns to stdout:

- the first line must be decimal values
- the second line is binary values

The output should be ordered in the same way as in the example output below:

- in columns
- column width 10 characters (in binary you need to add two spaces between columns to separate octets among themselves)

Example output for address 10.1.1.1:

```
10        1         1         1
00001010  00000001  00000001  00000001
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
ip = "192.168.3.1"
```

5. Basic scripts

Generally speaking, script is a regular file. This file stores the sequence of commands that you want to execute.

Let's start with basic script and print several strings on standard output. To do this, you need to create an access_template.py file with this content:

```
access_template = ['switchport mode access',
                   'switchport access vlan {}',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('\n'.join(access_template).format(5))
```

First, items in list are combined into a string that is separated by `\n` and VLAN number is inserted into string using string formatting. After this you should save file and go to command line. This is how the script execution looks like:

```
$ python access_template.py
switchport mode access
switchport access vlan 5
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

All scripts that will be created in this course have an extension `.py`. You can say that it is a «good manners» - to create Python scripts with .py extension.

Executable file

In order for a file to be executable and not have to write "python" every time before calling a file, you need to:

- make file executable (for Linux)
- the first line of file should have `#!/usr/bin/env python` or `#!/usr/bin/env python3` depending on which version of Python is used by default

Example of access_template_exec.py file:

```
#!/usr/bin/env python3

access_template = ['switchport mode access',
```

(continues on next page)

(continued from previous page)

```
                  'switchport access vlan {}',
                  'switchport nonegotiate',
                  'spanning-tree portfast',
                  'spanning-tree bpduguard enable']

print('\n'.join(access_template).format(5))
```

After that:

```
chmod +x access_template_exec.py
```

Now you can call file like this:

```
$ ./access_template_exec.py
```

Passing arguments to the script (sys.argv)

Very often script solves some common problem. For example, script processes a configuration file. Of course, in this case you don't want to edit name of file every time with your hands in script. It will be much better to pass file name as script argument and then use already specified file. The sys module allows working with script arguments via argv.

Example of access_template_argv.py:

```
from sys import argv

interface = argv[1]
vlan = argv[2]

access_template = ['switchport mode access',
                   'switchport access vlan {}',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('interface {}'.format(interface))
print('\n'.join(access_template).format(vlan))
```

Script output:

```
$ python access_template_argv.py Gi0/7 4
interface Gi0/7
switchport mode access
```

(continues on next page)

(continued from previous page)

```
switchport access vlan 4
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

Arguments that have been passed to script are substituted as values in template. Several points need to be clarified:

- argv is a list
- all arguments are in list and represented as strings
- argv contains not only arguments that passed to script but also name of script itself

In this case, argv list contains the following elements:

```
['access_template_argv.py', 'Gi0/7', '4']
```

First comes the name of script itself, then arguments in the same order.

User input

Sometimes it is necessary to get information from user. For example, request a password. The input function is used to get information from the user:

```
In [1]: print(input('What is your faivorite routing protocol? '))
What is your faivorite routing protocol? OSPF
OSPF
```

In this case, information is immediately displayed to user, but in addition, information entered by user can be stored in a variable and can be used later in script.

```
In [2]: protocol = input('What is your faivorite routing protocol? ')
What is your faivorite routing protocol? OSPF

In [3]: print(protocol)
OSPF
```

In parentheses, a question is usually written that specifies what information needs to be entered. A script that asks the user for information using input (file access_template_input.py):

```
interface = input('Enter interface type and number: ')
vlan = input('Enter VLAN number: ')

access_template = ['switchport mode access',
```

(continues on next page)

(continued from previous page)

```
                  'switchport access vlan {}',
                  'switchport nonegotiate',
                  'spanning-tree portfast',
                  'spanning-tree bpduguard enable']

print('\n' + '-' * 30)
print('interface {}'.format(interface))
print('\n'.join(access_template).format(vlan))
```

First two lines request information from user. Line `print('\n' + '-' * 30)` is used to visually separate information request from the output.

Script output:

```
$ python access_template_input.py
Enter interface type and number: Gi0/3
Enter VLAN number: 55

------------------------------
interface Gi0/3
switchport mode access
switchport access vlan 55
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 5.1

The task contains a dictionary with information about different devices.

In the task you need: ask the user to enter the device name (r1, r2 or sw1). Print information about the corresponding device to standard output (information will be in the form of a dictionary).

An example of script execution:

```
$ python task_5_1.py
Enter name of device: r1
{"location": "21 New Globe Walk", "vendor": "Cisco", "model": "4451", "ios": "15.4
↪", "ip": "10.255.0.1"}
```

Restriction: You cannot modify the london_co dictionary.

All tasks must be completed using only the topics covered. That is, this task can be solved without using the if condition. Restriction: You cannot change london_co dictionary.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
```

(continues on next page)

(continued from previous page)

```
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Task 5.1a

Modify the script from task 5.1 so that, in addition to the device name, the script requested and then printed the device parameter as well.

Display information about the corresponding parameter of the specified device.

An example of script execution:

```
$ python task_5_1a.py
Enter device name : r1
Enter parameter name: ios
15.4
```

Restriction: You cannot modify the london_co dictionary.

All tasks must be completed using only the topics covered. That is, this task can be solved without using the if condition.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
```

(continues on next page)

(continued from previous page)

```
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Task 5.1b

Modify the script from task 5.1a so that, when requesting a parameter, a list of possible parameters was displayed. The list of parameters must be obtained from the dictionary, rather than written manually.

Display information about the corresponding parameter of the specified device.

An example of script execution:

```
$ python task_5_1b.py
Enter device name: r1
Enter parameter name (ios, model, vendor, location, ip): ip
10.255.0.1
```

Restriction: You cannot modify the london_co dictionary.

All tasks must be completed using only the topics covered. That is, this task can be solved without using the if condition.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
```

(continues on next page)

(continued from previous page)

```
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Task 5.1c

Copy and modify the script from task 5.1b so that when you request a parameter that is not in the device dictionary, the message 'There is no such parameter' is displayed. The assignment applies only to the parameters of the devices, not to the devices themselves.

Note: Try typing a non-existent parameter, to see what the result will be. And then complete the task.

If an existing parameter is selected, print information about the corresponding parameter.

An example of script execution:

```
$ python task_5_1c.py
Enter device name: r1
Enter parameter name (ios, model, vendor, location, ip): ips
No such parameter
```

Restriction: You cannot modify the london_co dictionary.

All tasks must be completed using only the topics covered. That is, this task can be solved without using the if condition.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
```

(continues on next page)

(continued from previous page)

```
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Task 5.1d

Modify the script from task 5.1c so that, when requesting a parameter, the user could enter the parameter name in any case.

An example of script execution:

```
$ python task_5_1d.py
Enter device name: r1
Enter parameter name (ios, model, vendor, location, ip): IOS
15.4
```

Restriction: You cannot modify the london_co dictionary.

All tasks must be completed using only the topics covered. That is, this task can be solved without using the if condition.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
```

(continues on next page)

(continued from previous page)

```
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Task 5.2

Ask the user to enter the IP network in the format: `10.1.1.0/24`.

Then print information about the network and mask in this format:

```
Network:
10        1         1         0
00001010  00000001  00000001  00000000

Mask:
/24
255       255       255       0
11111111  11111111  11111111  00000000
```

Check the script work on different network/mask combinations.

Hint: You can get the mask in binary format like this:

```
In [1]: "1" * 28 + "0" * 4
Out[1]: "11111111111111111111111111110000"
```

You can then take 8 bits of the binary mask using slices and convert them to decimal.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 5.2a

Copy and modify the script from task 5.2 so that, if the user entered a host address rather than a network address, convert the host address to a network address and print the network address and mask, as in task 5.2.

An example of a network address (all host bits are equal to zero):

- 10.0.1.0/24
- 190.1.0.0/16

Host address example:

- 10.0.1.1/24 - host from network 10.0.1.0/24
- 10.0.5.195/28 - host from network 10.0.5.192/28

If the user entered the address 10.0.1.1/24, the output should look like this:

```
Network:
10        0         1         0
00001010  00000000  00000001  00000000

Mask:
/24
255       255       255       0
11111111  11111111  11111111  00000000
```

Check the script work on different host/mask combinations, for example: 10.0.5.195/28, 10.0.1.1/24

Hint:

The network address can be calculated from the binary host address and the netmask. If the mask is 28, then the network address is the first 28 bits host addresses + 4 zeros. For example, the host address 10.1.1.195/28 in binary will be:

```
bin_ip = "00001010000000010000000111000011"
```

Then the network address will be the first 28 characters from bin_ip + 0000 (4 because in total there can be 32 bits in the address, and 32 - 28 = 4)

```
00001010000000010000000111000000
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 5.3

The script should prompt the user for input:

- interface mode (access/trunk)
- interface number (Gi0/3)
- VLAN number (for trunk mode, a list of VLANs will be entered)

Depending on the selected mode, print corresponding access or trunk configuration on stdout (command templates are in the lists access_template and trunk_template).

The output should first print the interface line and the interface number, and then the corresponding template in which the VLAN number (or the list of VLANs) is inserted.

Restriction: All tasks must be done using the topics covered in this and previous chapters. This task can be solved without using the if condition and for/while loops.

Hint: Leading up to this task was task 5.1. To make it easier to solve this task, you can look at task 5.1 and figure out exactly how different information is displayed in the task, depending on user input.

Below are examples of script execution to make it easier to understand the task.

An example of script execution when the access mode is selected:

```
$ python task_5_3.py
Enter interface mode (access/trunk): access
Enter type and interface number: Fa0/6
Enter number of vlan (vlans): 3

interface Fa0/6
switchport mode access
switchport access vlan 3
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

An example of script execution when the trunk mode is selected:

```
$ python task_5_3.py
Enter interface mode (access/trunk): trunk
Enter type and interface number: Fa0/7
Enter number of vlan (vlans): 2,3,4,5

interface Fa0/7
switchport trunk encapsulation dot1q
```

(continues on next page)

(continued from previous page)

```
switchport mode trunk
switchport trunk allowed vlan 2,3,4,5
```

```
access_template = [
    "switchport mode access", "switchport access vlan {}",
    "switchport nonegotiate", "spanning-tree portfast",
    "spanning-tree bpduguard enable"
]

trunk_template = [
    "switchport trunk encapsulation dot1q", "switchport mode trunk",
    "switchport trunk allowed vlan {}"
]
```

Task 5.3a

Copy and change the script from task 5.3 in such a way that, depending on the selected mode, different questions were asked in the request for the VLAN number or VLAN list:

- for access: 'Enter VLAN number:'
- for trunk: 'Enter the allowed VLANs:'

Restriction: All tasks must be done using the topics covered in this and previous chapters. This task can be solved without using the if condition and for/while loops.

```
access_template = [
    "switchport mode access", "switchport access vlan {}",
    "switchport nonegotiate", "spanning-tree portfast",
    "spanning-tree bpduguard enable"
]

trunk_template = [
    "switchport trunk encapsulation dot1q", "switchport mode trunk",
    "switchport trunk allowed vlan {}"
]
```

6. Control structures

So far, all code has been executed sequentially - all lines of script have been executed in order in which they are written in file. This section covers program flow control:

- branching with `if/elif/else`
- repeating actions using for and while loops
- exception handling with `try/except`

if/elif/else

The `if/elif/else` statement allows make branches during program execution. The program goes into branch when a certain condition is met.

In this statement only `if` is mandatory, `elif` and `else` are optional:

- `if` condition is always checked first.
- After `if` statement there must be some condition: if this condition is met (returns True), then actions in block `if` are executed.
- `elif` can be used to make multiple branches, that is, to check incoming data for different conditions.
- `elif` block is the same as `if` but it checked next. Roughly speaking, it is "otherwise if ..."
- There can be many `elif` blocks.
- `else` block is executed if none of conditions `if` or `elif` were true.

Example of if statement:

```
In [1]: a = 9

In [2]: if a == 10:
   ...:     print('a equal to 10')
   ...: elif a < 10:
   ...:     print('a less than 10')
   ...: else:
   ...:     print('a greater than 10')
   ...:
a less than 10
```

Condition

`If` expression is based on conditions: conditions are always written after `if` and `elif`. Blocks if/elif are executed only when condition returns True, so the first thing to deal with is what is true and what is false in Python.

True and False

In Python, apart from obvious True and False values, all other objects also have false or true value:

- True value:
  - any non-zero number
  - any non-empty string
  - any non-empty object
- False value:
  - 0
  - None
  - empty string
  - empty object

For example, since an empty list is a false value, it is possible to check whether list is empty:

```
In [12]: list_to_test = [1, 2, 3]

In [13]: if list_to_test:
   ....:     print("The list has objects")
   ....:
List has objects
```

The same result could have been achieved somewhat differently:

```
In [14]: if len(list_to_test) != 0:
   ....:     print("The list has objects")
   ....:
List has objects
```

Comparison operators

Comparison operators can be used in conditions like:

```
In [3]: 5 > 6
Out[3]: False

In [4]: 5 > 2
Out[4]: True

In [5]: 5 < 2
Out[5]: False

In [6]: 5 == 2
Out[6]: False

In [7]: 5 == 5
Out[7]: True

In [8]: 5 >= 5
Out[8]: True

In [9]: 5 <= 10
Out[9]: True

In [10]: 8 != 10
Out[10]: True
```

Note: Note that equality is checked by double ==.

Example of use of comparison operators:

```
In [1]: a = 9

In [2]: if a == 10:
   ...:     print('a equal to 10')
   ...: elif a < 10:
   ...:     print('a less than 10')
   ...: else:
   ...:     print('a greater than 10')
   ...:
a less than 10
```

Operator in

Operator `in` allows checking for the presence of element in a sequence (for example, element in a list or substrings in a string):

```
In [8]: 'Fast' in 'FastEthernet'
Out[8]: True

In [9]: 'Gigabit' in 'FastEthernet'
Out[9]: False

In [10]: vlan = [10, 20, 30, 40]

In [11]: 10 in vlan
Out[11]: True

In [12]: 50 in vlan
Out[12]: False
```

When used with dictionaries, `in` condition performs check by dictionary keys:

```
In [15]: r1 = {
   ....:  'IOS': '15.4',
   ....:  'IP': '10.255.0.1',
   ....:  'hostname': 'london_r1',
   ....:  'location': '21 New Globe Walk',
   ....:  'model': '4451',
   ....:  'vendor': 'Cisco'}

In [16]: 'IOS' in r1
Out[16]: True

In [17]: '4451' in r1
Out[17]: False
```

Operators and, or, not

Conditions can also use *logical operators* `and`, `or`, `not`:

```
In [15]: r1 = {
   ....:  'IOS': '15.4',
   ....:  'IP': '10.255.0.1',
   ....:  'hostname': 'london_r1',
```

(continues on next page)

(continued from previous page)

```
   ....:  'location': '21 New Globe Walk',
   ....:  'model': '4451',
   ....:  'vendor': 'Cisco'}

In [18]: vlan = [10, 20, 30, 40]

In [19]: 'IOS' in r1 and 10 in vlan
Out[19]: True

In [20]: '4451' in r1 and 10 in vlan
Out[20]: False

In [21]: '4451' in r1 or 10 in vlan
Out[21]: True

In [22]: not '4451' in r1
Out[22]: True

In [23]: '4451' not in r1
Out[23]: True
```

Operator and

In Python and operator returns not a boolean value but a value of one of operands.

If both operands are true, result is the last value:

```
In [24]: 'string1' and 'string2'
Out[24]: 'string2'

In [25]: 'string1' and 'string2' and 'string3'
Out[25]: 'string3'
```

If one of operators is a false, result of expression will be the first false value:

```
In [26]: '' and 'string1'
Out[26]: ''

In [27]: '' and [] and 'string1'
Out[27]: ''
```

Operator or

Operator `or`, like operator and, returns one of operands value.

When checking operands, the first true operand is returned:

```
In [28]: '' or 'string1'
Out[28]: 'string1'

In [29]: '' or [] or 'string1'
Out[29]: 'string1'

In [30]: 'string1' or 'string2'
Out[30]: 'string1'
```

If all values are false, the last value is returned:

```
In [31]: '' or [] or {}
Out[31]: {}
```

An important feature of `or` operator - operands, which are after the true operand, are not calculated:

```
In [33]: '' or sorted([44, 1, 67])
Out[33]: [1, 44, 67]

In [34]: '' or 'string1' or sorted([44, 1, 67])
Out[34]: 'string1'
```

Example of if/elif/else statement

An example of a check_password.py script that checks length of password and whether password contains username:

```
# -*- coding: utf-8 -*-

username = input('Enter username: ')
password = input('Enter password: ')

if len(password) < 8:
    print('Password is too short')
elif username in password:
    print('Password contains username')
else:
    print('Password for user {} is set'.format(username))
```

Script check:

```
$ python check_password.py
Enter username: nata
Enter password: nata1234
Password contains username

$ python check_password.py
Enter username: nata
Enter password: 123nata123
Password contains username

$ python check_password.py
Enter username: nata
Enter password: 1234
Password is too short

$ python check_password.py
Enter username: nata
Enter password: 123456789
Password for user nata is set
```

Ternary expression

It is sometimes more convenient to use a ternary operator than an extended form:

```
s = [1, 2, 3, 4]
result = True if len(s) > 5 else False
```

It is best not to abuse it but in simple terms such a record can be useful.

for

Very often the same step should be performed for a set of the same data type. For example, convert all strings in list to uppercase. Python uses `for` loop for such purposes.

`For` loop iterates elements of specified sequence and performs actions specified for each element.

Examples of sequences of elements that can be iterated by `for`:

- string
- list
- dictionary

- *range*
- Any *Iterable*

An example of converting strings in a list to uppercase without `for` loop:

```
In [1]: words = ['list', 'dict', 'tuple']

In [2]: upper_words = []

In [3]: words[0]
Out[3]: 'list'

In [4]: words[0].upper() # converting word to uppercase
Out[4]: 'LIST'

In [5]: upper_words.append(words[0].upper()) # converting and adding to new list

In [6]: upper_words
Out[6]: ['LIST']

In [7]: upper_words.append(words[1].upper())

In [8]: upper_words.append(words[2].upper())

In [9]: upper_words
Out[9]: ['LIST', 'DICT', 'TUPLE']
```

This solution has several nuances:

- the same action need to be repeated several times
- code is tied to a certain number of elements in *words* list

The same steps with the `for` loop:

```
In [10]: words = ['list', 'dict', 'tuple']

In [11]: upper_words = []

In [12]: for word in words:
    ...:     upper_words.append(word.upper())
    ...:

In [13]: upper_words
Out[13]: ['LIST', 'DICT', 'TUPLE']
```

Expression `for word in words: upper_words.append(word.upper())` means “for each word in

words list to perform actions in block for". In this case, word is the name of the variable, which refers to different values each iteration of the loop.

Note: The pythontutor project can be very helpful in understanding loops. The project visualize code execution and allows you to see what happens at every stage of code execution, which is especially useful in first steps of learning loops. The pythontutor allows you to upload your code, for instance, see example above.

For loop can work with any sequence of elements. For example, the above code used a list and the loop iterated over the elements of the list. The for loop works in a similar way with tuples.

When working with strings for loop iterates through string characters, for example:

```
In [1]: for letter in 'Test string':
   ...:     print(letter)
   ...:
T
e
s
t

s
t
r
i
n
g
```

Note: Loop uses a variable named *letter*. Although, it could be any name, it is better when name tells you which objects go through a loop.

Sometimes it is necessary to use sequence of numbers in loop. In this case, it is best to use *range*

Example of loop for with range() function:

```
In [2]: for i in range(10):
   ...:     print('interface FastEthernet0/{}'.format(i))
   ...:
interface FastEthernet0/0
interface FastEthernet0/1
interface FastEthernet0/2
interface FastEthernet0/3
interface FastEthernet0/4
```

(continues on next page)

(continued from previous page)

```
interface FastEthernet0/5
interface FastEthernet0/6
interface FastEthernet0/7
interface FastEthernet0/8
interface FastEthernet0/9
```

This loop uses `range(10)`. Function range() generates numbers in range from zero to specified number (in this example, up to 10) not including it.

In this example, loop runs through *vlans* list, so variable can be called *vlan*:

```
In [3]: vlans = [10, 20, 30, 40, 100]
In [4]: for vlan in vlans:
   ...:     print('vlan {}'.format(vlan))
   ...:     print(' name VLAN_{}'.format(vlan))
   ...:
vlan 10
 name VLAN_10
vlan 20
 name VLAN_20
vlan 30
 name VLAN_30
vlan 40
 name VLAN_40
vlan 100
 name VLAN_100
```

When a loop runs through dictionary, it actually goes through keys:

```
In [34]: r1 = {
    ...:       'ios': '15.4',
    ...:       'ip': '10.255.0.1',
    ...:       'hostname': 'london_r1',
    ...:       'location': '21 New Globe Walk',
    ...:       'model': '4451',
    ...:       'vendor': 'Cisco'}
    ...:

In [35]: for k in r1:
    ...:     print(k)
    ...:
ios
ip
hostname
```

(continues on next page)

(continued from previous page)

```
location
model
vendor
```

If you want to print key-value pairs in loop, you can do this:

```
In [36]: for key in r1:
    ...:     print(key + ' => ' + r1[key])
    ...:
ios => 15.4
ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe Walk
model => 4451
vendor => Cisco
```

Or use items() method which allows you to run loop over a key-value pair:

```
In [37]: for key, value in r1.items():
    ...:     print(key + ' => ' + value)
    ...:
ios => 15.4
ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe Walk
model => 4451
vendor => Cisco
```

Method items() returns a special view object that displays key-value pairs:

```
In [38]: r1.items()
Out[38]: dict_items([('ios', '15.4'), ('ip', '10.255.0.1'), ('hostname', 'london_
↪r1'), ('location', '21 New Globe Walk'), ('model', '4451'), ('vendor', 'Cisco
↪')])
```

Nested for

Loops `for` can be nested in each other.

In this example, *commands* is a list of commands to execute on each interface in the *fast_int* list:

```
In [7]: commands = ['switchport mode access', 'spanning-tree portfast', 'spanning-
↪tree bpduguard enable']
```

(continues on next page)

(continued from previous page)

```
In [8]: fast_int = ['0/1', '0/3', '0/4', '0/7', '0/9', '0/10', '0/11']

In [9]: for intf in fast_int:
   ...:     print('interface FastEthernet {}'.format(intf))
   ...:     for command in commands:
   ...:         print(' {}'.format(command))
   ...:
interface FastEthernet 0/1
 switchport mode access
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet 0/3
 switchport mode access
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet 0/4
 switchport mode access
 spanning-tree portfast
 spanning-tree bpduguard enable
...
```

The first `for` loop passes through interfaces in the *fast_int* list and the second through commands in *commands* list.

Combination for and if

Consider example of combining `for` and `if`.

Generate_access_port_config.py file:

```
1  access_template = ['switchport mode access',
2                     'switchport access vlan',
3                     'spanning-tree portfast',
4                     'spanning-tree bpduguard enable']
5
6  fast_int = {'access': {'0/12':10,
7                         '0/14':11,
8                         '0/16':17,
9                         '0/17':150}}
10
11 for intf, vlan in fast_int['access'].items():
12     print('interface FastEthernet' + intf)
13     for command in access_template:
```

(continues on next page)

(continued from previous page)

```
14          if command.endswith('access vlan'):
15              print(' {} {}'.format(command, vlan))
16          else:
17              print(' {}'.format(command))
```

Comments to the code:

- The first `for` loop iterates keys and values in nested fast_int['access'] dictionary
- At this moment of loop the current key is stored in *intf* variable
- At this moment of loop the current value is stored in *vlan* variable
- String "interface Fastethernet" is displayed with interface number added
- The second loop `for` iterates commands from *access_template* list
- Since *switchport access to vlan* command requires a VLAN number:
  - within second loop `for` commands are checked
  - if command ends with "access vlan"
    - command is displayed and VLAN number is added to it
  - in all other cases, command is simply displayed

Result of script execution:

```
$ python generate_access_port_config.py
interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/17
 switchport mode access
 switchport access vlan 150
```

(continues on next page)

(continued from previous page)

```
spanning-tree portfast
spanning-tree bpduguard enable
```

while

A `while` loop is another type of loop in Python.

In the while loop, as in the if statement, you need to write a condition. If the condition is true, the actions inside the while block are executed. In this case, unlike if, after executing the code in the block, while returns to the beginning of the loop.

When using `while` loops it is necessary to pay attention to whether the result when condition of loop is false will be reached.

Consider an example:

```
In [1]: a = 5

In [2]: while a > 0:
   ...:     print(a)
   ...:     a -= 1 # This record is equal to: a = a - 1
   ...:
5
4
3
2
1
```

First, variable A is created with the value 5.

Then, in `while` loop the condition `a > 0` is specified. That is, as long as the value of a is greater than 0, actions in the body of the loop will be executed. In this case, value of variable a will be displayed.

In addition, in the body of the loop, with each pass, the value of a is decreased by one.

Note: Record `a -= 1` can be a bit unusual. Python allows this format to be used instead of `a = a - 1`.

Similarly, you can write: `a += 1`, `a *= 2`, `a /= 2`.

Since the value of a is decreasing, the loop will not be infinite, and at some point the expression `a > 0` will become false.

The following example is based on example about password from section which describes `if` statement use *Example of if/elif/else statement*. In that example, you had to re-run the script if the

password did not meet the requirements. Using a while loop, you can make the script ask for a password again if it does not meet the requirements (check_password_with_while.py):

```
# -*- coding: utf-8 -*-

username = input('Enter username: ')
password = input('Enter password: ')

password_correct = False

while not password_correct:
    if len(password) < 8:
        print('Password is too short\n')
        password = input('Enter password once again: ')
    elif username in password:
        print('Password contains username\n')
        password = input('Enter password once again: ' )
    else:
        print('Password for user {} is set'.format(username))
        password_correct = True
```

In this case, `while` loop is useful because it returns script back to the beginning of checks and allows password to be typed again but does not require script to restart.

Now script works like this:

```
$ python check_password_with_while.py
Enter username: nata
Enter password: nata
Password is too short

Enter password once again: natanata
Password contains username

Enter password once again: 123345345345
Password for user nata is set
```

break, continue, pass

Python has several operators that allow to change default loop behavior.

Break operator

Operator `break` allows early termination of loop:

- break breaks current loop and continues executing the next expressions
- if multiple nested loops are used, break interrupts internal loop and continues to execute expressions following the block. Break can be used in loops for and while

Example of loop for:

```
In [1]: for num in range(10):
   ...:     if num < 7:
   ...:         print(num)
   ...:     else:
   ...:         break
   ...:
0
1
2
3
4
5
6
```

Example of a loop while:

```
In [2]: i = 0
In [3]: while i < 10:
   ...:     if i == 5:
   ...:         break
   ...:     else:
   ...:         print(i)
   ...:         i += 1
   ...:
0
1
2
3
4
```

Using break in the password request example (check_password_with_while_break.py file):

```
username = input('Enter username: ')
password = input('Enter password: ')

while True:
    if len(password) < 8:
        print('Password is too short\n')
    elif username in password:
```

(continues on next page)

(continued from previous page)

```
        print('Password contains username\n')
    else:
        print('Password for user {} is set'.format(username))
        # exit while loop
        break
    password = input('Enter password once again: ')
```

Now it is possible not to repeat string `password = input('Enter password once again: ')` in each branch, it is enough to move it to the end of loop.

And as soon as correct password is entered, `break` will take the program out of loop `while`.

Continue operator

Operator `continue` returns control to the beginning of loop. That is, `continue` allows to «jump» remaining expressions in loop and go to the next iteration.

Example of a loop `for`:

```
In [4]: for num in range(5):
   ...:     if num == 3:
   ...:         continue
   ...:     else:
   ...:         print(num)
   ...:
0
1
2
4
```

Example of a loop `while`:

```
In [5]: i = 0
In [6]: while i < 6:
   ....:     i += 1
   ....:     if i == 3:
   ....:         print("Skip 3")
   ....:         continue
   ....:         print("No one will see it")
   ....:     else:
   ....:         print("Current value: ", i)
   ....:
Current value:  1
Current value:  2
```

(continues on next page)

(continued from previous page)

```
Skip 3
Current value:  4
Current value:  5
Current value:  6
```

Use of continue in example with password request (check_password_with_while_continue.py file):

```
username = input('Enter username: ')
password = input('Enter password: ')

password_correct = False

while not password_correct:
    if len(password) < 8:
        print('Password is too short\n')
    elif username in password:
        print('Password contains username\n')
    else:
        print('Password for user {} is set'.format(username))
        password_correct = True
        continue
    password = input('Enter password once again: ')
```

Here you can exit loop by checking password_correct flag. When correct password is entered, flag is set to True and with continue a jump to the beginning of loop is occurred by skipping the last line with password request.

The result will be:

```
$ python check_password_with_while_continue.py
Enter username: nata
Enter password: nata12
Password is too short

Enter password once again: natalksdjflsdjf
Password contains username

Enter password once again: asdfsujljhdflaskjdfh
Password for user nata is set
```

Pass operator

Operator pass does nothing. Basically it is a placeholder.

For example, pass can help when you need to specify a script structure. It can be set in loops, functions, classes. And it won't affect execution of code.

Example of using pass:

```
In [6]: for num in range(5):
   ....:     if num < 3:
   ....:         pass
   ....:     else:
   ....:         print(num)
   ....:
3
4
```

for/else, while/else

In loops `for` and `while` you may optionally use `else` block.

for/else

In loop `for`:

- block `else` is executed if loop has completed iteration of list
- but it *does not execute* if `break` was applied in loop.

Example of loop `for` with `else` (block `else` is executed after loop `for`):

```
In [1]: for num in range(5):
   ....:     print(num)
   ....: else:
   ....:     print("Run out of numbers")
   ....:
0
1
2
3
4
Run out of numbers
```

An example of loop `for` with `else` and `break` in loop (because of `break`, block `else` is not applied):

```
In [2]: for num in range(5):
   ....:     if num == 3:
   ....:         break
```

(continues on next page)

(continued from previous page)

```
   ....:     else:
   ....:         print(num)
   ....: else:
   ....:     print("Run out of numbers")
   ....:
0
1
2
```

Example of loop `for` with `else` and `continue` in loop (`continue` does not affect `else` block):

```
In [3]: for num in range(5):
   ....:     if num == 3:
   ....:         continue
   ....:     else:
   ....:         print(num)
   ....: else:
   ....:     print("Run out of numbers")
   ....:
0
1
2
4
Run out of numbers
```

while/else

In loop `while`:

- block `else` is executed if loop has completed iteration of list
- but it *does not execute* if `break` was applied in loop.

Example of a loop `while` with `else` (block `else` runs after loop `while`):

```
In [4]: i = 0
In [5]: while i < 5:
   ....:     print(i)
   ....:     i += 1
   ....: else:
   ....:     print("The End")
   ....:
0
1
```

(continues on next page)

(continued from previous page)

```
2
3
4
The End
```

An example of a loop `while` with `else` and `break` in loop (because of `break`, block `else` is not applied):

```
In [6]: i = 0

In [7]: while i < 5:
   ....:     if i == 3:
   ....:         break
   ....:     else:
   ....:         print(i)
   ....:         i += 1
   ....: else:
   ....:     print("The End")
   ....:
0
1
2
```

Working with try/except/else/finally

try/except

If you repeated examples that were used before, there could be situations where a mistake was made. It was probably a syntax error when a colon was missing, for example. Python generally reacts quite understandably to such errors and they can easily be corrected. However, even if the code is syntactically correct, errors can occur. In Python, these errors are called *exceptions*.

Examples of exceptions:

```
In [1]: 2/0
---------------------------------------------------------
ZeroDivisionError: division by zero

In [2]: 'test' + 2
---------------------------------------------------------
TypeError: must be str, not int
```

In this case, two exceptions were raised: `ZeroDivisionError` and `TypeError`.

Most often, it is possible to predict what kind of exceptions will occur during execution of the program. For example, if program expects two numbers in input and output returns their sum and user has entered a string instead of one of numbers a TypeError error will appear as in example above.

Python allows working with exceptions. They can be intercepted and acted upon if an exception has been occurred.

Note: When an exception appears, program is immediately interrupted.

In order to work with exceptions `try/except` statement is used:

```
In [3]: try:
   ...:     2/0
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero")
   ...:
You can't divide by zero
```

The `try` statement works as follows:

- first execute expressions that are written in `try` block
- if there are no exceptions during execution of `try` block, block `except` is skipped and the following code is executed
- if there is an exception within `try` block, the rest part of `try` block is skipped
  - if `except` block contains an exception which has been occurred, code in `except` block is executed
  - if exception that has raised is not specified in `except` block, program execution is interrupted and an error is generated

Note that `Cool!` string in `try` block is not displayed:

```
In [4]: try:
   ...:     print("Let's divide some numbers")
   ...:     2/0
   ...:     print('Cool!')
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero")
   ...:
Let's divide some numbers
You can't divide by zero
```

try/except statement may have many `except` if different actions are needed depending on type of error.

For example, divide.py script divides two numbers entered by user:

```
# -*- coding: utf-8 -*-

try:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    print("Result: ", int(a)/int(b))
except ValueError:
    print("Please enter only numbers")
except ZeroDivisionError:
    print("You can't divide by zero")
```

Examples of script execution:

```
$ python divide.py
Enter first number: 3
Enter second number: 1
Result:  3

$ python divide.py
Enter first number: 5
Enter second number: 0
You can't divide by zero

$ python divide.py
Enter first number: qewr
Enter second number: 3
Please enter only numbers
```

In this case, ValueError exception raised when user has entered a string instead of a number. ZeroDivisionError exception raised if second number is 0.

If you do not need to print different messages on ValueError and ZeroDivisionError, you can do this (divide_ver2.py file):

```
# -*- coding: utf-8 -*-

try:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    print("Result: ", int(a)/int(b))
except (ValueError, ZeroDivisionError):
    print("Something went wrong...")
```

Verification:

```
$ python divide_ver2.py
Enter first number: wer
Enter second number: 4
Something went wrong...

$ python divide_ver2.py
Enter first number: 5
Enter second number: 0
Something went wrong...
```

Note: In block except you don't have to specify a specific exception or exceptions. In that case, all exceptions would be intercepted.

That is not recommended!

try/except/else

Try/except has an optional `else` block. It is implemented if there is no exception.

For example, if you need to perform any further operations with data that user entered, you can write them in `else` block (divide_ver3.py file):

```
# -*- coding: utf-8 -*-

try:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Something went wrong...")
else:
    print("Result is squared: ", result``2)
```

Example of execution:

```
$ python divide_ver3.py
Enter first number: 10
Enter second number: 2
Result is squared:  25

$ python divide_ver3.py
Enter first number: werq
```

(continues on next page)

(continued from previous page)

```
Enter second number: 3
Something went wrong...
```

try/except/finally

Block `finally` is another optional block in `try` statement. It is *always* implemented, whether an exception has been raised or not. It's about actions that you have to do anyway. For example, it could be a file closing.

File divide_ver4.py с блоком finally:

```
# -*- coding: utf-8 -*-

try:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Something went wrong...")
else:
    print("Result is squared: ", result``2)
finally:
    print("And they lived happily ever after.")
```

Verification:

```
$ python divide_ver4.py
Enter first number: 10
Enter second number: 2
Result is squared:  25
And they lived happily ever after.

$ python divide_ver4.py
Enter first number: qwerewr
Enter second number: 3
Something went wrong...
And they lived happily ever after.

$ python divide_ver4.py
Enter first number: 4
Enter second number: 0
Something went wrong...
And they lived happily ever after.
```

When to use exceptions

As a rule, same code can be written with or without exceptions.

For example, this version of code:

```
while True:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    try:
        result = int(a)/int(b)
    except ValueError:
        print("Only digits are supported")
    except ZeroDivisionError:
        print("You can't divide by zero")
    else:
        print(result)
        break
```

You can rewrite this without try/except (try_except_divide.py file):

```
while True:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
    if a.isdigit() and b.isdigit():
        if int(b) == 0:
            print("You can't divide by zero")
        else:
            print(int(a)/int(b))
            break
    else:
        print("Only digits are supported")
```

But the same option without exceptions will not always be simple and understandable.

It is important to assess in each specific situation which version of code is more comprehensible, compact and universal - with or without exceptions.

If you've used some other programming language before, it's possible that use of exceptions was considered a bad form. In Python this is not true. To get a little bit more into this issue, look at the links to additional material at the end of this section.

Further reading

Documentation:

- Compound statements (if, while, for, try)
- break, continue
- Errors and Exceptions
- Built-in Exceptions

Articles:

- Write Cleaner Python: Use Exceptions
- Robust exception handling
- Python Exception Handling Techniques

Stack Overflow:

- Why does python use 'else' after for and while loops?
- Is it a good practice to use try-except-else in Python?

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 6.1

The mac list contains MAC addresses in the format XXXX:XXXX:XXXX However, in Cisco equipment MAC addresses are in XXXX.XXXX.XXXX format.

Write a code that converts MAC addresses to cisco format and adds them to a new list named result. Print the result list to the stdout using print function.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
mac = ["aabb:cc80:7000", "aabb:dd80:7340", "aabb:ee80:7000", "aabb:ff80:7000"]
```

Task 6.2

Prompt the user to enter an IP address in the format 10.0.1.1. Depending on the type of address (described below), print to the stdout:

- 'unicast' - if the first byte is in the range 1-223
- 'multicast' - if the first byte is in the range 224-239
- 'local broadcast' - if the IP address is 255.255.255.255
- 'unassigned' - if the IP address is 0.0.0.0
- 'unused' - in all other cases

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 6.2a

Make a copy of the code from the task 6.2.

Add verification of the entered IP address. An IP address is considered correct if it:

- consists of 4 numbers (not letters or other symbols)

- numbers are separated by a dot
- every number in the range from 0 to 255

If the IP address is incorrect, print the message: 'Invalid IP address'

The message "Invalid IP address" should be printed only once, even if several points above are not met.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 6.2b

Make a copy of the code from the task 6.2a.

Add this functionality: If the address was entered incorrectly, request the address again.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 6.3

A configuration generator for access ports is made in the script. Make a similar configuration generator for trunk ports.

In trunks, the situation is complicated by the fact that there can be many VLANs, and you need to understand what to do with them (add, delete, overwrite).

Therefore, in accordance with each port there is a list and the first (zero index) element of the list specifies how to interpret VLAN numbers that follow.

Dict value and corresponding command:

- `['add', '10', '20']` - switchport trunk allowed vlan add 10,20
- `['del', '17']` - switchport trunk allowed vlan remove 17
- `['only', '11', '30']` - switchport trunk allowed vlan 11,30

Task for ports 0/1, 0/2, 0/4:

- generate a configuration based on the trunk_template template
- taking into account the keywords add, del, only

The code should not be tied to specific port numbers. I.e, if there are other interface numbers in the trunk dictionary, the code should work.

For data in the trunk_template dictionary, output to the standard output should be like this:

```
interface FastEthernet 0/1
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan add 10,20
interface FastEthernet 0/2
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 11,30
interface FastEthernet 0/4
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan remove 17
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
access_template = [
    "switchport mode access",
    "switchport access vlan",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable",
]

trunk_template = [
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk allowed vlan",
]

access = {"0/12": "10", "0/14": "11", "0/16": "17", "0/17": "150"}
trunk = {"0/1": ["add", "10", "20"], "0/2": ["only", "11", "30"], "0/4": ["del",
↪"17"]}

for intf, vlan in access.items():
    print("interface FastEthernet" + intf)
    for command in access_template:
        if command.endswith("access vlan"):
            print(f" {command} {vlan}")
        else:
            print(f" {command}")
```

7. Working with files

In real life, in order to make full use of everything covered before this section you need to understand how to work with files.

When working with network equipment (and not only), files can be:

- configurations (simple, non-structured text files)
  - They are discussed in this section
- configuration templates
  - usually a special file format.
  - section Jinja configuration temlates discusses the use of Jinja2 to create configuration templates
- files with connection options
  - usually they are structured files in some particular format: YAML, JSON, CSV
    - section Data serialization discusses how to handle such files
- other Python scripts
  - section Modules discusses how to work with modules (other Python scripts)

This section covers simple text files. For example, Cisco configuration file.

There are several aspects to working with files:

- opening/closing
- reading
- writing

This section covers only the minimum required for working with files. More in Python documentation.

File opening

To start working with a file you have to open it.

open

Function open is most often used to open files:

```
file = open('file_name.txt', 'r')
```

In open() function:

- `'file_name.txt'` - file name
- You can specify not only the name but also the path (absolute or relative)
- `'r'` - file opening mode

Function open creates a **file** object to which different methods can then be applied to work with it.

File opening modes:

- `r` - open file in read-only mode (default)
- `r+` - open file for reading and writing
- `w` - open file for writing only
- if file exists, its content is removed
- if file does not exist, a new one is created
- `w+` - open file for reading and writing
- if file exists, its content is removed
- if file does not exist, a new one is created
- `a` - open file to add a data. Data is added to the end of file
- `a+` - open file for reading and writing. Data is added to the end of file

---

Note: r - read; a - append; w - write

File reading

Python has several file reading methods:

- `read` - reads the contents of file to string
- `readline` - reads file line by line
- `readlines` - reads file lines and creates a list from the lines

Let's see how to read contents of files using the example of r1.txt:

```
!
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
```

(continues on next page)

(continued from previous page)

```
!
ip ssh version 2
!
```

read

Method read reads the entire file to one string:

```
In [1]: f = open('r1.txt')

In [2]: f.read()
Out[2]: '!\nservice timestamps debug datetime msec localtime show-timezone␣
↪year\nservice timestamps log datetime msec localtime show-timezone␣
↪year\nservice password-encryption\nservice sequence-numbers\n!\nno ip domain␣
↪lookup\n!\nip ssh version 2\n!\n'

In [3]: f.read()
Out[3]: ''
```

When reading a file once again an empty line is displayed in line 3. This is because the whole file is read when read method is called. And after the file has been read the cursor stays at the end of file. The cursor position can be controlled by seek method.

readline

File can be read line by line using readline method:

```
In [4]: f = open('r1.txt')

In [5]: f.readline()
Out[5]: '!\n'

In [6]: f.readline()
Out[6]: 'service timestamps debug datetime msec localtime show-timezone year\n'
```

But most often it is easier to walk through a **file** object in a loop without using read... methods:

```
In [7]: f = open('r1.txt')

In [8]: for line in f:
   ...:     print(line)
   ...:
```

(continues on next page)

(continued from previous page)

```
!

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

!
```

readlines

Another useful method is `readlines`. It reads file lines to the list:

```
In [9]: f = open('r1.txt')

In [10]: f.readlines()
Out[10]:
['!\n',
 'service timestamps debug datetime msec localtime show-timezone year\n',
 'service timestamps log datetime msec localtime show-timezone year\n',
 'service password-encryption\n',
 'service sequence-numbers\n',
 '!\n',
 'no ip domain lookup\n',
 '!\n',
 'ip ssh version 2\n',
 '!\n']
```

If you want to get lines of a file but without a new line character at the end, you can use `split` method and specify symbol `\n` as a separator:

```
In [11]: f = open('r1.txt')

In [12]: f.read().split('\n')
Out[12]:
['!',
 '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',
 '!',
 '']
```

Note that the last item in list is an empty string.

If you use `split` before `rstrip`, list will be without empty string at the end:

```
In [13]: f = open('r1.txt')

In [14]: f.read().rstrip().split('\n')
Out[14]:
['!',
 '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',
 '!']
```

seek

Until now, file had to be reopened to read it again. This is because after reading methods a cursor is at the end of the file. And second reading returns an empty string.

To read information from a file again you need to use the seek method which moves the cursor to the desired position.

Example of file opening and content reading:

```
In [15]: f = open('r1.txt')

In [16]: print(f.read())
!
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
!
```

If you call read method again an empty string returns:

```
In [17]: print(f.read())
```

But with seek method you can go to the beginning of file (0 means the beginning of file):

```
In [18]: f.seek(0)
```

Once cursor has been set to the beginning of file you can read the content again:

```
In [19]: print(f.read())
!
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
!
```

File writing

When writing information to a file, it is very important to specify the correct mode for opening the file, so as not to accidentally delete it:

- w - open file for writing. If file exists, its content is removed
- a - open file to add data. Data is appended to the end of the file

Both modes create a file if it does not exist.

The following methods are used to write to a file:

- `write` - write one line to file
- `writelines` - allows to send as argument a list of strings

write

Method `write` expects string to write.

For example, take a list of lines with configuration:

```
In [1]: cfg_lines = ['!',
   ...:  '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',
   ...:  '!']
```

Open r2.txt file in write mode:

```
In [2]: f = open('r2.txt', 'w')
```

Convert the list of commands to one large string using `join`:

```
In [3]: cfg_lines_as_string = '\n'.join(cfg_lines)

In [4]: cfg_lines_as_string
Out[4]: '!\nservice timestamps debug datetime msec localtime show-timezone␣
↪year\nservice timestamps log datetime msec localtime show-timezone␣
↪year\nservice password-encryption\nservice sequence-numbers\n!\nno ip domain␣
↪lookup\n!\nip ssh version 2\n!'
```

Write a string to a file:

```
In [5]: f.write(cfg_lines_as_string)
```

Similarly, you can add a string manually:

```
In [6]: f.write('\nhostname r2')
```

After work with file is finished, it should be closed:

```
In [7]: f.close()
```

Since ipython supports *cat* command, you can easily see the content of file:

```
In [8]: cat r2.txt
!
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
!
hostname r2
```

writelines

Method `writelines` expects list of strings as an argument.

Writing cfg_lines list into the file:

```
In [1]: cfg_lines = ['!',
   ...:  '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',
   ...:  '!']

In [9]: f = open('r2.txt', 'w')

In [10]: f.writelines(cfg_lines)

In [11]: f.close()

In [12]: cat r2.txt
!service timestamps debug datetime msec localtime show-timezone yearservice␣
↪timestamps log datetime msec localtime show-timezone yearservice password-
↪encryptionservice sequence-numbers!no ip domain lookup!ip ssh version 2!
```

(continues on next page)

(continued from previous page)

As a result, all lines in the list were written into one line because there was no symbol \n at the end of lines. You can add newline character in different ways. For example, you can loop through a list:

```
In [13]: cfg_lines2 = []

In [14]: for line in cfg_lines:
   ....:     cfg_lines2.append(line + '\n')
   ....:

In [15]: cfg_lines2
Out[15]:
['!\n',
 'service timestamps debug datetime msec localtime show-timezone year\n',
 'service timestamps log datetime msec localtime show-timezone year\n',
 'service password-encryption\n',
 'service sequence-numbers\n',
 '!\n',
 'no ip domain lookup\n',
 '!\n',
 'ip ssh version 2\n',
```

If the final list is written anew to the file, then it will already contain newlines:

```
In [18]: f = open('r2.txt', 'w')

In [19]: f.writelines(cfg_lines2)

In [20]: f.close()

In [21]: cat r2.txt
!
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
!
```

File closing

Note: In real life, the most common way to close files is use of `with` statement. It's much more convenient way than to close file explicitly. But since you can also find `close` method in life, this section discusses how to use it.

After you finish working with file you have to close it. In some cases Python can close file itself. But it's best not to count on it and close file explicitly.

close

Method close() met in File writing section. It was there to make sure that the content of file was written on disk.

For this, Python has a separate `flush` method. But since in example with file writing there was no need to perform any more operations, file could be closed.

Open the r1.txt file:

```
In [1]: f = open('r1.txt', 'r')
```

You can now read the content:

```
In [2]: print(f.read())
!
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
!
```

The **file** object has a special `closed` attribute that lets you check whether a file is closed or not. If file is open, it returns `False`:

```
In [3]: f.closed
Out[3]: False
```

Now close file and check `closed` again:

```
In [4]: f.close()

In [5]: f.closed
Out[5]: True
```

If you try to read file an exception will be raised:

```
In [6]: print(f.read())
---------------------------------------------------------------------
ValueError                          Traceback (most recent call last)
<ipython-input-53-2c962247edc5> in <module>()
----> 1 print(f.read())

ValueError: I/O operation on closed file
```

with statement

The with statement is called the context manager.

Python has a more convenient way of working with files than the ones used so far - statement with:

```
In [1]: with open('r1.txt', 'r') as f:
  ....:     for line in f:
  ....:         print(line)
  ....:
!

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

```

(continues on next page)

(continued from previous page)

```
!
```

In addition, statement `with` guarantees file closure automatically.

Pay attention to how lines of the file are read:

```
for line in f:
    print(line)
```

When file needs to be run line by line, it is best to use this option.

In previous output there were extra empty lines between lines of the file because `print` adds another new line character.

To get rid of this you can use `rstrip` method:

```
In [2]: with open('r1.txt', 'r') as f:
  ....:     for line in f:
  ....:         print(line.rstrip())
  ....:
!
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
!

In [3]: f.closed
Out[3]: True
```

And of course, `with` statement can be used not only as a line-by-line reader, all methods that have been covered before also work:

```
In [4]: with open('r1.txt', 'r') as f:
  ....:     print(f.read())
  ....:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
```

(continues on next page)

(continued from previous page)

```
!
no ip domain lookup
!
ip ssh version 2
!
```

Open two files

Sometimes you have to work with two files simultaneously. For example, write some lines from one file to another.

In this case you can open two files in `with` block as follows:

```
In [5]: with open('r1.txt') as src, open('result.txt', 'w') as dest:
   ...:     for line in src:
   ...:         if line.startswith('service'):
   ...:             dest.write(line)
   ...:

In [6]: cat result.txt
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
```

This is equivalent to:

```
In [7]: with open('r1.txt') as src:
   ...:     with open('result.txt', 'w') as dest:
   ...:         for line in src:
   ...:             if line.startswith('service'):
   ...:                 dest.write(line)
   ...:
```

Examples of working with files

This subsection covers working with files and brings together topics: files, loops, and conditions.

When processing output of commands or configuration, often it will be necessary to write summary data to the dictionary. It is not always obvious how to handle the output of commands and how to deal with the output in general. This subsection discusses several examples with increasing complexity.

Parsing column output

This example will deal with the output of `sh ip int br` command. From the output of command we need to get interface name and IP address. Interface name is dictionary key and IP address is value. At the same time, match must be made only for those interfaces with IP address assigned.

An example of `sh ip int br` output (sh_ip_int_br.txt file):

```
R1#show ip interface brief
Interface              IP-Address      OK? Method Status        Protocol
FastEthernet0/0        15.0.15.1       YES manual up            up
FastEthernet0/1        10.0.12.1       YES manual up            up
FastEthernet0/2        10.0.13.1       YES manual up            up
FastEthernet0/3        unassigned      YES unset  up            down
Loopback0              10.1.1.1        YES manual up            up
Loopback100            100.0.0.1       YES manual up            up
```

Working_with_dict_example_1.py file:

```
result = {}

with open('sh_ip_int_br.txt') as f:
    for line in f:
        line = line.split()
        if line and line[1][0].isdigit():
            interface, address, *other = line
            result[interface] = address

print(result)
```

Command `sh ip int br` displays the output with columns. So desired fields are in the same line. Script processes the output line by line and divides each line using split() method.

The resulting list contains output columns. Because we need only interfaces on which IP address is configured, first character of second column is checked: if first character is a number the address is assigned to interface and string has to be processed.

In `interface, address, *other = line` - variables are unpacked. Variable *interface* will have interface name, *address* will have IP address and *other* - all other fields. Since each line has a key-value pair, they are assigned to dictionary: `result[interface] = address`.

The result of script execution will be a dictionary (here it is split into key-value pairs for convenience, in real script the dictionary output will be displayed in one line):

```
{'FastEthernet0/0': '15.0.15.1',
 'FastEthernet0/1': '10.0.12.1',
 'FastEthernet0/2': '10.0.13.1',
```

(continues on next page)

(continued from previous page)

```
 'Loopback0': '10.1.1.1',
 'Loopback100': '100.0.0.1'}
```

Getting key and value from different output lines

Very often the output of commands looks like that key and value are in different lines. And you have to figure out how to process the output to get right match. For example, from the output of *sh ip int br* command you need to get match *interface name – MTU* (sh_ip_interface.txt file):

```
Ethernet0/0 is up, line protocol is up
  Internet address is 192.168.100.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/1 is up, line protocol is up
  Internet address is 192.168.200.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/2 is up, line protocol is up
  Internet address is 19.1.1.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
```

Interface name is in `Ethernet0/0 is up, line protocol is up` line and MTU in `MTU is 1500 bytes` line.

For example, try to remember interface each time and print its value when MTU parameter is detected, together with MTU value:

```
In [2]: with open('sh_ip_interface.txt') as f:
   ...:     for line in f:
   ...:         if 'line protocol' in line:
   ...:             interface = line.split()[0]
   ...:         elif 'MTU is' in line:
   ...:             mtu = line.split()[-2]
```

(continues on next page)

(continued from previous page)

```
   ...:             print('{:15}{}'.format(interface, mtu))
   ...:
Ethernet0/0    1500
Ethernet0/1    1500
Ethernet0/2    1500
Ethernet0/3    1500
Loopback0      1514
```

Command output is organized in such a way that there is always a line with interface first and then a line with MTU after several lines. If you remember the name of interface every time it appears and at the time when line matches MTU, the last memorized interface is the one which matches this MTU. Now, if you want to create a dictionary that matches *interface – MTU*, it's enough to write values when MTU was found.

Working_with_dict_example_2.py file:

```
result = {}

with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'MTU is' in line:
            mtu = line.split()[-2]
            result[interface] = mtu

print(result)
```

The result of script execution will be a dictionary (here it is split into key-value pairs for convenience, in real script the dictionary output will be displayed in one line):

```
{'Ethernet0/0': '1500',
 'Ethernet0/1': '1500',
 'Ethernet0/2': '1500',
 'Ethernet0/3': '1500',
 'Loopback0': '1514'}
```

This technique will be quite often useful because command output is generally organized in a very similar way.

Nested dictionary

If you want to get several parameters from the output, it is very convenient to use a dictionary with a nested dictionary. For example, from output `sh ip interface` you need to get two parameters:

IP address and MTU. First, output of information:

```
Ethernet0/0 is up, line protocol is up
  Internet address is 192.168.100.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/1 is up, line protocol is up
  Internet address is 192.168.200.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/2 is up, line protocol is up
  Internet address is 19.1.1.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
```

In the first step, each value is stored in a variable and then all three values are displayed. Values are displayed when a string has MTU because it is the last string:

```
In [2]: with open('sh_ip_interface.txt') as f:
   ...:     for line in f:
   ...:         if 'line protocol' in line:
   ...:             interface = line.split()[0]
   ...:         elif 'Internet address' in line:
   ...:             ip_address = line.split()[-1]
   ...:         elif 'MTU' in line:
   ...:             mtu = line.split()[-2]
   ...:             print('{:15}{:17}{}'.format(interface, ip_address, mtu))
   ...:
Ethernet0/0    192.168.100.1/24 1500
Ethernet0/1    192.168.200.1/24 1500
Ethernet0/2    19.1.1.1/24      1500
Ethernet0/3    192.168.230.1/24 1500
Loopback0      4.4.4.4/32       1514
```

It uses the same technique as in previous example but adds another nested dictionary:

```
result = {}

with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
            result[interface] = {}
        elif 'Internet address' in line:
            ip_address = line.split()[-1]
            result[interface]['ip'] = ip_address
        elif 'MTU' in line:
            mtu = line.split()[-2]
            result[interface]['mtu'] = mtu

print(result)
```

Each time an interface is detected, result dictionary creates a key with the name of interface that corresponds to an empty dictionary. This blank is used so that at the time when IP address or MTU is detected, parameter can be written into nested dictionary of the corresponding interface.

The result of script execution will be a dictionary (here it is split into key-value pairs for convenience, in real script the dictionary output will be displayed in one line):

```
{'Ethernet0/0': {'ip': '192.168.100.1/24', 'mtu': '1500'},
 'Ethernet0/1': {'ip': '192.168.200.1/24', 'mtu': '1500'},
 'Ethernet0/2': {'ip': '19.1.1.1/24', 'mtu': '1500'},
 'Ethernet0/3': {'ip': '192.168.230.1/24', 'mtu': '1500'},
 'Loopback0': {'ip': '4.4.4.4/32', 'mtu': '1514'}}
```

Output with empty values

Sometimes, sections with empty values will be found in the output. For example, in case of output `sh ip interface`, interfaces may look like:

```
Ethernet0/1 is up, line protocol is up
  Internet protocol processing disabled
Ethernet0/2 is administratively down, line protocol is down
  Internet protocol processing disabled
Ethernet0/3 is administratively down, line protocol is down
  Internet protocol processing disabled
```

Consequently, there is no MTU or IP address. And if you execute previous script for a file with such interfaces, the result is this (output for file sh_ip_interface2.txt):

```
{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Ethernet0/1': {},
 'Ethernet0/2': {},
 'Ethernet0/3': {},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}
```

If you need to add interfaces to dictionary only when an IP address is assigned to interface, you need to move the creation of key with interface name to a moment when line with IP address is detected (working_with_dict_example_4.py file):

```
result = {}

with open('sh_ip_interface2.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'Internet address' in line:
            ip_address = line.split()[-1]
            result[interface] = {}
            result[interface]['ip'] = ip_address
        elif 'MTU' in line:
            mtu = line.split()[-2]
            result[interface]['mtu'] = mtu

print(result)
```

In this case, the result will be a dictionary:

```
{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}
```

Further reading

Documentation:

- Reading and Writing Files
- The with statement

Articles:

- The Python “with” Statement by Example

Stack Overflow:

- What is the python “with” statement designed for?

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 7.1

Process the lines from the ospf.txt file and print information for each line in this form to the stdout:

```
Prefix                10.0.24.0/24
AD/Metric             110/41
Next-Hop              10.0.13.3
Last update           3d18h
Outbound Interface    FastEthernet0/0
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 7.2

Create a script that will process the config_sw1.txt configuration file. The filename is passed as an argument to the script.

The script should return to the stdout commands from the passed configuration file, excluding lines that start with '!'.

There should be no blank lines in the output.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Output example:

```
$ python task_7_2.py config_sw1.txt
Current configuration : 2033 bytes
version 15.0
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
hostname sw1
interface Ethernet0/0
```

(continues on next page)

(continued from previous page)

```
 duplex auto
interface Ethernet0/1
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan 100
 switchport mode trunk
 duplex auto
 spanning-tree portfast edge trunk
interface Ethernet0/2
 duplex auto
interface Ethernet0/3
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan 100
 duplex auto
 switchport mode trunk
 spanning-tree portfast edge trunk
...
```

Task 7.2a

Make a copy of the code from the task 7.2.

Add this functionality: The script should not print to the stdout commands, which contain words from the ignore list. The script should also not print lines that begin with !.

Check the script on the config_sw1.txt configuration file. The filename is passed as an argument to the script.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
ignore = ["duplex", "alias", "Current configuration"]
```

Task 7.2b

Make a copy of the code from the task 7.2a. Add this functionality: instead of printing to stdout, the script should write the resulting lines to a file.

File names must be passed as arguments to the script:

1. name of the source configuration file
2. name of the destination configuration file

In this case, the lines that are contained in the ignore list and lines that start with ! must be filtered.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
ignore = ["duplex", "alias", "Current configuration"]
```

Task 7.3

The script should process the lines in the CAM_table.txt file. Each line, where there is a MAC address, must be handled in such a way that the following table was printed on the stdout:

```
100    01bb.c580.7000   Gi0/1
200    0a4b.c380.7000   Gi0/2
300    a2ab.c5a0.7000   Gi0/3
100    0a1b.1c80.7000   Gi0/4
500    02b1.3c80.7000   Gi0/5
200    1a4b.c580.7000   Gi0/6
300    0a1b.5c80.7000   Gi0/7
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 7.3a

Make a copy of the code from the task 7.3.

Add this functionality: Sort output by VLAN number

As a result, you should get the following output:

```
10       01ab.c5d0.70d0      Gi0/8
10       0a1b.1c80.7000      Gi0/4
100      01bb.c580.7000      Gi0/1
200      0a4b.c380.7c00      Gi0/2
200      1a4b.c580.7000      Gi0/6
300      0a1b.5c80.70f0      Gi0/7
300      a2ab.c5a0.700e      Gi0/3
500      02b1.3c80.7b00      Gi0/5
1000     0a4b.c380.7d00      Gi0/9
```

Pay attention to vlan 1000 - it should be displayed last. Correct sorting can be achieved if vlan is a number, not a string.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 7.3b

Make a copy of the code from the task 7.3a.

Add this functionality:

- Ask the user to enter the VLAN number.
- Print information only for the specified VLAN.

Output example:

```
Enter VLAN number: 10
10        0a1b.1c80.7000      Gi0/4
10        01ab.c5d0.70d0      Gi0/8
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

8. Python basic examples

This section covers topics that were not included in the previous sections and also provides examples of using Python to solve problems.

While most examples will be file-oriented the same data-processing principles can be applied to network equipment. Only part with reading from file will be replaced to get output from hardware.

Formatting lines with f-strings

Python 3.6 added a new version of string formatting - f-strings or interpolation of strings. The f-strings allow not only to set values to template but also to perform calls to functions, methods, etc.

In many situations f-strings are easier to use than `format` and f-strings work faster than `format` and other methods of string formatting.

Syntax

F-string is a string literal with a letter f in front of it. Inside f-string, in curly braces there are names of variables that will be substituted:

```
In [1]: ip = '10.1.1.1'

In [2]: mask = 24

In [3]: f"IP: {ip}, mask: {mask}"
Out[3]: 'IP: 10.1.1.1, mask: 24'

The same result with ``format`` method you can achieve by:
``"IP: {ip}, mask: {mask}".format(ip=ip, mask=mask)``.
```

A very important difference between f-strings and `format`: f-strings are expressions that are processed, not just strings. That is, in case of ipython, as soon as we wrote the expression and pressed Enter, it was performed and instead of expressions `{ip}` and `{mask}` the values of variables were substituted.

Therefore, for example, you cannot first write a template and then define variables that are used in template:

```
In [1]: f"IP: {ip}, mask: {mask}"
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-e6f8e01ac9c4> in <module>()
```

(continues on next page)

(continued from previous page)

```
----> 1 f"IP: {ip}, mask: {mask}"

NameError: name 'ip' is not defined
```

In addition to substituting variable values you can write expressions in curly braces:

```
In [1]: octets = ['10', '1', '1', '1']

In [2]: mask = 24

In [3]: f"IP: {'.'.join(octets)}, mask: {mask}"
Out[3]: 'IP: 10.1.1.1, mask: 24'
```

After colon in f-strings you can specify the same values as in `format`:

```
In [9]: oct1, oct2, oct3, oct4 = [10, 1, 1, 1]

In [10]: print(f'''
    ...: IP address:
    ...: {oct1:<8} {oct2:<8} {oct3:<8} {oct4:<8}
    ...: {oct1:08b} {oct2:08b} {oct3:08b} {oct4:08b}''')

IP address:
10       1        1        1
00001010 00000001 00000001 00000001
```

Special aspects of f-strings

When using f-strings you cannot first create a template and then use it as in `format` method.

F-string is immediately executed and contains the values of variables that were defined earlier:

```
In [7]: ip = '10.1.1.1'

In [8]: mask = 24

In [9]: print(f"IP: {ip}, mask: {mask}")
IP: 10.1.1.1, mask: 24
```

If you want to set other values you must create new variables (with the same names) and write f-string again:

```
In [11]: ip = '10.2.2.2'

In [12]: mask = 24

In [13]: print(f"IP: {ip}, mask: {mask}")
IP: 10.2.2.2, mask: 24
```

When using f-strings in loops an f-string must be written in body of the loop to «catch» new variable values within each iteration:

```
In [1]: ip_list = ['10.1.1.1/24', '10.2.2.2/24', '10.3.3.3/24']

In [2]: for ip_address in ip_list:
   ...:     ip, mask = ip_address.split('/')
   ...:     print(f"IP: {ip}, mask: {mask}")
   ...:
IP: 10.1.1.1, mask: 24
IP: 10.2.2.2, mask: 24
IP: 10.3.3.3, mask: 24
```

Examples of f-string usage

Basic variable substitution:

```
In [1]: intf_type = 'Gi'

In [2]: intf_name = '0/3'

In [3]: f'interface {intf_type}/{intf_name}'
Out[3]: 'interface Gi/0/3'
```

Column alignment:

```
In [6]: topology = [['sw1', 'Gi0/1', 'r1', 'Gi0/2'],
   ...:             ['sw1', 'Gi0/2', 'r2', 'Gi0/1'],
   ...:             ['sw1', 'Gi0/3', 'r3', 'Gi0/0'],
   ...:             ['sw1', 'Gi0/5', 'sw4', 'Gi0/2']]
   ...:

In [7]: for connection in topology:
   ...:     l_device, l_port, r_device, r_port = connection
   ...:     print(f'{l_device:10} {l_port:7} {r_device:10} {r_port:7}')
   ...:
```

(continues on next page)

(continued from previous page)

```
sw1         Gi0/1    r1          Gi0/2
sw1         Gi0/2    r2          Gi0/1
sw1         Gi0/3    r3          Gi0/0
sw1         Gi0/5    sw4         Gi0/2
```

Column width can be specified by variable:

```
In [6]: topology = [['sw1', 'Gi0/1', 'r1', 'Gi0/2'],
   ...:             ['sw1', 'Gi0/2', 'r2', 'Gi0/1'],
   ...:             ['sw1', 'Gi0/3', 'r3', 'Gi0/0'],
   ...:             ['sw1', 'Gi0/5', 'sw4', 'Gi0/2']]
   ...:

In [7]: width = 10

In [8]: for connection in topology:
   ...:     l_device, l_port, r_device, r_port = connection
   ...:     print(f'{l_device:{width}} {l_port:{width}} {r_device:{width}} {r_
↪port:{width}}')
   ...:
sw1         Gi0/1      r1         Gi0/2
sw1         Gi0/2      r2         Gi0/1
sw1         Gi0/3      r3         Gi0/0
sw1         Gi0/5      sw4        Gi0/2
```

Accessing a dictionary key:

```
In [1]: session_stats = {'done': 10, 'todo': 5}

In [2]: if session_stats['todo']:
   ...:     print(f"Pomodoros done: {session_stats['done']}, TODO: {session_stats[
↪'todo']}")
   ...: else:
   ...:     print(f"Good job! All {session_stats['done']} pomodoros done!")
   ...:
Pomodoros done: 10, TODO: 5
```

Call len() function inside f-string:

```
In [2]: topology = [['sw1', 'Gi0/1', 'r1', 'Gi0/2'],
   ...:             ['sw1', 'Gi0/2', 'r2', 'Gi0/1'],
   ...:             ['sw1', 'Gi0/3', 'r3', 'Gi0/0'],
   ...:             ['sw1', 'Gi0/5', 'sw4', 'Gi0/2']]
   ...:
```

(continues on next page)

(continued from previous page)

```
In [3]: print(f'Number of connections in topology: {len(topology)}')
Number of connections in topology: 4
```

Call upper() method inside f-string:

```
In [1]: name = 'python'

In [2]: print(f'Zen of {name.upper()}')
Zen of PYTHON
```

Converting numbers to binary format:

```
In [7]: ip = '10.1.1.1'

In [8]: oct1, oct2, oct3, oct4 = ip.split('.')

In [9]: print(f'{int(oct1):08b} {int(oct2):08b} {int(oct3):08b} {int(oct4):08b}')
00001010 00000001 00000001 00000001
```

What to use format or f-strings

In many cases f-strings are more convenient to use as template looks more understandable and compact. However, there are cases when `format` method is more convenient. For example:

```
In [6]: ip = [10, 1, 1, 1]

In [7]: oct1, oct2, oct3, oct4 = ip
   ...: print(f'{oct1:08b} {oct2:08b} {oct3:08b} {oct4:08b}')
   ...:
00001010 00000001 00000001 00000001

In [8]: template = "{:08b} "*4

In [9]: template.format(*ip)
Out[9]: '00001010 00000001 00000001 00000001 '
```

Another situation where `format` is usually more convenient to use: the need to use the same template many times in script. F-string will execute the first time and will set current values of variables and to use template again it has to be rewritten. This means that script will contain copies of the same line. At the same time `format` allows to create a template in one place and then use it again substituting variables as needed.

This can be avoided by creating a function but creating a function to print a string based on template is not always justified. Example of creating a function:

```
In [1]: def show_me_ip(ip, mask):
   ...:     return f"IP: {ip}, mask: {mask}"
   ...:

In [2]: show_me_ip('10.1.1.1', 24)
Out[2]: 'IP: 10.1.1.1, mask: 24'

In [3]: show_me_ip('192.16.10.192', 28)
Out[3]: 'IP: 192.16.10.192, mask: 28'
```

Variable unpacking

Variable unpacking is a special syntax that allows to assign elements of an iterable to variables.

Note: This functionality is often referred to as tuple unpacking but unpacking works on any iterable object, not only with tuples

---

Example of variable unpacking:

```
In [1]: interface = ['FastEthernet0/1', '10.1.1.1', 'up', 'up']

In [2]: intf, ip, status, protocol = interface

In [3]: intf
Out[3]: 'FastEthernet0/1'

In [4]: ip
Out[4]: '10.1.1.1'
```

This option is much more convenient than the use of indexes:

```
In [5]: intf, ip, status, protocol = interface[0], interface[1], interface[2],␣
↪interface[3]
```

When you unpack variables, each item in list falls into the corresponding variable. It is important to take into account that there should be exactly as many variables on the left as there are elements in the list.

If amount of variables are less or more, there will be an exception:

```
In [6]: intf, ip, status = interface
---------------------------------------------------------------
ValueError                 Traceback (most recent call last)
<ipython-input-11-a304c4372b1a> in <module>()
----> 1 intf, ip, status = interface

ValueError: too many values to unpack (expected 3)

In [7]: intf, ip, status, protocol, other = interface
---------------------------------------------------------------
ValueError                 Traceback (most recent call last)
<ipython-input-12-ac93e78b978c> in <module>()
----> 1 intf, ip, status, protocol, other = interface

ValueError: not enough values to unpack (expected 5, got 4)
```

Replacement of unnecessary elements _

Often only some of elements of an iterable are needed. Unpacking syntax requires that exactly as many variables as elements in the object being iterated be specified.

If, for example, only VLAN, MAC and interface should be obtained from line, you still need to specify a variable for “DYNAMIC”:

```
In [8]: line = '100    01bb.c580.7000    DYNAMIC     Gi0/1'

In [9]: vlan, mac, item_type, intf = line.split()

In [10]: vlan
Out[10]: '100'

In [11]: intf
Out[11]: 'Gi0/1'
```

If record type is no longer needed, you can replace item_type variable with underline character:

```
In [12]: vlan, mac, _, intf = line.split()
```

This clearly indicates that this element is not needed.

Underline character can be used more than once:

```
In [13]: dhcp = '00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping␣
↪10    FastEthernet0/1'

In [14]: mac, ip, _, _, vlan, intf = dhcp.split()

In [15]: mac
Out[15]: '00:09:BB:3D:D6:58'

In [16]: vlan
Out[16]: '10'
```

Use *

Variable unpacking supports a special syntax that allows unpacking of several elements into one. If you put * in front of variable name, all elements except those that are explicitly assigned will be written into it.

For example, you can get the first element in *first* variable and the rest in *rest*:

```
In [18]: vlans = [10, 11, 13, 30]

In [19]: first, *rest = vlans

In [20]: first
Out[20]: 10

In [21]: rest
Out[21]: [11, 13, 30]
```

Variable with an asterisk will always contain a list:

```
In [22]: vlans = (10, 11, 13, 30)

In [22]: first, *rest = vlans

In [23]: first
Out[23]: 10

In [24]: rest
Out[24]: [11, 13, 30]
```

If there is only one item, unpacking will still work:

```
In [25]: first, *rest = vlans

In [26]: first
Out[26]: 55

In [27]: rest
Out[27]: []
```

There can be only one variable with an asterisk in the unpacking expression.

```
In [28]: vlans = (10, 11, 13, 30)

In [29]: first, *rest, *others = vlans
  File "<ipython-input-37-dedf7a08933a>", line 1
    first, *rest, *others = vlans
                            ^
SyntaxError: two starred expressions in assignment
```

This variable may not only be at the end of expression:

```
In [30]: vlans = (10, 11, 13, 30)

In [31]: *rest, last = vlans

In [32]: rest
Out[32]: [10, 11, 13]

In [33]: last
Out[33]: 30
```

Thus, the first, second and last element can be specified:

```
In [34]: cdp = 'SW1     Eth 0/0    140   S I   WS-C3750-  Eth 0/1'

In [35]: name, l_intf, *other, r_intf = cdp.split()

In [36]: name
Out[36]: 'SW1'

In [37]: l_intf
Out[37]: 'Eth'

In [38]: r_intf
Out[38]: '0/1'
```

Unpacking examples

Unpacking of iterable objects

These examples show that you can unpack not only lists, tuples and strings but also any other iterable objects.

Unpacking the range:

```
In [39]: first, *rest = range(1, 6)

In [40]: first
Out[40]: 1

In [41]: rest
Out[41]: [2, 3, 4, 5]
```

Unpacking zip:

```
In [42]: a = [1, 2, 3, 4, 5]

In [43]: b = [100, 200, 300, 400, 500]

In [44]: zip(a, b)
Out[44]: <zip at 0xb4df4fac>

In [45]: list(zip(a, b))
Out[45]: [(1, 100), (2, 200), (3, 300), (4, 400), (5, 500)]

In [46]: first, *rest, last = zip(a, b)

In [47]: first
Out[47]: (1, 100)

In [48]: rest
Out[48]: [(2, 200), (3, 300), (4, 400)]

In [49]: last
Out[49]: (5, 500)
```

Example of unpacking in *for* loop

Example of a loop that runs through the keys:

```
In [50]: access_template = ['switchport mode access',
    ...:                    'switchport access vlan',
    ...:                    'spanning-tree portfast',
    ...:                    'spanning-tree bpduguard enable']
    ...:

In [51]: access = {'0/12':10,
    ...:           '0/14':11,
    ...:           '0/16':17}
    ...:

In [52]: for intf in access:
    ...:     print('interface FastEthernet' + intf)
    ...:     for command in access_template:
    ...:         if command.endswith('access vlan'):
    ...:             print(' {} {}'.format(command, access[intf]))
    ...:         else:
    ...:             print(' {}'.format(command))
    ...:
interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable
```

Instead, you can run through key-value pairs and immediately unpack them into different variables:

```
In [53]: for intf, vlan in access.items():
    ...:     print('interface FastEthernet' + intf)
    ...:     for command in access_template:
    ...:         if command.endswith('access vlan'):
    ...:             print(' {} {}'.format(command, vlan))
    ...:         else:
    ...:             print(' {}'.format(command))
```

(continues on next page)

(continued from previous page)

```
    ...:
```

Example of unpacking list items in the loop:

```
In [54]: table
Out[54]:
[['100', 'a1b2.ac10.7000', 'DYNAMIC', 'Gi0/1'],
 ['200', 'a0d4.cb20.7000', 'DYNAMIC', 'Gi0/2'],
 ['300', 'acb4.cd30.7000', 'DYNAMIC', 'Gi0/3'],
 ['100', 'a2bb.ec40.7000', 'DYNAMIC', 'Gi0/4'],
 ['500', 'aa4b.c550.7000', 'DYNAMIC', 'Gi0/5'],
 ['200', 'a1bb.1c60.7000', 'DYNAMIC', 'Gi0/6'],
 ['300', 'aa0b.cc70.7000', 'DYNAMIC', 'Gi0/7']]


In [55]: for line in table:
    ...:     vlan, mac, _, intf = line
    ...:     print(vlan, mac, intf)
    ...:
100 a1b2.ac10.7000 Gi0/1
200 a0d4.cb20.7000 Gi0/2
300 acb4.cd30.7000 Gi0/3
100 a2bb.ec40.7000 Gi0/4
500 aa4b.c550.7000 Gi0/5
200 a1bb.1c60.7000 Gi0/6
300 aa0b.cc70.7000 Gi0/7
```

But it's better to do this:

```
In [56]: for vlan, mac, _, intf in table:
    ...:     print(vlan, mac, intf)
    ...:
100 a1b2.ac10.7000 Gi0/1
200 a0d4.cb20.7000 Gi0/2
300 acb4.cd30.7000 Gi0/3
100 a2bb.ec40.7000 Gi0/4
500 aa4b.c550.7000 Gi0/5
200 a1bb.1c60.7000 Gi0/6
300 aa0b.cc70.7000 Gi0/7
```

List, dict, set comprehensions

Python supports special expressions that allow for compact creation of lists, dictionaries, and sets:

- list comprehensions
- dict comprehensions
- set comprehensions

These expressions not only enable more compact objects to be created but also create them faster. Although they require a certain habit of use and understanding at first, they are very often used.

List comprehensions

List comprehension is an expression like:

```
In [1]: vlans = [f'vlan {num}' for num in range(10, 16)]

In [2]: print(vlans)
['vlan 10', 'vlan 11', 'vlan 12', 'vlan 13', 'vlan 14', 'vlan 15']
```

In general, it is an expression that converts an iterable object into a list. That is, a sequence of elements is converted and added to a new list.

Expression above is similar to this loop:

```
In [3]: vlans = []

In [4]: for num in range(10, 16):
   ...:     vlans.append(f'vlan {num}')
   ...:

In [5]: print(vlans)
['vlan 10', 'vlan 11', 'vlan 12', 'vlan 13', 'vlan 14', 'vlan 15']
```

In list comprehensions you can use **if**. Thus, you can only add some objects to the list. For example, a loop selects only those elements that are digits, converts them and adds them to the resulting list only_digits:

```
In [6]: items = ['10', '20', 'a', '30', 'b', '40']

In [7]: only_digits = []

In [8]: for item in items:
   ...:     if item.isdigit():
   ...:         only_digits.append(int(item))
   ...:
```

(continues on next page)

(continued from previous page)

```
In [9]: print(only_digits)
[10, 20, 30, 40]
```

A similar version with list comprehensions:

```
In [10]: items = ['10', '20', 'a', '30', 'b', '40']

In [11]: only_digits = [int(item) for item in items if item.isdigit()]

In [12]: print(only_digits)
[10, 20, 30, 40]
```

Of course, not all loops can be rewritten as a list comprehension but when it is possible to do so without making the expression more complex, it is better to use list comprehension.

Note: In Python, list comprehensions can also replace filter and map functions and are considered a clearer option.

With list comprehension it is also convenient to get elements from nested dictionaries:

```
In [13]: london_co = {
    ...:     'r1' : {
    ...:     'hostname': 'london_r1',
    ...:     'location': '21 New Globe Walk',
    ...:     'vendor': 'Cisco',
    ...:     'model': '4451',
    ...:     'IOS': '15.4',
    ...:     'IP': '10.255.0.1'
    ...:     },
    ...:     'r2' : {
    ...:     'hostname': 'london_r2',
    ...:     'location': '21 New Globe Walk',
    ...:     'vendor': 'Cisco',
    ...:     'model': '4451',
    ...:     'IOS': '15.4',
    ...:     'IP': '10.255.0.2'
    ...:     },
    ...:     'sw1' : {
    ...:     'hostname': 'london_sw1',
    ...:     'location': '21 New Globe Walk',
    ...:     'vendor': 'Cisco',
    ...:     'model': '3850',
```

(continues on next page)

(continued from previous page)

```
    ...:     'IOS': '3.6.XE',
    ...:     'IP': '10.255.0.101'
    ...:     }
    ...: }

In [14]: [london_co[device]['IOS'] for device in london_co]
Out[14]: ['15.4', '15.4', '3.6.XE']

In [15]: [london_co[device]['IP'] for device in london_co]
Out[15]: ['10.255.0.1', '10.255.0.2', '10.255.0.101']
```

In fact, syntax of list comprehension looks like:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN ]
```

This means you can use several **for** in expression.

For example, *vlans* list contains several nested lists with VLANs:

```
In [16]: vlans = [[10,21,35], [101, 115, 150], [111, 40, 50]]
```

It's necessary to form only one list with VLAN numbers. The first option is to use **for** loop:

```
In [17]: result = []

In [18]: for vlan_list in vlans:
    ...:     for vlan in vlan_list:
    ...:         result.append(vlan)
    ...:

In [19]: print(result)
[10, 21, 35, 101, 115, 150, 111, 40, 50]
```

List comprehension:

```
In [20]: vlans = [[10,21,35], [101, 115, 150], [111, 40, 50]]

In [21]: result = [vlan for vlan_list in vlans for vlan in vlan_list]

In [22]: print(result)
[10, 21, 35, 101, 115, 150, 111, 40, 50]
```

Two sequences can be processed simultaneously using zip():

```
In [23]: vlans = [100, 110, 150, 200]

In [24]: names = ['mngmt', 'voice', 'video', 'dmz']

In [25]: result = [f'vlan {vl}\n name {name}' for vl, name in zip(vlans, names)]

In [26]: print('\n'.join(result))
vlan 100
 name mngmt
vlan 110
 name voice
vlan 150
 name video
vlan 200
 name dmz
```

Dict comprehensions

Dict comprehensions are similar to list comprehensions but they are used to create dictionaries. For example, the expression:

```
In [27]: d = {}

In [28]: for num in range(1, 11):
    ...:     d[num] = num**2
    ...:

In [29]: print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Can be replaced with a dict comprehension:

```
In [30]: d = {num: num**2 for num in range(1, 11)}

In [31]: print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Another example in which you need to change an existing dictionary and convert all keys to lowercase. First, a solution without a dict comprehension:

```
In [32]: r1 = {'IOS': '15.4',
    ...:       'IP': '10.255.0.1',
```

(continues on next page)

(continued from previous page)

```
    ...:       'hostname': 'london_r1',
    ...:       'location': '21 New Globe Walk',
    ...:       'model': '4451',
    ...:       'vendor': 'Cisco'}
    ...:

In [33]: lower_r1 = {}

In [34]: for key, value in r1.items():
    ...:     lower_r1[key.lower()] = value
    ...:

In [35]: lower_r1
Out[35]:
{'hostname': 'london_r1',
 'ios': '15.4',
 'ip': '10.255.0.1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

Dict comprehension version:

```
In [36]: r1 = {'IOS': '15.4',
    ...:   'IP': '10.255.0.1',
    ...:   'hostname': 'london_r1',
    ...:   'location': '21 New Globe Walk',
    ...:   'model': '4451',
    ...:   'vendor': 'Cisco'}
    ...:

In [37]: lower_r1 = {key.lower(): value for key, value in r1.items()}

In [38]: lower_r1
Out[38]:
{'hostname': 'london_r1',
 'ios': '15.4',
 'ip': '10.255.0.1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

Like list comprehensions, dict comprehensions can be nested. Try to convert keys in nested dictionaries in the same way:

```
In [39]: london_co = {
    ...:     'r1' : {
    ...:     'hostname': 'london_r1',
    ...:     'location': '21 New Globe Walk',
    ...:     'vendor': 'Cisco',
    ...:     'model': '4451',
    ...:     'IOS': '15.4',
    ...:     'IP': '10.255.0.1'
    ...:     },
    ...:     'r2' : {
    ...:     'hostname': 'london_r2',
    ...:     'location': '21 New Globe Walk',
    ...:     'vendor': 'Cisco',
    ...:     'model': '4451',
    ...:     'IOS': '15.4',
    ...:     'IP': '10.255.0.2'
    ...:     },
    ...:     'sw1' : {
    ...:     'hostname': 'london_sw1',
    ...:     'location': '21 New Globe Walk',
    ...:     'vendor': 'Cisco',
    ...:     'model': '3850',
    ...:     'IOS': '3.6.XE',
    ...:     'IP': '10.255.0.101'
    ...:     }
    ...: }

In [40]: lower_london_co = {}

In [41]: for device, params in london_co.items():
    ...:     lower_london_co[device] = {}
    ...:     for key, value in params.items():
    ...:         lower_london_co[device][key.lower()] = value
    ...:

In [42]: lower_london_co
Out[42]:
{'r1': {'hostname': 'london_r1',
  'ios': '15.4',
  'ip': '10.255.0.1',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
```

(continues on next page)

(continued from previous page)

```
 'r2': {'hostname': 'london_r2',
  'ios': '15.4',
  'ip': '10.255.0.2',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'sw1': {'hostname': 'london_sw1',
  'ios': '3.6.XE',
  'ip': '10.255.0.101',
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

Similar conversion with dict comprehensions:

```
In [43]: result = {device: {key.lower(): value for key, value in params.items()}
          for device, params in london_co.items()}

In [44]: result
Out[44]:
{'r1': {'hostname': 'london_r1',
  'ios': '15.4',
  'ip': '10.255.0.1',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'r2': {'hostname': 'london_r2',
  'ios': '15.4',
  'ip': '10.255.0.2',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'sw1': {'hostname': 'london_sw1',
  'ios': '3.6.XE',
  'ip': '10.255.0.101',
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

Set comprehensions

Set comprehensions are generally similar to list comprehensions. For example, get a set with unique VLAN numbers:

```
In [45]: vlans = [10, '30', 30, 10, '56']

In [46]: unique_vlans = {int(vlan) for vlan in vlans}

In [47]: unique_vlans
Out[47]: {10, 30, 56}
```

Similar solution without using of set comprehensions:

```
In [48]: vlans = [10, '30', 30, 10, '56']

In [49]: unique_vlans = set()

In [50]: for vlan in vlans:
    ...:     unique_vlans.add(int(vlan))
    ...:

In [51]: unique_vlans
Out[51]: {10, 30, 56}
```

Further reading

Documentation:

- PEP 3132 – Extended Iterable Unpacking

Articles:

- List, Dict And Set Comprehensions By Example
- Python List Comprehensions: Explained Visually

Stack Overflow:

- Answer with many unpacking examples

4

II. Code reuse

When writing code, some of the steps are often repeated. It can be a small block of 3-5 lines, or it can be a fairly large sequence of steps. Copying code is a bad idea. Because if you have to update one of copies later, you have to update others.

Instead, you create a special code block with name - function. And every time code has to be repeated, you just call a function. Function allows not only to name a block of code but also to make it more abstract through parameters. Parameters make it possible to pass different data for function. And get different results depending on input parameters.

Section *9. Functions* covers with creation of functions, section *10. Useful functions* discusses useful built-in functions.

Once code is divided into functions, there comes a time when you need to use function in another script. Of course, copying a function is as inconvenient as copying a normal code. Modules are used to reuse code from another Python script.

Section *11. Modules* is dedicated to creating your own modules and section *12. Useful modules* covers useful modules from Python standard library.

The last section *13. Iterators, iterable and generators* is dedicated to iterable objects, iterators and generators.

9. Functions

Function:

- has a name to run this code block as many times as you want
  - launch of function code is called a function call
- function parameters are usually defined when creating a function.
  - function parameters determine which arguments a function can accept
  - arguments can be passed to functions
  - function code will be executed taking into account the specified arguments

What are functions for?

Typically, problems that code solves are very similar and often have something in common. For example, when working with configuration files each time it is necessary to perform such actions:

- file opening
- deletion (or skipping) of lines starting with exclamation mark (for Cisco)
- deleting (or skipping) empty lines
- deleting new line characters at the end of lines
- converting the result to a list

Beyond that, actions can vary depending on what needs to be done.

Often there's a piece of code that repeats itself. Of course, you can copy it from one script to another. But this is very inconvenient because when you change code you have to update it in all files in which it is copied.

It is much easier and more accurate to put this code into a function (it can also be several functions). And then you will call this function - in this file or another one. This section discusses when a function is in the same file. And in *11. Modules* we will see how to reuse objects that are in other scripts.

Creation of functions

Creation of function:

- functions are created with a reserved word `def`
- `def` followed by function name and parentheses
- parameters that function accepts inside parentheses
- after parentheses goes colon and from a new line with indent there is a block of code that function executes

- optionally, the first line can be `docstring`
- function can use `return` operator
  - it is used to terminate and exit a function
  - most often `return` operator returns some value

Note: Function code used in this subsection can be copied from create_func file.

Example of function:

```
In [1]: def configure_intf(intf_name, ip, mask):
   ...:     print('interface', intf_name)
   ...:     print('ip address', ip, mask)
   ...:
```

Function `configure_intf` creates an interface configuration with specified name and IP address. Function has three parameters: intf_name, ip, mask. When function is called the real data will replace these parameters.

Note: When function is created, it does nothing yet. Actions listed in it will be executed only when you call function. This is something like ACL in network equipment: when creating ACL in configuration, it does nothing until it is applied.

Function call

When calling a function you must specify its name and pass arguments if necessary.

Note: Parameters are variables that are used to create a function. Arguments are the actual values (data) that are passed to functions when called.

Function `configure_intf` expects three values when called because it was created with three parameters:

```
In [2]: configure_intf('F0/0', '10.1.1.1', '255.255.255.0')
interface F0/0
ip address 10.1.1.1 255.255.255.0

In [3]: configure_intf('Fa0/1', '94.150.197.1', '255.255.255.248')
interface Fa0/1
ip address 94.150.197.1 255.255.255.248
```

The current version of the `configure_intf` function prints commands to a standard output, commands can be seen but the result of function cannot be saved to a variable.

For example, `sorted` function does not simply print the sorting result to standard output stream but **returns** it, so it can be saved to variable in this way:

```
In [4]: items = [40, 2, 0, 22]

In [5]: sorted(items)
Out[5]: [0, 2, 22, 40]

In [6]: sorted_items = sorted(items)

In [7]: sorted_items
Out[7]: [0, 2, 22, 40]
```

Note: Note string `Out[5]` in ipython: this is how ipython shows that the function/method is returning something and shows what it returns.

If you try to write the result of the `configure_intf` function to a variable, the value of the variable will be None:

```
.. code:: python
```

In [8]: result = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0') interface Fa0/0 ip address 10.1.1.1 255.255.255.0

In [9]: print(result) None

For a function to return a value, use `return` operator.

Operator return

Operator `return` is used to return a value, and at the same time it exits the function. Function can return any Python object. By default, function always returns None.

In order for `configure_intf` function to return a value that can then be assigned to a variable, you must use `return` operator:

```
In [10]: def configure_intf(intf_name, ip, mask):
    ...:     config = f'interface {intf_name}\nip address {ip} {mask}'
    ...:     return config
    ...:
```

(continues on next page)

(continued from previous page)

```
In [11]: result = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')

In [12]: print(result)
interface Fa0/0
ip address 10.1.1.1 255.255.255.0

In [13]: result
Out[13]: 'interface Fa0/0\nip address 10.1.1.1 255.255.255.0'
```

Now the result variable contains a line with commands to configure interface. In real life, function will almost always return some value.

Another important aspect of `return` operator is that after `return` the function closes, meaning that the expressions that follow `return` are not executed.

For example, in function below the line «Configuration is ready» will not be displayed because it stands after `return`:

```
In [14]: def configure_intf(intf_name, ip, mask):
    ...:     config = f'interface {intf_name}\nip address {ip} {mask}'
    ...:     return config
    ...:     print('Configuration is ready')
    ...:

In [15]: configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')
Out[15]: 'interface Fa0/0\nip address 10.1.1.1 255.255.255.0'
```

Function can return multiple values. In this case, they are separated by a comma after `return` operator. In fact, function returns tuple:

```
In [16]: def configure_intf(intf_name, ip, mask):
    ...:     config_intf = f'interface {intf_name}\n'
    ...:     config_ip = f'ip address {ip} {mask}'
    ...:     return config_intf, config_ip
    ...:

In [17]: result = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')

In [18]: result
Out[18]: ('interface Fa0/0\n', 'ip address 10.1.1.1 255.255.255.0')

In [19]: type(result)
Out[19]: tuple
```

(continues on next page)

(continued from previous page)

```
In [20]: intf, ip_addr = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')

In [21]: intf
Out[21]: 'interface Fa0/0\n'

In [22]: ip_addr
Out[22]: 'ip address 10.1.1.1 255.255.255.0'
```

Documentation (docstring)

The first line in function definition is docstring, documentation string. This is a comment that is used to describe a function:

```
In [23]: def configure_intf(intf_name, ip, mask):
    ...:     '''
    ...:     Fucntion generates interface configuration
    ...:     '''
    ...:     config_intf = f'interface {intf_name}\n'
    ...:     config_ip = f'ip address {ip} {mask}'
    ...:     return config_intf, config_ip
    ...:

In [24]: configure_intf?
Signature: configure_intf(intf_name, ip, mask)
Docstring: Fucntion generates interface configuration
File:      ~/repos/pyneng-examples-exercises/examples/06_control_structures/
↪<ipython-input-23-2b2bd970db8f>
Type:      function
```

It is best to write short comments that describe function. For example, describe what function expects to input, what type of arguments should be and what will be the output. Besides, it is better to write a couple of sentences about what function does. This will help when in a month or two you will be trying to understand what function you wrote is doing.

Namespace. Scope of variables

Variables in Python have a scope. Depending on location in code where variable has been defined, scope is also defined, it determines where variable will be available.

When using variable names in a program, Python searches, creates or changes these names in the corresponding namespace each time. Namespace that is available at each moment depends on area in which code is located.

Python has a LEGB rule that it uses for variables search. For example, when accessing a variable within a function, Python searches for a variable in this order in scopes (before the first match):

- L (local) - in local (within function)
- E (enclosing) - in local area of outer functions (these are functions within which our function is located)
- G (global) - in global (in script)
- B (built-in) - in built-in (reserved Python values)

Accordingly, there are local and global variables:

- local variables:
  - variables that are defined within function
  - these variables become unavailable after exit from function
- global variables:
  - variables that are defined outside the function
  - these variables are 'global' only within a module
  - for example, to be available in another module they must be imported

Example of local intf_config:

```
In [1]: def configure_intf(intf_name, ip, mask):
   ...:     intf_config = f'interface {intf_name}\nip address {ip} {mask}'
   ...:     return intf_config
   ...:

In [2]: intf_config
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-5983e972fb1c> in <module>
----> 1 intf_config

NameError: name 'intf_config' is not defined
```

Note that intf_config variable is not available outside of function. To get the result of a function you must call a function and assign result to a variable:

```
In [3]: result = configure_intf('F0/0', '10.1.1.1', '255.255.255.0')

In [4]: result
Out[4]: 'interface F0/0\nip address 10.1.1.1 255.255.255.0'
```

Function parameters and arguments

The purpose of creating a function is typically to take a piece of code that performs a particular task to a separate object. This allows you to use this piece of code multiple times without having to re-create it in program.

Typically, a function must perform some actions with input values and produce an output.

When working with functions it is important to distinguish:

- **parameters** - variables that are used when creating a function.
- **arguments** - actual values (data) that are passed to function when called.

For a function to receive incoming values, it must be created with parameters (func_check_passwd.py file):

```
In [1]: def check_passwd(username, password):
   ...:     if len(password) < 8:
   ...:         print('Password is too short')
   ...:         return False
   ...:     elif username in password:
   ...:         print('Password contains username')
   ...:         return False
   ...:     else:
   ...:         print(f'Password for user {username} has passed all checks')
   ...:         return True
   ...:
```

In this case, function has two parameters: username and password.

Function checks password and returns False if checks fail and True if password passed checks:

```
In [2]: check_passwd('nata', '12345')
Password is too short
Out[2]: False

In [3]: check_passwd('nata', '12345lsdkjflskfdjsnata')
Password contains username
Out[3]: False

In [4]: check_passwd('nata', '12345lsdkjflskfdjs')
Password for user nata has passed all checks
Out[4]: True
```

When defining a function in this way it is necessary to pass both arguments. If only one argument is passed, there is an error:

```
In [5]: check_passwd('nata')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-e07773bb4cc8> in <module>
----> 1 check_passwd('nata')

TypeError: check_passwd() missing 1 required positional argument: 'password'
```

Similarly, an error will occur if three or more arguments are passed.

Function parameter types

When creating a function you can specify which arguments must be passed and which must not. Accordingly, a function can be created with:

- **required parameters**
- **optional parameters** (with default values)

Required parameters

Required parameters - determine which arguments must be passed to functions. At the same time, they need to be passed exactly as many as parameters of function are specified (you cannot specify more or less)

Function with mandatory parameters (func_params_types.py file):

```
In [1]: def check_passwd(username, password):
   ...:     if len(password) < 8:
   ...:         print('Password is too short')
   ...:         return False
   ...:     elif username in password:
   ...:         print('Password contains username')
   ...:         return False
   ...:     else:
   ...:         print(f'Password for user {username} has passed all checks')
   ...:         return True
   ...:
```

Function check_passwd() expects two arguments: username and password.

Function checks password and returns False if checks fail and True if password passed all checks:

```
In [2]: check_passwd('nata', '12345')
Password is too short
Out[2]: False

In [3]: check_passwd('nata', '12345lsdkjflskfdjsnata')
Password contains username
Out[3]: False

In [4]: check_passwd('nata', '12345lsdkjflskfdjs')
Password for user nata has passed all checks
Out[4]: True
```

Optional parameters (default parameters)

When creating a function you can specify default value for parameter in this way: `def check_passwd(username, password, min_length=8)`. In this case, min_length option is specified with default value and may not be passed when a function is called.

Example of a check_passwd function with default parameter (func_check_passwd_optional_param.py file):

```
In [6]: def check_passwd(username, password, min_length=8):
   ...:     if len(password) < min_length:
   ...:         print('Password is too short')
   ...:         return False
   ...:     elif username in password:
   ...:         print('Password contains username')
   ...:         return False
   ...:     else:
   ...:         print(f'Password for user {username} has passed all checks')
   ...:         return True
   ...:
```

Since min_length parameter has a default value the corresponding argument can be omitted when a function is called if default value fits you:

```
In [7]: check_passwd('nata', '12345')
Password is too short
Out[7]: False
```

If you want to change default value:

```
In [8]: check_passwd('nata', '12345', 3)
Password for user nata has passed all checks
Out[8]: True
```

Function argument types

When a function is called the arguments can be passed in two ways:

- as **positional** - passed in the same order in which they are defined at creation of function. That is, the order in which arguments are passed determines what value each argument will receive.
- as **keyword** - passed with argument name and its value. In such a case, arguments can be stated in any order as their name is clearly indicated.

Positional and keyword arguments can be mixed when calling a function. That is, it is possible to use both methods when passing arguments of the same function. In this process, Positional arguments must be indicated before keyword arguments.

Look at different ways to pass arguments using check_passwd (func_check_check_passwd_optional_param.py file):

```
In [1]: def check_passwd(username, password):
   ...:     if len(password) < 8:
   ...:         print('Password is too short')
   ...:         return False
   ...:     elif username in password:
   ...:         print('Password contains username')
   ...:         return False
   ...:     else:
   ...:         print(f'Password for user {username} has passed all checks')
   ...:         return True
   ...:
```

Positional argument

Positional arguments when calling a function must be passed in the correct order (therefore they are called positional arguments).

```
In [2]: check_passwd('nata', '12345')
Password is too short
Out[2]: False
```

(continues on next page)

(continued from previous page)

```
In [3]: check_passwd('nata', '12345lsdkjflskfdjsnata')
Password contains username
Out[3]: False

In [4]: check_passwd('nata', '12345lsdkjflskfdjs')
Password for user nata has passed all checks
Out[4]: True
```

If you swap arguments when calling a function the error will likely occur depending on function.

Keyword arguments

Keyword arguments:

- are passed with name of argument
- thus they can be passed in any order

If both arguments are keyword, they can be passed in any order:

```
In [9]: check_passwd(password='12345', username='nata', min_length=4)
Password for user nata has passed all checks
Out[9]: True
```

> **Warning:** Please note that first there should always be positional arguments and then keyword arguments.

If you do the opposite, there's an error:

```
In [10]: check_passwd(password='12345', username='nata', 4)
  File "<ipython-input-10-7e8246b6b402>", line 1
    check_passwd(password='12345', username='nata', 4)
                                                    ^
SyntaxError: positional argument follows keyword argument
```

But in that combination it works:

```
In [11]: check_passwd('nata', '12345', min_length=3)
Password for user nata has passed all checks
Out[11]: True
```

In real life, it is often better to specify flags (parameters with True/False values) or numerical values as a keyword argument. If you set a good name for the parameter you can immediately know by its

name what it does.

For example, you can add a flag that will control whether or not a username should be checked in password:

```
In [12]: def check_passwd(username, password, min_length=8, check_username=True):
    ...:     if len(password) < min_length:
    ...:         print('Password is too short')
    ...:         return False
    ...:     elif check_username and username in password:
    ...:         print('Password contains username')
    ...:         return False
    ...:     else:
    ...:         print(f'Password for user {username} has passed all checks')
    ...:         return True
    ...:
```

By default, flag is True which means check should be done:

```
In [14]: check_passwd('nata', '12345nata', min_length=3)
Password contains username
Out[14]: False

In [15]: check_passwd('nata', '12345nata', min_length=3, check_username=True)
Password contains username
Out[15]: False
```

If you specify a value equal to False the verification will not be performed:

```
In [16]: check_passwd('nata', '12345nata', min_length=3, check_username=False)
Password for user nata has passed all checks
Out[16]: True
```

Variable length arguments

Sometimes it is necessary to make function accept not a fixed number of arguments, but any number. For such a case, in Python it is possible to create a function with a special parameter that accepts variable length arguments. This parameter can be both keyword and positional.

Note: Even if you don't use it in your scripts there's a good chance you'll find it in someone else's code.

Variable length positional arguments

Parameter that takes positional variable length arguments is created by adding an asterisk before parameter name. Parameter can have any name but by agreement *args is the most common name.

Example of a function:

```
In [1]: def sum_arg(a, *args):
  ....:     print(a, args)
  ....:     return a + sum(args)
  ....:
```

Function sum_arg is created with two parameters:

- parameter a
  - if passed as positional argument, should be first
  - if passed as a keyword argument, the order does not matter
- parameter *args - expects variable length arguments
  - all other arguments as a tuple
  - these arguments may be missed

Call a function with different number of arguments:

```
In [2]: sum_arg(1, 10, 20, 30)
1 (10, 20, 30)
Out[2]: 61

In [3]: sum_arg(1, 10)
1 (10,)
Out[3]: 11

In [4]: sum_arg(1)
1 ()
Out[4]: 1
```

You can also create such a function:

```
In [5]: def sum_arg(*args):
  ....:     print(args)
  ....:     return sum(args)
  ....:
```

(continues on next page)

(continued from previous page)

```
In [6]: sum_arg(1, 10, 20, 30)
(1, 10, 20, 30)
Out[6]: 61

In [7]: sum_arg()
()
Out[7]: 0
```

Keyword variable length arguments

Parameter that accepts keyword variable length arguments is created by adding two asterisk in front of parameter name. Name of parameter can be any but by agreement most commonly use name **kwargs (from keyword arguments).

Example of a function:

```
In [8]: def sum_arg(a, **kwargs):
  ....:     print(a, kwargs)
  ....:     return a + sum(kwargs.values())
  ....:
```

Function sum_arg is created with two parameters:

- parameter a
  - if passed as positional argument, should be first
  - if passed as a keyword argument, the order does not matter
- parameter **kwargs - expects keyword variable length arguments
  - all other keyword arguments as a dictionary
  - these arguments may be missed

Calling a function with different number of keyword arguments:

```
In [9]: sum_arg(a=10, b=10, c=20, d=30)
10 {'c': 20, 'b': 10, 'd': 30}
Out[9]: 70

In [10]: sum_arg(b=10, c=20, d=30, a=10)
10 {'c': 20, 'b': 10, 'd': 30}
Out[10]: 70
```

Unpacking arguments

In Python the expressions *args and **kwargs allow for another task - **unpacking arguments**.

So far we've called all functions manually. Hence, we passed on all relevant arguments.

In reality, it is usually necessary to pass data to function programmatically. And often data comes in the form of a Python object.

Unpacking positional arguments

For example, when formatting strings you often need to pass multiple arguments to format method. And often these arguments are already in list or tuple. To pass them to format method you have to use indexes:

```
In [1]: items = [1, 2, 3]

In [2]: print('One: {}, Two: {}, Three: {}'.format(items[0], items[1], items[2]))
One: 1, Two: 2, Three: 3
```

Instead, you can take advantage of unpacking argument and do this:

```
In [4]: items = [1, 2, 3]

In [5]: print('One: {}, Two: {}, Three: {}'.format(*items))
One: 1, Two: 2, Three: 3
```

Another example is config_interface function (func_config_interface_unpacking.py file):

```
In [8]: def config_interface(intf_name, ip_address, mask):
   ...:     interface = f'interface {intf_name}'
   ...:     no_shut = 'no shutdown'
   ...:     ip_addr = f'ip address {ip_address} {mask}'
   ...:     result = [interface, no_shut, ip_addr]
   ...:     return result
   ...:
```

Function expects such arguments:

- intf_name - interface name
- ip_address - IP address
- mask - subnet mask

Function returns a list of strings to configure interface:

```
In [9]: config_interface('Fa0/1', '10.0.1.1', '255.255.255.0')
Out[9]: ['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0']

In [11]: config_interface('Fa0/10', '10.255.4.1', '255.255.255.0')
Out[11]: ['interface Fa0/10', 'no shutdown', 'ip address 10.255.4.1 255.255.255.0
↪']
```

Suppose you call a function and pass it information that has been obtained from another source, for example from database. For example, interfaces_info list contains parameters for configuring interfaces:

```
In [14]: interfaces_info = [['Fa0/1', '10.0.1.1', '255.255.255.0'],
    ...:                    ['Fa0/2', '10.0.2.1', '255.255.255.0'],
    ...:                    ['Fa0/3', '10.0.3.1', '255.255.255.0'],
    ...:                    ['Fa0/4', '10.0.4.1', '255.255.255.0'],
    ...:                    ['Lo0', '10.0.0.1', '255.255.255.255']]
    ...:
```

If you go through list in the loop and pass nested list as a function argument, an error will occur:

```
In [15]: for info in interfaces_info:
    ...:     print(config_interface(info))
    ...:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-d34ced60c796> in <module>
      1 for info in interfaces_info:
----> 2     print(config_interface(info))
      3

TypeError: config_interface() missing 2 required positional arguments: 'ip_address
↪' and 'mask'
```

Error is quite logical: function expects three arguments and it is given 1 argument - a list. In such a situation it is necessary to unpack arguments. Just add * before passing the list as an argument and there is no error anymore:

```
In [16]: for info in interfaces_info:
    ...:     print(config_interface(*info))
    ...:
['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0']
['interface Fa0/2', 'no shutdown', 'ip address 10.0.2.1 255.255.255.0']
['interface Fa0/3', 'no shutdown', 'ip address 10.0.3.1 255.255.255.0']
['interface Fa0/4', 'no shutdown', 'ip address 10.0.4.1 255.255.255.0']
```

(continues on next page)

(continued from previous page)

```
['interface Lo0', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']
```

Python will unpack info list itself and pass list elements to function as arguments.

Note: Tuple can also be unpacked in this way.

Unpacking keyword alrguments

Similarly, you can unpack dictionary to pass it as keyword arguments.

Check_passwd function (func_check_pass_optional_param_2.py file):

```
In [19]: def check_passwd(username, password, min_length=8, check_username=True):
    ...:     if len(password) < min_length:
    ...:         print('Password is too short')
    ...:         return False
    ...:     elif check_username and username in password:
    ...:         print('Password contains username')
    ...:         return False
    ...:     else:
    ...:         print(f'Password for user {username} has passed all checks')
    ...:         return True
    ...:
```

List of dictionaries username_passwd where username and password are specified:

```
In [20]: username_passwd = [{'username': 'cisco', 'password': 'cisco'},
    ...:                    {'username': 'nata', 'password': 'natapass'},
    ...:                    {'username': 'user', 'password': '123456789'}]
```

If you pass dictionary to check_passwd function, there is an error:

```
In [21]: for data in username_passwd:
    ...:     check_passwd(data)
    ...:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-ad848f85c77f> in <module>
      1 for data in username_passwd:
----> 2     check_passwd(data)
      3
```

(continues on next page)

(continued from previous page)

```
TypeError: check_passwd() missing 1 required positional argument: 'password'
```

Error is because the function has taken dictionary as one argument and believes that it lacks only password argument.

If you add ** before passing a dictionary to function, function will work properly:

```
In [22]: for data in username_passwd:
    ...:     check_passwd(**data)
    ...:
Password is too short
Password contains username
Password for user user has passed all checks

In [23]: for data in username_passwd:
    ...:     print(data)
    ...:     check_passwd(**data)
    ...:
{'username': 'cisco', 'password': 'cisco'}
Password is too short
{'username': 'nata', 'password': 'natapass'}
Password contains username
{'username': 'user', 'password': '123456789'}
Password for user user has passed all checks
```

Python unpacks dictionary and passes it to function as keyword arguments. The `check_passwd(**data)` is converted to a `check_passwd(username='cisco', password='cisco')`.

Example of using variable length keyword arguments and unpacking arguments

Using variable length arguments and unpacking arguments you can pass arguments between functions. Let me give you an example.

Function check_passwd (func_add_user_kwargs_example.py file):

```
In [1]: def check_passwd(username, password, min_length=8, check_username=True):
   ...:     if len(password) < min_length:
   ...:         print('Password is too short')
   ...:         return False
   ...:     elif check_username and username in password:
```

(continues on next page)

(continued from previous page)

```
   ...:         print('Password contains username')
   ...:         return False
   ...:     else:
   ...:         print(f'Password for user {username} has passed all checks')
   ...:         return True
   ...:
```

Function checks password and returns True if password has passed verification and False if not.

Call function in ipython:

```
In [3]: check_passwd('nata', '12345', min_length=3)
Password for user nata has passed all checks
Out[3]: True

In [4]: check_passwd('nata', '12345nata', min_length=3)
Password contains username
Out[4]: False

In [5]: check_passwd('nata', '12345nata', min_length=3, check_username=False)
Password for user nata has passed all checks
Out[5]: True

In [6]: check_passwd('nata', '12345nata', min_length=3, check_username=True)
Password contains username
Out[6]: False
```

We will create add_user_to_users_file function that requests password for specified user, checks it and requests it again if password has not been checked or writes user and password to file if password has been verified

```
In [7]: def add_user_to_users_file(user, users_filename='users.txt'):
   ...:     while True:
   ...:         passwd = input(f'Enter password for user {user}: ')
   ...:         if check_passwd(user, passwd):
   ...:             break
   ...:     with open(users_filename, 'a') as f:
   ...:         f.write(f'{user},{passwd}\n')
   ...:

In [8]: add_user_to_users_file('nata')
Enter password for user nata: natasda
Password is too short
Enter password for user nata: natasdlajsl;fjd
```

(continues on next page)

(continued from previous page)

```
Password contains username
Enter password for user nata: salkfdjsalkdjfsal;dfj
Password for user nata has passed all checks

In [9]: cat users.txt
nata,salkfdjsalkdjfsal;dfj
```

In this version of add_user_to_users_file() function, it is not possible to regulate the minimum password length and whether to verify the presence of a username in password. In the following version of add_user_to_users_file() function, these features are added:

```
In [5]: def add_user_to_users_file(user, users_filename='users.txt', min_length=8,
↪ check_username=True):
   ...:     while True:
   ...:         passwd = input(f'Enter password for user {user}: ')
   ...:         if check_passwd(user, passwd, min_length, check_username):
   ...:             break
   ...:     with open(users_filename, 'a') as f:
   ...:         f.write(f'{user},{passwd}\n')
   ...:

In [6]: add_user_to_users_file('nata', min_length=5)
Enter password for user nata: natas2342
Password contains username
Enter password for user nata: dlfjgkd
Password for user nata has passed all checks
```

You can now specify min_length or check_username when calling a function. However, it was necessary to repeat parameters of check_passwd function in defining of add_user_to_users_file function. This is not very good and when there are many parameters it is just inconvenient, especially considering that check_passwd function can have other parameters.

This happens quite often and Python has a common solution to this problem: all arguments for internal function (in this case it is check_passwd) will be taken in **kwargs. Then, when calling check_passwd() function they will be unpacked into keyword arguments by the same **kwargs syntax.

```
In [7]: def add_user_to_users_file(user, users_filename='users.txt', **kwargs):
   ...:     while True:
   ...:         passwd = input(f'Enter password for user {user}: ')
   ...:         if check_passwd(user, passwd, **kwargs):
   ...:             break
   ...:     with open(users_filename, 'a') as f:
   ...:         f.write(f'{user},{passwd}\n')
```

(continues on next page)

(continued from previous page)

```
   ...:

In [8]: add_user_to_users_file('nata', min_length=5)
Enter password for user nata: alskfdjlksadjf
Password for user nata has passed all checks

In [9]: add_user_to_users_file('nata', min_length=5)
Enter password for user nata: 345
Password is too short
Enter password for user nata: 309487538
Password for user nata has passed all checks
```

In this version you can add arguments to check_passwd() function without having to duplicate them in add_user_to_users_file function.

Further reading

Documenation:

- Defining Functions
- Built-in Functions
- Sorting HOW TO
- Functional Programming HOWTO
- Range function

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 9.1

Create generate_access_config function that generates configuration for access ports.

The function expects arguments:

- a dictionary with interface as a key and VLAN as a value (access_config or access_config_2 dict)
- access ports configuration template as a list of commands (access_mode_template list)

The function should return a list of all ports in access mode with configuration based on the access_mode_template template.

In this task, the beginning of the function is written and you just need to continue writing the function body itself.

An example of a final list (each string is written on a new line for readability):

```
[
"interface FastEthernet0/12",
"switchport mode access",
"switchport access vlan 10",
"switchport nonegotiate",
"spanning-tree portfast",
"spanning-tree bpduguard enable",
"interface FastEthernet0/17",
"switchport mode access",
"switchport access vlan 150",
"switchport nonegotiate",
"spanning-tree portfast",
"spanning-tree bpduguard enable",
...]
```

Check the operation of the function using the access_config dictionary and the list of commands access_mode_template. If the previous check was successful, check the function again using the

dictionary access_config_2 and make sure that the final list contains the correct interface numbers and vlans.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
access_mode_template = [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable",
]

access_config = {"FastEthernet0/12": 10, "FastEthernet0/14": 11, "FastEthernet0/16
↪": 17}

access_config_2 = {
    "FastEthernet0/3": 100,
    "FastEthernet0/7": 101,
    "FastEthernet0/9": 107,
}


def generate_access_config(intf_vlan_mapping, access_template):
    """
    intf_vlan_mapping is a dictionary with interface-VLAN mapping:
         {'FastEthernet0/12': 10,
          'FastEthernet0/14': 11,
          'FastEthernet0/16': 17}
    access_template - list of commands for the port in access mode

    Returns a list of commands.
    """
```

Task 9.1a

Make a copy of the code from the task 9.1.

Add this functionality: add an additional parameter that controls whether port-security configured

- parameter name 'psecurity'
- default is None
- to configure port-security, a list of commands must be passed as a value port-security (port_security_template list)

The function should return a list of all ports in access mode with configuration based on the access_mode_template template and the port_security_template template, if passed. There should not be a new line character at the end of lines in the list.

Check the operation of the function using the example of the access_config dictionary, with the generation of the configuration port-security and without.

An example of a function call:

```
print(generate_access_config(access_config, access_mode_template))
print(generate_access_config(access_config, access_mode_template, port_security_
↪template))
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
access_mode_template = [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable",
]

port_security_template = [
    "switchport port-security maximum 2",
    "switchport port-security violation restrict",
    "switchport port-security"
]

access_config = {"FastEthernet0/12": 10, "FastEthernet0/14": 11, "FastEthernet0/16
↪": 17}
```

Task 9.2

Create generate_trunk_config function that generates configuration for access ports.

The function expects arguments:

- intf_vlan_mapping: expects a dictionary with interface-VLAN mapping (trunk_config or trunk_config_2)
- trunk_template: expects trunk port configuration template as command list (trunk_mode_template list)

The function should return a list of commands with configuration based on the specified ports and trunk_mode_template.

Check the operation of the function using the example of the trunk_config dictionary and a list of commands trunk_mode_template. If the previous check was successful, check the function again on the trunk_config_2 dictionary and make sure that the final list contains the correct numbers interfaces and vlans.

An example of a final list (each string is written on a new line for readability):

```
[
"interface FastEthernet0/1",
"switchport mode trunk",
"switchport trunk native vlan 999",
"switchport trunk allowed vlan 10,20,30",
"interface FastEthernet0/2",
"switchport mode trunk",
"switchport trunk native vlan 999",
"switchport trunk allowed vlan 11,30",
...]
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
trunk_mode_template = [
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan",
]

trunk_config = {
    "FastEthernet0/1": [10, 20, 30],
    "FastEthernet0/2": [11, 30],
    "FastEthernet0/4": [17],
}

trunk_config_2 = {
    "FastEthernet0/11": [120, 131],
    "FastEthernet0/15": [111, 130],
    "FastEthernet0/14": [117],
}
```

Task 9.2a

Make a copy of the code from the task 9.2.

Change the function so that it returns a dictionary instead of a list of commands: - keys: interface names, like 'FastEthernet0/1' - values: the list of commands that you need execute on this interface

Check the operation of the function using the example of the trunk_config dictionary and the trunk_mode_template template.

An example of a final dict (each string is written on a new line for readability):

```
{
    "FastEthernet0/1": [
        "switchport mode trunk",
        "switchport trunk native vlan 999",
        "switchport trunk allowed vlan 10,20,30",
    ],
    "FastEthernet0/2": [
        "switchport mode trunk",
        "switchport trunk native vlan 999",
        "switchport trunk allowed vlan 11,30",
    ],
    "FastEthernet0/4": [
        "switchport mode trunk",
        "switchport trunk native vlan 999",
        "switchport trunk allowed vlan 17",
    ],
}
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
trunk_mode_template = [
    "switchport mode trunk", "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
]

trunk_config = {
    "FastEthernet0/1": [10, 20, 30],
    "FastEthernet0/2": [11, 30],
    "FastEthernet0/4": [17]
}
```

Task 9.3

Create a get_int_vlan_map function that handles the switch configuration file and returns a tuple of two dictionaries:

1. A dictionary of ports in access mode, where the keys are port numbers, and the access VLAN values (numbers):

```
{"FastEthernet0/12": 10,
 "FastEthernet0/14": 11,
 "FastEthernet0/16": 17}
```

2. A dictionary of ports in trunk mode, where the keys are port numbers, and the values are the list of allowed VLANs (list of numbers):

```
{"FastEthernet0/1": [10, 20],
 "FastEthernet0/2": [11, 30],
 "FastEthernet0/4": [17]}
```

The function must have one parameter, config_filename, which expects as an argument the name of the configuration file.

Check the operation of the function using the config_sw1.txt file.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 9.3a

Make a copy of the code from the task 9.3.

Add this functionality: add support for configuration when the port is in VLAN 1 and the access port setting looks like this:

```
interface FastEthernet0/20
    switchport mode access
    duplex auto
```

In this case, information should be added to the dictionary that the port in VLAN 1 Dictionary example:

```
{'FastEthernet0/12': 10,
 'FastEthernet0/14': 11,
 'FastEthernet0/20': 1 }
```

The function must have one parameter, config_filename, which expects as an argument the name of the configuration file.

Check the operation of the function using the config_sw2.txt file.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 9.4

Create a convert_config_to_dict function that handles the switch configuration file and returns a dictionary:

- All top-level commands (global configuration mode) will be keys.
- If the top-level team has subcommands, they must be in the value from the corresponding key, in the form of a list (spaces at the beginning of the line must be removed).
- If the top-level command has no subcommands, then the value will be an empty list

The function must have one parameter, config_filename, which expects as an argument the name of the configuration file.

Check the operation of the function using the config_sw1.txt file.

When processing the configuration file, you should ignore the lines that begin with '!', as well as lines containing words from the ignore list.

To check if a line should be ignored, use the ignore_command function.

The part of the dictionary that the function should return (the full output can be seen in test_task_9_4.py test):

```
{
    "version 15.0": [],
    "service timestamps debug datetime msec": [],
    "service timestamps log datetime msec": [],
    "no service password-encryption": [],
    "hostname sw1": [],
    "interface FastEthernet0/0": [
        "switchport mode access",
        "switchport access vlan 10",
    ],
    "interface FastEthernet0/1": [
        "switchport trunk encapsulation dot1q",
        "switchport trunk allowed vlan 100,200",
        "switchport mode trunk",
    ],
    "interface FastEthernet0/2": [
        "switchport mode access",
        "switchport access vlan 20",
    ],
}
```

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
ignore = ["duplex", "alias", "Current configuration"]


def ignore_command(command, ignore):
    """
    The function checks if the command contains a word from the ignore list.

    command is a string. Command to check
    ignore is a list. Word list

    Returns
    * True if the command contains a word from the ignore list
    * False - if not
    """
    ignore_status = False
    for word in ignore:
        if word in command:
            ignore_status = True
    return ignore_status
```

10. Useful functions

This section discusses the following functions:

print

Function `print` has been used many times in book, but its full syntax has not yet been discussed:

```
print(*items, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Function `print` outputs all elements by separating them by their sep value and finishes output with end value.

All elements that are passed as arguments are converted into strings:

```
In [4]: def f(a):
   ...:     return a
   ...:

In [5]: print(1, 2, f, range(10))
1 2 <function f at 0xb4de926c> range(0, 10)
```

For functions f and range the result is equivalent to str:

```
In [6]: str(f)
Out[6]: '<function f at 0xb4de926c>'

In [7]: str(range(10))
Out[7]: 'range(0, 10)'
```

sep

Parameter sep controls which separator will be used between elements.

By default, space is used:

```
In [8]: print(1, 2, 3)
1 2 3
```

You can change sep value to any other string:

```
In [9]: print(1, 2, 3, sep='|')
1|2|3
```

(continues on next page)

(continued from previous page)

```
In [10]: print(1, 2, 3, sep='\n')
1
2
3

In [11]: print(1, 2, 3, sep=f"\n{'-' * 10}\n")
1
----------
2
----------
3
```

Note: Note that all arguments that manage behavior of `print` function must be passed on as keyword, not positional.

In some situations `print` function can replace join method:

```
In [12]: items = [1, 2, 3, 4, 5]

In [13]: print(*items, sep=', ')
1, 2, 3, 4, 5
```

end

Parameter end controls which value will be displayed after all elements are printed. By default, new line character is used:

```
In [19]: print(1, 2, 3)
1 2 3
```

You can change end value to any other string:

```
In [20]: print(1, 2, 3, end='\n' + '-' * 10)
1 2 3
----------
```

file

Parameter `file` controls where values of `print` function are displayed. The default output is sys.stdout.

Python allows to pass to `file` as an argument any object with write(string) method.

```
In [1]: f = open('result.txt', 'w')

In [2]: for num in range(10):
   ...:     print('Item {}'.format(num), file=f)
   ...:

In [3]: f.close()

In [4]: cat result.txt
Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
```

flush

By default, when writing to a file or print to a standard output stream, the output is buffered. Function `print` allows to disable buffering. You can control it in a file.

Example script that displays a number from 0 to 10 every second (print_nums.py file):

```
import time

for num in range(10):
    print(num)
    time.sleep(1)
```

Try running a script and make sure the numbers are displayed once per second.

Now, a similar script but the numbers will appear in one line (print_nums_oneline.py file):

```
import time

for num in range(10):
    print(num, end=' ')
    time.sleep(1)
```

Try running a function. Numbers does not appear one per second but all appear after 10 seconds.

This is because when output is displayed on standard output, `flush` is performed after new line character.

In order to make script work properly `flush` should be set to True (print_nums_oneline_fixed.py file):

```
import time

for num in range(10):
    print(num, end=' ', flush=True)
    time.sleep(1)
```

range

Function range returns an immutable sequence of numbers as a range object.

Function syntax:

```
range(stop)
range(start, stop[, step])
```

Parameters of function:

- **start** - from what number the sequence begins. By default - 0
- **stop** - on which number the sequence of numbers ends. Mentioned number is not included in range
- **step** - with what step numbers increase. By default 1

Function range stores only **start**, **stop** and **step** values and calculates values as necessary. This means that regardless of the size of range that describes range function, it will always occupy a fixed amount of memory.

The easiest range option is to pass only **stop** value:

```
In [1]: range(5)
Out[1]: range(0, 5)

In [2]: list(range(5))
Out[2]: [0, 1, 2, 3, 4]
```

If two arguments are passed, the first is used as **start** and the second as **stop**:

```
In [3]: list(range(1, 5))
Out[3]: [1, 2, 3, 4]
```

And in order to indicate sequence step, you have to pass three arguments:

```
In [4]: list(range(0, 10, 2))
Out[4]: [0, 2, 4, 6, 8]

In [5]: list(range(0, 10, 3))
Out[5]: [0, 3, 6, 9]
```

Function range can also generate descending sequences of numbers:

```
In [6]: list(range(10, 0, -1))
Out[6]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

In [7]: list(range(5, -1, -1))
Out[7]: [5, 4, 3, 2, 1, 0]
```

To get a descending sequence use a negative step and specify **start** by a greater number and **stop** by a smaller number.

In descending sequence the steps can also be different:

```
In [8]: list(range(10, 0, -2))
Out[8]: [10, 8, 6, 4, 2]
```

Function supports negative **start** and **stop** values:

```
In [9]: list(range(-10, 0, 1))
Out[9]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]

In [10]: list(range(0, -10, -1))
Out[10]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

The range object supports all operations that support sequences in Python, except addition and multiplication.

Check whether a number falls within a range:

```
In [11]: nums = range(5)

In [12]: nums
Out[12]: range(0, 5)

In [13]: 3 in nums
Out[13]: True

In [14]: 7 in nums
```

(continues on next page)

(continued from previous page)

```
Out[14]: False
```

Note: Starting with Python 3.2 this check is performed in constant time (O(1)).

You can get a specific range element:

```
In [15]: nums = range(5)

In [16]: nums[0]
Out[16]: 0

In [17]: nums[-1]
Out[17]: 4
```

Range supports slices:

```
In [18]: nums = range(5)

In [19]: nums[1:]
Out[19]: range(1, 5)

In [20]: nums[:3]
Out[20]: range(0, 3)
```

You can get range length:

```
In [21]: nums = range(5)

In [22]: len(nums)
Out[22]: 5
```

And a minimum and maximum element:

```
In [23]: nums = range(5)

In [24]: min(nums)
Out[24]: 0

In [25]: max(nums)
Out[25]: 4
```

In addition, `range` object supports `index` method:

```
In [26]: nums = range(1, 7)

In [27]: nums.index(3)
Out[27]: 2
```

sorted

Function `sorted` returns a new sorted list that is obtained from an iterable object that has been passed as an argument. Function also supports additional options that allow you to control sorting.

The first aspect that is important to pay attention to - `sorted` returns a list. If you sort a list of items, a new list is returned:

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [2]: sorted(list_of_words)
Out[2]: ['', 'dict', 'list', 'one', 'two']
```

When sorting a tuple also a list returns:

```
In [3]: tuple_of_words = ('one', 'two', 'list', '', 'dict')

In [4]: sorted(tuple_of_words)
Out[4]: ['', 'dict', 'list', 'one', 'two']
```

Sorting set:

```
In [5]: set_of_words = {'one', 'two', 'list', '', 'dict'}

In [6]: sorted(set_of_words)
Out[6]: ['', 'dict', 'list', 'one', 'two']
```

Sorting string:

```
In [7]: string_to_sort = 'long string'

In [8]: sorted(string_to_sort)
Out[8]: [' ', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r', 's', 't']
```

If you pass a dictionary to `sorted` the function will return sorted list of keys:

```
In [9]: dict_for_sort = {
   ...:         'id': 1,
   ...:         'name': 'London',
```

(continues on next page)

(continued from previous page)

```
   ...:         'it_vlan': 320,
   ...:         'user_vlan': 1010,
   ...:         'mngmt_vlan': 99,
   ...:         'to_name': None,
   ...:         'to_id': None,
   ...:         'port': 'G1/0/11'
   ...: }

In [10]: sorted(dict_for_sort)
Out[10]:
['id',
 'it_vlan',
 'mngmt_vlan',
 'name',
 'port',
 'to_id',
 'to_name',
 'user_vlan']
```

reverse

The reverse flag allows you to control the sorting order. By default, sorting will be in ascending order of items:

```
In [11]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [12]: sorted(list_of_words)
Out[12]: ['', 'dict', 'list', 'one', 'two']

In [13]: sorted(list_of_words, reverse=True)
Out[13]: ['two', 'one', 'list', 'dict', '']
```

key

With key option you can specify how to perform sorting. The key parameter expects function by which the comparison should be performed.

For example you can sort a list of strings by string length:

```
In [14]: list_of_words = ['one', 'two', 'list', '', 'dict']

```

(continues on next page)

(continued from previous page)

```
In [15]: sorted(list_of_words, key=len)
Out[15]: ['', 'one', 'two', 'list', 'dict']
```

If you want to sort dictionary keys but ignore string register:

```
In [16]: dict_for_sort = {
    ...:         'id': 1,
    ...:         'name': 'London',
    ...:         'IT_VLAN': 320,
    ...:         'User_VLAN': 1010,
    ...:         'Mngmt_VLAN': 99,
    ...:         'to_name': None,
    ...:         'to_id': None,
    ...:         'port': 'G1/0/11'
    ...: }

In [17]: sorted(dict_for_sort, key=str.lower)
Out[17]:
['id',
 'IT_VLAN',
 'Mngmt_VLAN',
 'name',
 'port',
 'to_id',
 'to_name',
 'User_VLAN']
```

The key option can accept any functions, not only embedded ones. It is also convenient to use anonymous lambda() function.

Using key option you can sort objects by any element. However, this requires either lambda() or special functions from **operator** module.

For example, in order to sort the list of tuples with two items in the second element, you should use this technique:

```
In [18]: from operator import itemgetter

In [19]: list_of_tuples = [('IT_VLAN', 320),
    ...:  ('Mngmt_VLAN', 99),
    ...:  ('User_VLAN', 1010),
    ...:  ('DB_VLAN', 11)]

In [20]: sorted(list_of_tuples, key=itemgetter(1))
```

(continues on next page)

(continued from previous page)

```
Out[20]: [('DB_VLAN', 11), ('Mngmt_VLAN', 99), ('IT_VLAN', 320), ('User_VLAN',␣
↪1010)]
```

enumerate

Sometimes, when iterating objects in **for** loop, it is necessary not only to get object itself but also its sequence number. This can be done by creating an additional variable that will increase by one with each iteration. However, it is much more convenient to do this with iterator `enumerate`.

Basic example:

```
In [15]: list1 = ['str1', 'str2', 'str3']

In [16]: for position, string in enumerate(list1):
    ...:     print(position, string)
    ...:
0 str1
1 str2
2 str3
```

`enumerate` can count not only from scratch but from any value that has been given to it after object:

```
In [17]: list1 = ['str1', 'str2', 'str3']

In [18]: for position, string in enumerate(list1, 100):
    ...:     print(position, string)
    ...:
100 str1
101 str2
102 str3
```

Sometimes it is necessary to check what iterator has generated. If you want to see full content that iterator generates you can use list() function:

```
In [19]: list1 = ['str1', 'str2', 'str3']

In [20]: list(enumerate(list1, 100))
Out[20]: [(100, 'str1'), (101, 'str2'), (102, 'str3')]
```

An example of using enumerate for EEM

This example uses Cisco EEM. In a nutshell, EEM allows you to perform some actions in response to an event.

EEM applet looks like this:

```
event manager applet Fa0/1_no_shut
 event syslog pattern "Line protocol on Interface FastEthernet0/0, changed state␣
↪to down"
 action 1 cli command "enable"
 action 2 cli command "conf t"
 action 3 cli command "interface fa0/1"
 action 4 cli command "no sh"
```

In EEM, in a situation where many actions need to be performed it is inconvenient to type `action x cli command` each time. Plus, most often, there is already a ready piece of configuration that must be executed by EEM.

A simple Python script can generate EEM commands based on existing command list (enumerate_eem.py file):

```
import sys

config = sys.argv[1]

with open(config, 'r') as f:
    for i, command in enumerate(f, 1):
        print('action {:04} cli command "{}"'.format(i, command.rstrip()))
```

In this example, commands are read from a file and then EEM prefix is added to each line.

File with commands looks like this (r1_config.txt):

```
en
conf t
no int Gi0/0/0.300
no int Gi0/0/0.301
no int Gi0/0/0.302
int range gi0/0/0-2
 channel-group 1 mode active
interface Port-channel1.300
 encapsulation dot1Q 300
 vrf forwarding Management
 ip address 10.16.19.35 255.255.255.248
```

The output is:

```
$ python enumerate_eem.py r1_config.txt
action 0001 cli command "en"
action 0002 cli command "conf t"
action 0003 cli command "no int Gi0/0/0.300"
action 0004 cli command "no int Gi0/0/0.301"
action 0005 cli command "no int Gi0/0/0.302"
action 0006 cli command "int range gi0/0/0-2"
action 0007 cli command " channel-group 1 mode active"
action 0008 cli command "interface Port-channel1.300"
action 0009 cli command " encapsulation dot1Q 300"
action 0010 cli command " vrf forwarding Management"
action 0011 cli command " ip address 10.16.19.35 255.255.255.248"
```

zip

zip function:

- sequences are passed to function
- zip returns an iterator with tuples in which n-tuple consists of n-elements of sequences that have been passed as arguments
- for example, 10th tuple will contain 10th element of each of passed sequences
- if sequences with different lengths have been passed to input, they will all be cut by the shortest sequence
- the order of elements is respected

Note: Since zip is an iterator, list is used to show its contents

---

Example of using zip:

```
In [1]: a = [1, 2, 3]

In [2]: b = [100, 200, 300]

In [3]: list(zip(a, b))
Out[3]: [(1, 100), (2, 200), (3, 300)]
```

Use zip with lists of different lengths:

```
In [4]: a = [1, 2, 3, 4, 5]

```

(continues on next page)

(continued from previous page)

```
In [5]: b = [10, 20, 30, 40, 50]

In [6]: c = [100, 200, 300]

In [7]: list(zip(a, b, c))
Out[7]: [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
```

Using zip to create a dictionary

Example of using zip to create a dictionary:

```
In [4]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
In [5]: d_values = ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4',
↪'10.255.0.1']

In [6]: list(zip(d_keys, d_values))
Out[6]:
[('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('vendor', 'Cisco'),
 ('model', '4451'),
 ('IOS', '15.4'),
 ('IP', '10.255.0.1')]

In [7]: dict(zip(d_keys, d_values))
Out[7]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
In [8]: r1 = dict(zip(d_keys, d_values))

In [9]: r1
Out[9]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

In example below there is a separate list which stores keys and a dictionary which stores information about each device in form of list (to preserve order).

Collect them in dictionary with keys from list and information from dictionary *data*:

```
In [10]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

In [11]: data = {
   ....: 'r1': ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.
↪255.0.1'],
   ....: 'r2': ['london_r2', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.
↪255.0.2'],
   ....: 'sw1': ['london_sw1', '21 New Globe Walk', 'Cisco', '3850', '3.6.XE',
↪'10.255.0.101']
   ....: }

In [12]: london_co = {}

In [13]: for k in data.keys():
   ....:     london_co[k] = dict(zip(d_keys, data[k]))
   ....:

In [14]: london_co
Out[14]:
{'r1': {'IOS': '15.4',
  'IP': '10.255.0.1',
  'hostname': 'london_r1',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'r2': {'IOS': '15.4',
  'IP': '10.255.0.2',
  'hostname': 'london_r2',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'sw1': {'IOS': '3.6.XE',
  'IP': '10.255.0.101',
  'hostname': 'london_sw1',
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

all

Function `all` returns True if all elements are true (or object is empty).

```
In [1]: all([False, True, True])
Out[1]: False

In [2]: all([True, True, True])
Out[2]: True

In [3]: all([])
Out[3]: True
```

For example, it is possible to check that all octets in an IP address are numbers:

```
In [4]: ip = '10.0.1.1'

In [5]: all([i.isdigit() for i in ip.split('.')])
Out[5]: True

In [6]: all([i.isdigit() for i in '10.1.1.a'.split('.')])
Out[6]: False
```

any

Function any returns True if at least one element is true.

```
In [7]: any([False, True, True])
Out[7]: True

In [8]: any([False, False, False])
Out[8]: False

In [9]: any([])
Out[9]: False

In [10]: any([i.isdigit() for i in '10.1.1.a'.split('.')])
Out[10]: True
```

For example, with any you can replace `ignore_command` function:

```
def ignore_command(command):
    '''
```

(continues on next page)

(continued from previous page)

```
    Function checks if command contains a word from ignore list.
    * command is a string. Command that need to be checked returns True
    * if command contains a word from ignore list, False - if not.
    '''
    ignore = ['duplex', 'alias', 'Current configuration']

    for word in ignore:
        if word in command:
            return True
    return False
```

To this option:

```
def ignore_command(command):
    '''
    Function checks if command contains a word from ignore list.
    * command is a string. Command that need to be checked returns True
    * if command contains a word from ignore list, False - if not.
    '''
    ignore = ['duplex', 'alias', 'Current configuration']

    return any([word in command for word in ignore])
```

Anonymous function (lambda expression)

In Python, lambda expression allows creation of anonymous functions - functions that are not tied to a name.

Anonymous function:

- may contain only one expression
- can pass as many arguments as you want

Standard function:

```
In [1]: def sum_arg(a, b): return a + b

In [2]: sum_arg(1, 2)
Out[2]: 3
```

Similar anonymous function or lambda function:

```
In [3]: sum_arg = lambda a, b: a + b

In [4]: sum_arg(1, 2)
Out[4]: 3
```

Note that there is no **return** operator in `lambda` function definition because there can only be one expression in this function that always returns a value and closes the function.

Function `lambda` is convenient to use in expressions where you need to write a small function for data processing. For example, in `sorted` function you can use lambda expression to specify sorting key:

```
In [5]: list_of_tuples = [('IT_VLAN', 320),
   ...:  ('Mngmt_VLAN', 99),
   ...:  ('User_VLAN', 1010),
   ...:  ('DB_VLAN', 11)]

In [6]: sorted(list_of_tuples, key=lambda x: x[1])
Out[6]: [('DB_VLAN', 11), ('Mngmt_VLAN', 99), ('IT_VLAN', 320), ('User_VLAN',␣
↪1010)]
```

Function `lambda` is also useful in `map` and `filter` functions which will be discussed in the following sections.

map

Function `map` applies function to each element of sequence and returns iterator with result.

For example, `map` can be used to perform element transformations. Convert all strings to uppercase:

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [2]: map(str.upper, list_of_words)
Out[2]: <map at 0xb45eb7ec>

In [3]: list(map(str.upper, list_of_words))
Out[3]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

Converting to numbers:

```
In [3]: list_of_str = ['1', '2', '5', '10']

In [4]: list(map(int, list_of_str))
Out[4]: [1, 2, 5, 10]
```

With map it is convenient to use lambda expressions:

```
In [5]: vlans = [100, 110, 150, 200, 201, 202]

In [6]: list(map(lambda x: 'vlan {}'.format(x), vlans))
Out[6]: ['vlan 100', 'vlan 110', 'vlan 150', 'vlan 200', 'vlan 201', 'vlan 202']
```

If map function expects two arguments, two lists are passed:

```
In [7]: nums = [1, 2, 3, 4, 5]

In [8]: nums2 = [100, 200, 300, 400, 500]

In [9]: list(map(lambda x, y: x*y, nums, nums2))
Out[9]: [100, 400, 900, 1600, 2500]
```

List comprehension instead of map

As a rule, you can use list comprehension instead of map. The list comprehension option is often clearer, and in some cases even faster.

> Alex Martelli response with comparison of map and list comprehension

But map can be more effective when you have to generate a large number of elements because map is an iterator and list comprehension generates a list.

Examples similar to those above in list comprehension version.

Convert all strings to uppercase:

```
In [48]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [49]: [str.upper(word) for word in list_of_words]
Out[49]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

Converting to numbers:

```
In [50]: list_of_str = ['1', '2', '5', '10']

In [51]: [int(i) for i in list_of_str]
Out[51]: [1, 2, 5, 10]
```

String formatting:

```
In [52]:  vlans = [100, 110, 150, 200, 201, 202]

```

(continues on next page)

(continued from previous page)

```
In [53]: ['vlan {}'.format(x) for x in vlans]
Out[53]: ['vlan 100', 'vlan 110', 'vlan 150', 'vlan 200', 'vlan 201', 'vlan 202']
```

Use `zip` to get pairs of elements:

```
In [54]: nums = [1, 2, 3, 4, 5]

In [55]: nums2 = [100, 200, 300, 400, 500]

In [56]: [x * y for x, y in zip(nums, nums2)]
Out[56]: [100, 400, 900, 1600, 2500]
```

filter

Function `filter()` applies function to all sequence elements and returns an iterator with those objects for which function has returned True.

For example, return only those strings that contain numbers:

```
In [1]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']

In [2]: filter(str.isdigit, list_of_strings)
Out[2]: <filter at 0xb45eb1cc>

In [3]: list(filter(str.isdigit, list_of_strings))
Out[3]: ['100', '1', '50']
```

From the list of numbers leave only odd:

```
In [3]: list(filter(lambda x: x % 2, [10, 111, 102, 213, 314, 515]))
Out[3]: [111, 213, 515]
```

Similarly, only even ones:

```
In [4]: list(filter(lambda x: not x % 2, [10, 111, 102, 213, 314, 515]))
Out[4]: [10, 102, 314]
```

From the list of words leave only those with more than two letters:

```
In [5]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [6]: list(filter(lambda x: len(x) > 2, list_of_words))
Out[6]: ['one', 'two', 'list', 'dict']
```

List comprehension instead of filter

As a rule, you can use list comprehension instead of `filter`.

Examples similar to those above in list comprehension version.

Return only those strings that contain numbers:

```
In [7]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']

In [8]: [s for s in list_of_strings if s.isdigit()]
Out[8]: ['100', '1', '50']
```

Odd/even numbers:

```
In [9]: nums = [10, 111, 102, 213, 314, 515]

In [10]: [n for n in nums if n % 2]
Out[10]: [111, 213, 515]

In [11]: [n for n in nums if not n % 2]
Out[11]: [10, 102, 314]
```

From the list of words leave only those with more than two letters:

```
In [12]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [13]: [word for word in list_of_words if len(word) > 2]
Out[13]: ['one', 'two', 'list', 'dict']
```

11. Modules

Module in Python is a plain text file with Python code and **.py** extention. It allows logical ordering and grouping of the code. Division into modules can be done, for example, by this logic:

- division of data, formatting and code logic
- grouping functions and other objects by functionality

The good thing about modules is that they allow you to reuse already written code and not copy it (for example, do not copy a previously written function).

Module import

Python has several ways to import a module:

- `import module`
- `import module as`
- `from module import object`
- `from module import *`

`import module`

Example of **import module**:

```
In [1]: dir()
Out[1]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'quit']

In [2]: import os

In [3]: dir()
Out[3]:
['In',
 'Out',
 ...
 'exit',
```

(continues on next page)

(continued from previous page)

```
 'get_ipython',
 'os',
 'quit']
```

After importing the **os** module appeared in the output `dir`.This means that it is now in the current namespace.

To call some function or method from **os** module you should specify `os.` and then object name:

```
In [4]: os.getlogin()
Out[4]: 'natasha'
```

This import method is good because module objects do not enter the namespace of current program. That is, if you create a function named `getlogin` it will not conflict with the same function of **os** module.

Note: If file name contains a dot, the standard way of importing will not work. In such cases, another method is used.

---

import module as

Construction **import module as** allows importing a module under a different name (usually shorter):

```
In [1]: import subprocess as sp

In [2]: sp.check_output('ping -c 2 -n  8.8.8.8', shell=True)
Out[2]: 'PING 8.8.8.8 (8.8.8.8): 56 data bytes\n64 bytes from 8.8.8.8: icmp_seq=0␣
↪ttl=48 time=49.880 ms\n64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=46.875␣
↪ms\n\n--- 8.8.8.8 ping statistics ---\n2 packets transmitted, 2 packets␣
↪received, 0.0% packet loss\nround-trip min/avg/max/stddev = 46.875/48.377/49.
↪880/1.503 ms\n'
```

from module import object

Option **from module import object** is convenient to use when only few functions are needed from whole module:

```
In [1]: from os import getlogin, getcwd
```

These functions are now available in the current namespace:

```
In [2]: dir()
Out[2]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'getcwd',
 'getlogin',
 'quit']
```

They can be called without module name:

```
In [3]: getlogin()
Out[3]: 'natasha'

In [4]: getcwd()
Out[4]: '/Users/natasha/Desktop/Py_net_eng/code_test'
```

`from module import *`

Option `from module import *` imports all module names into the current namespace:

```
In [1]: from os import *

In [2]: dir()
Out[2]:
['EX_CANTCREAT',
 'EX_CONFIG',
 ...
 'wait',
 'wait3',
 'wait4',
 'waitpid',
 'walk',
 'write']

In [3]: len(dir())
Out[3]: 218
```

There are many objects in **os** module, so the output is shortened. At the end, length of the list of names of current namespace is specified.

This import option is best not to use. With such code import it is not clear which function is taken,

for example from **os** module. This makes it much harder to understand the code.

Create your own modules

Module is a file with `.py` extension and Python code.

Example of creating your own modules and importing a function from one module to another.

File check_ip_function.py:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


ip1 = '10.1.1.1'
ip2 = '10.1.1'

print('Checking IP...')
print(ip1, check_ip(ip1))
print(ip2, check_ip(ip2))
```

File check_ip_function.py has created check_ip function which checks that argument is an IP address. This is done by using **ipaddress** module which will be discussed in the next section.

Function `ipaddress.ip_address` checks the correctness of IP address and generates ValueError exception if address is not validated. Function `check_ip` returns True if address is valid and False if not.

If you run check_ip_function.py script, the output is:

```
$ python check_ip_function.py
Checking IP...
10.1.1.1 True
10.1.1 False
```

The second script imports `check_ip` function and uses it to select from address list only those that passed the check (get_correct_ip.py file):

```
from check_ip_function import check_ip


def return_correct_ip(ip_addresses):
    correct = []
    for ip in ip_addresses:
        if check_ip(ip):
            correct.append(ip)
    return correct

print('Cheking list of IP addresses')
ip_list = ['10.1.1.1', '8.8.8.8', '2.2.2']
correct = return_correct_ip(ip_list)
print(correct)
```

First line imports check_ip function from check_ip_function.py module.

Result of script execution:

```
$ python get_correct_ip.py
Cheking IP...
10.1.1.1 True
10.1.1 False
Cheking list of IP addresses
['10.1.1.1', '8.8.8.8']
```

Note that not only information from get_correct_ip.py script is displayed, but also information from check_ip_function.py. This is because any type of import executes the entire script. That is, even when import looks like `from check_ip_function import check_ip`, entire check_ip_function.py script is executed, not just check_ip function. As a result, all messages of imported script will be displayed.

Messages from imported script are not scary, they are just confusing. Worse when script performed some kind of connection to equipment and when importing a function from it, we will have to wait for connection to take place.

Python can specify that some strings should not be executed when importing. This is discussed in the following subsection.

Note: Function `return_correct_ip` can be replaced by a `filter` or a list comprehension. Above is used the longer but most likely more understandable option:

```
In [19]: list(filter(check_ip, ip_list))
Out[19]: ['10.1.1.1', '8.8.8.8']
```

(continues on next page)

(continued from previous page)

```
In [20]: [ip for ip in ip_list if check_ip(ip)]
Out[20]: ['10.1.1.1', '8.8.8.8']

In [21]: def return_correct_ip(ip_addresses):
    ...:     return [ip for ip in ip_addresses if check_ip(ip)]
    ...:

In [22]: return_correct_ip(ip_list)
Out[22]: ['10.1.1.1', '8.8.8.8']
```

if __name__ == "__main__"

Often script can be executed independently and can be imported as a module by another script. Since importing a script runs this script, it is often necessary to specify that some strings should not be executed when importing.

In previous example there were two scripts: check_ip_function.py and get_correct_ip.py. And when starting get_correct_ip.py, print() from check_ip_function.py was displayed.

Python has a special technique that specifies that a code must not be executed at import: all lines that are in `if __name__ == '__main__'` block are not executed at import.

Variable __name__ is a special variable that will be equal to "__main__" only if file is run as the main program and is set equal to module name when importing the module. That is, `if __name__ == '__main__'` condition checks whether a file was run directly.

As a rule, `if __name__ == '__main__'` block includes all function calls and information output on the standard output stream. That is, in check_ip_function.py script this block conytains everything except import and `return_correct_ip` function:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


if __name__ == '__main__':
    ip1 = '10.1.1.1'
```

(continues on next page)

(continued from previous page)

```
    ip2 = '10.1.1'

    print('Cheking IP...')
    print(ip1, check_ip(ip1))
    print(ip2, check_ip(ip2))
```

Result of script execution:

```
$ python check_ip_function.py
Cheking IP...
10.1.1.1 True
10.1.1 False
```

When you start check_ip_function.py script directly, all lines are executed, because __name__ variable in this case is equal to `'__main__'`.

Script get_correct_ip.py remains unchanged

```
from check_ip_function import check_ip


def return_correct_ip(ip_addresses):
    correct = []
    for ip in ip_addresses:
        if check_ip(ip):
            correct.append(ip)
    return correct


print('Checking list of IP addresses')
ip_list = ['10.1.1.1', '8.8.8.8', '2.2.2']
correct = return_correct_ip(ip_list)
print(correct)
```

Execution of get_correct_ip.py script:

```
$ python get_correct_ip.py
Checking list of IP addresses
['10.1.1.1', '8.8.8.8']
```

Now the output contains only information from script getcorrect__ip.py.

In general, it is better to write all code that calls functions and outputs something to the standard output stream inside `if __name__ == '__main__'` block.

Warning: Starting with Section 11, for tests to work correctly you have to always write a function call in task file within `if __name__ == '__main__'` block. The absence of this block will not cause errors in all tasks, but it will still avoid problems.

Further reading

The Python Standard Library:

- Python Module Index
- Python 3 Module of the Week

Docs:

- Python tutorial. Modules
- os
- argparse
- subprocess
- ipaddress

Video:

- David Beazley - Modules and Packages: Live and Let Die! - PyCon 2015

argparse

- Argparse docs
- PyMOTW

tabulate

- tabulate docs
- Pretty printing tables in Python
- Tabulate 0.7.1 with LaTeX & MediaWiki tables

Stack Overflow:

- Printing Lists as Tabular Data.

pprint

- pprint — Data pretty printer
- PyMOTW. pprint — Pretty-Print Data Structures

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 11.1

Create a function parse_cdp_neighbors that handles show cdp neighbors command output.

The function must have one parameter, command_output, which expects a single string of command output as an argument (not a filename). To do this, you need to read the entire contents of the file into a string, and then pass the string as an argument to the function (how to pass the command output is shown in the code below).

The function should return a dictionary that describes the connections between devices.

For example, if the following output was passed as an argument:

```
R4>show cdp neighbors

Device ID    Local Intrfce   Holdtme     Capability       Platform    Port ID
R5           Fa 0/1          122           R S I           2811       Fa 0/1
R6           Fa 0/2          143           R S I           2811       Fa 0/0
```

Function should return a dictionary:

```
{("R4", "Fa0/1"): ("R5", "Fa0/1"),
 ("R4", "Fa0/2"): ("R6", "Fa0/0")}
```

In the dictionary, interfaces must be written without a space between type and name. That is, so Fa0/0, and not so Fa 0/0.

Check the operation of the function on the contents of the sh_cdp_n_sw1.txt file. In this case, the function should work on other files (the test checks the operation of the function on the output from sh_cdp_n_sw1.txt and sh_cdp_n_r3.txt).

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
def parse_cdp_neighbors(command_output):
    """
```

(continues on next page)

(continued from previous page)

```
    Here we pass the output of the command as single string because it is in this
    form that received command output from equipment. Taking the output of
    the command as an argument, instead of a filename, we make the function more
    generic: it can work both with files and with output from equipment.
    Plus, we learn to work with such a output.
    """


if __name__ == "__main__":
    with open("sh_cdp_n_sw1.txt") as f:
        print(parse_cdp_neighbors(f.read()))
```

Task 11.2

Create a create_network_map function that processes the show cdp neighbors command output from multiple files and merges it into one common topology.

The function must have one parameter, filenames, which expects as an argument a list of filenames containing the output of the show cdp neighbors command.

The function should return a dictionary that describes the connections between devices. The structure of the dictionary is the same as in task 11.1:

```
{("R4", "Fa0/1"): ("R5", "Fa0/1"),
 ("R4", "Fa0/2"): ("R6", "Fa0/0")}
```

Generate topology that matches the output from the files:

- sh_cdp_n_sw1.txt
- sh_cdp_n_r1.txt
- sh_cdp_n_r2.txt
- sh_cdp_n_r3.txt

Do not copy the code of the parse_cdp_neighbors and draw_topology functions. If the parse_cdp_neighbors function cannot process the output of one of the command output files, you need to correct the function code in task 11.1.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
infiles = [
    "sh_cdp_n_sw1.txt",
    "sh_cdp_n_r1.txt",
    "sh_cdp_n_r2.txt",
```

(continues on next page)

(continued from previous page)

```
    "sh_cdp_n_r3.txt",
]
```

Task 11.2a

Note: To complete this task, graphviz must be installed: apt-get install graphviz

And a python module to work with graphviz: pip install graphviz

---

Use the create_network_map function from task 11.2 to create the topology dict for files:

- sh_cdp_n_sw1.txt
- sh_cdp_n_r1.txt
- sh_cdp_n_r2.txt
- sh_cdp_n_r3.txt

Using the draw_topology function from the draw_network_graph.py file, draw schema for the topology dict received with create_network_map function. You need to figure out how to work with the draw_topology function on your own, by reading the function description in the draw_network_graph.py file. The resulting scheme will be written to the svg file - it can be opened with a browser.

With the current topology dictionary, extra connections are drawn on the diagram. They occur because one CDP file (sh_cdp_n_r1.txt) describes connection `("R1", "Eth0/0"): ("SW1", "Eth0/1")` and another (sh_cdp_n_sw1.txt) describes connection `("SW1", "Eth0/1"): ("R1", "Eth0/0")`.

In this task, you need to create a new function unique_network_map, which of these two connections will leave only one, for correct drawing of the schema. In this case, it does not matter which of the connections to leave.

The unique_network_map function must have one topology_dict parameter, which expects a dictionary as an argument. It should be a dictionary resulting from the create_network_map function call.

Dict example:

```
{
    ("R1", "Eth0/0"): ("SW1", "Eth0/1"),
    ("R2", "Eth0/0"): ("SW1", "Eth0/2"),
    ("R2", "Eth0/1"): ("SW2", "Eth0/11"),
    ("R3", "Eth0/0"): ("SW1", "Eth0/3"),
```

(continues on next page)

(continued from previous page)

```
    ("R3", "Eth0/1"): ("R4", "Eth0/0"),
    ("R3", "Eth0/2"): ("R5", "Eth0/0"),
    ("SW1", "Eth0/1"): ("R1", "Eth0/0"),
    ("SW1", "Eth0/2"): ("R2", "Eth0/0"),
    ("SW1", "Eth0/3"): ("R3", "Eth0/0"),
    ("SW1", "Eth0/5"): ("R6", "Eth0/1"),
}
```

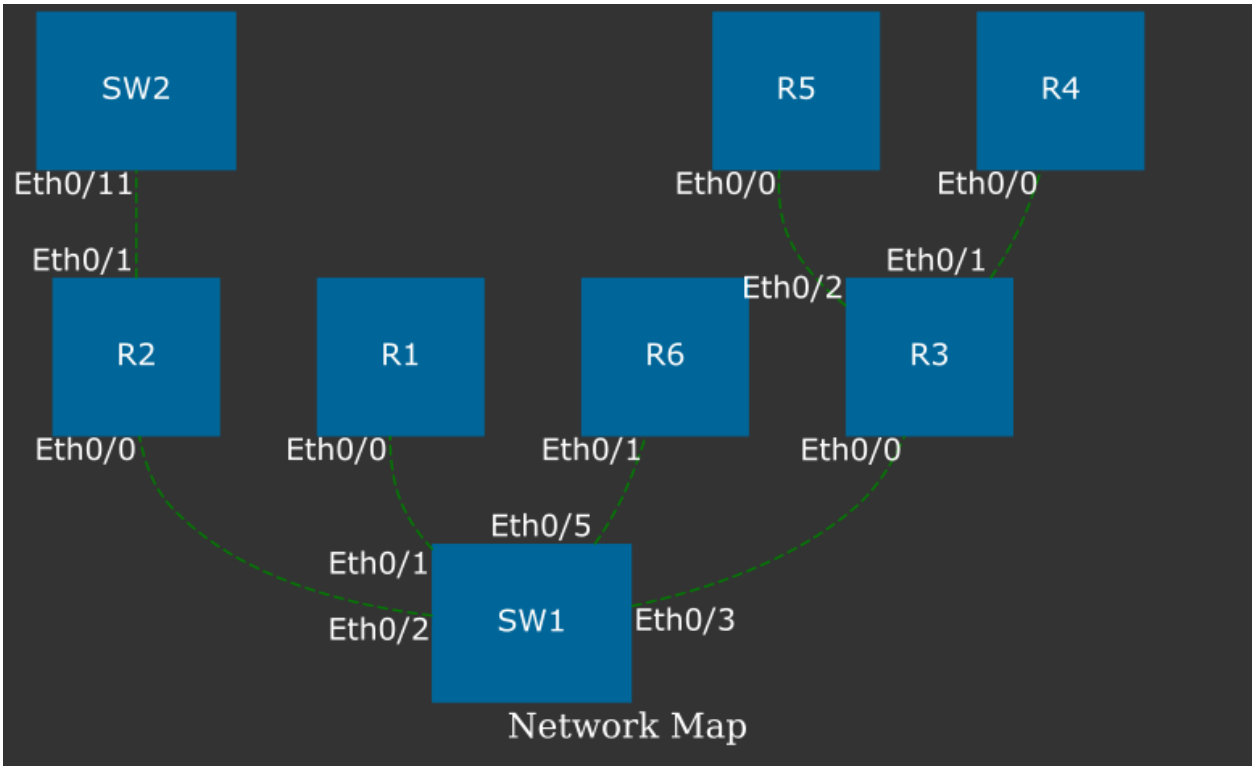
The function should return a dictionary that describes the connections between devices. In the dictionary, you need to get rid of "duplicate" connections and leave only one of them.

The structure of the final dict is the same as in task 11.2:

```
{("R4", "Fa0/1"): ("R5", "Fa0/1"),
 ("R4", "Fa0/2"): ("R6", "Fa0/0")}
```

After creating the function, try drawing the topology again, now for the dictionary returned by the unique_network_map function.

The result should look the same as the diagram in task_11_2a_topology.svg



Wherein:

- The arrangement of devices on the diagram may be different

- Connections must match the diagram

Do not copy the code of the create_network_map and draw_topology functions.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

```
infiles = [
    "sh_cdp_n_sw1.txt",
    "sh_cdp_n_r1.txt",
    "sh_cdp_n_r2.txt",
    "sh_cdp_n_r3.txt",
]
```

12. Useful modules

This section covers these modules:

subprocess

Subprocess module allows you to create new processes. It can then connect to standard input/output/error streams and receive a return code.

Subprocess can for example execute any Linux commands from script. And depending on situation, get the output or just check that command has been performed correctly.

Note: In Python 3.5, syntax of subprocess module has changed.

Function `subprocess.run()`

Function `subprocess.run()` is the main way of working with subprocess module.

The easiest way to use a function is to call it in this way:

```
In [1]: import subprocess

In [2]: result = subprocess.run('ls')
ipython_as_mngmt_console.md  README.md         version_control.md
module_search.md             useful_functions
naming_conventions           useful_modules
```

The **result** variable now contains a special CompletedProcess object. From this object you can get information about execution of process, such as return code:

```
In [3]: result
Out[3]: CompletedProcess(args='ls', returncode=0)

In [4]: result.returncode
Out[4]: 0
```

Code 0 means that program was executed successfully.

If it is necessary to call a command with arguments, it should be passed in this way (as a list):

```
In [5]: result = subprocess.run(['ls', '-ls'])
total 28
```

(continues on next page)

(continued from previous page)

```
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:28 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Trying to execute a command using wildcard expressions, for example using *, will cause an error:

```
In [6]: result = subprocess.run(['ls', '-ls', '*md'])
ls: cannot access *md: No such file or directory
```

To call commands in which wildcard expressions are used, you add **shell** argument and call a command:

```
In [7]: result = subprocess.run('ls -ls *md', shell=True)
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Another feature of `run()` If you try to run a ping command, for example, this aspect will be visible:

```
In [8]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=55.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=54.4 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 54.498/54.798/55.116/0.252 ms
```

Getting the result of a command execution

By default, run() function returns the result of a command execution to a standard output stream. If you want to get the result of command execution, add **stdout** argument with value **subprocess.PIPE**:

```
In [9]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE)
```

Now you can get the result of command executing in this way:

```
In [10]: print(result.stdout)
b'total 28\n4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_
↪console.md\n4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.
↪md\n4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions\n4 -rw-
↪r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md\n4 drwxr-xr-x 2 vagrant␣
↪vagrant 4096 Jun 16 05:11 useful_functions\n4 drwxr-xr-x 2 vagrant vagrant 4096␣
↪Jun 17 16:30 useful_modules\n4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35␣
↪version_control.md\n'
```

Note letter **b** before line. It means that module returned the output as a byte string. There are two options to translate it into unicode:

- decode received string
- specify encoding argument

Example with decode:

```
In [11]: print(result.stdout.decode('utf-8'))
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:30 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Example with encoding:

```
In [12]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE, encoding=
↪'utf-8')

In [13]: print(result.stdout)
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:31 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Output disabling

Sometimes it is enough to get only return code and need to disable output of execution result on standard output stream. This can be done by passing to run() function the **stdout** argument with value **subprocess.DEVNULL**:

```
In [14]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.DEVNULL)

In [15]: print(result.stdout)
None

In [16]: print(result.returncode)
0
```

Working with standard error stream

If command was executed with error or failed, the output of command will fall on standard error stream.

This can be obtained in the same way as the standard output stream:

```
In [17]: result = subprocess.run(['ping', '-c', '3', '-n', 'a'],␣
↪stderr=subprocess.PIPE, encoding='utf-8')
```

Now result.stdout has empty string and result.stderr has standard output stream:

```
In [18]: print(result.stdout)
None

In [19]: print(result.stderr)
ping: unknown host a


In [20]: print(result.returncode)
2
```

Examples of module use

Example of subprocess module use (subprocess_run_basic.py file):

```
import subprocess

reply = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])
```

(continues on next page)

(continued from previous page)

```
if reply.returncode == 0:
    print('Alive')
else:
    print('Unreachable')
```

The result will be:

```
$ python subprocess_run_basic.py
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=54.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 53.962/54.145/54.461/0.293 ms
Alive
```

That is, the result of command execution is printed to standard output stream.

Function ping_ip() checks the availability of IP address and returns True and **stdout** if address is available, or False and **stderr** if address is not available (subprocess_ping_function.py file):

```
import subprocess


def ping_ip(ip_address):
    """
    Ping IP address and return tuple:
    On success:
        * True
        * command output (stdout)
    On failure:
        * False
        * error output (stderr)
    """
    reply = subprocess.run(['ping', '-c', '3', '-n', ip_address],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
                           encoding='utf-8')
    if reply.returncode == 0:
        return True, reply.stdout
    else:
```

(continues on next page)

(continued from previous page)

```
        return False, reply.stderr

print(ping_ip('8.8.8.8'))
print(ping_ip('a'))
```

The result will be:

```
$ python subprocess_ping_function.py
(True, 'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8: icmp_
↪seq=1 ttl=43 time=63.8 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=55.6␣
↪ms\n64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=55.9 ms\n\n--- 8.8.8.8 ping␣
↪statistics ---\n3 packets transmitted, 3 received, 0% packet loss, time␣
↪2003ms\nrtt min/avg/max/mdev = 55.643/58.492/63.852/3.802 ms\n')
(False, 'ping: unknown host a\n')
```

Based on this function you can make a function that will check list of IP addresses and return as a result two lists: reachable and unreachable addresses.

Note: You will find it in tasks of section

If number of IP addresses to check is large, you can use threading or multiprocessing modules to speed up verification.

os

Module os allows working with filesystem, environment and managing processes.

This subsection addresses only several useful features. For a more complete description of capabilities of module please refer to documentation or article on Pymotw.

Module os allows you to create directories:

```
In [1]: import os

In [2]: os.mkdir('test')

In [3]: ls -ls
total 0
0 drwxr-xr-x  2 nata  nata  68 Jan 23 18:58 test/
```

In addition, module contains relevant existence checks. For example, if you try to re-create a directory, an error will occur:

```
In [4]: os.mkdir('test')
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
<ipython-input-4-cbf3b897c095> in <module>()
----> 1 os.mkdir('test')

FileExistsError: [Errno 17] File exists: 'test'
```

In this case, testing with `os.path.exists` is useful:

```
In [5]: os.path.exists('test')
Out[5]: True

In [6]: if not os.path.exists('test'):
   ...:     os.mkdir('test')
   ...:
```

Method `listdir` allows you to view the content of directory:

```
In [7]: os.listdir('.')
Out[7]: ['cover3.png', 'dir2', 'dir3', 'README.txt', 'test']
```

By checking `os.path.isdir` and `os.path.isfile` you can get a separate list of files and list of directories:

```
In [8]: dirs = [d for d in os.listdir('.') if os.path.isdir(d)]

In [9]: dirs
Out[9]: ['dir2', 'dir3', 'test']

In [10]: files = [f for f in os.listdir('.') if os.path.isfile(f)]

In [11]: files
Out[11]: ['cover3.png', 'README.txt']
```

Also in module there are separate methods for working with paths:

```
In [12]: file = 'Programming/PyNEng/book/25_additional_info/README.md'

In [13]: os.path.basename(file)
Out[13]: 'README.md'

In [14]: os.path.dirname(file)
Out[14]: 'Programming/PyNEng/book/25_additional_info'
```

(continues on next page)

(continued from previous page)

```
In [15]: os.path.split(file)
Out[15]: ('Programming/PyNEng/book/25_additional_info', 'README.md')
```

ipaddress

Module ipaddress simplifies work with IP addresses.

Note: Since Python 3.3, ipaddress module is part of standard Python library.

ipaddress.ip_address

Function ipaddress.ip_address allows to create an IPv4Address or IPv6Address respectively:

```
In [1]: import ipaddress

In [2]: ipv4 = ipaddress.ip_address('10.0.1.1')

In [3]: ipv4
Out[3]: IPv4Address('10.0.1.1')

In [4]: print(ipv4)
10.0.1.1
```

Object has several methods and attributes:

```
In [5]: ipv4.
ipv4.compressed       ipv4.is_loopback      ipv4.is_unspecified   ipv4.version
ipv4.exploded         ipv4.is_multicast     ipv4.max_prefixlen
ipv4.is_global        ipv4.is_private       ipv4.packed
ipv4.is_link_local    ipv4.is_reserved      ipv4.reverse_pointer
```

With is_ attributes you can check to what range the address belongs to:

```
In [6]: ipv4.is_loopback
Out[6]: False

In [7]: ipv4.is_multicast
Out[7]: False

```

(continues on next page)

(continued from previous page)

```
In [8]: ipv4.is_reserved
Out[8]: False

In [9]: ipv4.is_private
Out[9]: True
```

Different operations can be performed with received objects:

```
In [10]: ip1 = ipaddress.ip_address('10.0.1.1')

In [11]: ip2 = ipaddress.ip_address('10.0.2.1')

In [12]: ip1 > ip2
Out[12]: False

In [13]: ip2 > ip1
Out[13]: True

In [14]: ip1 == ip2
Out[14]: False

In [15]: ip1 != ip2
Out[15]: True

In [16]: str(ip1)
Out[16]: '10.0.1.1'

In [17]: int(ip1)
Out[17]: 167772417

In [18]: ip1 + 5
Out[18]: IPv4Address('10.0.1.6')

In [19]: ip1 - 5
Out[19]: IPv4Address('10.0.0.252')
```

ipaddress.ip_network

ipaddress.ip_network function allows you to create an object that describes the network (IPv4 or IPv6):

```
In [20]: subnet1 = ipaddress.ip_network('80.0.1.0/28')
```

As with an address, a network has various attributes and methods:

```
In [21]: subnet1.broadcast_address
Out[21]: IPv4Address('80.0.1.15')

In [22]: subnet1.with_netmask
Out[22]: '80.0.1.0/255.255.255.240'

In [23]: subnet1.with_hostmask
Out[23]: '80.0.1.0/0.0.0.15'

In [24]: subnet1.prefixlen
Out[24]: 28

In [25]: subnet1.num_addresses
Out[25]: 16
```

Method `hosts` returns generator, so to view all hosts you should apply list() function:

```
In [26]: list(subnet1.hosts())
Out[26]:
[IPv4Address('80.0.1.1'),
 IPv4Address('80.0.1.2'),
 IPv4Address('80.0.1.3'),
 IPv4Address('80.0.1.4'),
 IPv4Address('80.0.1.5'),
 IPv4Address('80.0.1.6'),
 IPv4Address('80.0.1.7'),
 IPv4Address('80.0.1.8'),
 IPv4Address('80.0.1.9'),
 IPv4Address('80.0.1.10'),
 IPv4Address('80.0.1.11'),
 IPv4Address('80.0.1.12'),
 IPv4Address('80.0.1.13'),
 IPv4Address('80.0.1.14')]
```

Method `subnets` allows dividing network (subnetting). By default, it splits network into two subnets:

```
In [27]: list(subnet1.subnets())
Out[27]: [IPv4Network('80.0.1.0/29'), IPv4Network(u'80.0.1.8/29')]
```

Prefixlen_diff parameter allows you to specify the number of bits for subnets:

```
In [28]: list(subnet1.subnets(prefixlen_diff=2))
Out[28]:
```

(continues on next page)

(continued from previous page)

```
[IPv4Network('80.0.1.0/30'),
 IPv4Network('80.0.1.4/30'),
 IPv4Network('80.0.1.8/30'),
 IPv4Network('80.0.1.12/30')]
```

With new_prefix parameter you can specify which mask should be configured:

```
In [29]: list(subnet1.subnets(new_prefix=30))
Out[29]:
[IPv4Network('80.0.1.0/30'),
 IPv4Network('80.0.1.4/30'),
 IPv4Network('80.0.1.8/30'),
 IPv4Network('80.0.1.12/30')]

In [30]: list(subnet1.subnets(new_prefix=29))
Out[30]: [IPv4Network('80.0.1.0/29'), IPv4Network(u'80.0.1.8/29')]
```

IP addresses of network can be iterated in a loop:

```
In [31]: for ip in subnet1:
   ....:     print(ip)
   ....:
80.0.1.0
80.0.1.1
80.0.1.2
80.0.1.3
80.0.1.4
80.0.1.5
80.0.1.6
80.0.1.7
80.0.1.8
80.0.1.9
80.0.1.10
80.0.1.11
80.0.1.12
80.0.1.13
80.0.1.14
80.0.1.15
```

And it is possible to get a specific address:

```
In [32]: subnet1[0]
Out[32]: IPv4Address('80.0.1.0')
```

(continues on next page)

(continued from previous page)

```
In [33]: subnet1[5]
Out[33]: IPv4Address('80.0.1.5')
```

This way you can check if IP address is in the network:

```
In [34]: ip1 = ipaddress.ip_address('80.0.1.3')

In [35]: ip1 in subnet1
Out[35]: True
```

ipaddress.ip_interface

The `ipaddress.ip_interface` function allows you to create an IPv4Interface or IPv6Interface object, respectively:

```
In [36]: int1 = ipaddress.ip_interface('10.0.1.1/24')
```

Using methods of IPv4Interface object you can get an address, mask or interface network:

```
In [37]: int1.ip
Out[37]: IPv4Address('10.0.1.1')

In [38]: int1.network
Out[38]: IPv4Network('10.0.1.0/24')

In [39]: int1.netmask
Out[39]: IPv4Address('255.255.255.0')
```

Example of module usage

Since module has built-in address correctness checks, you can use them, for example, to check whether an address is a network or host address:

```
In [40]: IP1 = '10.0.1.1/24'

In [41]: IP2 = '10.0.1.0/24'

In [42]: def check_if_ip_is_network(ip_address):
   ....:     try:
   ....:         ipaddress.ip_network(ip_address)
   ....:         return True
```

(continues on next page)

(continued from previous page)

```
   ....:     except ValueError:
   ....:         return False
   ....:

In [43]: check_if_ip_is_network(IP1)
Out[43]: False

In [44]: check_if_ip_is_network(IP2)
Out[44]: True
```

tabulate

tabulate is a module that allows you to display table data beautifully. It is not part of standard Python library, so tabulate needs to be installed:

```
pip install tabulate
```

Module supports such tabular data types as:

- list of lists (in general case - iterable of iterables)
- dictionary list (or any other iterable object with dictionaries). Keys are used as column names
- dictionary with iterable objects. Keys are used as column names

Function tabulate() is used to generate table:

```
In [1]: from tabulate import tabulate

In [2]: sh_ip_int_br = [('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
   ...:  ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
   ...:  ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
   ...:  ('Loopback0', '10.1.1.1', 'up', 'up'),
   ...:  ('Loopback100', '100.0.0.1', 'up', 'up')]
   ...:

In [4]: print(tabulate(sh_ip_int_br))
---------------  ---------  --  --
FastEthernet0/0  15.0.15.1  up  up
FastEthernet0/1  10.0.12.1  up  up
FastEthernet0/2  10.0.13.1  up  up
Loopback0        10.1.1.1   up  up
Loopback100      100.0.0.1  up  up
---------------  ---------  --  --
```

headers

Parameter **headers** allows you to pass an additional argument that specifies column names:

```
In [8]: columns = ['Interface', 'IP', 'Status', 'Protocol']

In [9]: print(tabulate(sh_ip_int_br, headers=columns))
Interface        IP         Status    Protocol
---------------  ---------  --------  ----------
FastEthernet0/0  15.0.15.1  up        up
FastEthernet0/1  10.0.12.1  up        up
FastEthernet0/2  10.0.13.1  up        up
Loopback0        10.1.1.1   up        up
Loopback100      100.0.0.1  up        up
```

Quite often, the first data set is headers. Then it is enough to specify headers equal to “firstrow”:

```
In [18]: data
Out[18]:
[('Interface', 'IP', 'Status', 'Protocol'),
 ('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]

In [20]: print(tabulate(data, headers='firstrow'))
Interface        IP         Status    Protocol
---------------  ---------  --------  ----------
FastEthernet0/0  15.0.15.1  up        up
FastEthernet0/1  10.0.12.1  up        up
FastEthernet0/2  10.0.13.1  up        up
Loopback0        10.1.1.1   up        up
Loopback100      100.0.0.1  up        up
```

If data is in the form of a list of dictionaries, you should specify headers equal to “keys”:

```
In [22]: list_of_dict
Out[22]:
[{'IP': '15.0.15.1',
  'Interface': 'FastEthernet0/0',
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '10.0.12.1',
  'Interface': 'FastEthernet0/1',
```

(continues on next page)

(continued from previous page)

```
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '10.0.13.1',
  'Interface': 'FastEthernet0/2',
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '10.1.1.1',
  'Interface': 'Loopback0',
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '100.0.0.1',
  'Interface': 'Loopback100',
  'Protocol': 'up',
  'Status': 'up'}]

In [23]: print(tabulate(list_of_dict, headers='keys'))
Interface        IP         Status    Protocol
---------------  ---------  --------  ----------
FastEthernet0/0  15.0.15.1  up        up
FastEthernet0/1  10.0.12.1  up        up
FastEthernet0/2  10.0.13.1  up        up
Loopback0        10.1.1.1   up        up
Loopback100      100.0.0.1  up        up
```

Dict with lists in values:

```
In [6]: vlans = {"sw1": [10, 20, 30, 40], "sw2": [1, 2, 10], "sw3": [1, 2, 3, 4,␣
↪5, 10, 11, 12]}

In [7]: print(tabulate(vlans, headers="keys"))
  sw1    sw2    sw3
-----  -----  -----
   10      1      1
   20      2      2
   30     10      3
   40             4
                  5
                 10
                 11
                 12
```

Table style

`tabulate` supports different table styles.

Table in Grid format:

```
In [24]: print(tabulate(list_of_dict, headers='keys', tablefmt="grid"))
+-----------------+-----------+----------+------------+
| Interface       | IP        | Status   | Protocol   |
+=================+===========+==========+============+
| FastEthernet0/0 | 15.0.15.1 | up       | up         |
+-----------------+-----------+----------+------------+
| FastEthernet0/1 | 10.0.12.1 | up       | up         |
+-----------------+-----------+----------+------------+
| FastEthernet0/2 | 10.0.13.1 | up       | up         |
+-----------------+-----------+----------+------------+
| Loopback0       | 10.1.1.1  | up       | up         |
+-----------------+-----------+----------+------------+
| Loopback100     | 100.0.0.1 | up       | up         |
+-----------------+-----------+----------+------------+
```

Table in Markdown format:

```
In [25]: print(tabulate(list_of_dict, headers='keys', tablefmt='pipe'))
| Interface       | IP        | Status   | Protocol   |
|:----------------|:----------|:---------|:-----------|
| FastEthernet0/0 | 15.0.15.1 | up       | up         |
| FastEthernet0/1 | 10.0.12.1 | up       | up         |
| FastEthernet0/2 | 10.0.13.1 | up       | up         |
| Loopback0       | 10.1.1.1  | up       | up         |
| Loopback100     | 100.0.0.1 | up       | up         |
```

Table in HTML format:

```
In [26]: print(tabulate(list_of_dict, headers='keys', tablefmt='html'))
<table>
<thead>
<tr><th>Interface      </th><th>IP       </th><th>Status  </th><th>Protocol  </th>
↪</tr>
</thead>
<tbody>
<tr><td>FastEthernet0/0</td><td>15.0.15.1</td><td>up      </td><td>up        </td>
↪</tr>
<tr><td>FastEthernet0/1</td><td>10.0.12.1</td><td>up      </td><td>up        </td>
↪</tr>
```

(continues on next page)

(continued from previous page)

```
<tr><td>FastEthernet0/2</td><td>10.0.13.1</td><td>up      </td><td>up        </td>
↪</tr>
<tr><td>Loopback0      </td><td>10.1.1.1 </td><td>up      </td><td>up        </td>
↪</tr>
<tr><td>Loopback100    </td><td>100.0.0.1</td><td>up      </td><td>up        </td>
↪</tr>
</tbody>
</table>
```

Alignment of columns

You can specify alignment for columns:

```
In [27]: print(tabulate(list_of_dict, headers='keys', tablefmt='pipe', stralign=
↪'center'))
|    Interface    |    IP     |  Status  |  Protocol  |
|:---------------:|:---------:|:--------:|:----------:|
| FastEthernet0/0 | 15.0.15.1 |    up    |     up     |
| FastEthernet0/1 | 10.0.12.1 |    up    |     up     |
| FastEthernet0/2 | 10.0.13.1 |    up    |     up     |
|    Loopback0    | 10.1.1.1  |    up    |     up     |
|   Loopback100   | 100.0.0.1 |    up    |     up     |
```

Note that not only columns are displayed centrally, but Markdown syntax has been changed accordingly.

Additional material

- tabulate documentation

Articles from author `tabulate`:

- Pretty printing tables in Python
- Tabulate 0.7.1 with LaTeX & MediaWiki tables

Stack Overflow:

- Printing Lists as Tabular Data. Note the answer - it contains other tabulate analogues.

pprint

Module `pprint` allows you to show Python objects beautifully. This saves the structure of object. You can use the result that produces `pprint` to create object. Module `pprint` is part of standard Python

library.

The simplest use of module is ``pprint`` function. For example, a dictionary with nested dictionaries is displayed as follows:

```
In [6]: london_co = {'r1': {'hostname': 'london_r1', 'location': '21 New Globe Wal
   ...: k', 'vendor': 'Cisco', 'model': '4451', 'IOS': '15.4', 'IP': '10.255.0.1'}
   ...: , 'r2': {'hostname': 'london_r2', 'location': '21 New Globe Walk', 'vendor
   ...: ': 'Cisco', 'model': '4451', 'IOS': '15.4', 'IP': '10.255.0.2'}, 'sw1': {'
   ...: hostname': 'london_sw1', 'location': '21 New Globe Walk', 'vendor': 'Cisco
   ...: ', 'model': '3850', 'IOS': '3.6.XE', 'IP': '10.255.0.101'}}
   ...:

In [7]: from pprint import pprint

In [8]: pprint(london_co)
{'r1': {'IOS': '15.4',
        'IP': '10.255.0.1',
        'hostname': 'london_r1',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'r2': {'IOS': '15.4',
        'IP': '10.255.0.2',
        'hostname': 'london_r2',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'sw1': {'IOS': '3.6.XE',
         'IP': '10.255.0.101',
         'hostname': 'london_sw1',
         'location': '21 New Globe Walk',
         'model': '3850',
         'vendor': 'Cisco'}}
```

List of lists:

```
In [13]: interfaces = [['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up
    ...: '], ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'], ['FastE
    ...: thernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]
    ...:

In [14]: pprint(interfaces)
[['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up'],
 ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'],
```

(continues on next page)

(continued from previous page)

```
 ['FastEthernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]
```

String:

```
In [18]: tunnel
Out[18]: '\ninterface Tunnel0\n ip address 10.10.10.1 255.255.255.0\n ip mtu␣
↪1416\n ip ospf hello-interval 5\n tunnel source FastEthernet1/0\n tunnel␣
↪protection ipsec profile DMVPN\n'

In [19]: pprint(tunnel)
('\n'
 'interface Tunnel0\n'
 ' ip address 10.10.10.1 255.255.255.0\n'
 ' ip mtu 1416\n'
 ' ip ospf hello-interval 5\n'
 ' tunnel source FastEthernet1/0\n'
 ' tunnel protection ipsec profile DMVPN\n')
```

Nesting restriction

Function `pprint` has an additional **depth** parameter that allows limiting the depth of data structure display.

For example, there's a dictionary:

```
In [3]: result = {
   ...:  'interface Tunnel0': [' ip unnumbered Loopback0',
   ...:   ' tunnel mode mpls traffic-eng',
   ...:   ' tunnel destination 10.2.2.2',
   ...:   ' tunnel mpls traffic-eng priority 7 7',
   ...:   ' tunnel mpls traffic-eng bandwidth 5000',
   ...:   ' tunnel mpls traffic-eng path-option 10 dynamic',
   ...:   ' no routing dynamic'],
   ...:  'ip access-list standard LDP': [' deny   10.0.0.0 0.0.255.255',
   ...:   ' permit 10.0.0.0 0.255.255.255'],
   ...:  'router bgp 100': {' address-family vpnv4': ['  neighbor 10.2.2.2 activat
   ...: e',
   ...:    '  neighbor 10.2.2.2 send-community both',
   ...:    '  exit-address-family'],
   ...:   ' bgp bestpath igp-metric ignore': [],
   ...:   ' bgp log-neighbor-changes': [],
   ...:   ' neighbor 10.2.2.2 next-hop-self': [],
   ...:   ' neighbor 10.2.2.2 remote-as 100': [],
```

(continues on next page)

(continued from previous page)

```
   ...:    ' neighbor 10.2.2.2 update-source Loopback0': [],
   ...:    ' neighbor 10.4.4.4 remote-as 40': []},
   ...:  'router ospf 1': [' mpls ldp autoconfig area 0',
   ...:    ' mpls traffic-eng router-id Loopback0',
   ...:    ' mpls traffic-eng area 0',
   ...:    ' network 10.0.0.0 0.255.255.255 area 0']}
   ...:
```

You can only display keys with depth equal to 1:

```
In [5]: pprint(result, depth=1)
{'interface Tunnel0': [...],
 'ip access-list standard LDP': [...],
 'router bgp 100': {...},
 'router ospf 1': [...]}
```

Hidden nesting levels are replaced with `...`.

If you specify a depth of 2, the next level is displayed:

```
In [6]: pprint(result, depth=2)
{'interface Tunnel0': [' ip unnumbered Loopback0',
                       ' tunnel mode mpls traffic-eng',
                       ' tunnel destination 10.2.2.2',
                       ' tunnel mpls traffic-eng priority 7 7',
                       ' tunnel mpls traffic-eng bandwidth 5000',
                       ' tunnel mpls traffic-eng path-option 10 dynamic',
                       ' no routing dynamic'],
 'ip access-list standard LDP': [' deny   10.0.0.0 0.0.255.255',
                                 ' permit 10.0.0.0 0.255.255.255'],
 'router bgp 100': {' address-family vpnv4': [...],
                    ' bgp bestpath igp-metric ignore': [],
                    ' bgp log-neighbor-changes': [],
                    ' neighbor 10.2.2.2 next-hop-self': [],
                    ' neighbor 10.2.2.2 remote-as 100': [],
                    ' neighbor 10.2.2.2 update-source Loopback0': [],
                    ' neighbor 10.4.4.4 remote-as 40': []},
 'router ospf 1': [' mpls ldp autoconfig area 0',
                   ' mpls traffic-eng router-id Loopback0',
                   ' mpls traffic-eng area 0',
                   ' network 10.0.0.0 0.255.255.255 area 0']}
```

pformat

pformat() is a function that displays the result as a string. It is convenient to use if you want to write a data structure into a file, for example to log.

```
In [15]: from pprint import pformat

In [16]: formatted_result = pformat(result)

In [17]: print(formatted_result)
{'interface Tunnel0': [' ip unnumbered Loopback0',
                       ' tunnel mode mpls traffic-eng',
                       ' tunnel destination 10.2.2.2',
                       ' tunnel mpls traffic-eng priority 7 7',
                       ' tunnel mpls traffic-eng bandwidth 5000',
                       ' tunnel mpls traffic-eng path-option 10 dynamic',
                       ' no routing dynamic'],
 'ip access-list standard LDP': [' deny   10.0.0.0 0.0.255.255',
                                 ' permit 10.0.0.0 0.255.255.255'],
 'router bgp 100': {' address-family vpnv4': ['  neighbor 10.2.2.2 activate',
                                              '  neighbor 10.2.2.2 '
                                              'send-community both',
                                              '  exit-address-family'],
                    ' bgp bestpath igp-metric ignore': [],
                    ' bgp log-neighbor-changes': [],
                    ' neighbor 10.2.2.2 next-hop-self': [],
                    ' neighbor 10.2.2.2 remote-as 100': [],
                    ' neighbor 10.2.2.2 update-source Loopback0': [],
                    ' neighbor 10.4.4.4 remote-as 40': []},
 'router ospf 1': [' mpls ldp autoconfig area 0',
                   ' mpls traffic-eng router-id Loopback0',
                   ' mpls traffic-eng area 0',
                   ' network 10.0.0.0 0.255.255.255 area 0']}
```

Additional material

Documentation:

- pprint — Data pretty printer
- PyMOTW. pprint — Pretty-Print Data Structures

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 12.1

Create a ping_ip_addresses function that checks if IP addresses are pingable.

The function expects a list of IP addresses as an argument.

The function must return a tuple with two lists:

- list of available IP addresses
- list of unavailable IP addresses

To check the availability of an IP address, use the ping command.

Restriction: All tasks must be done using the topics covered in this and previous chapters.

Task 12.2

The ping_ip_addresses function from task 12.1 only accepts a list of addresses, but it would be convenient to be able to specify addresses using a range, for example 192.168.100.1-10.

In this task, you need to create a function convert_ranges_to_ip_list that converts a list of IP addresses in different formats into a list where each IP address is listed separately.

The function expects as an argument a list containing IP addresses and/or ranges of IP addresses.

List items can be in the format:

- 10.1.1.1
- 10.1.1.1-10.1.1.10
- 10.1.1.1-10

If the address is specified as a range, the range must be expanded into individual addresses, including the last address in the range. To simplify the task, we can assume that only the last octet of the address changes in the range.

The function returns a list of IP addresses.

For example, if you pass the following list to the convert_ranges_to_ip_list function:

```
['8.8.4.4', '1.1.1.1-3', '172.21.41.128-172.21.41.132']
```

The function should return a list like this:

```
['8.8.4.4', '1.1.1.1', '1.1.1.2', '1.1.1.3', '172.21.41.128',
 '172.21.41.129', '172.21.41.130', '172.21.41.131', '172.21.41.132']
```

Task 12.3

Create a function print_ip_table that prints a table of available and unavailable IP addresses.

The function expects two lists as arguments:

- list of available IP addresses
- list of unavailable IP addresses

The result of the function is printing a table to the stdout:

```
Reachable    Unreachable
-----------  -------------
10.1.1.1     10.1.1.7
10.1.1.2     10.1.1.8
             10.1.1.9
```

13. Iterators, iterable and generators

This section discusses:

- iterable
- iterators
- generator expressions

Iterable

Iteration is a generic term that describes the procedure for taking elements of something in turn. In a more general sense, it is a sequence of instructions that is repeated a certain number of times or before the specified condition is fulfilled.

An iterable is an object that can return elements one at a time. It is also an object from which an iterator can be derived.

Examples of iterables:

- all sequences: list, string, tuple
- dicts
- files

In Python, the `iter` function is responsible for getting an iterator:

```
In [1]: lista = [1, 2, 3]

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

`iter` function will work on any object that has `__iter__` or `__getitem__` method. `__iter__` method returns an iterator. If this method is not available, `iter` function checks if there is `__getitem__` method that allows getting elements by index.

If method `__getitem__` is present an iterator is returned, which iterates through the elements using index (starting with 0). In practice, the use of `__getitem__` means that all sequence elements are iterable objects. For example, a list, a tuple, a string. Although these data types have `__iter__` method.

Iterators

Iterator is an object that returns its elements one at a time.

From Python point of view, it is any object that has __next__ method. This method returns the next item if any, or returns StopIteration exception when items are finished. In addition, iterator remembers which object it stopped at in the last iteration.

In Python, each iterator has __iter__ method - that is, every iterator is an iterable. This method simply returns iterator itself.

An example of creating an iterator from a list:

```
In [3]: numbers = [1, 2, 3]

In [4]: i = iter(numbers)
```

Now you can use next function that calls __next__ method to take the next element:

```
In [5]: next(i)
Out[5]: 1

In [6]: next(i)
Out[6]: 2

In [7]: next(i)
Out[7]: 3

In [8]: next(i)
---------------------------------------------------------------
StopIteration                Traceback (most recent call last)
<ipython-input-8-bed2471d02c1> in <module>()
----> 1 next(i)

StopIteration:
```

After elements are finished, StopIteration exception is raised.

Note: To make iterator to return elements again, it has to be re-created.

Similar actions are performed when loop **for** processes a list:

```
In [9]: for item in numbers:
   ...:     print(item)
   ...:
1
2
3
```

When we iterate over the list items, the `iter` function is first applied to the list to create the iterator, and then its `__next__` method is called until a `StopIteration` exception is raised.

Iterators are useful because they give elements one at a time. For example, when working with a file, it is useful that memory will not contain the whole file, but only one line of a file.

File as iterator

One of the most common examples of an iterator is a file (r1.txt):

```
!
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
!
```

If you open a file with the open function, an object is returned that represents the file:

```
In [10]: f = open('r1.txt')
```

This object is an iterator that can be verified by calling `__next__` method:

```
In [11]: f.__next__()
Out[11]: '!\n'

In [12]: f.__next__()
Out[12]: 'service timestamps debug datetime msec localtime show-timezone year\n'
```

You can also go through lines using **for** loop:

```
In [13]: for line in f:
    ...:     print(line.rstrip())
    ...:
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
```

(continues on next page)

(continued from previous page)

```
ip ssh version 2
!
```

When working with files, using a file as an iterator does not simply allow iterate file line by line - only one line is loaded into each iteration. This is very important when working with large files of thousands and hundreds of thousands of lines, such as log files.

Therefore, when working with files in Python, the most commonly used expression is:

```
In [14]: with open('r1.txt') as f:
    ...:     for line in f:
    ...:         print(line.rstrip())
    ...:
!
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
!
```

Generator

Generators are a special class of functions that allows you to easily create your own iterators. Unlike regular functions, a generator doesn't just return a value and exit, but returns an iterator that returns the elements one at a time.

Note: Generators are not covered in this book and are only mentioned here because they are a fairly straightforward way to create iterators. More about generators

---

A normal function exits if:

- execution reached the `return` statement
- function code is ended (this works as `return None` expression)
- an exception raised

After function execution is finished the control is returned and program execution goes further. All arguments that were passed to function like local variables, all of this is lost. Only the result that returned a function remains.

A function can return a list of elements, multiple objects or different results depending on arguments, but it always returns a single result.

Generator generates values. Values are then returned on demand and after return of one value a function-generator is suspended until the next value is requested. Between requests, generator retains its state.

Python allows generators to be created in two ways:

- generator expression
- generator function

generator expression

Generator expression uses the same syntax as a list comprehensions, but returns iterator, not list (note the parentheses instead of the square brackets):

```
In [1]: genexpr = (x**2 for x in range(10000))

In [2]: genexpr
Out[2]: <generator object <genexpr> at 0xb571ec8c>

In [3]: next(genexpr)
Out[3]: 0

In [4]: next(genexpr)
Out[4]: 1

In [5]: next(genexpr)
Out[5]: 4
```

It is useful when working with a large iterable object or infinite iterator.

Further reading

Documentation Python:

- Sequence types
- Iterator types
- Functional Programming HOWTO

Articles:

- Iterables vs. Iterators vs. Generators

5

III. Regular expressions

A regular expression is a sequence of ordinary and special characters. This sequence specifies a template that is later used to find search pattern.

When working with network equipment, regular expressions can be used, for example, to:

- retrieve information from show command output
- select a portion of lines from show command output that matches the template
- check whether there are certain settings in configuration

A few examples are:

- After processing the output of show version command, you can collect information about OS version and uptime.
- get from log file the lines that correspond to the template.
- get from configuration those interfaces that do not have a description

In addition, in network equipment the regular expressions can be used to filter the output of any show commands.

In general, use of regular expressions will involve getting part of a text out of a large output. But that's not the only thing they can be used for. For example, regular expressions can be used to perform string replacements or for dividing a string.

These areas of use overlap with methods that apply to strings. And if problemi is clear and simple to solve with string methods, it is better to use them. This code will be easier to understand and, in addition, string methods work faster.

But string methods may not solve all problems or may make problem much harder to solve. Regular expressions can help in this case.

14. Regular expression (regex) syntax

Regular expression syntax

Python uses re module to work with regular expressions (regex). To get started with regular expressions, you need to import re module.

This section will use search function for all examples. And in the next chapter, the rest of functions of re module will be covered.

Syntax of search function is:

```
match = re.search(pattern, string, flags=0)
```

Function search has three parameters:

- pattern - regular expression
- string - string in which search pattern is searched
- flags - change regex behavior (covered in next chapter)

If a match is found, function will return special object Match. If there is no match, function will return None.

Important distinction of search function is that it only looks for a first match. That is, if there are several substrings in a line that correspond to a regex, search will return only the first match found.

The simplest example of a regex is a substring:

```
In [1]: import re

In [2]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [3]: match = re.search('MTU', int_line)
```

In this example:

- first import module re
- then goes an example of string int_line
- in line 3 a search pattern is passed to search function plus string int_line in which the match is searched

In this case we are simply looking for whether there is 'MTU' substring in string int_line. If it exists, match variable will contain a special Match object:

```
In [4]: print(match)
<_sre.SRE_Match object; span=(2, 5), match='MTU'>
```

Match object has several methods that allow to get different information about received match. For example, group method shows that string matches an expression described.

In this case, it's just a 'MTU' substring:

```
In [5]: match.group()
Out[5]: 'MTU'
```

If there was no match, match variable will have None value:

```
In [6]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [7]: match = re.search('MU', int_line)

In [8]: print(match)
None
```

The full potential of regular expressions is revealed when using special characters. For example, symbol \d means a digit, + means repetition of previous symbol one or more times. If you combine them \d+, you get an expression that means one or more digits.

Using this expression, you can get the part of string that describes bandwidth:

```
In [9]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [10]: match = re.search('BW \d+', int_line)

In [11]: match.group()
Out[11]: 'BW 10000'
```

Regular expressions are particularly useful in getting certain substrings from a string. For example, it is necessary to get VLAN, MAC and ports from the output of such log message:

```
In [12]: log2 = 'Oct  3 12:49:15.941: %SW_MATM-4-MACFLAP_NOTIF: Host f04d.a206.
↪7fd6 in vlan 1 is flapping between port Gi0/5 and port Gi0/16'
```

This can be done with regex:

```
In [13]: re.search('Host (\S+) in vlan (\d+) is flapping between port (\S+) and␣
↪port (\S+)', log2).groups()
Out[13]: ('f04d.a206.7fd6', '1', 'Gi0/5', 'Gi0/16')
```

Method group returns only those parts of original string that are in parentheses. Thus, by placing a part of expression in parentheses, you can specify which parts of the line you want to remember.

Expression \d+ has been used before - it describes one or more digits. And expression \S+ describes all characters except whitespace (space, tab, etc.).

The following subsections deal with special characters that are used in regular expressions.

Note: If you know what special characters mean in regular expressions, you can skip the following subsection and immediately switch to subsection about module re.

Character sets

Python has special designations for character sets:

- \d - any digit
- \D - any non-numeric value
- \s - whitespace character
- \S - all except whitespace characters
- \w - any letter, digit or underline character
- \W - all except letter, digit or underline character

Note: These are not all character sets that support Python. See documentation for details.

Character sets allow you to write shorter expressions without having to list all necessary characters. For example, get time from log file string:

```
In [1]: log = '*Jul  7 06:15:18.695: %LINEPROTO-5-UPDOWN: Line protocol on␣
↪Interface Ethernet0/3, changed state to down'

In [2]: re.search('\d\d:\d\d:\d\d', log).group()
Out[2]: '06:15:18'
```

Expression \d\d:\d\d:\d\d describes 3 pairs of numbers separated by colons.

Getting MAC address from log message:

```
In [3]: log2 = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [4]: re.search('\w\w\w\w\.\w\w\w\w\.\w\w\w\w', log2).group()
Out[4]: 'f03a.b216.7ad7'
```

Expression \w\w\w\w\.\w\w\w\w\.\w\w\w\w describes 12 letters or digits that are divided into three groups of four characters and separated by dot.

Symbol groups are very convenient, but for now it is necessary to manually specify a character repetition. The following subsection covers repetition symbols which will simplify description of expressions.

Repeating characters

- regex+ - one or more repetitions of preceding element
- regex* - zero or more repetitions of preceding element
- regex? – zero or one repetition of preceding element
- regex{n} - exactly n repetitions of preceding element
- regex{n,m} - from n to m repetitions of preceding element
- regex{n,} - n or more repetitions of preceding element

+

Plus indicates that the previous expression can be repeated as many times as you like, but at least once. For example, here the repetition refers to letter 'a':

```
In [1]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [2]: re.search('a+', line).group()
Out[2]: 'aa'
```

And in this expression, string 'a1' is repeated:

```
In [3]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [4]: re.search('(a1)+', line).group()
Out[4]: 'a1a1'
```

Expresson (a1)+ uses parentheses to specify that repetition is related to sequence of symbols 'a1'.

IP address can be described by \d+\.\d+\.\d+\.\d+. Plus is used to indicate that there can be several digits. Expression \. is required because the dot is a special symbol (it denotes any symbol). And in order to indicate that we are interested in a dot as a symbol, you have to screen it - put a backslash in front of a dot.

Using this expression, you can get an IP address from sh_ip_int_br string:

```
In [5]: sh_ip_int_br = 'Ethernet0/1    192.168.200.1   YES NVRAM  up          up'
```

(continues on next page)

(continued from previous page)

```
In [6]: re.search('\d+\.\d+\.\d+\.\d+', sh_ip_int_br).group()
Out[6]: '192.168.200.1'
```

Another example of an expression: \d+\s+\S+ - describes a string which has digits first, then whitespace characters, and then non-whitespace characters (all except space, tab, and other similar characters). Using it you can get VLAN and MAC address from string:

```
In [7]: line = '1500     aab1.a1a1.a5d3    FastEthernet0/1'

In [8]: re.search('\d+\s+\S+', line).group()
Out[8]: '1500     aab1.a1a1.a5d3'
```

*

Asterisk indicates that the previous expression can be repeated 0 or more times. For example, if an asterisk stands after a symbol, it means a repetition of that symbol.

Expression ba* means b and then zero or more repetitions of a:

```
In [9]: line = '100     a011.baaa.a5d3    FastEthernet0/1'

In [10]: re.search('ba*', line).group()
Out[10]: 'baaa'
```

If b occurs in line before baaa, then b will match:

```
In [11]: line = '100     ab11.baaa.a5d3    FastEthernet0/1'

In [12]: re.search('ba*', line).group()
Out[12]: 'b'
```

Suppose you write a regex that describes email addresses in two formats: user@example.com and user.test@example.com. That is, the left side of address can have either one word or two words separated by a dot.

The first version is an example of email without a dot:

```
In [13]: email1 = 'user1@gmail.com'
```

This address can be described by \w+@\w+\.\w+:

```
In [14]: re.search('\w+@\w+\.\w+', email1).group()
Out[14]: 'user1@gmail.com'
```

But such an expression is not suitable for an email address with a dot:

```
In [15]: email2 = 'user2.test@gmail.com'

In [16]: re.search('\w+@\w+\.\w+', email2).group()
Out[16]: 'test@gmail.com'
```

Regex for email with a dot:

```
In [17]: re.search('\w+\.\w+@\w+\.\w+', email2).group()
Out[17]: 'user2.test@gmail.com'
```

To describe both email, you have to specify that the dot is optional:

```
'\w+\.*\w+@\w+\.\w+'
```

This regex describes both options:

```
In [18]: email1 = 'user1@gmail.com'

In [19]: email2 = 'user2.test@gmail.com'

In [20]: re.search('\w+\.*\w+@\w+\.\w+', email1).group()
Out[20]: 'user1@gmail.com'

In [21]: re.search('\w+\.*\w+@\w+\.\w+', email2).group()
Out[21]: 'user2.test@gmail.com'
```

?

In the last example, regex indicates that the dot is optional, but at the same time determines that it can appear many times.

In this situation, it is more logical to use a question mark. It denotes zero or one repetition of a preceding expression or symbol. Now regex looks like `\w+\.?\w+@\w+\.\w+`:

```
In [22]: mail_log = ['Jun 18 14:10:35 client-ip=154.10.180.10 from=user1@gmail.
↪com, size=551',
   ...:              'Jun 18 14:11:05 client-ip=150.10.180.10 from=user2.
↪test@gmail.com, size=768']

In [23]: for message in mail_log:
     ...:     match = re.search('\w+\.?\w+@\w+\.\w+', message)
     ...:     if match:
     ...:         print("Found email: ", match.group())
     ...:
```

(continues on next page)

(continued from previous page)

```
Found email:  user1@gmail.com
Found email:  user2.test@gmail.com
```

{n}

You can set how many times the previous expression should be repeated with curly braces.

For example, expression \w{4}\.\w{4}\.\w{4} describes 12 letters or digits that are divided into three groups of four characters and separated by dot. This way you can get a MAC address:

```
In [24]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [25]: re.search('\w{4}\.\w{4}\.\w{4}', line).group()
Out[25]: 'aab1.a1a1.a5d3'
```

You can specify a repetition range in curly braces. For example, try to get all VLAN numbers from string mac_table:

```
In [26]: mac_table = '''
    ...: sw1#sh mac address-table
    ...:           Mac Address Table
    ...: ------------------------------------------
    ...:
    ...: Vlan    Mac Address       Type        Ports
    ...: ----    -----------       --------    -----
    ...:  100    a1b2.ac10.7000    DYNAMIC     Gi0/1
    ...:  200    a0d4.cb20.7000    DYNAMIC     Gi0/2
    ...:  300    acb4.cd30.7000    DYNAMIC     Gi0/3
    ...: 1100    a2bb.ec40.7000    DYNAMIC     Gi0/4
    ...:  500    aa4b.c550.7000    DYNAMIC     Gi0/5
    ...: 1200    a1bb.1c60.7000    DYNAMIC     Gi0/6
    ...: 1300    aa0b.cc70.7000    DYNAMIC     Gi0/7
    ...: '''
```

Since search only looks for the first match, expression \d{1,4} will have VLAN number:

```
In [27]: for line in mac_table.split('\n'):
    ...:     match = re.search('\d{1,4}', line)
    ...:     if match:
    ...:         print('VLAN: ', match.group())
    ...:
VLAN:  1
VLAN:  100
```

(continues on next page)

(continued from previous page)

```
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

Expression `\d{1,4}` describes one to four digits.

Note that the output of command from equipment does not have a VLAN with number 1. Regex got a number 1 from somewhere. Number 1 was in the output from hostname in line `sw1#sh mac address-table`.

To correct this, it suffices to complete an expression and indicate that at least one space must follow the numbers:

```
In [28]: for line in mac_table.split('\n'):
    ...:     match = re.search('\d{1,4} +', line)
    ...:     if match:
    ...:         print('VLAN: ', match.group())
    ...:
VLAN:  100
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

Special symbols

- `.` - any character except new line character
- `^` - beginning of line
- `$` - end of line
- `[abc]` - any symbol in square brackets
- `[^abc]` - any symbol except those in square brackets
- `a|b` - element a or b
- `(regex)` - expression is treated as one element. In addition, substring that matches an expression is memorized

.

Dot represents any symbol. Most often, a dot is used with repetition symbols + and * to indicate that any character can be found between certain expressions.

For example, using expression `Interface.+Port ID.+` you can describe a line with interfaces in the output "sh cdp neighbors detail":

```
In [1]: cdp = '''
   ...: SW1#show cdp neighbors detail
   ...: -------------------------
   ...: Device ID: SW2
   ...: Entry address(es):
   ...:   IP address: 10.1.1.2
   ...: Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
   ...: Interface: GigabitEthernet1/0/16,  Port ID (outgoing port):␣
↪GigabitEthernet0/1
   ...: Holdtime : 164 sec
   ...: '''

In [2]: re.search('Interface.+Port ID.+', cdp).group()
Out[2]: 'Interface: GigabitEthernet1/0/16,  Port ID (outgoing port):␣
↪GigabitEthernet0/1'
```

The result was only one string as the dot represents any character except line feed character. In addition, repetition characters + and * by default capture the longest string possible. This aspect is addressed in subsection "Greedy qualifiers".

^

Character ^ means the beginning of line. Expression ^\d+ corresponds to substring:

```
In [3]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [4]: re.search('^\d+', line).group()
Out[4]: '100'
```

Characters from beginning of line to pound sign (including pound):

```
In [5]: prompt = 'SW1#show cdp neighbors detail'

In [6]: re.search('^.+#', prompt).group()
Out[6]: 'SW1#'
```

$

Symbol $ represents the end of a line.

Expression \S+$ describes any characters except whitespace at the end of line:

```
In [7]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [8]: re.search('\S+$', line).group()
Out[8]: 'FastEthernet0/1'
```

[]

Symbols that are listed in square brackets mean that any of these symbols will be a match. Thus, different registers can be described:

```
In [9]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [10]: re.search('[Ff]ast', line).group()
Out[10]: 'Fast'

In [11]: re.search('[Ff]ast[Ee]thernet', line).group()
Out[11]: 'FastEthernet'
```

Using square brackets, you can specify which characters may meet at a specific position. For example, expression ^.+[>#] describes characters from the beginning of a line to # or > sign (including them). This expression can be used to get the name of device:

```
In [12]: commands = ['SW1#show cdp neighbors detail',
    ...:             'SW1>sh ip int br',
    ...:             'r1-london-core# sh ip route']
    ...:

In [13]: for line in commands:
    ...:     match = re.search('^.+[>#]', line)
    ...:     if match:
    ...:         print(match.group())
    ...:
SW1#
SW1>
r1-london-core#
```

You can specify character ranges in square brackets. For example, any number from 0 to 9:

```
In [14]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [15]: re.search('[0-9]+', line).group()
Out[15]: '100'
```

Letters:

```
In [16]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [17]: re.search('[a-z]+', line).group()
Out[17]: 'aa'

In [18]: re.search('[A-Z]+', line).group()
Out[18]: 'F'
```

Several ranges may be indicated in square brackets:

```
In [19]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [20]: re.search('[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+', line).group()
Out[20]: 'aa12.35fe.a5d3'
```

Expression `[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+` describes three groups of symbols separated by a dot. Characters in each group can be letters a-f or digits 0-9. This expression describes MAC address.

Another feature of square brackets is that the special symbols within square brackets lose their special meaning and are simply a symbol. For example, a dot inside square brackets will denote a dot, not any symbol.

Expression `[a-f0-9]+[./][a-f0-9]+` describes three groups of symbols:

1. letters a-f or digits 0-9
2. dot or slash
3. letters a-f or digits 0-9

For `line` string the match will be a such substring:

```
In [21]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [22]: re.search('[a-f0-9]+[./][a-f0-9]+', line).group()
Out[22]: 'aa12.35fe'
```

If first symbol in square brackets is ^, match will be any symbol except those in brackets.

```
In [23]: line = 'FastEthernet0/0    15.0.15.1       YES manual up         up'

In [24]: re.search('[^a-zA-Z]+', line).group()
Out[24]: '0/0    15.0.15.1       '
```

In this case, expression describes everything except letters.

|

Pipe symbol works like 'or':

```
In [25]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [26]: re.search('Fast|0/1', line).group()
Out[26]: 'Fast'
```

Note how | works - Fast и 0/1 are treated as an whole expression. So in the end, expression means that we're looking for Fast or 0/1.

()

Parentheses are used to group expressions. As in mathematical expressions, parentheses can be used to indicate which elements the operation is applied to.

For example, expression `[0-9]([a-f]|[0-9])[0-9]` describes three characters: digit, then a letter or digit and digit:

```
In [27]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [28]: re.search('[0-9]([a-f]|[0-9])[0-9]', line).group()
Out[28]: '100'
```

Parentheses allow to indicate which expression is a one entity. This is particularly useful when using repetition symbols:

```
In [29]: line = 'FastEthernet0/0    15.0.15.1       YES manual up         up'

In [30]: re.search('([0-9]+\.)+[0-9]+', line).group()
Out[30]: '15.0.15.1'
```

Parentheses not only allow you to group expressions. String that matches expression in parentheses is memorized. It can be obtained separately by special methods `groups` and `group(n)`. This is covered in subsection "Grouping".

Greedy qualifiers

By default, *, +, and ? qualifiers are all greedy - they match as much text as possible.

An example of greedy behavior:

```
In [1]: import re

In [2]: line = '<text line> some text>'

In [3]: match = re.search('<.*>', line)

In [4]: match.group()
Out[4]: '<text line> some text>'
```

In this case, expression captured maximum possible piece of symbols contained in <>. If greedy behavior need to be disabled, just add a question mark after the repetition symbols:

```
In [5]: line = '<text line> some text>'

In [6]: match = re.search('<.*?>', line)

In [7]: match.group()
Out[7]: '<text line>'
```

But greed is often useful. For example, without turning off greed of the last plus, expression \d+\s+\S+ describes line:

```
In [8]: line = '1500     aab1.a1a1.a5d3    FastEthernet0/1'

In [9]: re.search('\d+\s+\S+', line).group()
Out[9]: '1500     aab1.a1a1.a5d3'
```

Symbol \S denotes everything except whitespace characters. Therefore, expression \S+ with greedy repetition symbol describes maximum long string until the first whitespace character. In this case up to the first space.

If greed is disabled, the result is:

```
In [10]: re.search('\d+\s+\S+?', line).group()
Out[10]: '1500     a'
```

Grouping

Grouping indicates that sequence of symbols should be considered as a one. However, this is not the only advantage of grouping. In addition, by use of groups you can get only a certain portion of string that has been described by expression.

For example, from a log file you should select strings in which "%SW_MATM-4-MACFLAP_NOTIF" match occur and then from each such string get MAC address, VLAN and interfaces. In this case, regex has to describe a string and all parts of string to be remembered are placed in parentheses.

For example, from the log file, you need to select the lines that contain "%SW_MATM-4-MACFLAP_NOTIF", and then get the MAC address, VLAN and interfaces from each such line. In this case, the regex not only describes the string, but also indicates all parts of the string to be returned in parentheses.

Python has two options for using groups:

- Numbered groups
- Named groups

Numbered groups

Group is defined by placing expression in parentheses `()`.

Inside expression, group are numbered from left to right starting with 1. Groups can then be selected by numbers to get text that corresponds to group expression.

Example of groups use:

```
In [8]: line = "FastEthernet0/1       10.0.12.1   YES manual up          up"
In [9]: match = re.search('(\S+)\s+([\w.]+)\s+.*', line)
```

In this example, two groups are specified:

- first group - any characters other than whitespaces
- second group - any letter or digit (symbol \w) or dot

The second group could be described as the first. Other version is just for example.

You can now access a group by number. Group 0 is a string that corresponds to the entire match:

```
In [10]: match.group(0)
Out[10]: 'FastEthernet0/1            10.0.12.1       YES manual up                  ␣
↪    up'

In [11]: match.group(1)
Out[11]: 'FastEthernet0/1'
```

(continues on next page)

(continued from previous page)

```
In [12]: match.group(2)
Out[12]: '10.0.12.1'
```

If necessary, you can list several group numbers:

```
In [13]: match.group(1, 2)
Out[13]: ('FastEthernet0/1', '10.0.12.1')

In [14]: match.group(2, 1, 2)
Out[14]: ('10.0.12.1', 'FastEthernet0/1', '10.0.12.1')
```

Starting with Python 3.6, groups can be accessed as follows:

```
In [15]: match[0]
Out[15]: 'FastEthernet0/1            10.0.12.1       YES manual up
↪    up'

In [16]: match[1]
Out[16]: 'FastEthernet0/1'

In [17]: match[2]
Out[17]: '10.0.12.1'
```

Method `groups` is used to return all substrings that correspond to groups:

```
In [18]: match.groups()
Out[18]: ('FastEthernet0/1', '10.0.12.1')
```

Named groups

When expression is complex, it is not very convenient to determine number of group. Plus, when you modify an expression the order of groups can be changed and you will need to change the code that refers to groups.

Named groups allow you to give a name to the group. Syntax of named group `(?P<name>regex)`:

```
In [19]: line = "FastEthernet0/1            10.0.12.1       YES manual up
↪           up"

In [20]: match = re.search('(?P<intf>\S+)\s+(?P<address>\S+)\s+', line)
```

These groups can now be accessed by name:

```
In [21]: match.group('intf')
Out[21]: 'FastEthernet0/1'

In [22]: match.group('address')
Out[22]: '10.0.12.1'
```

It is also very useful that with `groupdict` method you can get a dictionary where keys are the names of groups and values are the substrings that correspond to them:

```
In [23]: match.groupdict()
Out[23]: {'address': '10.0.12.1', 'intf': 'FastEthernet0/1'}
```

And then you can add groups to regex and rely on their name instead of order:

```
In [24]: match = re.search('(?P<intf>\S+)\s+(?P<address>\S+)\s+\w+\s+\w+\s+(?P
↪<status>up|down)\s+(?P<protocol>up|down)', line)

In [25]: match.groupdict()
Out[25]:
{'address': '10.0.12.1',
 'intf': 'FastEthernet0/1',
 'protocol': 'up',
 'status': 'up'}
```

Parsing the output of 'show ip dhcp snooping' command using named groups

Consider another example of using named groups. In this example, the task is to get from the output of 'show ip dhcp snooping binding' the fields: MAC address, IP address, VLAN and interface.

File dhcp_snooping.txt contains the output of command 'show ip dhcp snooping binding':

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  ------------
↪--------
00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10   ␣
↪FastEthernet0/1
00:04:A3:3E:5B:69   10.1.5.2         63951       dhcp-snooping   5    ␣
↪FastEthernet0/10
00:05:B3:7E:9B:60   10.1.5.4         63253       dhcp-snooping   5    ␣
↪FastEthernet0/9
00:09:BC:3F:A6:50   10.1.10.6        76260       dhcp-snooping   10   ␣
↪FastEthernet0/3
```

(continues on next page)

(continued from previous page)

```
Total number of bindings: 4
```

Let's start with one string:

```
In [1]: line = '00:09:BB:3D:D6:58  10.1.10.2 86250   dhcp-snooping   10 ␣
↪FastEthernet0/1'
```

In regex terms, named groups are used for those parts of the output that need to be remembered:

```
In [2]: match = re.search('(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P
↪<port>\S+)', line)
```

Comments on regex:

- `(?P<mac>\S+) +` - group with name 'mac' matches any characters except whitespace characters. regex describes a sequence of any characters before space
- `(?P<ip>\S+) +` - the same here: a sequence of any non-whitespace characters up to space. Group name - 'ip'
- `\d+ +` - numerical sequence (one or more digits) followed by one or more spaces. *Lease* value gets here
- `\S+ +`- sequence of any characters other than whitespace. This matches *Type* (in this case all of them 'dhcp-snooping')
- `(?P<vlan>\d+) +` - named group 'vlan'. Only numerical sequences with one or more characters are included here
- `(?P<port>\S+)` - named group 'port'. All characters except whitespace are included here

As a result, `groupdict` method will return such a dictionary:

```
In [3]: match.groupdict()
Out[3]:
{'int': 'FastEthernet0/1',
 'ip': '10.1.10.2',
 'mac': '00:09:BB:3D:D6:58',
 'vlan': '10'}
```

Since regex has worked well, you can create a script. In script all lines of dhcp_snooping.txt file are iterated and information about devices is displayed on the standard output stream (parse_dhcp_snooping.py):

```
import re

#'00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10   ␣
↪FastEthernet0/1'
```

(continues on next page)

(continued from previous page)

```
regex = re.compile('(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P<port>
↪\S+)')
result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groupdict())

print(f'{len(result)} devices connected to switch')

for num, comp in enumerate(result, 1):
    print(f'Parameters of device {num}:')
    for key in comp:
        print(f'{key:10}: {comp[key]:10}')
```

Script output:

```
$ python parse_dhcp_snooping.py
4 devices connected to switch
Parameters of device 1:
    int:    FastEthernet0/1
    ip:    10.1.10.2
    mac:    00:09:BB:3D:D6:58
    vlan:    10
Parameters of device 2:
    int:    FastEthernet0/10
    ip:    10.1.5.2
    mac:    00:04:A3:3E:5B:69
    vlan:    5
Parameters of device 3:
    int:    FastEthernet0/9
    ip:    10.1.5.4
    mac:    00:05:B3:7E:9B:60
    vlan:    5
Parameters of device 4:
    int:    FastEthernet0/3
    ip:    10.1.10.6
    mac:    00:09:BC:3F:A6:50
    vlan:    10
```

Non-capturing group

By default, everything that fell into the group is remembered. It's called a capturing group.

Sometimes parentheses are needed to indicate a part of expression that repeats. And, in doing so, you don't need to remember an expression.

For example, get a MAC address, VLAN and ports from log message:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7␣
↪in vlan 10 is flapping between port Gi0/5 and port Gi0/15'
```

A regex that describes substrings:

```
In [2]: match = re.search('((\w{4}\.){2}\w{4}).+vlan (\d+).+port (\S+).+port (\S+)
↪', log)
```

Expression consists of the following parts:

- `((\w{4}\.){2}\w{4})` - MAC address gets here
- `\w{4}\.` - this part describes 4 letters or digits and a dot
- `(\w{4}\.){2}` - here parentheses are used to indicate that 4 letters or digits and a dot are repeated twice
- `\w{4}` - then 4 letters or numbers
- `.+vlan (\d+)` - VLAN number falls into the group
- `.+port (\S+)` - first interface
- `.+port (\S+)` - second interface

Method `groups` returns:

```
In [3]: match.groups()
Out[3]: ('f03a.b216.7ad7', 'b216.', '10', 'Gi0/5', 'Gi0/15')
```

The second element is essentially superfluous. It appeared in the output because of parentheses in expression `(\w{4}\.){2}`.

In this case, you need to disable capture in the group. This is done by adding `?:` after the group's opening parenthesis.

Now the expression looks like this:

```
In [4]: match = re.search('((?:\w{4}\.){2}\w{4}).+vlan (\d+).+port (\S+).+port␣
↪(\S+)', log)
```

Accordingly, `groups` method returns:

```
In [5]: match.groups()
Out[5]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Repeating the captured result

When working with groups, you can use the result that matched with the group, further in the same expression.

For example, in the output of 'sh ip bgp' the last column describes AS Path attribute (through which autonomous systems the route passed):

```
In [1]: bgp = '''
   ...: R9# sh ip bgp | be Network
   ...:    Network          Next Hop       Metric LocPrf Weight Path
   ...: *  192.168.66.0/24  192.168.79.7                       0 500 500 500 i
   ...: *>                  192.168.89.8                       0 800 700 i
   ...: *  192.168.67.0/24  192.168.79.7         0             0 700 700 700 i
   ...: *>                  192.168.89.8                       0 800 700 i
   ...: *  192.168.88.0/24  192.168.79.7                       0 700 700 700 i
   ...: *>                  192.168.89.8         0             0 800 800 i
   ...: '''
```

Suppose you get those prefixes where the same AS number repeats several times in the path.

This can be done by reference to a result that has been captured by the group. For example, such an expression displays all lines in which the same number is repeated at least twice:

```
In [2]: for line in bgp.split('\n'):
   ...:     match = re.search(r'(\d+) \1', line)
   ...:     if match:
   ...:         print(line)
   ...:
*  192.168.66.0/24  192.168.79.7                       0 500 500 500 i
*  192.168.67.0/24  192.168.79.7         0             0 700 700 700 i
*  192.168.88.0/24  192.168.79.7                       0 700 700 700 i
*>                  192.168.89.8         0             0 800 800 i
```

In this expression, \1 denotes the result that falls into the group. Number one indicates a specific group. In this case, it's Group 1, it's only one group here.

Similarly, you can describe strings where the same number occurs three times:

```
In [3]: for line in bgp.split('\n'):
   ...:     match = re.search(r'(\d+) \1 \1', line)
```

(continues on next page)

(continued from previous page)

```
   ...:     if match:
   ...:         print(line)
   ...:
*  192.168.66.0/24  192.168.79.7                          0 500 500 500 i
*  192.168.67.0/24  192.168.79.7          0               0 700 700 700 i
*  192.168.88.0/24  192.168.79.7                          0 700 700 700 i
```

You can reffer to the result which was captured by named group:

```
In [129]: for line in bgp.split('\n'):
     ...:     match = re.search('(?P<as>\d+) (?P=as)', line)
     ...:     if match:
     ...:         print(line)
     ...:
*  192.168.66.0/24   192.168.79.7                          0 500 500 500 i
*  192.168.67.0/24   192.168.79.7          0               0 700 700 700 i
*  192.168.88.0/24   192.168.79.7                          0 700 700 700 i
*>                   192.168.89.8          0               0 800 800 i
```

15. Module re

Python uses re module to work with regular expressions.

Core functions of re module:

- match - searches a sequence at the beginning of the line
- search - searches for first match with template
- findall - searches for all matches with template. Returns the resulting strings as a list
- finditer - searches for any matches with template. Returns an iterator
- compile - compiles regex. You can then apply all of listed functions to this object
- fullmatch - the entire line must conform to regex described

In addition to functions that search matches, module has the following functions:

- re.sub - for replacement in strings
- re.split - to split string into parts

Match object

In re module, several functions return Match object if a match is found:

- search
- match
- finditer - returns an iterator with Match objects

This subsection covers methods of Match object.

Example of Match object:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7␣
↪in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [2]: match = re.search(r'Host (\S+) in vlan (\d+) .* port (\S+) and port (\S+)
↪', log)

In [3]: match
Out[3]: <_sre.SRE_Match object; span=(47, 124), match='Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping betwee>'
```

The 3rd line output simply displays information about object. Therefore, it is not necessary to rely on what is displayed in match part as displayed line is cut by a fixed number of characters.

group

Method group returns a substring that matches an expression or an expression in a group.

If method is called without arguments, the whole substring is displayed:

```
In [4]: match.group()
Out[4]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port␣
↪Gi0/15'
```

The same result returns group 0:

```
In [5]: match.group(0)
Out[5]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port␣
↪Gi0/15'
```

Other numbers show only the contents of relevant group:

```
In [6]: match.group(1)
Out[6]: 'f03a.b216.7ad7'

In [7]: match.group(2)
Out[7]: '10'

In [8]: match.group(3)
Out[8]: 'Gi0/5'

In [9]: match.group(4)
Out[9]: 'Gi0/15'
```

If you call a group method with a group number that is larger than number of existing groups, there is an error:

```
In [10]: match.group(5)
---------------------------------------------------------------------
IndexError                          Traceback (most recent call last)
<ipython-input-18-9df93fa7b44b> in <module>()
----> 1 match.group(5)

IndexError: no such group
```

If you call a method with multiple group numbers, the result is a tuple with strings that correspond to matches:

```
In [11]: match.group(1, 2, 3)
Out[11]: ('f03a.b216.7ad7', '10', 'Gi0/5')
```

Group may not get anything, then it will be matched with an empty string:

```
In [12]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [13]: match = re.search(r'Host (\S+) in vlan (\D*)', log)

In [14]: match.group(2)
Out[14]: ''
```

If group describes a part of template and there are more than one match, method displays the last match:

```
In [15]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [16]: match = re.search(r'Host (\w{4}\.)+', log)

In [17]: match.group(1)
Out[17]: 'b216.'
```

This is because expression in parentheses describes four letters or numbers, dot and then there is a plus. The first and the second part of MAC address matched to expression in parentheses. But only the last expression is remembered and returned.

If named groups are used in expression, group name can be passed to `group` method and the corresponding substring can be obtained:

```
In [18]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [19]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [20]: match.group('mac')
Out[20]: 'f03a.b216.7ad7'

In [21]: match.group('int2')
Out[21]: 'Gi0/15'
```

Groups are also available via number:

```
In [22]: match.group(3)
Out[22]: 'Gi0/5'

In [23]: match.group(4)
Out[23]: 'Gi0/15'
```

groups

Method group returns a tuple with strings in which the elements are those substrings that fall into respective groups:

```
In [24]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [25]: match = re.search(r'Host (\S+) '
    ...:                   r'in vlan (\d+) .* '
    ...:                   r'port (\S+) '
    ...:                   r'and port (\S+)',
    ...:                   log)
    ...:

In [26]: match.groups()
Out[26]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Method group has an optional parameter - default. It returned when anything that comes into group is optional.

For example, with this line the match will be in both the first group and the second:

```
In [26]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [27]: match = re.search(r'(\d+) +(\w+)?', line)

In [28]: match.groups()
Out[28]: ('100', 'aab1')
```

If there is nothing in the line after space, nothing will get into the group. But the match will be because it is stated in regex that the group is optional:

```
In [30]: line = '100     '

In [31]: match = re.search(r'(\d+) +(\w+)?', line)

```

(continues on next page)

(continued from previous page)

```
In [32]: match.groups()
Out[32]: ('100', None)
```

Accordingly, for the second group the value is None.

If group method is given a default value, it will be returned instead of None:

```
In [33]: line = '100     '

In [34]: match = re.search(r'(\d+) +(\w+)?', line)

In [35]: match.groups(default=0)
Out[35]: ('100', 0)

In [36]: match.groups(default='No match')
Out[36]: ('100', 'No match')
```

groupdict

Method groupdict returns a dictionary in which keys are group names and values are corresponding lines:

```
In [37]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [38]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [39]: match.groupdict()
Out[39]: {'int1': 'Gi0/5', 'int2': 'Gi0/15', 'mac': 'f03a.b216.7ad7', 'vlan': '10
↪'}
```

start, end

start and end methods return indexes of the beginning and end of the match of regex.

If methods are called without arguments, they return indexes for whole match:

```
In [40]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [41]: match = re.search(r'(\d+) +([0-9a-f.]+) +(\S+)', line)

In [42]: match.start()
Out[42]: 2

In [43]: match.end()
Out[43]: 42

In [45]: line[match.start():match.end()]
Out[45]: '10     aab1.a1a1.a5d3    FastEthernet0/1'
```

You can pass number or name of the group to methods. Then they return indexes for this group:

```
In [46]: match.start(2)
Out[46]: 9

In [47]: match.end(2)
Out[47]: 23

In [48]: line[match.start(2):match.end(2)]
Out[48]: 'aab1.a1a1.a5d3'
```

Similarly for named groups:

```
In [49]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [50]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [51]: match.start('mac')
Out[51]: 52

In [52]: match.end('mac')
Out[52]: 66
```

span

Method span returns a tuple with an index of the beginning and end of substring. It works in a similar way to `start` and end methods, but returns a pair of numbers.

Without arguments span returns indexes for whole match:

```
In [53]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [54]: match = re.search(r'(\d+) +([0-9a-f.]+) +(\S+)', line)

In [55]: match.span()
Out[55]: (2, 42)
```

But you can also pass number of the group:

```
In [56]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [57]: match = re.search(r'(\d+) +([0-9a-f.]+) +(\S+)', line)

In [58]: match.span(2)
Out[58]: (9, 23)
```

Similarly for named groups:

```
In [59]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.
↪7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [60]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [64]: match.span('mac')
Out[64]: (52, 66)

In [65]: match.span('vlan')
Out[65]: (75, 77)
```

Search function

Function `search`:

- is used to find a substring that matches a template
- returns Match object if a substring is found
- returns None if no substring was found

Function `search` is suitable when you need to find only one match in a string, for example when a regex describes the entire string or part of a string.

Consider an example of using `search` function to parse a log file. File log.txt contains log messages indicating that the same MAC is too often re-learned on one or another interface. One of the reasons for these messages is loop in network.

Contents of log.txt file:

```
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port␣
↪Gi0/16 and port Gi0/24
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port␣
↪Gi0/16 and port Gi0/24
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port␣
↪Gi0/24 and port Gi0/19
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port␣
↪Gi0/24 and port Gi0/16
```

MAC address can jump between several ports. In this case it is very important to know from which ports MAC comes.

Try to figure out which ports and which VLAN was the problem. Check regex with one line from log file:

```
In [1]: import re

In [2]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is␣
↪flapping between port Gi0/16 and port Gi0/24'

In [3]: match = re.search(r'Host \S+ '
   ...:                   r'in vlan (\d+) '
   ...:                   r'is flapping between port '
   ...:                   r'(\S+) and port (\S+)', log)
   ...:
```

Regex is divided into parts for ease of reading. It has three groups:

- `(\d+)` - describes VLAN number
- `(\S+) and port (\S+)` - describes port numbers

As a result, the following parts of line fell into the groups:

```
In [4]: match.groups()
Out[4]: ('10', 'Gi0/16', 'Gi0/24')
```

In the resulting script, log.txt is processed line by line and port information is collected from each line. Since ports can be duplicated we add them immediately to the set in order to get a compilation of unique interfaces (parse_log_search.py file):

```
import re

regex = ('Host \S+ '
         'in vlan (\d+) '
         'is flapping between port '
         '(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for line in f:
        match = re.search(regex, line)
        if match:
            vlan = match.group(1)
            ports.add(match.group(2))
            ports.add(match.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

The result of script execution:

```
$ python parse_log_search.py
Loop between ports Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Processing of 'show cdp neighbors detail' output

Try to get device parameters from 'sh cdp neighbors detail' output.

Example of output for one neighbor:

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
```

(continues on next page)

(continued from previous page)

```
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9,␣
↪RELEASE SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

The goal is to get such fields:

- neighbor name (Device ID: SW2)
- IP address of neighbor (IP address: 10.1.1.2)
- neighbor platform (Platform: cisco WS-C2960-8TC-L)
- IOS version (Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE SOFTWARE (fc1))

And for convenience you need to get data in the form of a dictionary. Example of the resulting dictionary for SW2 switch:

```
{'SW2': {'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L',
         'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9'}}
```

Example is checked on file sh_cdp_neighbors_sw1.txt.

The first solution (parse_sh_cdp_neighbors_detail_ver1.py file):

```
import re
from pprint import pprint


def parse_cdp(filename):
    result = {}
```

(continues on next page)

(continued from previous page)

```
    with open(filename) as f:
        for line in f:
            if line.startswith('Device ID'):
                neighbor = re.search('Device ID: (\S+)', line).group(1)
                result[neighbor] = {}
            elif line.startswith('  IP address'):
                ip = re.search('IP address: (\S+)', line).group(1)
                result[neighbor]['ip'] = ip
            elif line.startswith('Platform'):
                platform = re.search('Platform: (\S+ \S+),', line).group(1)
                result[neighbor]['platform'] = platform
            elif line.startswith('Cisco IOS Software'):
                ios = re.search('Cisco IOS Software, (.+), RELEASE',
                                line).group(1)
                result[neighbor]['ios'] = ios

    return result


pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

The desired strings are selected using startswith() string method. And in a string, a regex takes required part of the string. It all ends up in a dictionary.

The result is:

```
$ python parse_sh_cdp_neighbors_detail_ver1.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

It worked out well, but it can be done in a more compact way.

The second version of solution (parse_sh_cdp_neighbors_detail_ver2.py file):

```
import re
from pprint import pprint
```

(continues on next page)

(continued from previous page)

```
def parse_cdp(filename):
    regex = ('Device ID: (?P<device>\S+)'
             '|IP address: (?P<ip>\S+)'
             '|Platform: (?P<platform>\S+ \S+),'
             '|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open(filename) as f:
        for line in f:
            match = re.search(regex, line)
            if match:
                if match.lastgroup == 'device':
                    device = match.group(match.lastgroup)
                    result[device] = {}
                else:
                    result[device][match.lastgroup] = match.group(
                        match.lastgroup)

    return result


pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Explanations for the second option:

- in regex, all lines written via | sign (or)
- if a match is found, `lastgroup` method is checked
- `lastgroup` method returns name of the last named group in regex for which a match has been found
- if a match was found for `device` group, the value that fells into the group is written to `device` variable
- otherwise the mapping of `'group name': 'corresponding value'` is written to dictionary

Result will be the same:

```
$ python parse_sh_cdp_neighbors_detail_ver2.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
```

(continues on next page)

(continued from previous page)

```
       'ip': '10.2.2.2',
       'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
        'ip': '10.1.1.2',
        'platform': 'cisco WS-C2960-8TC-L'}}
```

Match function

Function `match`:

- is used to search at the beginning of string that corresponds to regex
- returns Match object if match is found
- returns None if no match was found

`Match` function differs from `search` in that `match` always looks for a match at the beginning of the line. For example, if you repeat the example that was used for `search` function, but with `match`:

```
In [2]: import re

In [3]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is␣
↪flapping between port Gi0/16 and port Gi0/24'

In [4]: match = re.match(r'Host \S+ '
   ...:                  r'in vlan (\d+) '
   ...:                  r'is flapping between port '
   ...:                  r'(\S+) and port (\S+)', log)
   ...:
```

The result will be None:

```
In [6]: print(match)
None
```

This is because `match` searches for Host word at the beginning of the line. But this message is in the middle.

In this case it is easy to fix expression so that match() function finds match:

```
In [4]: match = re.match(r'\S+: Host \S+ '
   ...:                  r'in vlan (\d+) '
   ...:                  r'is flapping between port '
   ...:                  r'(\S+) and port (\S+)', log)
   ...:
```

Expression \S+: was added before *Host* word. Now match will be found:

```
In [11]: print(match)
<_sre.SRE_Match object; span=(0, 104), match='%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.
↪4c18.0156 in >

In [12]: match.groups()
Out[12]: ('10', 'Gi0/16', 'Gi0/24')
```

Example is similar to one used in search function with minor changes (parse_log_match match.py file):

```
import re

regex = (r'\S+: Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for line in f:
        match = re.match(regex, line)
        if match:
            vlan = match.group(1)
            ports.add(match.group(2))
            ports.add(match.group(3))

print('Loop between ports {} in VLAN {}'.format(', '.join(ports), vlan))
```

The result is:

```
$ python parse_log_match.py
Loop between ports Gi0/19, Gi0/24, Gi0/16 in VLAN 10
```

Finditer function

Function finditer:

- is used to search for all non-overlapping matches in string
- returns an iterator with Match objects
- finditer returns iterator even if no match is found

Function `finditer` is well suited to handle those commands whose output is displayed by columns. For example: 'sh ip int br', 'sh mac address-table', etc. In this case it can be applied to the entire output of command.

Example of 'sh ip int br' output:

```
In [8]: sh_ip_int_br = '''
   ...: R1#show ip interface brief
   ...: Interface              IP-Address      OK? Method Status           Protocol
   ...: FastEthernet0/0        15.0.15.1       YES manual up               up
   ...: FastEthernet0/1        10.0.12.1       YES manual up               up
   ...: FastEthernet0/2        10.0.13.1       YES manual up               up
   ...: FastEthernet0/3        unassigned      YES unset  up               up
   ...: Loopback0              10.1.1.1        YES manual up               up
   ...: Loopback100            100.0.0.1       YES manual up               up
   ...: '''
```

regex for output processing:

```
In [9]: result = re.finditer(r'(\S+) +'
   ...:                      r'([\d.]+) +'
   ...:                      r'\w+ +\w+ +'
   ...:                      r'(up|down|administratively down) +'
   ...:                      r'(up|down)',
   ...:                      sh_ip_int_br)
   ...:
```

`result` variable contains an iterator:

```
In [12]: result
Out[12]: <callable_iterator at 0xb583f46c>
```

Iterator contains Match objects:

```
In [16]: groups = []

In [18]: for match in result:
    ...:     print(match)
    ...:     groups.append(match.groups())
    ...:
<_sre.SRE_Match object; span=(103, 171), match='FastEthernet0/0       15.0.15.1  ␣
↪    YES manual >
<_sre.SRE_Match object; span=(172, 240), match='FastEthernet0/1       10.0.12.1  ␣
↪    YES manual >
<_sre.SRE_Match object; span=(241, 309), match='FastEthernet0/2       10.0.13.1  ␣
↪    YES manual >
```

(continues on next page)

(continued from previous page)

```
<_sre.SRE_Match object; span=(379, 447), match='Loopback0              10.1.1.1
↪    YES manual >
<_sre.SRE_Match object; span=(448, 516), match='Loopback100            100.0.0.1
↪    YES manual >'
```

Now in groups list there are tuples with strings that fallen into groups:

```
In [19]: groups
Out[19]:
[('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]
```

A similar result can be obtained by a list comprehension:

```
In [20]: regex = r'(\S+) +([\d.]+) +\w+ +\w+ +(up|down|administratively down)
↪+(up|down)'

In [21]: result = [match.groups() for match in re.finditer(regex, sh_ip_int_br)]

In [22]: result
Out[22]:
[('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]
```

Now we will analyze the same log file that was used in `search` and `match` subsections.

In this case it is possible to pass the entire contents of file (parse_log_finditer.py):

```
import re

regex = (r'Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
```

(continues on next page)

(continued from previous page)

```
    for m in re.finditer(regex, f.read()):
        vlan = m.group(1)
        ports.add(m.group(2))
        ports.add(m.group(3))

print('Loop between ports {} in VLAN {}'.format(', '.join(ports), vlan))
```

Warning: In real life, a log file can be very large. In that case, it's better to process it line by line.

Output will be the same:

```
$ python parse_log_finditer.py
Loop between ports Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Processing of 'show cdp neighbors detail' output

`finditer` can handle output of 'sh cdp neighbors detail' as well as in re.search subsection.

The script is almost identical to version with `re.search` (parse_sh_cdp_neighbors_detail_finditer.py file):

```
import re
from pprint import pprint


def parse_cdp(filename):
    regex = (r'Device ID: (?P<device>\S+)'
             r'|IP address: (?P<ip>\S+)'
             r'|Platform: (?P<platform>\S+ \S+),'
             r'|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open(filename) as f:
        match_iter = re.finditer(regex, f.read())
        for match in match_iter:
            if match.lastgroup == 'device':
                device = match.group(match.lastgroup)
                result[device] = {}
            elif device:
```

(continues on next page)

(continued from previous page)

```
                result[device][match.lastgroup] = match.group(match.lastgroup)

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Now matches are searched throughout the file, not in every line separately:

```
with open(filename) as f:
    match_iter = re.finditer(regex, f.read())
```

Then matches go through the loop:

```
with open(filename) as f:
    match_iter = re.finditer(regex, f.read())
    for match in match_iter:
```

The rest is the same.

The result will be:

```
$ python parse_sh_cdp_neighbors_detail_finditer.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

Although the result is similar, `finditer` has more features, as you can specify not only what should be in searched string but also in strings around it. For example, you can specify exactly which IP address to take:

```
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP

...

Native VLAN: 1
```

(continues on next page)

(continued from previous page)

```
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

If you want to take the first IP address you can supplement a regex like this:

```
regex = (r'Device ID: (?P<device>\S+)'
         r'|Entry address.*\n +IP address: (?P<ip>\S+)'
         r'|Platform: (?P<platform>\S+ \S+),'
         r'|Cisco IOS Software, (?P<ios>.+), RELEASE')
```

Findall function

Function `findall`:

- is used to search for all non-overlapping matches in string
- returns:
  - list of strings that are described by regex if there are no groups in regex
  - list of strings that match with regex in the group if there is only one group in regex
  - list of tuples containing strings that matches with expression in the group if there are more than one group

Consider the work of `findall` with an example of 'sh mac address-table output':

```
In [2]: mac_address_table = open('CAM_table.txt').read()

In [3]: print(mac_address_table)
sw1#sh mac address-table
          Mac Address Table
-------------------------------------------

Vlan    Mac Address       Type        Ports
----    -----------       --------    -----
 100    a1b2.ac10.7000    DYNAMIC     Gi0/1
 200    a0d4.cb20.7000    DYNAMIC     Gi0/2
 300    acb4.cd30.7000    DYNAMIC     Gi0/3
 100    a2bb.ec40.7000    DYNAMIC     Gi0/4
 500    aa4b.c550.7000    DYNAMIC     Gi0/5
 200    a1bb.1c60.7000    DYNAMIC     Gi0/6
 300    aa0b.cc70.7000    DYNAMIC     Gi0/7
```

The first example is a regex without groups. In this case `findall` returns a list of strings that matches with regex.

For example, with `findall` you can get a list of matching strings with `vlan - mac – interface` and get rid of header in the output of command:

```
In [4]: re.findall(r'\d+ +\S+ +\w+ +\S+', mac_address_table)
Out[4]:
['100    a1b2.ac10.7000    DYNAMIC     Gi0/1',
 '200    a0d4.cb20.7000    DYNAMIC     Gi0/2',
 '300    acb4.cd30.7000    DYNAMIC     Gi0/3',
 '100    a2bb.ec40.7000    DYNAMIC     Gi0/4',
 '500    aa4b.c550.7000    DYNAMIC     Gi0/5',
 '200    a1bb.1c60.7000    DYNAMIC     Gi0/6',
 '300    aa0b.cc70.7000    DYNAMIC     Gi0/7']
```

Note that `findall` returns a list of strings, not a Match object.

As soon as a group appears in regex, `findall` behaves differently. If one group is used in the expression, `findall` returns a list of strings that matches with expression in the group:

```
In [5]: re.findall(r'\d+ +(\S+) +\w+ +\S+', mac_address_table)
Out[5]:
['a1b2.ac10.7000',
 'a0d4.cb20.7000',
 'acb4.cd30.7000',
 'a2bb.ec40.7000',
 'aa4b.c550.7000',
 'a1bb.1c60.7000',
 'aa0b.cc70.7000']
```

`findall` searches for a match of the entire string but returns a result similar to `group` method in Match object. If there are several groups, `findall` will return the list of tuples:

```
In [6]: re.findall(r'(\d+) +(\S+) +\w+ +(\S+)', mac_address_table)
Out[6]:
[('100', 'a1b2.ac10.7000', 'Gi0/1'),
 ('200', 'a0d4.cb20.7000', 'Gi0/2'),
 ('300', 'acb4.cd30.7000', 'Gi0/3'),
 ('100', 'a2bb.ec40.7000', 'Gi0/4'),
 ('500', 'aa4b.c550.7000', 'Gi0/5'),
 ('200', 'a1bb.1c60.7000', 'Gi0/6'),
 ('300', 'aa0b.cc70.7000', 'Gi0/7')]
```

If such features of `findall` function prevent you from getting the needed result, it is better to use `finditer` function, but sometimes this behavior is appropriate and convenient to use.

An example of using `findall` in a log file parsing (parse_log_findall.py file):

```
import re

regex = (r'Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    result = re.findall(regex, f.read())
    for vlan, port1, port2 in result:
        ports.add(port1)
        ports.add(port2)

print('Loop between ports {} in VLAN {}'.format(', '.join(ports), vlan))
```

The result is:

```
$ python parse_log_findall.py
Loop between ports Gi0/19, Gi0/16, Gi0/24 в VLAN 10
```

Compile function

Python has the ability to pre-compile a regular expression and then use it. This is particularly useful when regex is used a lot in the script.

The use of a compiled expression can speed up processing and it is generally more convenient to use this option as the program divides the creation of a regex and its use. In addition, using `re.compile` function creates a RegexObject object that has several additional features that are not present in MatchObject object.

To compile a regex, use `re.compile`:

```
In [52]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')
```

It returns RegexObject object:

```
In [53]: regex
Out[53]: re.compile(r'\d+ +\S+ +\w+ +\S+', re.UNICODE)
```

RegexObject has such methods and attributes:

```
In [55]: [method for method in dir(regex) if not method.startswith('_')]
Out[55]:
['findall',
 'finditer',
 'flags',
 'fullmatch',
 'groupindex',
 'groups',
 'match',
 'pattern',
 'scanner',
 'search',
 'split',
 'sub',
 'subn']
```

Note that Regex object has `search`, `match`, `finditer`, `findall` methods available. These are the same functions that are available in module globally, but now they have to be applied to object.

An example of using `search` method:

```
In [67]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [68]: match = regex.search(line)
```

Now `search` should be called as method of regex object. The result is a Match object:

```
In [69]: match
Out[69]: <_sre.SRE_Match object; span=(1, 43), match='100    a1b2.ac10.7000    ␣
↪DYNAMIC     Gi0/1'>

In [70]: match.group()
Out[70]: '100    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

An example of compiling a regex and its use based on example of a log file (parse_log_compile.py file):

```
import re

regex = re.compile(r'Host \S+ '
                   r'in vlan (\d+) '
                   r'is flapping between port '
                   r'(\S+) and port (\S+)')

ports = set()
```

(continues on next page)

(continued from previous page)

```
with open('log.txt') as f:
    for m in regex.finditer(f.read()):
        vlan = m.group(1)
        ports.add(m.group(2))
        ports.add(m.group(3))

print('Loop between ports {} in VLAN {}'.format(', '.join(ports), vlan))
```

This is a modified example of `finditer` usage. Description of regex changed:

```
regex = re.compile(r'Host \S+ '
                   r'in vlan (\d+) '
                   r'is flapping between port '
                   r'(\S+) and port (\S+)')
```

And now the call of `finditer` is executed as a regex object method:

```
for m in regex.finditer(f.read()):
```

Options that are available only when using re.compile

When using re.compile in `search`, `match`, `findall`, `finditer` and `fullmatch` methods, additional parameters appear:

- `pos` - allows you to specify an index in string from where to start looking for a match
- `endpos` - specifies from which index the search should be started

Their use is similar to execution of a string slice.

For example, this is the result without specifying `pos`, `endpos` parameters:

```
In [75]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [76]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [77]: match = regex.search(line)

In [78]: match.group()
Out[78]: '100    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

In this case, the initial search position should be indicated:

```
In [79]: match = regex.search(line, 2)

In [80]: match.group()
Out[80]: '00    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

The initial entry is the same as string slice:

```
In [81]: match = regex.search(line[2:])

In [82]: match.group()
Out[82]: '00    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

A final example is the use of two indexes:

```
In [90]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [91]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [92]: match = regex.search(line, 2, 40)

In [93]: match.group()
Out[93]: '00    a1b2.ac10.7000    DYNAMIC     Gi'
```

And a similar string slice:

```
In [94]: match = regex.search(line[2:40])

In [95]: match.group()
Out[95]: '00    a1b2.ac10.7000    DYNAMIC     Gi'
```

In `match`, `findall`, `finditer` and `fullmatch` methods `pos` and `endpos` parameters work similarly.

Flags

When using re functions or creating a compiled regex you can specify additional flags that affect the behavior of regex.

The `re` module supports flags (in parentheses - a short version of flag):

- `re.ASCII` (`re.A`)
- `re.IGNORECASE` (`re.I`)
- `re.MULTILINE` (`re.M`)
- `re.DOTALL` (`re.S`)

- re.VERBOSE (re.X)
- re.LOCALE (re.L)
- re.DEBUG

In this subsection the re.DOTALL flag is covered. Information about other flags is available in documentation.

re.DOTALL

Regex can also be used for multiline string.

For example, from sh_cdp string you need to get a device name, platform and IOS:

```
In [2]: sh_cdp = '''
   ...: Device ID: SW2
   ...: Entry address(es):
   ...:   IP address: 10.1.1.2
   ...: Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
   ...: Interface: GigabitEthernet1/0/16,  Port ID (outgoing port):␣
↪GigabitEthernet0/1
   ...: Holdtime : 164 sec
   ...:
   ...: Version :
   ...: Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.
↪2(55)SE9, RELEASE SOFTWARE (fc1)
   ...: Technical Support: http://www.cisco.com/techsupport
   ...: Copyright (c) 1986-2014 by Cisco Systems, Inc.
   ...: Compiled Mon 03-Mar-14 22:53 by prod_rel_team
   ...:
   ...: advertisement version: 2
   ...: VTP Management Domain: ''
   ...: Native VLAN: 1
   ...: Duplex: full
   ...: Management address(es):
   ...:   IP address: 10.1.1.2
   ...: '''
```

Of course, in this case it is possible to divide a string into parts and work with each string separately, but you can get necessary data without splitting.

In this expression, strings with required data are described:

```
In [3]: regex = r'Device ID: (\S+).+Platform: \w+ (\S+),.+Cisco IOS Software.+␣
↪Version (\S+),'
```

In this case, there will be no match because by default a dot means any character other than a new line character:

```
In [4]: print(re.search(regex, sh_cdp))
None
```

You can change default behavior by using `re.DOTALL` flag:

```
In [5]: match = re.search(regex, sh_cdp, re.DOTALL)

In [6]: match.groups()
Out[6]: ('SW2', 'WS-C2960-8TC-L', '12.2(55)SE9')
```

Since new line character is now included, combination `.+` captures everything between data.

Now try to use this regex to get information about all neighbors from sh_cdp_neighbors_sw1.txt file.

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9,␣
↪RELEASE SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport

-------------------------
Device ID: R1
Entry address(es):
  IP address: 10.1.1.1
Platform: Cisco 3825,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/22,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1,␣
↪RELEASE SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport

-------------------------
```

(continues on next page)

(continued from previous page)

```
Device ID: R2
Entry address(es):
  IP address: 10.2.2.2
Platform: Cisco 2911,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/21,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1,␣
↪RELEASE SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport
```

Search for all regex matches:

```
In [7]: with open('sh_cdp_neighbors_sw1.txt') as f:
   ...:     sh_cdp = f.read()
   ...:

In [8]: regex = r'Device ID: (\S+).+Platform: \w+ (\S+),.+Cisco IOS Software.+␣
↪Version (\S+),'

In [9]: match = re.finditer(regex, sh_cdp, re.DOTALL)

In [10]: for m in match:
    ...:     print(m.groups())
    ...:
('SW2', '2911', '15.2(2)T1')
```

At first glance, it seems that instead of three devices there was only one device in output. However, if you look at the results the tuple has Device ID from the first neighbor and platform and IOS from the last neighbor.

A short output to ease understanding of result:

```
Device ID        Local Intrfce     Holdtme    Capability  Platform  Port ID
SW2              Gi 1/0/16         171              R S   C2960     Gi 0/1
R1               Gi 1/0/22         158              R     C3825     Gi 0/0
R2               Gi 1/0/21         177              R     C2911     Gi 0/0
```

This is because there is a `.+` combination between desired parts of the output. Without `re.DOTALL` flag, such an expression would capture everything before new line character, but with a flag it captures the longest possible piece of text because + is greedy. As a result, regex describes a string from the first Device ID to the last place where `Cisco IOS Software.+ Version` match occurs.

This situation occurs very often when using `re.DOTALL` and in order to correct it remember to disable

greedy behavior:

```
In [10]: regex = r'Device ID: (\S+).+?Platform: \w+ (\S+),.+?Cisco IOS Software.+?
↪ Version (\S+),'

In [11]: match = re.finditer(regex, sh_cdp, re.DOTALL)

In [12]: for m in match:
    ...:     print(m.groups())
    ...:
('SW2', 'WS-C2960-8TC-L', '12.2(55)SE9')
('R1', '3825', '12.4(24)T1')
('R2', '2911', '15.2(2)T1')
```

Function re.split

Function `split` works similary to `split` method in strings, but in `re.split` function you can use regular expressions which means dividing a string into parts using more complex conditions.

For example, `ospf_route` string should be split by spaces (as in `str.split` method):

```
In [1]: ospf_route = 'O     10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h,␣
↪FastEthernet0/0'

In [2]: re.split(r' +', ospf_route)
Out[2]:
['O',
 '10.0.24.0/24',
 '[110/41]',
 'via',
 '10.0.13.3,',
 '3d18h,',
 'FastEthernet0/0']
```

Similarly, commas can be removed:

```
In [3]: re.split(r'[ ,]+', ospf_route)
Out[3]:
['O',
 '10.0.24.0/24',
 '[110/41]',
 'via',
 '10.0.13.3',
```

(continues on next page)

(continued from previous page)

```
 '3d18h',
 'FastEthernet0/0']
```

And if necessary, get rid of square brackets:

```
In [4]: re.split(r'[ ,\[\]]+', ospf_route)
Out[4]: ['O', '10.0.24.0/24', '110/41', 'via', '10.0.13.3', '3d18h',
↪'FastEthernet0/0']
```

Function `split` has a peculiarity of working with groups (expressions in parentheses). If you specify the same expression with parentheses, the resulting list will include separators.

For example, word via is specified as a separator:

```
In [5]: re.split(r'(via|[ ,\[\]])+', ospf_route)
Out[5]:
['O',
 ' ',
 '10.0.24.0/24',
 '[',
 '110/41',
 ' ',
 '10.0.13.3',
 ' ',
 '3d18h',
 ' ',
 'FastEthernet0/0']
```

To disable such behavior you should make a noncapture group. That is, disable capturing of group elements:

```
In [6]: re.split(r'(?:via|[ ,\[\]])+', ospf_route)
Out[6]: ['O', '10.0.24.0/24', '110/41', '10.0.13.3', '3d18h', 'FastEthernet0/0']
```

Function re.sub

Function `re.sub` works similary to `replace` method in strings. But in `re.sub` you can use regex and therefore make substitutions using more complex conditions. Replace commas, square brackets and via word with space in ospf_route string:

```
In [7]: ospf_route = 'O    10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h,␣
↪FastEthernet0/0'
```

(continues on next page)

(continued from previous page)

```
In [8]: re.sub(r'(via|[,\[\]])', ' ', ospf_route)
Out[8]: 'O        10.0.24.0/24  110/41    10.0.13.3  3d18h  FastEthernet0/0'
```

With `re.sub` you can transform a string. For example, convert mac_table string to:

```
In [9]: mac_table = '''
   ...:  100    aabb.cc10.7000    DYNAMIC     Gi0/1
   ...:  200    aabb.cc20.7000    DYNAMIC     Gi0/2
   ...:  300    aabb.cc30.7000    DYNAMIC     Gi0/3
   ...:  100    aabb.cc40.7000    DYNAMIC     Gi0/4
   ...:  500    aabb.cc50.7000    DYNAMIC     Gi0/5
   ...:  200    aabb.cc60.7000    DYNAMIC     Gi0/6
   ...:  300    aabb.cc70.7000    DYNAMIC     Gi0/7
   ...: '''

In [4]: print(re.sub(r' *(\d+) +'
   ...:              r'([a-f0-9]+)\.'
   ...:              r'([a-f0-9]+)\.'
   ...:              r'([a-f0-9]+) +\w+ +'
   ...:              r'(\S+)',
   ...:              r'\1 \2:\3:\4 \5',
   ...:              mac_table))
   ...:

100 aabb:cc10:7000 Gi0/1
200 aabb:cc20:7000 Gi0/2
300 aabb:cc30:7000 Gi0/3
100 aabb:cc40:7000 Gi0/4
500 aabb:cc50:7000 Gi0/5
200 aabb:cc60:7000 Gi0/6
300 aabb:cc70:7000 Gi0/7
```

Regex is divided into groups:

- `(\d+)` - the first group. VLAN number gets here
- `([a-f0-9]+).([a-f0-9]+).([a-f0-9]+)` - the following three groups (2, 3, 4) describe MAC address
- `(\S+)` - the fifth group. Describes an interface.

In a second regex these groups are used. To refer to a group a backslash and a group number are used. To avoid backslash screening, raw string is used. As a result, the corresponding substrings will be substituted instead of group numbers. For example, format of MAC address record was also changed.

Further reading

Regular expressions in Python:

- Regular Expression HOWTO
- Python 3 Module of the Week. Module re

Websites for regular expressions checking:

- regex101
- for Python - you can specify search, match, findall methods and flags. An example of a regular expression. Unfortunately, sometimes not all expressions are perceived.
- Another site for Python - does not support methods but works well and has worked out the expressions which didn't work in previous site. It's perfect for one-line text. With the multiline, it worth considering that Python will have a different situation.

General guidance on the use of regular expressions:

- Many examples of the use of regular expressions from basics to more complex themes
- Book Mastering Regular Expressions

Assistance in the study of regular expressions:

- Visualize regular expression
- Regex Crossword

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 15.1

Create a get_ip_from_cfg function that expects the name of the file containing the device configuration as an argument.

The function should process the configuration and return the IP addresses and masks that are configured on the interfaces as a list of tuples:

- the first element of the tuple is the IP address
- the second element of the tuple is a mask

For example (arbitrary addresses are taken):

```
[("10.0.1.1", "255.255.255.0"), ("10.0.2.1", "255.255.255.0")]
```

To get this result, use regular expressions.

Check the operation of the function using the config_r1.txt file.

Please note that in this case, you can not check the correctness of the IP address, address ranges, and so on, since the command output from network device is processed, not user input.

Task 15.1a

Copy the get_ip_from_cfg function from task 15.1 and redesign it so that it returns a dictionary:

- key: interface name
- value: a tuple with two lines:
  - IP address
  - mask

Add to the dictionary only those interfaces on which IP addresses are configured.

Dict example (arbitrary addresses are taken):

```
{"FastEthernet0/1": ("10.0.1.1", "255.255.255.0"),
 "FastEthernet0/2": ("10.0.2.1", "255.255.255.0")}
```

To get this result, use regular expressions.

Check the operation of the function using the example of the config_r1.txt file.

Please note that in this case, you can not check the correctness of the IP address, address ranges, and so on, since the command output from network device is processed, not user input.

Task 15.1b

Check the get_ip_from_cfg function from task 15.1a on the config_r2.txt configuration.

Note that there are two IP addresses assigned on the e0/1 interface:

```
interface Ethernet0/1
 ip address 10.255.2.2 255.255.255.0
 ip address 10.254.2.2 255.255.255.0 secondary
```

And in the dictionary returned by the get_ip_from_cfg function, only one of them (first or second) corresponds to the Ethernet0/1 interface.

Copy the get_ip_from_cfg function from 15.1a and redesign it to return a list of tuples for each interface in the dictionary value. If only one address is assigned on the interface, there will be one tuple in the list. If several IP addresses are configured on the interface, then the list will contain several tuples. The interface name remains the key.

Check the function in the config_r2.txt configuration and make sure the Ethernet0/1 interface matches a list of two tuples.

Please note that in this case, you can not check the correctness of the IP address, address ranges, and so on, since the command output from network device is processed, not user input.

Task 15.2

Create a function parse_sh_ip_int_br that expects as an argument the name of the file containing the output of the show ip int br command.

The function should process the output of the show ip int br command and return the following fields:

- Interface
- IP-Address
- Status
- Protocol

The information should be returned as a list of tuples:

```
[("FastEthernet0/0", "10.0.1.1", "up", "up"),
 ("FastEthernet0/1", "10.0.2.1", "up", "up"),
 ("FastEthernet0/2", "unassigned", "down", "down")]
```

To get this result, use regular expressions.

Check the operation of the function using the example of the sh_ip_int_br.txt file.

Task 15.3

Create a convert_ios_nat_to_asa function that converts NAT rules from cisco IOS syntax to cisco ASA.

The function expects such arguments:

- the name of the file containing the Cisco IOS NAT rules
- the name of the file in which to write the NAT rules for the ASA

The function returns None.

Check the function on the cisco_nat_config.txt file.

Example cisco IOS NAT rules

```
ip nat inside source static tcp 10.1.2.84 22 interface GigabitEthernet0/1 20022
ip nat inside source static tcp 10.1.9.5 22 interface GigabitEthernet0/1 20023
```

And the corresponding NAT rules for the ASA:

```
object network LOCAL_10.1.2.84
 host 10.1.2.84
 nat (inside,outside) static interface service tcp 22 20022
object network LOCAL_10.1.9.5
 host 10.1.9.5
 nat (inside,outside) static interface service tcp 22 20023
```

In the file with the rules for the ASA:

- there should be no blank lines between the rules
- there must be no spaces before the lines “object network”
- there must be one space before the rest of the lines

In all rules for ASA, the interfaces will be the same (inside, outside).

Task 15.4

Create a get_ints_without_description function that expects as an argument the name of the file containing the device configuration.

The function should process the configuration and return a list of interface names, which do not have a description (description command).

An example of an interface with a description:

```
interface Ethernet0/2
 description To P_r9 Ethernet0/2
 ip address 10.0.19.1 255.255.255.0
 mpls traffic-eng tunnels
 ip rsvp bandwidth
```

Interface without description:

```
interface Loopback0
 ip address 10.1.1.1 255.255.255.255
```

Check the operation of the function using the example of the config_r1.txt file.

Task 15.5

Create a generate_description_from_cdp function that expects as an argument the name of the file that contains the output of the show cdp neighbors command.

The function should process the show cdp neighbors command output and generate a description for the interfaces based on the command output.

For example, if R1 has the following command output:

```
R1>show cdp neighbors
Capability Codes: R - Router, T - Trans Bridge, B - Source Route Bridge
                  S - Switch, H - Host, I - IGMP, r - Repeater

Device ID        Local Intrfce     Holdtme    Capability  Platform  Port ID
SW1              Eth 0/0           140          S I      WS-C3750-  Eth 0/1
```

For the Eth 0/0 interface, you need to generate the following description:

```
description Connected to SW1 port Eth 0/1
```

The function must return a dictionary, in which the keys are the names of the interfaces, and the values are the command specifying the description of the interface:

```
'Eth 0/0': 'description Connected to SW1 port Eth 0/1'
```

Check the operation of the function on the sh_cdp_n_sw1.txt file.

6

IV. Data writing and transmission

This part of the book covers data writing and transmission. Data can be, for example:

- command output
- processed output of commands as dictionary, list or similar
- information from monitoring system

So far, only the simplest option has been covered - writing information to a plain text file.

This section covers data reading and writing in CSV, JSON and YAML formats:

- CSV - a tabular format of data presentation. It can be obtained, for example, by exporting data from a table or database. Similarly, data can be written in this format for further import into the table.
- JSON - a format that is often used in API. In addition, this format will allow you to save data structures such as dictionaries or lists in a structured format and then read them from a JSON file and get the same data structures in Python.
- YAML format is often used to describe playbooks. For example, it is used in Ansible. In addition, in this format it is convenient to write manually the parameters that should be read by scripts.

Note: Python allows objects of language itself to be written into files and read through Pickle module, but this topic is not covered in this book.

---

16. Unicode

Programs we write are not isolated. They download data from the Internet, read and write data on disk, transmit data over the network.

So it's very important to understand the difference between how a computer stores and transmits data and how that data is perceived by a person. We take text, computer takes bytes.

Python 3, respectively, has two concepts:

- text - an immutable sequence of unicode characters. Type string (str) is used to store these characters
- data - an immutable sequence of bytes. Type bytes is used for storage

---

Note: It is more correct to say that text is an immutable sequence of Unicode codes (codepoints).

Unicode standard

Unicode is a standard that describes the representation and encoding of almost all languages and other characters.

A few facts about Unicode:

- version 13.0 (March 2020) describes 143 859 codes
- each code is a number that corresponds to a certain character
- standard also defines the encoding - the way of representing the symbol code in bytes

Each character in Unicode has a specific code. This is a number that is usually written as follows: U+0073, where 0073 - hexadecimal digits. Apart from the code, each symbol has its own unique name. For example, letter "s" corresponds to code U+0073 and the name "LATIN SMALL LETTER S".

Examples of codes, names and corresponding symbols:

- U+0073, "LATIN SMALL LETTER S" - s
- U+00F6, "LATIN SMALL LETTER O WITH DIAERESIS" - ö
- U+1F383, "JACK-O-LANTERN" - 🎃
- U+2615, "HOT BEVERAGE" - ☕
- U+1f600, "GRINNING FACE" - 😀

Encodings

Encodings allow to write character code in bytes.

Unicode supports several encodings:

- UTF-8
- UTF-16
- UTF-32

One of the most popular encoding to date is UTF-8. This encoding uses a variable number of bytes to write Unicode characters.

Examples of Unicode characters and their representation in bytes in UTF-8 encoding:

- H - 48
- i - 69
- □ - 01 f6 c0
- □ - 01 f6 80
- ☃ - 26 03

Unicode in Python 3

Python 3 has:

- strings - an immutable sequence of Unicode characters. Type string (str) is used to store these characters
- bytes - an immutable sequence of bytes. Type bytes is used for storage

Strings

Examples of strings:

```
In [11]: hi = 'привет'

In [12]: hi
Out[12]: 'привет'

In [15]: type(hi)
Out[15]: str

In [13]: beautiful = 'schön'
```

(continues on next page)

(continued from previous page)

```
In [14]: beautiful
Out[14]: 'schön'
```

Since strings are a sequence of Unicode codes you can write a string in different ways.

Unicode symbol can be written using its name:

```
In [1]: "\N{LATIN SMALL LETTER O WITH DIAERESIS}"
Out[1]: 'ö'
```

Or by using this format:

```
In [4]: "\u00F6"
Out[4]: 'ö'
```

You can write a string as a sequence of Unicode codes:

```
In [19]: hi1 = 'привет'

In [20]: hi2 = '\u043f\u0440\u0438\u0432\u0435\u0442'

In [21]: hi2
Out[21]: 'привет'

In [22]: hi1 == hi2
Out[22]: True

In [23]: len(hi2)
Out[23]: 6
```

Function ord() returns value of Unicode code for character:

```
In [6]: ord('ö')
Out[6]: 246
```

Function chr() returns Unicode character that corresponds to the code:

```
In [7]: chr(246)
Out[7]: 'ö'
```

Bytes

Bytes are an immutable sequence of bytes.

Bytes are denoted in the same way as strings but with addition of letter b before string:

```
In [30]: b1 = b'\xd0\xb4\xd0\xb0'

In [31]: b2 = b"\xd0\xb4\xd0\xb0"

In [32]: b3 = b'''\xd0\xb4\xd0\xb0'''

In [36]: type(b1)
Out[36]: bytes

In [37]: len(b1)
Out[37]: 4
```

In Python, bytes that correspond to ASCII symbols are displayed as these symbols, not as their corresponding bytes. This may be a bit confusing but it is always possible to recognize bytes type by letter b:

```
In [38]: bytes1 = b'hello'

In [39]: bytes1
Out[39]: b'hello'

In [40]: len(bytes1)
Out[40]: 5

In [41]: bytes1.hex()
Out[41]: '68656c6c6f'

In [42]: bytes2 = b'\x68\x65\x6c\x6c\x6f'

In [43]: bytes2
Out[43]: b'hello'
```

If you try to write not an ASCII character in a byte literal, an error will occur:

```
In [44]: bytes3 = b'привет'
  File "<ipython-input-44-dc8b23504fa7>", line 1
    bytes3 = b'привет'
             ^
SyntaxError: bytes can only contain ASCII literal characters.
```

Conversion between bytes and strings

You can't avoid working with bytes. For example, when working with a network or a filesystem, most often the result is returned in bytes. Accordingly, you need to know how to convert bytes to string and vice versa. That's what the encoding is for.

Encoding can be represented as an encryption key that specifies:

- how to "encrypt" a string to bytes (str -> bytes). Encode method used (similar to encrypt)
- how to "decrypt" bytes to string (bytes -> str). Decode method used (similar to decrypt)

This analogy makes it clear that string-byte and byte-string transformations must use the same encoding.

encode, decode

encode method is used to convert string to bytes:

```
In [1]: hi = 'привет'

In [2]: hi.encode('utf-8')
Out[2]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [3]: hi_bytes = hi.encode('utf-8')
```

decode method to get a string from bytes:

```
In [4]: hi_bytes
Out[4]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [5]: hi_bytes.decode('utf-8')
Out[5]: 'привет'
```

str.encode, bytes.decode

Method encode is also present in str class (as are other methods of working with strings):

```
In [6]: hi
Out[6]: 'привет'

In [7]: str.encode(hi, encoding='utf-8')
Out[7]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
```

And decode method is available in bytes class (like other methods):

```
In [8]: hi_bytes
Out[8]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [9]: bytes.decode(hi_bytes, encoding='utf-8')
Out[9]: 'привет'
```

In these methods, encoding can be used as a key argument (examples above) or as a positional argument:

```
In [10]: hi_bytes
Out[10]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [11]: bytes.decode(hi_bytes, 'utf-8')
Out[11]: 'привет'
```

How to work with Unicode and bytes

There is a rule called a “Unicode sandwich”:

- bytes that the program reads must be converted to Unicode (string) as early as possible
- inside the program work with Unicode
- Unicode must be converted to bytes as soon as possible before transmitting

Examples of converting between bytes and strings

Consider a few examples of working with bytes and converting bytes to string.

subprocess

Module subprocess returns the result of command as bytes:

```
In [1]: import subprocess

In [2]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'],
   ...:                         stdout=subprocess.PIPE)
   ...:

In [3]: result.stdout
Out[3]: b'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8:␣
↪icmp_seq=1 ttl=43 time=59.4 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43␣
↪time=54.4 ms\n64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=55.1 ms\n\n--- 8.8.
↪8.8 ping statistics ---\n3 packets transmitted, 3 received, 0% packet loss,␣
↪time 2002ms\nrtt min/avg/max/mdev = 54.470/56.346/59.440/2.220 ms\n'
```

(continues on next page)

(continued from previous page)

If it is necessary to work with this output further you should immediately convert it to string:

```
In [4]: output = result.stdout.decode('utf-8')

In [5]: print(output)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=59.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=55.1 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 54.470/56.346/59.440/2.220 ms
```

Module subprocess supports another conversion option - encoding parameter. If you specify it when you call run() function, the result will be as a string:

```
In [6]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'],
   ...:                         stdout=subprocess.PIPE, encoding='utf-8')
   ...:

In [7]: result.stdout
Out[7]: 'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8:␣
↪icmp_seq=1 ttl=43 time=55.5 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43␣
↪time=54.6 ms\n64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.3 ms\n\n--- 8.8.
↪8.8 ping statistics ---\n3 packets transmitted, 3 received, 0% packet loss,␣
↪time 2003ms\nrtt min/avg/max/mdev = 53.368/54.534/55.564/0.941 ms\n'

In [8]: print(result.stdout)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=55.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.3 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 53.368/54.534/55.564/0.941 ms
```

telnetlib

Depending on module, conversion between strings and bytes can be performed automatically or may be required explicitly.

For example, `telnetlib` module must pass bytes to `read_until` and `write` methods:

```
import telnetlib
import time

t = telnetlib.Telnet('192.168.100.1')

t.read_until(b'Username:')
t.write(b'cisco\n')

t.read_until(b'Password:')
t.write(b'cisco\n')
t.write(b'sh ip int br\n')

time.sleep(5)

output = t.read_very_eager().decode('utf-8')
print(output)
```

Method returns bytes, so penultimate line uses decode.

pexpect

Module `pexpect` waits for a string as an argument and returns bytes:

```
In [9]: import pexpect

In [10]: output = pexpect.run('ls -ls')

In [11]: output
Out[11]: b'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_
↪futures\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug  3 07:59 iterator_
↪generator\r\n'

In [12]: output.decode('utf-8')
Out[12]: 'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_
↪futures\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug  3 07:59 iterator_
↪generator\r\n'
```

And it also supports `encoding` parameter:

```
In [13]: output = pexpect.run('ls -ls', encoding='utf-8')

In [14]: output
Out[14]: 'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_
↪futures\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug  3 07:59 iterator_
↪generator\r\n'
```

Working with files

Until now, when working with files, the following expression was used:

```
with open(filename) as f:
    for line in f:
        print(line)
```

But actually, when you read a file you convert bytes to a string. And default encoding was used:

```
In [1]: import locale

In [2]: locale.getpreferredencoding()
Out[2]: 'UTF-8'
```

Default encoding in file:

```
In [2]: f = open('r1.txt')

In [3]: f
Out[3]: <_io.TextIOWrapper name='r1.txt' mode='r' encoding='UTF-8'>
```

When working with files it is better to specify encoding explicitly because it may differ in different operating systems:

```
In [4]: with open('r1.txt', encoding='utf-8') as f:
   ...:     for line in f:
   ...:         print(line, end='')
   ...:
!
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
```

(continues on next page)

(continued from previous page)

```
!
ip ssh version 2
!
```

Conclusion

These examples are shown here to show that different modules can treat the issue of conversion between strings and bytes differently. And different functions and methods of these modules can expect arguments and return values of different types. However, all of these items are in documentation.

Converting errors

When converting between strings and bytes it is very important to know exactly which encoding is used as well as to know the possibilities of different encodings.

For example, ASCII codec cannot encode Cyrillic:

```
In [32]: hi_unicode = 'привет'

In [33]: hi_unicode.encode('ascii')
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-33-ec69c9fd2dae> in <module>()
----> 1 hi_unicode.encode('ascii')

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-5:␣
↪ordinal not in range(128)
```

Similarly, if the string “привет” is converted to bytes and you try to convert it into a string with ascii, we will also get an error:

```
In [34]: hi_unicode = 'привет'

In [35]: hi_bytes = hi_unicode.encode('utf-8')

In [36]: hi_bytes.decode('ascii')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-36-aa0ada5e44e9> in <module>()
----> 1 hi_bytes.decode('ascii')
```

(continues on next page)

(continued from previous page)

```
UnicodeDecodeError: 'ascii' codec can't decode byte 0xd0 in position 0: ordinal␣
↪not in range(128)
```

Another version of error where different encodings are used to conversion:

```
In [37]: de_hi_unicode = 'grüezi'

In [38]: utf_16 = de_hi_unicode.encode('utf-16')

In [39]: utf_16.decode('utf-8')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-39-4b4c731e69e4> in <module>()
----> 1 utf_16.decode('utf-8')

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0: invalid␣
↪start byte
```

Having mistakes is good. They're telling what the problem is. It's worse when it's like this:

```
In [40]: hi_unicode = 'привет'

In [41]: hi_bytes = hi_unicode.encode('utf-8')

In [42]: hi_bytes
Out[42]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [43]: hi_bytes.decode('utf-16')
Out[43]: '□□□□□□'
```

Error processing

Encode and decode methods have error-processing modes that indicate how to respond to a conversion error.

Parameter errors in encode

By default `encode` uses `strict` mode - UnicodeError exception is generated when encoding errors occur. Examples of such behaviour are above.

Instead, you can use `replace` to substitute character with a question mark:

```
In [44]: de_hi_unicode = 'grüezi'

In [45]: de_hi_unicode.encode('ascii', 'replace')
Out[45]: b'gr?ezi'
```

Or namereplace to replace character with the name:

```
In [46]: de_hi_unicode = 'grüezi'

In [47]: de_hi_unicode.encode('ascii', 'namereplace')
Out[47]: b'gr\\N{LATIN SMALL LETTER U WITH DIAERESIS}ezi'
```

In addition, characters that cannot be encoded may be completely ignored:

```
In [48]: de_hi_unicode = 'grüezi'

In [49]: de_hi_unicode.encode('ascii', 'ignore')
Out[49]: b'grezi'
```

Parameter errors in decode

The decode method also uses strict mode by default and generates a UnicodeDecodeError exception.

If you change mode to ignore, as in encode, characters will simply be ignored:

```
In [50]: de_hi_unicode = 'grüezi'

In [51]: de_hi_utf8 = de_hi_unicode.encode('utf-8')

In [52]: de_hi_utf8
Out[52]: b'gr\xc3\xbcezi'

In [53]: de_hi_utf8.decode('ascii', 'ignore')
Out[53]: 'grezi'
```

Mode replace substitutes characters:

```
In [54]: de_hi_unicode = 'grüezi'

In [55]: de_hi_utf8 = de_hi_unicode.encode('utf-8')

In [56]: de_hi_utf8.decode('ascii', 'replace')
Out[56]: 'gr��ezi'
```

Further reading

Python documentation:

- What's New In Python 3: Text Vs. Data Instead Of Unicode Vs. 8-bit
- Unicode HOWTO

Articles:

- Pragmatic Unicode - article, presentation and video
- Section «Strings» of the book "Dive Into Python 3" - very well written about Unicode, encodings and how all this works in Python

Without binding to Python:

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
- The Unicode Consortium
- Unicode (Wikipedia)
- UTF-8 (Wikipedia)

17. Working with CSV, JSON, YAML files

Data serialization is about storing data in some format that is often structured.

For example, it could be:

- files in YAML or JSON format
- files in CSV format
- database

In addition, Python allows you to write down objects of language itself (this aspect is not covered, but if you are interested, look at the Pickle module).

This section covers CSV, JSON, YAML formats and chapter 25 covers databases.

YAML, JSON, CSV formats usage:

- you may have data about IP address and similar information to process in tables
  - table can be exported to CSV format and processed by Python
- software can return data in JSON format. Accordingly, by converting this data into a Python object you can work with it and do whatever you want
- YAML is very convenient to use to describe parameters
  - for example, it can be settings for different objects (IP addresses, VLANs, etc.)
  - at least knowing YAML format will be useful when using Ansible

For each of these formats, Python has a module that makes them easier to work with.

Work with CSV files

CSV (comma-separated value) - a tabular data format (for example, it may be data from a table or data from a database).

In this format, each line of a file is a line of a table. Despite format name the separator can be not only a comma. Formats with a different separator may have their own name, for example, TSV (tab separated values), however, the name CSV usually means any separators).

Example of a CSV file (sw_data.csv):

```
hostname,vendor,model,location
sw1,Cisco,3750,London
sw2,Cisco,3850,Liverpool
sw3,Cisco,3650,Liverpool
sw4,Cisco,3650,London
```

The standard Python library has a csv module that allows working with files in CSV format.

Reading

Example of reading a file in CSV format (csv_read.py file):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

The output is:

```
$ python csv_read.py
['hostname', 'vendor', 'model', 'location']
['sw1', 'Cisco', '3750', 'London']
['sw2', 'Cisco', '3850', 'Liverpool']
['sw3', 'Cisco', '3650', 'Liverpool']
['sw4', 'Cisco', '3650', 'London']
```

First list contains column names and remaining list contains the corresponding values.

Note that `csv.reader` returns an iterator:

```
In [1]: import csv

In [2]: with open('sw_data.csv') as f:
   ...:     reader = csv.reader(f)
   ...:     print(reader)
   ...:
<_csv.reader object at 0x10385b050>
```

If necessary it could be converted into a list in the following way:

```
In [3]: with open('sw_data.csv') as f:
   ...:     reader = csv.reader(f)
   ...:     print(list(reader))
   ...:
[['hostname', 'vendor', 'model', 'location'], ['sw1', 'Cisco', '3750', 'London'],␣
↪['sw2', 'Cisco', '3850', 'Liverpool'], ['sw3', 'Cisco', '3650', 'Liverpool'], [
↪'sw4', 'Cisco', '3650', 'London']]
```

Most often column headers are more convenient to get by a separate object. This can be done in this way (csv_read_headers.py file):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.reader(f)
    headers = next(reader)
    print('Headers: ', headers)
    for row in reader:
        print(row)
```

Sometimes it is more convenient to get dictionaries in which keys are column names and values are column values.

For this purpose, module has `DictReader` (csv_read_dict.py file):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row)
        print(row['hostname'], row['model'])
```

The output is:

```
$ python csv_read_dict.py
{'hostname': 'sw1', 'vendor': 'Cisco', 'model': '3750', 'location': 'London,␣
↪Globe Str 1 '}
sw1 3750
{'hostname': 'sw2', 'vendor': 'Cisco', 'model': '3850', 'location': 'Liverpool'}
sw2 3850
{'hostname': 'sw3', 'vendor': 'Cisco', 'model': '3650', 'location': 'Liverpool'}
sw3 3650
{'hostname': 'sw4', 'vendor': 'Cisco', 'model': '3650', 'location': 'London,␣
↪Grobe Str 1'}
sw4 3650
```

Note: Prior to Python 3.8 OrderedDict type was returned, not dict.

Writing

Similarly, a csv module can be used to write data to file in CSV format (csv_write.py file):

```
import csv

data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f)
    for row in data:
        writer.writerow(row)

with open('sw_data_new.csv') as f:
    print(f.read())
```

In example above, strings from list are written to the file and then the content of file is displayed on standard output stream.

The output will be as follows:

```
$ python csv_write.py
hostname,vendor,model,location
sw1,Cisco,3750,"London, Best str"
sw2,Cisco,3850,"Liverpool, Better str"
sw3,Cisco,3650,"Liverpool, Better str"
sw4,Cisco,3650,"London, Best str"
```

Note the interesting thing: strings in the last column are quoted and other values are not.

This is because all strings in the last column have a comma. And quotes indicate what is an entire string. When a comma is inside quotation marks the csv module does not perceive it as a separator.

Sometimes it's better to have all strings quoted. Of course, in this case, example is simple enough but when there are more values in the strings, the quotes indicate where value begins and ends.

Csv module allows you to control this. For all strings to be written in a CSV file with quotes you should change script this way (csv_write_quoting.py file):

```
import csv


data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
```

(continues on next page)

(continued from previous page)

```
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    for row in data:
        writer.writerow(row)

with open('sw_data_new.csv') as f:
    print(f.read())
```

Now the output is this:

```
$ python csv_write_quoting.py
"hostname","vendor","model","location"
"sw1","Cisco","3750","London, Best str"
"sw2","Cisco","3850","Liverpool, Better str"
"sw3","Cisco","3650","Liverpool, Better str"
"sw4","Cisco","3650","London, Best str"
```

Now all values are quoted. And because model number is given as a string in original list, it is quoted here as well.

Besides `writerow` method, `writerows` method is supported. It accepts any iterable object.

So, previous example can be written this way (csv_writerows.py file):

```
import csv

data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    writer.writerows(data)

with open('sw_data_new.csv') as f:
    print(f.read())
```

DictWriter

With `DictWriter` you can write dictionaries in CSV format.

In general, `DictWriter` works as `writer` but since dictionaries are not ordered it is necessary to specify the order of columns in file. The `fieldnames` option is used for this purpose (csv_write_dict.py file):

```
import csv

data = [{
    'hostname': 'sw1',
    'location': 'London',
    'model': '3750',
    'vendor': 'Cisco'
}, {
    'hostname': 'sw2',
    'location': 'Liverpool',
    'model': '3850',
    'vendor': 'Cisco'
}, {
    'hostname': 'sw3',
    'location': 'Liverpool',
    'model': '3650',
    'vendor': 'Cisco'
}, {
    'hostname': 'sw4',
    'location': 'London',
    'model': '3650',
    'vendor': 'Cisco'
}]

with open('csv_write_dictwriter.csv', 'w') as f:
    writer = csv.DictWriter(
        f, fieldnames=list(data[0].keys()), quoting=csv.QUOTE_NONNUMERIC)
    writer.writeheader()
    for d in data:
        writer.writerow(d)
```

Delimiter

Sometimes other values are used as a separator. In this case, it should be possible to tell module which separator to use.

For example, if the file uses separator `;` (sw_data2.csv file):

```
hostname;vendor;model;location
sw1;Cisco;3750;London
sw2;Cisco;3850;Liverpool
sw3;Cisco;3650;Liverpool
sw4;Cisco;3650;London
```

Simply specify which separator is used in `reader` (csv_read_delimiter.py file):

```
import csv

with open('sw_data2.csv') as f:
    reader = csv.reader(f, delimiter=';')
    for row in reader:
        print(row)
```

Work with JSON files

JSON (JavaScript Object Notation) - a text format for data storage and exchange.

JSON syntax is very similar to Python and is user-friendly.

As for CSV, Python has a module that allows easy writing and reading of data in JSON format.

Reading

File sw_templates.json:

```
{
  "access": [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable"
  ],
  "trunk": [
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
  ]
}
```

There are two methods for reading in json module:

- `json.load` - method reads JSON file and returns Python objects
- `json.loads` - method reads string in JSON format and returns Python objects

json.load

Reading JSON file to Python object (json_read_load.py file):

```
import json

with open('sw_templates.json') as f:
    templates = json.load(f)

print(templates)

for section, commands in templates.items():
    print(section)
    print('\n'.join(commands))
```

The output will be as follows:

```
$ python json_read_load.py
{'access': ['switchport mode access', 'switchport access vlan', 'switchport␣
↪nonegotiate', 'spanning-tree portfast', 'spanning-tree bpduguard enable'],
↪'trunk': ['switchport trunk encapsulation dot1q', 'switchport mode trunk',
↪'switchport trunk native vlan 999', 'switchport trunk allowed vlan']}
access
switchport mode access
switchport access vlan
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
trunk
switchport trunk encapsulation dot1q
switchport mode trunk
switchport trunk native vlan 999
switchport trunk allowed vlan
```

json.loads

Reading JSON string to Python object (json_read_loads.py file):

```
import json

with open('sw_templates.json') as f:
    file_content = f.read()
    templates = json.loads(file_content)

print(templates)

for section, commands in templates.items():
    print(section)
    print('\n'.join(commands))
```

The result will be similar to previous output.

Writing

Writing a file in JSON format is also fairly easy.

There are also two methods for writing information in JSON format in json module:

- `json.dump` - method writes Python object to file in JSON format
- `json.dumps` - method returns string in JSON format

json.dumps()

Convert object to string in JSON format (json_write_dumps.py):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

access_template = [
    'switchport mode access', 'switchport access vlan',
    'switchport nonegotiate', 'spanning-tree portfast',
    'spanning-tree bpduguard enable'
]

to_json = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.json', 'w') as f:
```

(continues on next page)

(continued from previous page)

```
    f.write(json.dumps(to_json))

with open('sw_templates.json') as f:
    print(f.read())
```

Method `json.dumps` is suitable for situations where you want to return a string in JSON format. For example, to pass it to the API.

json.dump

Write a Python object to a JSON file (json_write_dump.py file):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

access_template = [
    'switchport mode access', 'switchport access vlan',
    'switchport nonegotiate', 'spanning-tree portfast',
    'spanning-tree bpduguard enable'
]

to_json = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.json', 'w') as f:
    json.dump(to_json, f)

with open('sw_templates.json') as f:
    print(f.read())
```

When you want to write information in JSON format into a file, it is better to use dump method.

Additional parameters of write methods

Methods dump and dumps can pass additional parameters to manage the output format.

By default, these methods write information in a compact view. As a rule, when data is used by other programs, visual presentation of data is not important. If data in file needs to be read by person, this format is not very convenient to perceive. Fortunately, `json` module allows you to manage such things.

By passing additional parameters to dump method (or dumps method) you can get a more readable output (json_write_indent.py file):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

access_template = [
    'switchport mode access', 'switchport access vlan',
    'switchport nonegotiate', 'spanning-tree portfast',
    'spanning-tree bpduguard enable'
]

to_json = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.json', 'w') as f:
    json.dump(to_json, f, sort_keys=True, indent=2)

with open('sw_templates.json') as f:
    print(f.read())
```

Now the content of sw_templates.json file is:

```
{
  "access": [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable"
  ],
  "trunk": [
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
  ]
}
```

Changing data type

Another important aspect of data conversion to JSON format is that data will not always be the same type as source data in Python.

For example, when you write a tuple to JSON it becomes a list:

```
In [1]: import json

In [2]: trunk_template = ('switchport trunk encapsulation dot1q',
   ...:                   'switchport mode trunk',
   ...:                   'switchport trunk native vlan 999',
   ...:                   'switchport trunk allowed vlan')

In [3]: print(type(trunk_template))
<class 'tuple'>

In [4]: with open('trunk_template.json', 'w') as f:
   ...:     json.dump(trunk_template, f, sort_keys=True, indent=2)
   ...:

In [5]: cat trunk_template.json
[
  "switchport trunk encapsulation dot1q",
  "switchport mode trunk",
  "switchport trunk native vlan 999",
  "switchport trunk allowed vlan"
]
In [6]: templates = json.load(open('trunk_template.json'))

In [7]: type(templates)
Out[7]: list

In [8]: print(templates)
['switchport trunk encapsulation dot1q', 'switchport mode trunk', 'switchport␣
↪trunk native vlan 999', 'switchport trunk allowed vlan']
```

This is because JSON uses different data types and does not have matches for all Python data types.

Python data conversion table to JSON:

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

JSON conversion table to Python data:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Limitation on data types

It's not possible to write a dictionary in JSON format if it has tuples as a keys.

```
In [23]: to_json = {('trunk', 'cisco'): trunk_template, 'access': access_template}

In [24]: with open('sw_templates.json', 'w') as f:
    ...:     json.dump(to_json, f)
    ...:
...
TypeError: key ('trunk', 'cisco') is not a string
```

By using additional parameter you can ignore such keys:

```
In [25]: to_json = {('trunk', 'cisco'): trunk_template, 'access': access_template}

In [26]: with open('sw_templates.json', 'w') as f:
    ...:     json.dump(to_json, f, skipkeys=True)
    ...:
    ...:
```

(continues on next page)

(continued from previous page)

```
In [27]: cat sw_templates.json
{"access": ["switchport mode access", "switchport access vlan", "switchport␣
↪nonegotiate", "spanning-tree portfast", "spanning-tree bpduguard enable"]}
```

Beside that, dictionary keys can only be strings in JSON. But if numbers are used in Python dictionary there will be no error. But conversion from numbers to strings will take place:

```
In [28]: d = {1:100, 2:200}

In [29]: json.dumps(d)
Out[29]: '{"1": 100, "2": 200}'
```

Work with YAML files

YAML (YAML Ain't Markup Language) - another text format for writing data.

YAML is more human-friendly than JSON, so it is often used to describe actions in software. Playbooks in Ansible, for example.

YAML syntax

Like Python, YAML uses indents to specify the structure of document. But YAML can only use spaces and cannot use tabs. Another similarity with Python is that comments start with # and continue until the end of line.

List

A list can be written in one line:

```
[switchport mode access, switchport access vlan, switchport nonegotiate, spanning-
↪tree portfast, spanning-tree bpduguard enable]
```

Or every item in the list in separate row:

```
- switchport mode access
- switchport access vlan
- switchport nonegotiate
- spanning-tree portfast
- spanning-tree bpduguard enable
```

When a list is written in such a block, each row must start with "- " (minus and space) and all lines in the list must be at the same indentation level.

Dictionary

A dictionary can also be written in one line:

```
{vlan: 100, name: IT}
```

Or a block:

```
vlan: 100
name: IT
```

Strings

Strings in YAML don't have to be quoted. This is convenient, but sometimes quotes should be used. For example, when a special character (special for YAML) is used in a string.

This line, for example, should be quoted to be correctly understood by YAML:

```
command: "sh interface | include Queueing strategy:"
```

Combination of elements

A dictionary with two keys: access and trunk. Values that correspond to these keys - command lists:

```
access:
- switchport mode access
- switchport access vlan
- switchport nonegotiate
- spanning-tree portfast
- spanning-tree bpduguard enable

trunk:
- switchport trunk encapsulation dot1q
- switchport mode trunk
- switchport trunk native vlan 999
- switchport trunk allowed vlan
```

List of dictionaries:

```
- BS: 1550
  IT: 791
  id: 11
  name: Liverpool
```

(continues on next page)

(continued from previous page)

```
  to_id: 1
  to_name: LONDON
- BS: 1510
  IT: 793
  id: 12
  name: Bristol
  to_id: 1
  to_name: LONDON
- BS: 1650
  IT: 892
  id: 14
  name: Coventry
  to_id: 2
  to_name: Manchester
```

PyYAML module

Python uses a PyYAML module to work with YAML. It is not part of the standard module library, so it needs to be installed:

```
pip install pyyaml
```

Work with it is similar to csv and json modules.

Reading from YAML

Converting data from YAML file to Python objects (info.yaml file):

```
- BS: 1550
  IT: 791
  id: 11
  name: Liverpool
  to_id: 1
  to_name: LONDON
- BS: 1510
  IT: 793
  id: 12
  name: Bristol
  to_id: 1
  to_name: LONDON
- BS: 1650
```

(continues on next page)

(continued from previous page)

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

Reading from YAML (yaml_read.py file):

```
import yaml
from pprint import pprint

with open('info.yaml') as f:
    templates = yaml.safe_load(f)

pprint(templates)
```

The result is:

```
$ python yaml_read.py
[{'BS': 1550,
  'IT': 791,
  'id': 11,
  'name': 'Liverpool',
  'to_id': 1,
  'to_name': 'LONDON'},
 {'BS': 1510,
  'IT': 793,
  'id': 12,
  'name': 'Bristol',
  'to_id': 1,
  'to_name': 'LONDON'},
 {'BS': 1650,
  'IT': 892,
  'id': 14,
  'name': 'Coventry',
  'to_id': 2,
  'to_name': 'Manchester'}]
```

YAML format is very convenient for storing different parameters, especially if they are filled manually.

Writing to YAML

Write Python objects to YAML (yaml_write.py file):

```
import yaml

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

access_template = [
    'switchport mode access', 'switchport access vlan',
    'switchport nonegotiate', 'spanning-tree portfast',
    'spanning-tree bpduguard enable'
]

to_yaml = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.yaml', 'w') as f:
    yaml.dump(to_yaml, f, default_flow_style=False)

with open('sw_templates.yaml') as f:
    print(f.read())
```

File sw_templates.yaml:

```
access:
- switchport mode access
- switchport access vlan
- switchport nonegotiate
- spanning-tree portfast
- spanning-tree bpduguard enable
trunk:
- switchport trunk encapsulation dot1q
- switchport mode trunk
- switchport trunk native vlan 999
- switchport trunk allowed vlan
```

Further reading

In this section only basic read and write operations were covered with no additional parameters. More details can be found in the module documentation.

- CSV
- JSON

- YAML

In addition, PyMOTW has very good description of all Python modules that are part of the standard library (installed with Python):

- CSV
- JSON

Example of getting JSON data via Github API:

- Example of working with Github API with requests
- Writing Cyrillic and other non-ASCI characters in JSON format

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 17.1

Create the write_dhcp_snooping_to_csv function, which processes the output of the show dhcp snooping binding command from different files and writes the processed data to the csv file.

Function arguments:

- filenames - list of filenames with "show dhcp snooping binding" command output
- output - the name of the csv file into which the result will be written

The function returns None.

For example, if a list with one file sw3_dhcp_snooping.txt was passed as an argument:

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  ------------
↪--------
00:E9:BC:3F:A6:50   100.1.1.6        76260       dhcp-snooping   3    ␣
↪FastEthernet0/20
00:E9:22:11:A6:50   100.1.1.7        76260       dhcp-snooping   3    ␣
↪FastEthernet0/21
Total number of bindings: 2
```

The resulting csv file should contain the following content:

```
switch,mac,ip,vlan,interface
sw3,00:E9:BC:3F:A6:50,100.1.1.6,3,FastEthernet0/20
sw3,00:E9:22:11:A6:50,100.1.1.7,3,FastEthernet0/21
```

The first column in the csv file, the name of the switch, must be obtained from the file name, the rest - from the contents in the files.

Check the function on the contents of the files sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt.

Task 17.2

In this task you need:

- take the contents of several files with the output of the sh version command
- parse command output using regular expressions and get device information
- write this information to a file in CSV format

To complete the task, you need to create two functions.

parse_sh_version function:

- expects the output of the sh version command as an argument in single string (not a filename)
- processes output using regular expressions
- returns a tuple of three elements:
  - ios - “12.4(5)T”
  - image - “flash:c2800-advipservicesk9-mz.124-5.T.bin”
  - uptime - “5 days, 3 hours, 3 minutes”

The write_inventory_to_csv function must have two parameters:

- data_filenames - expects a list of filenames as an argument with the output of sh version
- csv_filename - expects as an argument the name of a file (for example, routers_inventory.csv) to which information will be written in CSV format

write_inventory_to_csv function writes the contents to a file, in CSV format and returns nothing.

The write_inventory_to_csv function should do the following:

- process information from each file with sh version output: sh_version_r1.txt, sh_version_r2.txt, sh_version_r3.txt
- using the parse_sh_version function, ios, image, uptime information should be obtained from each output
- from the file name you need to get the hostname
- after that all information should be written to a CSV file

The routers_inventory.csv file should have the following columns (in this order): hostname, ios, image, uptime

The code below has created a list of files using the glob module. You can uncomment the print(sh_version_files) line to see the content of the list.

In addition, a list of headers has been created, which should be written to CSV.

```
import glob

sh_version_files = glob.glob("sh_vers*")
#print(sh_version_files)

headers = ["hostname", "ios", "image", "uptime"]
```

Task 17.3

Create a function parse_sh_cdp_neighbors that processes the output of the show cdp neighbors command.

The function expects, as an argument, the output of the command as a single string (not a filename). The function should return a dictionary that describes the connections between devices.

For example, if the following output was passed as an argument:

```
R4>show cdp neighbors

Device ID    Local Intrfce   Holdtme     Capability       Platform    Port ID
R5           Fa 0/1          122           R S I           2811       Fa 0/1
R6           Fa 0/2          143           R S I           2811       Fa 0/0
```

The function should return a dictionary like this:

```
{"R4": {"Fa 0/1": {"R5": "Fa 0/1"},
        "Fa 0/2": {"R6": "Fa 0/0"}}}
```

Interfaces must be written with a space. That is, so Fa 0/0, and not so Fa0/0.

Check the function on the contents of the sh_cdp_n_sw1.txt file

Task 17.3a

Create a generate_topology_from_cdp function that processes the show cdp neighbor command output from multiple files and writes the resulting topology to a single dictionary.

The generate_topology_from_cdp function must be created with parameters:

- list_of_files - list of files from which to read the output of the sh cdp neighbor command
- save_to_filename is the name of the YAML file where the topology will be saved.
  - default is None. By default, the topology is not saved to a file.
  - topology is saved only if save_to_filename is file name as argument

The function should return a dictionary that describes the connections between devices, regardless of whether the topology is saved to a file.

Dictionary example:

```
{"R4": {"Fa 0/1": {"R5": "Fa 0/1"},
        "Fa 0/2": {"R6": "Fa 0/0"}},
 "R5": {"Fa 0/1": {"R4": "Fa 0/1"}},
 "R6": {"Fa 0/0": {"R4": "Fa 0/2"}}}
```

Interfaces must be written with a space. That is, so Fa 0/0, and not so Fa0/0.

Check the work of the generate_topology_from_cdp function on the list of files:

- sh_cdp_n_sw1.txt
- sh_cdp_n_r1.txt
- sh_cdp_n_r2.txt
- sh_cdp_n_r3.txt
- sh_cdp_n_r4.txt
- sh_cdp_n_r5.txt
- sh_cdp_n_r6.txt

Check the operation of the save_to_filename parameter and write the resulting dictionary to the topology.yaml file. You will need it in the next task.

Task 17.3b

Create a transform_topology function that converts the topology to a format suitable for the draw_topology function.

The function expects a YAML filename as an argument in which the topology is stored.

The function must read data from the YAML file, transform it accordingly, so that the function returns a dictionary of the following form:

```
{("R4", "Fa 0/1"): ("R5", "Fa 0/1"),
 ("R4", "Fa 0/2"): ("R6", "Fa 0/0")}
```

The transform_topology function should not only change the format of the topology representation, but also remove the “duplicate” connections (they are best seen in the diagram that the draw_topology function generates from the draw_network_graph.py file). “Duplicate” connections are connections of this kind:

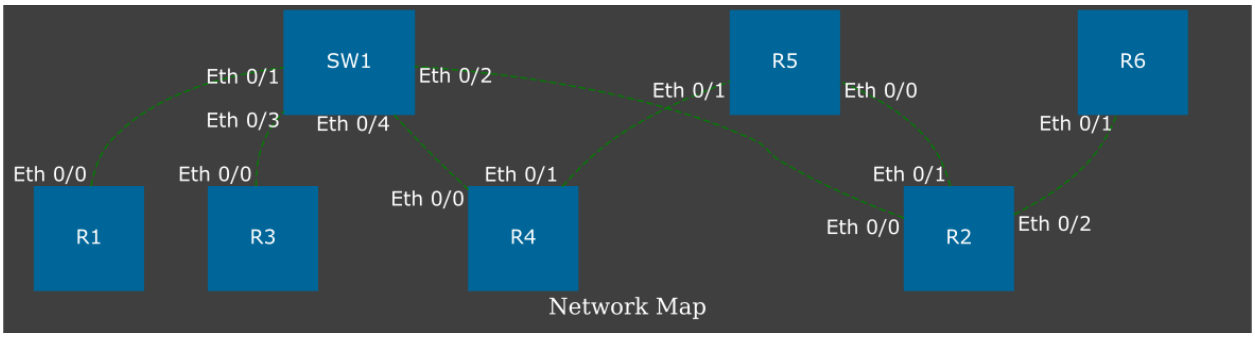
```
("R1", "Eth0/0"): ("SW1", "Eth0/1")
("SW1", "Eth0/1"): ("R1", "Eth0/0")
```

Due to the fact that the same link is described twice, there will be extra connections on the diagram. The task is to leave only one of these links in the final dictionary, does not matter which one.

Check the operation of the function on the topology.yaml file (must be created in task 17.3a). Based on the resulting dictionary, you need to generate a topology image using the draw_topology function. Do not copy draw_topology function code from draw_network_graph.py file.

The result should look the same as the diagram in the task_17_3b_topology.svg file:

- Interfaces must be written with a space Fa 0/0
- The arrangement of devices on the diagram may be different
- Connections must match the diagram
- There should be no “duplicate” links on the diagram



Note: To complete this task, graphviz must be installed: apt-get install graphviz

And a python module to work with graphviz: pip install graphviz

Task 17.4

Create function write_last_log_to_csv.

Function arguments:

- source_log - the name of the csv file from which the data is read (mail_log.csv)
- output - the name of the csv file into which the result will be written

The function returns None.

The write_last_log_to_csv function processes the csv file mail_log.csv. The mail_log.csv file contains the logs of the username change. User cannot change email, only username.

The write_last_log_to_csv function should select from the mail_log.csv file only the most recent entries for each user and write them to another csv file. In the output file, the first line should be the column headers as in the source_log file.

For some users, there is only one record, and then it is necessary to write to the final file only her. For some users, there are multiple entries with different names. For example, a user with email c3po@gmail.com changed his username several times:

```
C=3PO,c3po@gmail.com,16/12/2019 17:10
C3PO,c3po@gmail.com,16/12/2019 17:15
C-3PO,c3po@gmail.com,16/12/2019 17:24
```

Of these three records, only one should be written to the final file - the most recent:

```
C-3PO,c3po@gmail.com,16/12/2019 17:24
```

It is convenient to use datetime objects from the datetime module for comparing dates. To make it easier to work with dates, the convert_str_to_datetime function has been created - it converts a date string in the format 11/10/2019 14:05 into a datetime object. The resulting datetime objects can be compared with each other. The second function, convert_datetime_to_str, does the opposite — it turns a datetime object into a string.

It is not necessary to use the functions convert_str_to_datetime and convert_datetime_to_str

```
import datetime


def convert_str_to_datetime(datetime_str):
    """
    Converts a date string formatted as 11/10/2019 14:05 to a datetime object.
    """
    return datetime.datetime.strptime(datetime_str, "%d/%m/%Y %H:%M")


def convert_datetime_to_str(datetime_obj):
    """
    Converts a datetime object to a date string in the format 11/10/2019 14:05.
    """
    return datetime.datetime.strftime(datetime_obj, "%d/%m/%Y %H:%M")
```

7

V. Working with network devices

In this part, the following topics are discussed:

- SSH and Telnet connection
- simultaneous connection to multiple devices
- creating configuration templates with Jinja2
- command output processing with TextFSM

18. Connection to network devices

This section discusses how to connect to network devices via:

- SSH
- Telnet

Python has several modules that allow you to connect to network devices and execute commands:

- `pexpect` - an implementation of `expect` in Python
  - this module allows working with any interactive session: ssh, telnet, sftp, etc.
  - in addition, it makes possible to execute different commands in OS (this can also be done with other modules)
  - while pexpect may be less user-friendly than other modules, it implements a more general functionality and allows it to be used in situations where other modules do not work
- `telnetlib` - this module allows you connecting via Telnet
  - `netmiko` version >= 1.0 also has Telnet support, so if `netmiko` supports the network devices you use, it is more convenient to use it
- `paramiko` - his module allows you connecting via SSHv2
  - it is more convenient to use than `pexpect` but with narrower functionality (only supports SSH)
- `netmiko` - module that simplifies the use of paramiko for network devices
  - `netmiko` is a "wrapper" which is oriented to work with network devices
- `scrapli` - is a module that allows you to connect to network equipment using Telnet, SSH or NETCONF

This section covers all five modules and describes how to connect to several devices in parallel. Three routers are used in section examples. There are no requirements for them, only configured SSHv2 and Telnet.

Parameters used in these section:

- user: cisco
- password: cisco
- password for enable mode: cisco
- SSH version 2, Telnet
- IP addresses: 192.168.100.1, 192.168.100.2, 192.168.100.3

Password input

During manual connection to device the password is also manually entered.

When automating connection it is necessary to decide how password will be transmitted:

- Request password at start of the script and read user input. Disadvantage is that you can see which characters user is typing
- Write login and password in some file (it's not secure).

As a rule, the same user uses the same login and password to connect to devices. And usually it's enough to request login and password at the start of the script and then use them to connect to different devices.

Unfortunately, if you use `input` the typed password will be visible. But it is better if no characters are displayed when entering a password.

Module getpass

Module getpass allows you to request a password without displaying input characters:

```
In [1]: import getpass

In [2]: password = getpass.getpass()
Password:

In [3]: print(password)
testpass
```

Environment variables

Another way to store a password (or even a username) is by environment variables.

For example, login and password are written in variables:

```
$ export SSH_USER=user
$ export SSH_PASSWORD=userpass
```

And then Python reads values to variables in the script:

```
import os

USERNAME = os.environ.get('SSH_USER')
PASSWORD = os.environ.get('SSH_PASSWORD')
```

Module pexpect

Module pexpect allows to automate interactive connections such as:

- telnet
- ssh
- ftp

Note: Pexpect is an implementation of expect in Python.

First, pexpect module needs to be installed:

```
pip install pexpect
```

The logic of pexpect is:

- some program is running
- pexpect expects a certain output (prompt, password request, etc.)
- after receiving the output, it sends commands/data
- last two actions are repeated as many times as necessary

At the same time, pexpect does not implement utilities but uses ready-made ones.

pexpect.spawn

Class spawn allows you to interact with called program by sending data and waiting for a response.

For example, you can initiate SSH connecton:

```
In [5]: ssh = pexpect.spawn('ssh cisco@192.168.100.1')
```

After executing this line, connection is established. Now you must specify which line to expect. In this case, wait for password request:

```
In [6]: ssh.expect('[Pp]assword')
Out[6]: 0
```

Note how line that pexpect expects is written as `[Pp]assword`. This is a regex that describes a `password` or `Password` string. That is, `expect` method can be used to pass a regex as an argument.

Method `expect` returned number 0 as a result of the work. This number indicates that a match has been found and that this element with index zero. Index appears here because you can pass a list of strings. For example, you can pass a list with two elements:

```
In [7]: ssh = pexpect.spawn('ssh cisco@192.168.100.1')

In [8]: ssh.expect(['password', 'Password'])
Out[8]: 1
```

Note that it now returns 1. This means that `Password` word matched.

Now you can send password using `sendline` method:

```
In [9]: ssh.sendline('cisco')
Out[9]: 6
```

Method `sendline` sends a string, automatically adds a new line character to it based on the value of `os.linesep` and then returns a number indicating how many bytes were written.

Note: Pexpect has several methods for sending commands, not just sendline.

To get into enable mode expect-sendline cycle repeats:

```
In [10]: ssh.expect('[>#]')
Out[10]: 0

In [11]: ssh.sendline('enable')
Out[11]: 7

In [12]: ssh.expect('[Pp]assword')
Out[12]: 0

In [13]: ssh.sendline('cisco')
Out[13]: 6

In [14]: ssh.expect('[>#]')
Out[14]: 0
```

Now we can send a command:

```
In [15]: ssh.sendline('sh ip int br')
Out[15]: 13
```

After sending the command, pexpect must be told until what point to read the output. We specify that it should read untill #:

```
In [16]: ssh.expect('#')
Out[16]: 0
```

Command output is in before attribute:

```
In [17]: ssh.before
Out[17]: b'sh ip int br\r\nInterface                  IP-Address      OK? Method␣
↪Status                Protocol\r\nEthernet0/0               192.168.100.1  ␣
↪YES NVRAM  up                    up      \r\nEthernet0/1               192.168.
↪200.1   YES NVRAM  up                    up      \r\nEthernet0/2                ␣
↪19.1.1.1        YES NVRAM  up                    up      \r\nEthernet0/3       ␣
↪        192.168.230.1   YES NVRAM  up                    up      \r\nEthernet0/
↪3.100            10.100.0.1      YES NVRAM  up                    up      ␣
↪\r\nEthernet0/3.200            10.200.0.1      YES NVRAM  up                  ␣␣
↪up      \r\nEthernet0/3.300            10.30.0.1       YES NVRAM  up          ␣
↪        up      \r\nR1'
```

Since the result is displayed as a sequence of bytes you should convert it to a string:

```
In [18]: show_output = ssh.before.decode('utf-8')

In [19]: print(show_output)
sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                19.1.1.1        YES NVRAM  up                    up
Ethernet0/3                192.168.230.1   YES NVRAM  up                    up
Ethernet0/3.100            10.100.0.1      YES NVRAM  up                    up
Ethernet0/3.200            10.200.0.1      YES NVRAM  up                    up
Ethernet0/3.300            10.30.0.1       YES NVRAM  up                    up
R1
```

Session ends with a close call:

```
In [20]: ssh.close()
```

Special characters in shell

Pexpect does not interpret special shell characters such as >, |, *.

For example, in order make command `ls -ls | grep SUMMARY` work, shell must be run as follows:

```
In [1]: import pexpect

In [2]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep pexpect"')
```

(continues on next page)

(continued from previous page)

```
In [3]: p.expect(pexpect.EOF)
Out[3]: 0

In [4]: print(p.before)
b'4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py\r\n'

In [5]: print(p.before.decode('utf-8'))
4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py
```

pexpect.EOF

In the previous example we met pexpect.EOF.

Note: EOF — end of file

This is a special value that allows you to react to the end of a command or session that has been run in spawn.

When calling `ls -ls` command, pexpect does not receive an interactive session. Command is simply executed and that ends its work.

Therefore, if you run this command and set prompt in expect, there is an error:

```
In [5]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep SUMMARY"')

In [6]: p.expect('nattaur')
---------------------------------------------------------------------------
EOF                                       Traceback (most recent call last)
<ipython-input-9-9c71777698c2> in <module>()
----> 1 p.expect('nattaur')
...
```

If EOF passed to expect, there will be no error.

Method pexpect.expect

In pexpect.expect as a value can be used:

- regex
- EOF - this template allows you to react to EOF exception

- TIMEOUT - timeout exception (default timeout = 30 seconds)
- compiled regex

Another very useful feature of pexpect.expect is that you can pass not a single value, but a list.

For example:

```
In [7]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep netmiko"')

In [8]: p.expect(['py3_convert', pexpect.TIMEOUT, pexpect.EOF])
Out[8]: 2
```

Here are some important points:

- when pexpect.expect is called with a list, you can specify different expected strings
- apart strings, exceptions also can be specified
- pexpect.expect returns number of element that matched
  - in this case number 2 because EOF exception is number two in the list
- with this format you can make branches in the program depending on the element which had a match

Example of pexpect use

Example of using pexpect when connecting to equipment and passing show command (file 1_pexpect.py):

```
import pexpect
import re
from pprint import pprint


def send_show_command(ip, username, password, enable, commands, prompt="#"):
    with pexpect.spawn(f"ssh {username}@{ip}", timeout=10, encoding="utf-8") as␣
↪ssh:
        ssh.expect("[Pp]assword")
        ssh.sendline(password)
        enable_status = ssh.expect([">", "#"])
        if enable_status == 0:
            ssh.sendline("enable")
            ssh.expect("[Pp]assword")
            ssh.sendline(enable)
            ssh.expect(prompt)
```

(continues on next page)

(continued from previous page)

```
        ssh.sendline("terminal length 0")
        ssh.expect(prompt)

        result = {}
        for command in commands:
            ssh.sendline(command)
            match = ssh.expect([prompt, pexpect.TIMEOUT, pexpect.EOF])
            if match == 1:
                print(
                    f"Symbol {prompt} is not found in output. Resulting output is␣
↪written to
                    dictionary")
            if match == 2:
                print("Connection was terminated by server")
                return result
            else:
                output = ssh.before
                result[command] = output.replace("\r\n", "\n")
        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh clock", "sh int desc"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", commands)
        pprint(result, width=120)
```

This part of function is responsible for switching to enable mode:

```
enable_status = ssh.expect([">", "#"])
if enable_status == 0:
    ssh.sendline("enable")
    ssh.expect("[Pp]assword")
    ssh.sendline(enable)
    ssh.expect(prompt)
```

If `ssh.expect([">", "#"])` does not return index 0, it means that connection was not switched to enable mode automaticaly and it should be done separately. If index 1 is returned, then we are already in enable mode, for example, because device is configured with privilege 15.

Another interesting point about this function:

```
for command in commands:
    ssh.sendline(command)
    match = ssh.expect([prompt, pexpect.TIMEOUT, pexpect.EOF])
    if match == 1:
        print(
            f"Symbol {prompt} is not found in output. Resulting output is written␣
↪to dictionary"
        )
    if match == 2:
        print("Connection was terminated by server")
        return result
    else:
        output = ssh.before
        result[command] = output.replace("\r\n", "\n")
return result
```

Here commands are sent in turn and expect waits for three options: prompt, timeout or EOF. If expect method didn't catch #, value 1 will be returned and in this case a message is displayed, that symbol was not found. But in both cases, when a match is found or timeout the resulting output is written to dictionary. Thus, you can see what was received from device, even if prompt is not found.

Output after script execution:

```
{'sh clock': 'sh clock\n*13:13:47.525 UTC Sun Jul 19 2020\n',
 'sh int desc': 'sh int desc\n'
                'Interface          Status         Protocol Description\n'
                'Et0/0              up             up       \n'
                'Et0/1              up             up       \n'
                'Et0/2              up             up       \n'
                'Et0/3              up             up       \n'
                'Lo22               up             up       \n'
                'Lo33               up             up       \n'
                'Lo45               up             up       \n'
                'Lo55               up             up       \n'}
{'sh clock': 'sh clock\n*13:13:50.450 UTC Sun Jul 19 2020\n',
 'sh int desc': 'sh int desc\n'
                'Interface          Status         Protocol Description\n'
                'Et0/0              up             up       \n'
                'Et0/1              up             up       \n'
                'Et0/2              admin down     down     \n'
                'Et0/3              admin down     down     \n'
                'Lo0                up             up       \n'
                'Lo9                up             up       \n'
                'Lo19               up             up       \n'
```

(continues on next page)

(continued from previous page)

```
                'Lo33               up              up       \n'
                'Lo100              up              up       \n'}
{'sh clock': 'sh clock\n*13:13:53.360 UTC Sun Jul 19 2020\n',
 'sh int desc': 'sh int desc\n'
                'Interface          Status          Protocol Description\n'
                'Et0/0              up              up       \n'
                'Et0/1              up              up       \n'
                'Et0/2              admin down      down     \n'
                'Et0/3              admin down      down     \n'
                'Lo33               up              up       \n'}
```

Working with pexpect without disabling commands pagination

Sometimes the output of a command is very large and cannot be read completely or device is not makes it possible to disable pagination. In this case, a slightly different approach is needed.

Note: The same task will be repeated for other modules in this section.

Example of using pexpect to work with paginated output of show command (1_pexpect_more.py file):

```
import pexpect
import re
from pprint import pprint


def send_show_command(ip, username, password, enable, command, prompt="#"):
    with pexpect.spawn(f"ssh {username}@{ip}", timeout=10, encoding="utf-8") as␣
↪ssh:
        ssh.expect("[Pp]assword")
        ssh.sendline(password)
        enable_status = ssh.expect([">", "#"])
        if enable_status == 0:
            ssh.sendline("enable")
            ssh.expect("[Pp]assword")
            ssh.sendline(enable)
            ssh.expect(prompt)

        ssh.sendline(command)
        output = ""
```

(continues on next page)

(continued from previous page)

```
        while True:
            match = ssh.expect([prompt, "--More--", pexpect.TIMEOUT])
            page = ssh.before.replace("\r\n", "\n")
            page = re.sub(" +\x08+ +\x08+", "\n", page)
            output += page
            if match == 0:
                break
            elif match == 1:
                ssh.send(" ")
            else:
                print("Error: timeout")
                break
        output = re.sub("\n +\n", "\n", output)
        return output


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", "sh run")
        with open(f"{ip}_result.txt", "w") as f:
            f.write(result)
```

Now after sending the command, expect method waits for another option `--More--` - sign, that there will be one more page further. Since it's not known in advance how many pages will be in the output, reading is performed in a loop `while True`. Loop is interrupted if prompt is met # or no prompt appears within 10 seconds or `--More--`.

If `--More--` is met, pages are not over yet and you have to scroll through the next one. In Cisco, you need to press space bar to do this (without new line). Therefore, send method is used here, not `sendline` - sendline automatically adds a new line character.

This string `page = re.sub(" +\x08+ +\x08+", "\n", page)` removes backspace symbols which are around `--More--` so they don't end up in the final output.

Module telnetlib

Module telnetlib is part of standard Python library. This is telnet client implementation.

Note: It is also possible to connect via telnet using pexpect. The advantage of telnetlib is that this module is part of standard Python library.

Telnetlib resembles pexpect but has several differences. The most notable difference is that telnetlib requires a pass of a byte string, rather than normal one.

Connection is performed as follows:

```
In [1]: telnet = telnetlib.Telnet('192.168.100.1')
```

Method read_until

Method read_until specifies till which line the output should be read. However, as an argument, it is necessary to pass bytes, not the usual string:

```
In [2]: telnet.read_until(b'Username')
Out[2]: b'\r\n\r\nUser Access Verification\r\n\r\nUsername'
```

Method read_until returns everything it has read before specified string.

Method write

The write method is used to transmit data. You must pass a byte string as an argument:

```
In [3]: telnet.write(b'cisco\n')
```

Read output till Password and pass the password:

```
In [4]: telnet.read_until(b'Password')
Out[4]: b': cisco\r\nPassword'

In [5]: telnet.write(b'cisco\n')
```

You can now specify what should be read untill prompt and then send the command:

```
In [6]: telnet.read_until(b'>')
Out[6]: b': \r\nR1>'

In [7]: telnet.write(b'sh ip int br\n')
```

After sending a command, you can continue to use read_until method:

```
In [8]: telnet.read_until(b'>')
Out[8]: b'sh ip int br\r\nInterface                  IP-Address      OK? Method␣
↪Status                Protocol\r\nEthernet0/0                192.168.100.1  ␣
↪YES NVRAM  up                    up      \r\nEthernet0/1                192.168.
↪200.1   YES NVRAM  up                    up      \r\nEthernet0/2                ␣
↪19.1.1.1        YES NVRAM  up                    up      \r\nEthernet0/3        ␣
↪        192.168.230.1   YES NVRAM  up                    up      \r\nEthernet0/
↪3.100           10.100.0.1      YES NVRAM  up                    up             ␣
↪\r\nEthernet0/3.200           10.200.0.1      YES NVRAM  up                    ␣
↪up      \r\nEthernet0/3.300           10.30.0.1       YES NVRAM  up             ␣
↪        up      \r\nR1>'
```

(continues on next page)

(continued from previous page)

Method read_very_eager

Or use another read method read_very_eager. When using read_very_eager method, you can send multiple commands and then read all available output:

```
In [9]: telnet.write(b'sh arp\n')

In [10]: telnet.write(b'sh clock\n')

In [11]: telnet.write(b'sh ip int br\n')

In [12]: all_result = telnet.read_very_eager().decode('utf-8')

In [13]: print(all_result)
sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.30.0.1               -   aabb.cc00.6530  ARPA   Ethernet0/3.300
Internet  10.100.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.100
Internet  10.200.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.200
Internet  19.1.1.1                -   aabb.cc00.6520  ARPA   Ethernet0/2
Internet  192.168.100.1           -   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2         124   aabb.cc00.6600  ARPA   Ethernet0/0
Internet  192.168.100.3         143   aabb.cc00.6700  ARPA   Ethernet0/0
Internet  192.168.100.100       160   aabb.cc80.c900  ARPA   Ethernet0/0
Internet  192.168.200.1           -   0203.e800.6510  ARPA   Ethernet0/1
Internet  192.168.200.100        13   0800.27ac.16db  ARPA   Ethernet0/1
Internet  192.168.230.1           -   aabb.cc00.6530  ARPA   Ethernet0/3
R1>sh clock
*19:18:57.980 UTC Fri Nov 3 2017
R1>sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                19.1.1.1        YES NVRAM  up                    up
Ethernet0/3                192.168.230.1   YES NVRAM  up                    up
Ethernet0/3.100            10.100.0.1      YES NVRAM  up                    up
Ethernet0/3.200            10.200.0.1      YES NVRAM  up                    up
Ethernet0/3.300            10.30.0.1       YES NVRAM  up                    up
R1>
```

Warning: You should always set `time.sleep(n)` before using `read_very_eager`.

With `read_until` will be a slightly different approach. You can execute the same three commands, but then get the output one by one because of reading till prompt string:

```
In [14]: telnet.write(b'sh arp\n')

In [15]: telnet.write(b'sh clock\n')

In [16]: telnet.write(b'sh ip int br\n')

In [17]: telnet.read_until(b'>')
Out[17]: b'sh arp\r\nProtocol  Address          Age (min)  Hardware Addr   Type
↪Interface\r\nInternet  10.30.0.1               -   aabb.cc00.6530  ARPA
↪Ethernet0/3.300\r\nInternet  10.100.0.1             -   aabb.cc00.6530  ARPA
↪Ethernet0/3.100\r\nInternet  10.200.0.1             -   aabb.cc00.6530  ARPA
↪Ethernet0/3.200\r\nInternet  19.1.1.1               -   aabb.cc00.6520  ARPA
↪Ethernet0/2\r\nInternet  192.168.100.1          -   aabb.cc00.6500  ARPA
↪Ethernet0/0\r\nInternet  192.168.100.2        126   aabb.cc00.6600  ARPA
↪Ethernet0/0\r\nInternet  192.168.100.3        145   aabb.cc00.6700  ARPA
↪Ethernet0/0\r\nInternet  192.168.100.100      162   aabb.cc80.c900  ARPA
↪Ethernet0/0\r\nInternet  192.168.200.1          -   0203.e800.6510  ARPA
↪Ethernet0/1\r\nInternet  192.168.200.100       15   0800.27ac.16db  ARPA
↪Ethernet0/1\r\nInternet  192.168.230.1          -   aabb.cc00.6530  ARPA
↪Ethernet0/3\r\nR1>'

In [18]: telnet.read_until(b'>')
Out[18]: b'sh clock\r\n*19:20:39.388 UTC Fri Nov 3 2017\r\nR1>'

In [19]: telnet.read_until(b'>')
Out[19]: b'sh ip int br\r\nInterface                  IP-Address      OK? Method
↪Status                Protocol\r\nEthernet0/0               192.168.100.1
↪YES NVRAM  up                    up      \r\nEthernet0/1               192.168.
↪200.1   YES NVRAM  up                    up      \r\nEthernet0/2
↪19.1.1.1        YES NVRAM  up                    up      \r\nEthernet0/3
↪        192.168.230.1   YES NVRAM  up                    up      \r\nEthernet0/
↪3.100            10.100.0.1      YES NVRAM  up                    up
↪\r\nEthernet0/3.200           10.200.0.1      YES NVRAM  up
↪up      \r\nEthernet0/3.300           10.30.0.1       YES NVRAM  up
↪        up      \r\nR1>'
```

read_until vs read_very_eager

An important difference between read_until and read_very_eager is how they react to the lack of output.

Method read_until waits for a certain string. By default, if it does not exist, method will "freeze". Timeout option allows you to specify how long to wait for the desired string:

```
In [20]: telnet.read_until(b'>', timeout=5)
Out[20]: b''
```

If no string appears during the specified time, an empty string is returned.

Method read_very_eager simply returns an empty string if there is no output:

```
In [21]: telnet.read_very_eager()
Out[21]: b''
```

Method expect

Method expect allows you to specify a list with regular expressions. It works like pexpect but telnetlib always has to pass a list of regular expressions.

You can then pass byte strings or compiled regular expressions:

```
In [22]: telnet.write(b'sh clock\n')

In [23]: telnet.expect([b'[>#]'])
Out[23]:
(0,
 <_sre.SRE_Match object; span=(46, 47), match=b'>'>,
 b'sh clock\r\n*19:35:10.984 UTC Fri Nov 3 2017\r\nR1>')
```

Method expect returns tuple of their three elements:

- index of matched expression
- object Match
- byte string that contains everything read till regular expression including regular expression

Accordingly, if necessary you can continue working with these elements:

```
In [24]: telnet.write(b'sh clock\n')

In [25]: regex_idx, match, output = telnet.expect([b'[>#]'])
```

(continues on next page)

(continued from previous page)

```
In [26]: regex_idx
Out[26]: 0

In [27]: match.group()
Out[27]: b'>'

In [28]: match
Out[28]: <_sre.SRE_Match object; span=(46, 47), match=b'>'>

In [29]: match.group()
Out[29]: b'>'

In [30]: output
Out[30]: b'sh clock\r\n*19:37:21.577 UTC Fri Nov 3 2017\r\nR1>'

In [31]: output.decode('utf-8')
Out[31]: 'sh clock\r\n*19:37:21.577 UTC Fri Nov 3 2017\r\nR1>'
```

Method close

Method `close` closes connection but it's better to open and close connection using context manager:

```
In [32]: telnet.close()
```

Note: Using Telnet object as context manager added in version 3.6

Telnetlib usage example

Working principle of telnetlib resembles pexpect, so the example below should be clear (2_telnetlib.py):

```
import telnetlib
import time
from pprint import pprint


def to_bytes(line):
    return f"{line}\n".encode("utf-8")

```

(continues on next page)

(continued from previous page)

```
def send_show_command(ip, username, password, enable, commands):
    with telnetlib.Telnet(ip) as telnet:
        telnet.read_until(b"Username")
        telnet.write(to_bytes(username))
        telnet.read_until(b"Password")
        telnet.write(to_bytes(password))
        index, m, output = telnet.expect([b">", b"#"])
        if index == 0:
            telnet.write(b"enable\n")
            telnet.read_until(b"Password")
            telnet.write(to_bytes(enable))
            telnet.read_until(b"#", timeout=5)
        telnet.write(b"terminal length 0\n")
        telnet.read_until(b"#", timeout=5)
        time.sleep(3)
        telnet.read_very_eager()

        result = {}
        for command in commands:
            telnet.write(to_bytes(command))
            output = telnet.read_until(b"#", timeout=5).decode("utf-8")
            result[command] = output.replace("\r\n", "\n")
        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh ip int br", "sh arp"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", commands)
        pprint(result, width=120)
```

Since bytes need to be passed to `write` method and new line character should be added each time, a small function `to_bytes` is created that does the conversion to bytes and adds a new line.

Script execution:

```
{'sh int desc': 'sh int desc\n'
                'Interface            Status         Protocol Description\n'
                'Et0/0                up             up       \n'
                'Et0/1                up             up       \n'
                'Et0/2                up             up       \n'
```

(continues on next page)

(continued from previous page)

```
                'Et0/3                up             up       \n'
                'R1#',
 'sh ip int br': 'sh ip int br\n'
                'Interface         IP-Address      OK? Method Status                ␣
↪   Protocol\n'
                'Ethernet0/0       192.168.100.1   YES NVRAM  up                    ␣
↪   up      \n'
                'Ethernet0/1       192.168.200.1   YES NVRAM  up                    ␣
↪   up      \n'
                'Ethernet0/2       unassigned      YES NVRAM  up                    ␣
↪   up      \n'
                'Ethernet0/3       192.168.130.1   YES NVRAM  up                    ␣
↪   up      \n'
                'R1#'}
{'sh int desc': 'sh int desc\n'
                'Interface             Status         Protocol Description\n'
                'Et0/0                 up             up       \n'
                'Et0/1                 up             up       \n'
                'Et0/2                 admin down     down     \n'
                'Et0/3                 admin down     down     \n'
                'R2#',
 'sh ip int br': 'sh ip int br\n'
                'Interface         IP-Address      OK? Method Status                ␣
↪   Protocol\n'
                'Ethernet0/0       192.168.100.2   YES NVRAM  up                    ␣
↪   up      \n'
                'Ethernet0/1       unassigned      YES NVRAM  up                    ␣
↪   up      \n'
                'Ethernet0/2       unassigned      YES NVRAM  administratively␣
↪down down    \n'
                'Ethernet0/3       unassigned      YES NVRAM  administratively␣
↪down down    \n'
                'R2#'}
{'sh int desc': 'sh int desc\n'
                'Interface             Status         Protocol Description\n'
                'Et0/0                 up             up       \n'
                'Et0/1                 up             up       \n'
                'Et0/2                 admin down     down     \n'
                'Et0/3                 admin down     down     \n'
                'R3#',
 'sh ip int br': 'sh ip int br\n'
                'Interface         IP-Address      OK? Method Status                ␣
↪   Protocol\n'
```

(continues on next page)

(continued from previous page)

```
                'Ethernet0/0       192.168.100.3   YES NVRAM  up
↪    up      \n'
                'Ethernet0/1       unassigned      YES NVRAM  up
↪    up      \n'
                'Ethernet0/2       unassigned      YES NVRAM  administratively␣
↪down down    \n'
                'Ethernet0/3       unassigned      YES NVRAM  administratively␣
↪down down    \n'
```

Paginated command output

Example of using telnetlib to work with paginated output of show commands (2_telnetlib_more.py file):

```
import telnetlib
import time
from pprint import pprint
import re


def to_bytes(line):
    return f"{line}\n".encode("utf-8")


def send_show_command(ip, username, password, enable, command):
    with telnetlib.Telnet(ip) as telnet:
        telnet.read_until(b"Username")
        telnet.write(to_bytes(username))
        telnet.read_until(b"Password")
        telnet.write(to_bytes(password))
        index, m, output = telnet.expect([b">", b"#"])
        if index == 0:
            telnet.write(b"enable\n")
            telnet.read_until(b"Password")
            telnet.write(to_bytes(enable))
            telnet.read_until(b"#", timeout=5)
        time.sleep(3)
        telnet.read_very_eager()

        telnet.write(to_bytes(command))
        result = ""
```

(continues on next page)

(continued from previous page)

```
        while True:
            index, match, output = telnet.expect([b"--More--", b"#"], timeout=5)
            output = output.decode("utf-8")
            output = re.sub(" +--More--| +\x08+ +\x08+", "\n", output)
            result += output
            if index in (1, -1):
                break
            telnet.write(b" ")
            time.sleep(1)
            result.replace("\r\n", "\n")

        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", "sh run")
        pprint(result, width=120)
```

Module paramiko

Paramiko is an implementation of SSHv2 protocol on Python. Paramiko provides client-server functionality. Book covers only client functionality.

Since Paramiko is not part of standard Python module library, it needs to be installed:

```
pip install paramiko
```

Connection is established in this way: first, client is created and client configuration is set, then connection is initiated and an interactive session is returned:

```
In [2]: client = paramiko.SSHClient()

In [3]: client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

In [4]: client.connect(hostname="192.168.100.1", username="cisco", password="cisco
   ...: ", look_for_keys=False, allow_agent=False)

In [5]: ssh = client.invoke_shell()
```

SSHClient is a class that represents a connection to SSH server. It performs client authentication.

String `set_missing_host_key_policy` is optional, it indicates which policy to use when connecting to a server whose key is unknown. Policy `paramiko.AutoAddPolicy()` automatically add new hostname and key to local HostKeys object.

Method `connect` connects to SSH server and authenticates the connection. Parameters:

- `look_for_keys` - by default paramiko performs key authentication. To disable this, put the flag in False
- `allow_agent` - paramiko can connect to a local SSH agent. This is necessary when working with keys and since in this case authentication is done by login/password, it should be disabled.

After execution of previous command there is already a connection to server. Method `invoke_shell` allows to set an interactive SSH session with server.

Method send

Method `send` - sends specified string to session and returns amount of sent bytes.

```
In [7]: ssh.send("enable\n")
Out[7]: 7

In [8]: ssh.send("cisco\n")
Out[8]: 6

In [9]: ssh.send("sh ip int br\n")
Out[9]: 13
```

Warning: In code, after `send` you will need to put time.sleep, especially between `send` and `recv`. Since this is an interactive session and commands are slow to type, everything works without pauses.

Method recv

Method `recv` receives data from session. In parentheses, the maximum value in bytes that can be obtained is indicated. This method returns a received string

```
In [10]: ssh.recv(3000)
Out[10]: b'\r\nR1>enable\r\nPassword: \r\nR1#sh ip int br\r\nInterface
↪      IP-Address      OK? Method Status                Protocol\r\nEthernet0/0
↪              192.168.100.1   YES NVRAM  up                    up
↪\r\nEthernet0/1               192.168.200.1   YES NVRAM  up
↪up      \r\nEthernet0/2               unassigned      YES NVRAM  up
↪        up      \r\nEthernet0/3               192.168.130.1   YES NVRAM  up
↪                up      \r\nLoopback22                10.2.2.2        YES
↪manual up                    up      \r\nLoopback33                unassigned
↪    YES unset  up                    up      \r\nLoopback45
↪unassigned      YES unset  up                    up      \r\nLoopback55
↪        5.5.5.5         YES manual up                    up      \r\nR1#'
```

(continues on next page)

(continued from previous page)

Method close

Method `close` closes session:

```
In [11]: ssh.close()
```

Example of paramiko use

Example of paramiko use (3_paramiko.py file):

```
import paramiko
import time
import socket
from pprint import pprint


def send_show_command(
    ip,
    username,
    password,
    enable,
    command,
    max_bytes=60000,
    short_pause=1,
    long_pause=5,
):
    cl = paramiko.SSHClient()
    cl.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    cl.connect(
        hostname=ip,
        username=username,
        password=password,
        look_for_keys=False,
        allow_agent=False,
    )
    with cl.invoke_shell() as ssh:
        ssh.send("enable\n")
        ssh.send(f"{enable}\n")
        time.sleep(short_pause)
```

(continues on next page)

(continued from previous page)

```
        ssh.send("terminal length 0\n")
        time.sleep(short_pause)
        ssh.recv(max_bytes)

        result = {}
        for command in commands:
            ssh.send(f"{command}\n")
            ssh.settimeout(5)

            output = ""
            while True:
                try:
                    part = ssh.recv(max_bytes).decode("utf-8")
                    output += part
                    time.sleep(0.5)
                except socket.timeout:
                    break
            result[command] = output

        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh clock", "sh arp"]
    result = send_show_command("192.168.100.1", "cisco", "cisco", "cisco",␣
↪commands)
    pprint(result, width=120)
```

Result of script execution:

```
{'sh arp': 'sh arp\r\n'
           'Protocol  Address          Age (min)  Hardware Addr   Type   ␣
↪Interface\r\n'
           'Internet  192.168.100.1           -   aabb.cc00.6500  ARPA   ␣
↪Ethernet0/0\r\n'
           'Internet  192.168.100.2         124   aabb.cc00.6600  ARPA   ␣
↪Ethernet0/0\r\n'
           'Internet  192.168.100.3         183   aabb.cc00.6700  ARPA   ␣
↪Ethernet0/0\r\n'
           'Internet  192.168.100.100       208   aabb.cc80.c900  ARPA   ␣
↪Ethernet0/0\r\n'
           'Internet  192.168.101.1           -   aabb.cc00.6500  ARPA   ␣
↪Ethernet0/0\r\n'
```

(continues on next page)

(continued from previous page)

```
          'Internet  192.168.102.1           -   aabb.cc00.6500  ARPA  ␣
↪Ethernet0/0\r\n'
          'Internet  192.168.130.1           -   aabb.cc00.6530  ARPA  ␣
↪Ethernet0/3\r\n'
          'Internet  192.168.200.1           -   0203.e800.6510  ARPA  ␣
↪Ethernet0/1\r\n'
          'Internet  192.168.200.100        18   6ee2.6d8c.e75d  ARPA  ␣
↪Ethernet0/1\r\n'
          'R1#',
 'sh clock': 'sh clock\r\n*08:25:22.435 UTC Mon Jul 20 2020\r\nR1#'}
```

Paginated command output

Example of using paramiko to work with paginated output of show command (3_paramiko_more.py file):

```
import paramiko
import time
import socket
from pprint import pprint
import re


def send_show_command(
    ip,
    username,
    password,
    enable,
    command,
    max_bytes=60000,
    short_pause=1,
    long_pause=5,
):
    cl = paramiko.SSHClient()
    cl.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    cl.connect(
        hostname=ip,
        username=username,
        password=password,
        look_for_keys=False,
        allow_agent=False,
    )
```

(continues on next page)

(continued from previous page)

```
    with cl.invoke_shell() as ssh:
        ssh.send("enable\n")
        ssh.send(enable + "\n")
        time.sleep(short_pause)
        ssh.recv(max_bytes)

        result = {}
        for command in commands:
            ssh.send(f"{command}\n")
            ssh.settimeout(5)

            output = ""
            while True:
                try:
                    page = ssh.recv(max_bytes).decode("utf-8")
                    output += page
                    time.sleep(0.5)
                except socket.timeout:
                    break
                if "More" in page:
                    ssh.send(" ")
            output = re.sub(" +--More--| +\x08+ +\x08+", "\n", output)
            result[command] = output

        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh run"]
    result = send_show_command("192.168.100.1", "cisco", "cisco", "cisco",␣
↪commands)
    pprint(result, width=120)
```

Module netmiko

Netmiko is a module that makes it easier to use paramiko for network devices. Netmiko uses paramiko but also creates interface and methods needed to work with network devices.

First you need to install netmiko:

```
pip install netmiko
```

Supported device types

Netmiko supports several types of devices:

- Arista vEOS
- Cisco ASA
- Cisco IOS
- Cisco IOS-XR
- Cisco SG300
- HP Comware7
- HP ProCurve
- Juniper Junos
- Linux
- and other

The whole list can be viewed in module repository.

Dictionary for defining device parameters

Dictionary may have the next parameters:

```
cisco_router = {
    'device_type': 'cisco_ios',
    'host': '192.168.1.1',
    'username': 'user',
    'password': 'userpass',
    'secret': 'enablepass',
    'port': 20022,
}
```

Connect via SSH

```
ssh = ConnectHandler(**cisco_router)
```

Enable mode

Switch to enable mode:

```
ssh.enable()
```

Exit enable mode:

```
ssh.exit_enable_mode()
```

Sending commands

Netmiko has several ways to send commands:

- `send_command` - send one command
- `send_config_set` - send list of commands or command in configuration mode
- `send_config_from_file` - send commands from the file (uses `send_config_set` method inside)
- `send_command_timing` - send command and wait for the output based on timer

send_command

Method send_command allows you to send one command to device.

For example:

```
result = ssh.send_command('show ip int br')
```

Method works as follows:

- sends command to device and gets the output until string with prompt or until specified string
  - prompt is automatically determined
  - if your device does not determine it, you can simply specify a string till which to read the output
  - `send_command_expect` method previously worked this way, but since version

    1.0.0 this is how send_command works and send_command_expect method is left for compatibility
- method returns command output
- the following parameters can be passed to method:
  - `command_string` - command
  - `expect_string` - to which substring to read the output
  - `delay_factor` - option allows to increase delay before the start of string search

- `max_loops` - number of iterations before method gives out an error (exception). By default 500
- `strip_prompt` - remove prompt from the output. Removed by default
- `strip_command` - remove command from output

In most cases, only command will be sufficient to specify.

send_config_set

Method `send_config_set` allows you to send command or multiple commands in configuration mode.

Example:

```
commands = ['router ospf 1',
            'network 10.0.0.0 0.255.255.255 area 0',
            'network 192.168.100.0 0.0.0.255 area 1']

result = ssh.send_config_set(commands)
```

Method works as follows:

- goes into configuration mode,
- then passes all commands
- and exits configuration mode
- depending on device type, there may be no exit from configuration mode. For example, there will be no exit for IOS-XR because you first have to commit changes

send_config_from_file

Method `send_config_from_file` sends commands from specified file to configuration mode.

Example of use:

```
result = ssh.send_config_from_file('config_ospf.txt')
```

Method opens a file, reads commands and passes them to `send_config_set` method.

Additional methods

Besides the above methods for sending commands, netmiko supports such methods:

- `config_mode` - switch to configuration mode: `ssh.config_mode`

- `exit_config_mode` - exit configuration mode: `ssh.exit_config_mode`
- `check_config_mode` - check whether netmiko is in configuration mode (returns True if in configuration mode and False if not): `ssh.check_config_mode`
- `find_prompt` - returns the current prompt of device: `ssh.find_prompt`
- `commit` - commit on IOS-XR and Juniper: `ssh.commit`
- `disconnect` - terminate SSH connection

Note: Variable ssh is a pre-created SSH connection: `ssh = ConnectHandler(**cisco_router)`

Telnet support

Since version 1.0.0 netmiko supports Telnet connections, so far only for Cisco IOS devices. Inside netmiko uses telnetlib to connect via Telnet. But, at the same time, it provides the same interface for work as for SSH connection.

In order to connect via Telnet, it is enough in the dictionary that defines connection parameters specify device type `cisco_ios_telnet`:

```
device = {
    "device_type": "cisco_ios_telnet",
    "host": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}
```

Otherwise, methods that apply to SSH apply to Telnet. An example similar to SSH (4_netmiko_telnet.py file):

```
from pprint import pprint
import yaml
from netmiko import (
    ConnectHandler,
    NetmikoTimeoutException,
    NetmikoAuthenticationException,
)


def send_show_command(device, commands):
    result = {}
    try:
```

(continues on next page)

(continued from previous page)

```
        with ConnectHandler(**device) as ssh:
            ssh.enable()
            for command in commands:
                output = ssh.send_command(command)
                result[command] = output
        return result
    except (NetmikoTimeoutException, NetmikoAuthenticationException) as error:
        print(error)


if __name__ == "__main__":
    device = {
        "device_type": "cisco_ios_telnet",
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "secret": "cisco",
    }
    result = send_show_command(device, ["sh clock", "sh ip int br"])
    pprint(result, width=120)
```

Other methods works similarly:

- send_command_timing
- find_prompt
- send_config_set
- send_config_from_file
- check_enable_mode
- disconnect

Example of netmiko use

Example of netmiko use (4_netmiko.py file):

```
from pprint import pprint
import yaml
from netmiko import (
    ConnectHandler,
    NetmikoTimeoutException,
    NetmikoAuthenticationException,
```

(continues on next page)

(continued from previous page)

```
)


def send_show_command(device, commands):
    result = {}
    try:
        with ConnectHandler(**device) as ssh:
            ssh.enable()
            for command in commands:
                output = ssh.send_command(command)
                result[command] = output
        return result
    except (NetmikoTimeoutException, NetmikoAuthenticationException) as error:
        print(error)


if __name__ == "__main__":
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)
    for device in devices:
        result = send_show_command(device, ["sh clock", "sh ip int br"])
        pprint(result, width=120)
```

In this example `terminal length` command is not passed because netmiko executes this command by default.

The result of script execution:

```
{'sh clock': '*09:12:15.210 UTC Mon Jul 20 2020',
 'sh ip int br': 'Interface     IP-Address      OK? Method Status                ␣
↪Protocol\n'
                 'Ethernet0/0   192.168.100.1   YES NVRAM  up                    ␣
↪up      \n'
                 'Ethernet0/1   192.168.200.1   YES NVRAM  up                    ␣
↪up      \n'
                 'Ethernet0/2   unassigned      YES NVRAM  up                    ␣
↪up      \n'
                 'Ethernet0/3   192.168.130.1   YES NVRAM  up                    ␣
↪up      \n'}
{'sh clock': '*09:12:24.507 UTC Mon Jul 20 2020',
 'sh ip int br': 'Interface     IP-Address      OK? Method Status                ␣
↪Protocol\n'
                 'Ethernet0/0   192.168.100.2   YES NVRAM  up                    ␣
↪up      \n'
```

(continues on next page)

(continued from previous page)

```
                'Ethernet0/1   unassigned      YES NVRAM  up                    ␣
↪up      \n'
                'Ethernet0/2   unassigned      YES NVRAM  administratively down␣
↪down    \n'
                'Ethernet0/3   unassigned      YES NVRAM  administratively down␣
↪down    \n'}
{'sh clock': '*09:12:33.573 UTC Mon Jul 20 2020',
 'sh ip int br': 'Interface     IP-Address      OK? Method Status                ␣
↪Protocol\n'
                'Ethernet0/0   192.168.100.3   YES NVRAM  up                    ␣
↪up      \n'
                'Ethernet0/1   unassigned      YES NVRAM  up                    ␣
↪up      \n'
                'Ethernet0/2   unassigned      YES NVRAM  administratively down␣
↪down    \n'
                'Ethernet0/3   unassigned      YES NVRAM  administratively down␣
↪down    \n'}
```

Paginated command output

Example of using netmiko with paginated output of show command (4_netmiko_more.py file):

```
from netmiko import ConnectHandler, NetmikoTimeoutException
import yaml


def send_show_command(device_params, command):
    with ConnectHandler(**device_params) as ssh:
        ssh.enable()
        prompt = ssh.find_prompt()
        ssh.send_command("terminal length 100")
        ssh.write_channel(f"{command}\n")
        output = ""
        while True:
            try:
                page = ssh.read_until_pattern(f"More|{prompt}")
                output += page
                if "More" in page:
                    ssh.write_channel(" ")
                elif prompt in output:
                    break
            except NetmikoTimeoutException:
```

(continues on next page)

(continued from previous page)

```
                break
    return output


if __name__ == "__main__":
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)
    print(send_show_command(devices[0], "sh run"))
```

Module scrapli

scrapli is a module that allows you to connect to network equipment using Telnet, SSH or NETCONF.

Just like netmiko, scrapli can use paramiko or telnetlib (and other modules) for the connection itself, but it provides the same interface for different types of connections and different equipment.

Installing scrapli:

```
pip install scrapli
```

Note: Book covers scrapli version 2021.1.30.

The three main components of scrapli are:

- transport is a specific way to connect to equipment
- channel - the next level above the transport, which is responsible for sending commands, receiving output and other interactions with equipment
- driver is the interface for working with scrapli. There are both specific drivers, for example, IOSXEDriver, which understands how to interact with a specific type of equipment, and the basic Driver, which provides a minimal interface for working via SSH/Telnet.

Available transport options:

- system - the built-in SSH client, it is assumed that the client is used on Linux/MacOS
- paramiko - the paramiko module
- ssh2 - the ssh2-python module (wrapper around the C library libssh2)
- telnet - telnetlib
- asyncssh - asyncssh module
- asynctelnet - async telnet client

Most of the examples will be using the system transport. Since the module interface is the same for all synchronous transport options, to use a different transport, you just need to specify it (for telnet transport, you must also specify the port).

Note: Asynchronous transport options (asyncssh, asynctelnet) are covered in the Advanced Python for network engineers (russian)

Supported platforms:

- Cisco IOS-XE
- Cisco NX-OS
- Juniper JunOS
- Cisco IOS-XR
- Arista EOS

In addition to these platforms, there are also scrapli community platforms. And one of the advantages of scrapli is that it is relatively easy to add new platforms.

There are two connection options in scrapli: using the general Scrapli class, which selects the required driver by the platform parameter, or a specific driver, for example, IOSXEDriver. The same parameters are passed to the specific driver and Scrapli.

Note: In addition to these options, there are also generic (base) drivers.

If the scrapli (or scrapli community) does not support the required platform, you can add the platform to the scrapli community or use generic drivers (not covered in the book):

- Driver
- GenericDriver
- NetworkDriver

Connection parameters

Basic connection parameters:

- host - IP address or hostname
- auth_username - username for authentication
- auth_password - password for authentication
- auth_secondary - enable password

- auth_strict_key - strict key checking (True by default)
- platform - must be specified when using Scrapli
- transport - which transport to use
- transport_options - options for a specific transport

The connection process is slightly different depending on whether you are using a context manager or not. When connecting without a context manager, you first need to pass parameters to the driver or Scrapli, and then call the open method:

```
from scrapli import Scrapli

r1 = {
   "host": "192.168.100.1",
   "auth_username": "cisco",
   "auth_password": "cisco",
   "auth_secondary": "cisco",
   "auth_strict_key": False,
   "platform": "cisco_iosxe"
}

In [2]: ssh = Scrapli(**r1)

In [3]: ssh.open()
```

After that, you can send commands:

```
In [4]: ssh.get_prompt()
Out[4]: 'R1#'

In [5]: ssh.close()
```

When using a context manager, you don't need to call open:

```
In [8]: with Scrapli(**r1_driver) as ssh:
   ...:     print(ssh.get_prompt())
   ...:
R1#
```

Using the driver

Available drivers

Network equipment	Драйвер	Параметр platform
Cisco IOS-XE	IOSXEDriver	cisco_iosxe
Cisco NX-OS	NXOSDriver	cisco_nxos
Cisco IOS-XR	IOSXRDriver	cisco_iosxr
Arista EOS	EOSDriver	arista_eos
Juniper JunOS	JunosDriver	juniper_junos

Example of connection using the IOSXEDriver driver (connecting to Cisco IOS):

```
In [11]: from scrapli.driver.core import IOSXEDriver

In [12]: r1_driver = {
    ...:     "host": "192.168.100.1",
    ...:     "auth_username": "cisco",
    ...:     "auth_password": "cisco",
    ...:     "auth_secondary": "cisco",
    ...:     "auth_strict_key": False,
    ...: }

In [13]: with IOSXEDriver(**r1_driver) as ssh:
    ...:     print(ssh.get_prompt())
    ...:
R1#
```

Sending commands

Scrapli has several methods for sending commands:

- `send_command` - send one show command
- `send_commands` - send a list of show commands
- `send_commands_from_file` - send show commands from a file
- `send_config` - send one command in configuration mode
- `send_configs` - send a list of commands in configuration mode
- `send_configs_from_file` - send commands from file in configuration mode

All of these methods return a Response object, not the output of the command as a string.

Response object

The send_command method and other methods for sending commands return a Response object (not the output of the command). Response allows you to get not only the output of the command,

but also such things as the execution time of the command, whether there were errors during the execution of the command, structured output using textfsm, and so on.

```
In [15]: reply = ssh.send_command("sh clock")

In [16]: reply
Out[16]: Response <Success: True>
```

You can get the output of the command by accessing the result attribute:

```
In [17]: reply.result
Out[17]: '*17:31:54.232 UTC Wed Mar 31 2021'
```

The raw_result attribute contains a byte string with complete output:

```
In [18]: reply.raw_result
Out[18]: b'\n*17:31:54.232 UTC Wed Mar 31 2021\nR1#'
```

For commands that take longer than normal show, it may be necessary to know the command execution time:

```
In [18]: r = ssh.send_command("ping 10.1.1.1")

In [19]: r.result
Out[19]: 'Type escape sequence to abort.\nSending 5, 100-byte ICMP Echos to 10.1.
↪1.1, timeout is 2 seconds:\n.....\nSuccess rate is 0 percent (0/5)'

In [20]: r.elapsed_time
Out[20]: 10.047594

In [21]: r.start_time
Out[21]: datetime.datetime(2021, 4, 1, 7, 10, 56, 63697)

In [22]: r.finish_time
Out[22]: datetime.datetime(2021, 4, 1, 7, 11, 6, 111291)
```

The channel_input attribute returns the command that was sent to the equipment:

```
In [23]: r.channel_input
Out[23]: 'ping 10.1.1.1'
```

send_command method

The send_command method allows you to send one command to a device.

```
In [14]: reply = ssh.send_command("sh clock")
```

Method parameters (all these parameters must be passed as keyword arguments):

- `strip_prompt` - remove a prompt from the output. Deleted by default
- `failed_when_contains` - if the output contains the specified line or one of the lines in the list, the command will be considered as completed with an error
- `timeout_ops` - maximum time to execute a command, by default it is 30 seconds for IOSXEDriver

An example of using the send_command method:

```
In [15]: reply = ssh.send_command("sh clock")

In [16]: reply
Out[16]: Response <Success: True>
```

The timeout_ops parameter specifies how long to wait for the command to execute:

```
In [19]: ssh.send_command("ping 8.8.8.8", timeout_ops=20)
Out[19]: Response <Success: True>
```

If the command does not complete within the specified time, a ScrapliTimeout exception will be raised (output is truncated):

```
In [20]: ssh.send_command("ping 8.8.8.8", timeout_ops=2)
---------------------------------------------------------------------------
ScrapliTimeout                            Traceback (most recent call last)
<ipython-input-20-e062fb19f0e6> in <module>
----> 1 ssh.send_command("ping 8.8.8.8", timeout_ops=2)
```

In addition to receiving normal command output, scrapli also allows you to receive structured output, for example using the textfsm_parse_output method:

```
In [21]: reply = ssh.send_command("sh ip int br")

In [22]: reply.textfsm_parse_output()
Out[22]:
[{'intf': 'Ethernet0/0',
  'ipaddr': '192.168.100.1',
  'status': 'up',
  'proto': 'up'},
 {'intf': 'Ethernet0/1',
  'ipaddr': '192.168.200.1',
```

(continues on next page)

(continued from previous page)

```
 'status': 'up',
 'proto': 'up'},
{'intf': 'Ethernet0/2',
 'ipaddr': 'unassigned',
 'status': 'up',
 'proto': 'up'},
{'intf': 'Ethernet0/3',
 'ipaddr': '192.168.130.1',
 'status': 'up',
 'proto': 'up'}]
```

Note: What is TextFSM and how to work with it is covered in chapter 21. Scrapli uses ready-made templates in order to receive structured output and in basic cases does not require knowledge of TextFSM.

Error detection

Methods for sending commands automatically check the output for errors. For each vendor/type of equipment, these are different errors, plus you can specify which lines in the output will be considered an error. By default, IOSXEDriver will consider the following lines as errors:

```
In [21]: ssh.failed_when_contains
Out[21]:
['% Ambiguous command',
 '% Incomplete command',
 '% Invalid input detected',
 '% Unknown command']
```

The `failed` attribute of the Response object returns `False` if the command finished without error and `True` if it failed.

```
In [23]: reply = ssh.send_command("sh clck")

In [24]: reply.result
Out[24]: "        ^\n% Invalid input detected at '^' marker."

In [25]: reply
Out[25]: Response <Success: False>

In [26]: reply.failed
Out[26]: True
```

send_config method

The send_config method allows you to send one configuration mode command.

Example:

```
In [33]: r = ssh.send_config("username user1 password password1")
```

Since scrapli removes the command from the output, by default, when using send_config, the result attribute will contain an empty string (if there was no error while executing the command):

```
In [34]: r.result
Out[34]: ''
```

You can add the parameter strip_prompt=False and then the prompt will appear in the output:

```
In [37]: r = ssh.send_config("username user1 password password1", strip_
↪prompt=False)

In [38]: r.result
Out[38]: 'R1(config)#'
```

send_commands, send_configs

The send_commands, send_configs methods differ from send_command, send_config in that they can send several commands. In addition, these methods do not return a Response, but a MultiResponse object, which can generally be thought of as a list of Response objects, one for each command.

```
In [44]: reply = ssh.send_commands(["sh clock", "sh ip int br"])

In [45]: reply
Out[45]: MultiResponse <Success: True; Response Elements: 2>

In [46]: for r in reply:
    ...:     print(r)
    ...:     print(r.result)
    ...:
Response <Success: True>
*08:38:20.115 UTC Thu Apr 1 2021
Response <Success: True>
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                     up
```

(continues on next page)

(continued from previous page)

```
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                unassigned      YES NVRAM  up                    up
Ethernet0/3                192.168.130.1   YES NVRAM  up                    up

In [47]: reply.result
Out[47]: 'sh clock\n*08:38:20.115 UTC Thu Apr 1 2021sh ip int br\nInterface
↪           IP-Address      OK? Method Status                Protocol\nEthernet0/
↪0               192.168.100.1   YES NVRAM  up                    up\nEthernet0/
↪1               192.168.200.1   YES NVRAM  up                    up\nEthernet0/
↪2               unassigned      YES NVRAM  up                    up\nEthernet0/
↪3               192.168.130.1   YES NVRAM  up                    up'

In [48]: reply[0]
Out[48]: Response <Success: True>

In [49]: reply[1]
Out[49]: Response <Success: True>

In [50]: reply[0].result
Out[50]: '*08:38:20.115 UTC Thu Apr 1 2021'
```

When sending multiple commands, it is also very convenient to use the `stop_on_failed` parameter. By default, it is False, so all commands are executed, but if you specify `stop_on_failed=True`, after an error occurs in some command, the following commands will not be executed:

```
In [59]: reply = ssh.send_commands(["ping 192.168.100.2", "sh clck", "sh ip int br
↪"], stop_on_failed=True)

In [60]: reply
Out[60]: MultiResponse <Success: False; Response Elements: 2>

In [61]: reply.result
Out[61]: "ping 192.168.100.2\nType escape sequence to abort.\nSending 5, 100-byte
↪ICMP Echos to 192.168.100.2, timeout is 2 seconds:\n!!!!!\nSuccess rate is 100
↪percent (5/5), round-trip min/avg/max = 1/2/6 mssh clck\n        ^\n% Invalid
↪input detected at '^' marker."

In [62]: for r in reply:
    ...:     print(r)
    ...:     print(r.result)
    ...:
Response <Success: True>
Type escape sequence to abort.
```

(continues on next page)

(continued from previous page)

```
Sending 5, 100-byte ICMP Echos to 192.168.100.2, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 1/2/6 ms
Response <Success: False>
        ^
% Invalid input detected at '^' marker.
```

Telnet connection

To connect to equipment via Telnet, you must specify transport equal to `telnet` and be sure to specify the port parameter equal to 23 (or the port that you use to connect via Telnet):

```
from scrapli.driver.core import IOSXEDriver
from scrapli.exceptions import ScrapliException
import socket

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco2",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "transport": "telnet",
    "port": 23,  # must be specified when connecting telnet
}


def send_show(device, show_command):
    try:
        with IOSXEDriver(**r1) as ssh:
            reply = ssh.send_command(show_command)
            return reply.result
    except socket.timeout as error:
        print(error)
    except ScrapliException as error:
        print(error, device["host"])


if __name__ == "__main__":
    output = send_show(r1, "sh ip int br")
    print(output)
```

Scrapli examples

Basic example of sending show command:

```
from scrapli.driver.core import IOSXEDriver
from scrapli.exceptions import ScrapliException


r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "timeout_socket": 5,  # timeout for establishing socket/initial connection
    "timeout_transport": 10,  # timeout for ssh|telnet transport
}


def send_show(device, show_command):
    try:
        with IOSXEDriver(**r1) as ssh:
            reply = ssh.send_command(show_command)
            return reply.result
    except ScrapliException as error:
        print(error, device["host"])


if __name__ == "__main__":
    output = send_show(r1, "sh ip int br")
    print(output)
```

Basic example of sending config commands:

```
from pprint import pprint
from scrapli import Scrapli

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "platform": "cisco_iosxe",
```

(continues on next page)

(continued from previous page)

```
}


def send_show(device, show_commands):
    if type(show_commands) == str:
        show_commands = [show_commands]
    cmd_dict = {}
    with Scrapli(**r1) as ssh:
        for cmd in show_commands:
            reply = ssh.send_command(cmd)
            cmd_dict[cmd] = reply.result
    return cmd_dict


if __name__ == "__main__":
    print("show".center(20, "#"))
    output = send_show(r1, ["sh ip int br", "sh ver | i uptime"])
    pprint(output, width=120)
```

An example of sending configuration commands with error checking:

```
from pprint import pprint
from scrapli import Scrapli

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "platform": "cisco_iosxe",
}


def send_cfg(device, cfg_commands, strict=False):
    output = ""
    if type(cfg_commands) == str:
        cfg_commands = [cfg_commands]
    with Scrapli(**r1) as ssh:
        reply = ssh.send_configs(cfg_commands, stop_on_failed=strict)
        for cmd_reply in reply:
            if cmd_reply.failed:
                print(
```

(continues on next page)

(continued from previous page)

```
                    f"An error occurred while executing the command::\n{reply.
↪result}\n"
                )
        output = reply.result
    return output


if __name__ == "__main__":
    output_cfg = send_cfg(
        r1, ["interfacelo11", "ip address 11.1.1.1 255.255.255.255"], strict=True
    )
    print(output_cfg)
```

Further reading

Documentation:

- pexpect
- telnetlib
- paramiko Client
- paramiko Channel
- netmiko
- scrapli
- scrapli-cfg
- time
- datetime
- getpass

Articles:

- Netmiko Library
- Automate SSH connections with netmiko
- Network Automation Using Python: BGP Configuration
- A Tale of Five Python SSH Libraries Commentary

Code examples:

- netmiko

- scrapli
- netmiko, paramiko, telnetlib, scrapli, pexpect

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 18.1

Create send_show_command function.

The function connects via SSH (using netmiko) to ONE device and executes the specified command.

Function parameters:

- device - a dictionary with parameters for connecting to a device
- command - the command to be executed

The function should return a string with the command output.

The script should send command command to all devices from the devices.yaml file using the send_show_command function (this part of the code is written).

```
import yaml


if __name__ == "__main__":
    command = "sh ip int br"
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)

    for dev in devices:
        print(send_show_command(dev, command))
```

Task 18.1a

Copy the send_show_command function from task 18.1 and rewrite it to handle the exception that is thrown on authentication failure on the device.

When an error occurs, an exception message should be printed to stdout.

To verify, change the password on the device or in the devices.yaml file.

Task 18.1b

Copy the send_show_command function from task 18.1a and rewrite it to handle not only the exception that is raised when authentication fails on the device, but also the exception that is raised when the IP address of the device is not available.

When an error occurs, an exception message should be printed to standard output.

To check, change the IP address on the device or in the devices.yaml file.

Task 18.2

Create send_config_commands function

The function connects via SSH (using netmiko) to ONE device and executes a list of commands in configuration mode based on the passed arguments.

Function parameters:

- device - a dictionary with parameters for connecting to a device
- config_commands - list of configuration commands to be executed

The function should return a string with the results of the command:

```
In [7]: r1
Out[7]:
{'device_type': 'cisco_ios',
 'ip': '192.168.100.1',
 'username': 'cisco',
 'password': 'cisco',
 'secret': 'cisco'}

In [8]: commands
Out[8]: ['logging 10.255.255.1', 'logging buffered 20010', 'no logging console']

In [9]: result = send_config_commands(r1, commands)

In [10]: result
Out[10]: 'config term\nEnter configuration commands, one per line.  End with CNTL/
↪Z.\nR1(config)#logging 10.255.255.1\nR1(config)#logging buffered␣
↪20010\nR1(config)#no logging console\nR1(config)#end\nR1#'

In [11]: print(result)
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#logging 10.255.255.1
```

(continues on next page)

(continued from previous page)

```
R1(config)#logging buffered 20010
R1(config)#no logging console
R1(config)#end
R1#
```

The script should send command command to all devices from the devices.yaml file using the send_config_commands function.

```
commands = [
    'logging 10.255.255.1', 'logging buffered 20010', 'no logging console'
]
```

Task 18.2a

Copy the send_config_commands function from job 18.2 and add the log parameter. The log parameter controls whether information is displayed about which device the connection is to:

- if log is equal to True - information is printed
- if log is equal to False - information is not printed

By default, log is equal to True.

An example of how the function works:

```
In [13]: result = send_config_commands(r1, commands)
Connecting to 192.168.100.1...

In [14]: result = send_config_commands(r1, commands, log=False)

In [15]:
```

The script should send command command to all devices from the devices.yaml file using the send_config_commands function.

Task 18.2b

Copy the send_config_commands function from task 18.2a and add error checking.

When executing each command, the script should check the output for the following errors: Invalid input detected, Incomplete command, Ambiguous command

If an error occurs while executing any of the commands, the function should print a message to the stdout with information about what error occurred, in which command and on which device,

for example: The “logging” command was executed with the error “Incomplete command.” on the device 192.168.100.1

Errors should always be printed, regardless of the value of the log parameter. At the same time, the log parameter should still control whether the message “Connecting to 192.168.100.1…” will be displayed.

Send_config_commands should now return a tuple of two dictionaries:

- the first dict with the output of commands that were executed without error
- second dict with the output of commands that were executed with errors

In both dictionaries:

- key - command
- value - output with command execution

You can test the function on one device.

An example of how the send_config_commands function works:

```
In [16]: commands
Out[16]:
['logging 0255.255.1',
 'logging',
 'a',
 'logging buffered 20010',
 'ip http server']

In [17]: result = send_config_commands(r1, commands)
Connecting to 192.168.100.1...
"logging 0255.255.1" command was executed with error "Invalid input detected at '^
↪' marker." on device 192.168.100.1
"logging" command was executed with error "Incomplete command." on device 192.168.
↪100.1
"a" command was executed with error "Ambiguous command:  "a"" on device 192.168.
↪100.1

In [18]: pprint(result, width=120)
({'ip http server': 'config term\n'
                    'Enter configuration commands, one per line.  End with CNTL/Z.
↪\n'
                    'R1(config)#ip http server\n'
                    'R1(config)#',
  'logging buffered 20010': 'config term\n'
                            'Enter configuration commands, one per line.  End␣
↪with CNTL/Z.\n'
```

(continues on next page)

(continued from previous page)

```
                          'R1(config)#logging buffered 20010\n'
                          'R1(config)#'},
 {'a': 'config term\n'
       'Enter configuration commands, one per line.  End with CNTL/Z.\n'
       'R1(config)#a\n'
       '% Ambiguous command:  "a"\n'
       'R1(config)#',
  'logging': 'config term\n'
             'Enter configuration commands, one per line.  End with CNTL/Z.\n'
             'R1(config)#logging\n'
             '% Incomplete command.\n'
             '\n'
             'R1(config)#',
  'logging 0255.255.1': 'config term\n'
                        'Enter configuration commands, one per line.  End with␣
↪CNTL/Z.\n'
                        'R1(config)#logging 0255.255.1\n'
                        '                   ^\n'
                        "% Invalid input detected at '^' marker.\n"
                        '\n'
                        'R1(config)#'})

In [19]: good, bad = result

In [20]: good.keys()
Out[20]: dict_keys(['logging buffered 20010', 'ip http server'])

In [21]: bad.keys()
Out[21]: dict_keys(['logging 0255.255.1', 'logging', 'a'])
```

Examples of commands with errors:

```
R1(config)#logging 0255.255.1
                   ^
% Invalid input detected at '^' marker.

R1(config)#logging
% Incomplete command.

R1(config)#a
% Ambiguous command:  "a"
```

Lists of command lists with and without errors:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
correct_commands = ['logging buffered 20010', 'ip http server']

commands = commands_with_errors + correct_commands
```

Task 18.2c

Copy the send_config_commands function from task 18.2b and remake it as follows: If an error occurs while executing a command, ask the user whether to continue executing other commands.

Answer options [y]/n:

- y - execute other commands. This is the default, so any key is interpreted as y
- n or no - do not execute other commands

The send_config_commands function should still return a tuple of two dictionaries:

- the first dictionary with the output of commands that were executed without error
- second dictionary with the output of commands that were executed with errors

In both dictionaries:

- key - command
- value - output with command execution

You can test the function on one device.

An example of how the send_config_commands function works:

```
In [11]: result = send_config_commands(r1, commands)
Connecting to 192.168.100.1...
"logging 0255.255.1" command was executed with error "Invalid input detected at '^
↪' marker." on device 192.168.100.1
Continue commands execution? [y]/n: y
"logging" command was executed with error "Incomplete command." on device 192.168.
↪100.1
Continue commands execution? [y]/n: n

In [12]: pprint(result)
({},
 {'logging': 'config term\n'
             'Enter configuration commands, one per line.  End with CNTL/Z.\n'
             'R1(config)#logging\n'
             '% Incomplete command.\n'
             '\n'
```

(continues on next page)

(continued from previous page)

```
            'R1(config)#',
  'logging 0255.255.1': 'config term\n'
                        'Enter configuration commands, one per line.  End with '
                        'CNTL/Z.\n'
                        'R1(config)#logging 0255.255.1\n'
                        '                   ^\n'
                        "% Invalid input detected at '^' marker.\n"
                        '\n'
                        'R1(config)#'})
```

Lists of commands with and without errors:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
correct_commands = ['logging buffered 20010', 'ip http server']

commands = commands_with_errors + correct_commands
```

Task 18.3

Create a send_commands function (use netmiko to connect via SSH).

Function parameters:

- device - a dictionary with parameters for connecting to one device
- show - one show command (string)
- config - a list with commands to be executed in configuration mode

The show and config arguments should only be passed as keyword arguments. Passing these arguments as positional should raise a TypeError exception.

```
In [4]: send_commands(r1, 'sh clock')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-75adcfb4a005> in <module>
----> 1 send_commands(r1, 'sh clock')

TypeError: send_commands() takes 1 positional argument but 2 were given
```

Depending on which argument was passed, the send_commands function calls different functions internally. When calling the send_commands function, only one of the show, config arguments should always be passed. If both arguments are passed, a ValueError exception should be raised.

A combination of an argument and a corresponding function:

- show - the send_show_command function from task 18.1
- config - send_config_commands function from task 18.2

The function returns a string with the results of executing single command or multiple commands.

Check function operation:

- with a list of config commands in variable commands
- single show command in variable command

An example of how the function works:

```
In [14]: send_commands(r1, show='sh clock')
Out[14]: '*17:06:12.278 UTC Wed Mar 13 2019'

In [15]: send_commands(r1, config=['username user5 password pass5', 'username␣
↪user6 password pass6'])
Out[15]: 'config term\nEnter configuration commands, one per line.  End with CNTL/
↪Z.\nR1(config)#username user5 password pass5\nR1(config)#username user6␣
↪password pass6\nR1(config)#end\nR1#'
```

Commands example:

```
commands = ['logging 10.255.255.1', 'logging buffered 20010']
command = 'sh ip int br'
```

19. Concurent connections to multiple devices

When you need to connect to a large number of devices, it will take quite a long time to complete the connections one by one. Of course, it will be faster than manual connection, but it would be faster to connect to equipment in parallel.

Module concurrent.futures is used for parallel connection to devices in this section.

Note: All these "long" and "faster" are relative concepts, but in this section we will learn to measure exact script execution time to compare how quick the connection is established.

Measure script execution time

There are several options for estimating execution time of the script. The simplest options are:

- Linux time utility
- and Python datetime module

When measuring the execution time of script in this case, high accuracy is not important. The main thing is to compare the execution time of script in different variants.

time

Linux `time` utility allows you to measure the execution time of a script. To use `time` utility it is enough to write `time` before starting the script:

```
$ time python thread_paramiko.py
...
real    0m4.712s
user    0m0.336s
sys     0m0.064s
```

We are interested in real time. In this case, it's 4.7 seconds.

datetime

The second option is a `datetime` module. This module allows working with time and dates in Python.

Example:

```
from datetime import datetime
import time

start_time = datetime.now()

#Code is running here
time.sleep(5)

print(datetime.now() - start_time)
```

Output:

```
$ python test.py
0:00:05.004949
```

Processes and threads in Python (CPython)

First, we need to work out the terms:

- process - roughly speaking, it's a launched program. Separate resources are allocated to the process: memory, processor time
- thread - execution unit in the process. Thread share resources of the process to which they relate.

Python (or, more precisely, CPython - the implementation used in the book) is optimized to work in single-threaded mode. This is good if program uses only one thread. And, at the same time, Python has certain nuances of running in multithreaded mode. This is because CPython uses GIL (global interpreter lock).

GIL does not allow multiple threads to execute Python code at the same time. If you don't go into detail, GIL can be visualized as a sort of flag that carried over from thread to thread. Whoever has the flag can do the job. The flag is transmitted either every Python instruction or, for example, when some type of input-output operation is performed.

Therefore, different threads will not run in parallel and the program will simply switch between them executing them at different times. However, if in the program there is some "wait" (packages from the network, user request, time.sleep pause), then in such program the threads will be executed as if in parallel. This is because during such pauses the flag (GIL) can be passed to another thread.

That is, threads are well suited for tasks that involve input-output (IO) operations:

- Connection to equipment and network connectivity in general
- Working with file system
- Downloading files

Note: In the Internet it is often possible to find phrases like «In Python it is better not to use threads at all». Unfortunately, such phrases are not always written in context, namely that it is about specific tasks that are tied to CPU.

The next sections discuss how to use threads to connect via Telnet/SSH. Script execution time will be checked comparing the sequential execution and execution using processes.

Processes

Processes allow to execute tasks on different computer cores. This is important for tasks that are tied to CPU. For each process a copy of resources is created, a memory is allocated, each process has its own GIL. This also makes processes “heavier” than threads.

In addition, the number of processes that run in parallel depends on the number of cores and CPU and is usually estimated in dozens, while the number of threads for input-output operations can be estimated in hundreds.

Processes and threads can be combined but this complicates the program and at the base level for input-output operations it is better to stop at threads.

Note: Combining threads and processes, i.e., starting a process in a program and then starting threads inside it, makes troubleshooting difficult. And I’d recomend not use that option.

Although it is usually better to use threads for input-output tasks, for some modules it is better to use processes because they may not work correctly with threads.

Note: In addition to processes and threads, there is another version of concurrent connections to device: asynchronous programming. This option is not covered in the book.

Number of threads

How many threads you need to use when connecting to device? There is no clear answer to this question. The number of threads depends at least on which computer runs the script (OS, memory, processor), on network itself (delays).

So instead of looking for the perfect number of threads, you have to measure the number on your computer, your network, your script. For example, in the examples to this section there is a script netmiko_count_threads.py that runs the same function with different threads and displays runtime information. Function by default uses a small number of devices from the devices_all.yaml file and a small number of threads, but it can be adapted to any number based on your network.

Example of connecting to 5,000 devices with different number of threads:

```
Number of devices: 5460

#30 threads
----------------------------------------
Execution time: 0:09:17.187867

#50 threads
----------------------------------------
Execution time: 0:09:17.604252

#70 threads
----------------------------------------
Execution time: 0:09:17.117332

#90 threads
----------------------------------------
Execution time: 0:09:16.693774

#100 threads
----------------------------------------
Execution time: 0:09:17.083294

#120 threads
----------------------------------------
Execution time: 0:09:17.945270

#140 threads
----------------------------------------
Execution time: 0:09:18.114993

#200 threads
----------------------------------------
Execution time: 0:11:12.951247

#300 threads
----------------------------------------
Execution time: 0:14:03.790432
```

In this case, the execution time with 30 threads and 120 threads is the same and after time only increases. This is because switching between threads also takes a lot of time and the more streams the more switching. And from some moment it makes no sense to increase number of threads. For this example (this PC, code and number of devices), the optimal number is approximately 50 threads. We're not taking 30 here in order to make a reserve.

Thread safety

When working with threads there are several recommendations and rules. If they are respected, it is easier to work with threads and it is likely that there will be no problem with threads. Of course, from time to time, there will be tasks that will require violations of recommendations. However, before doing so, it is better to try to meet the task by adhering to recommendations. If this is not possible, then we should look for ways to secure the solution so that the data is not damaged.

Very important feature of working with threads: with a small number of threads and small test tasks "everything works". For example, printing output when connected to 20 devices in 5 threads will work normally. But when connected to a large number of devices with a large number of threads, it turns out that sometimes messages overlap. This peculiarity appears very often, so do not trust the version when "everything works" on basic examples, follow the rules of working with threads.

Before dealing with rules we have to deal with term "thread safety". Thread safety is a concept that describes work with multithreaded programs. Code is considered to be thread-safe if it can work normally with multiple threads.

For example, `print` function is not thread-safe. This is demonstrated by the fact that when code executes `print` from different threads, messages in the output can be mixed. There could be output with a part of message from the first thread, then a part from the second thread, then a part from the first thread, and so on. That is, `print` function does not work normally (as it should be) in thread. In this case, it is said that `print` function is not thread-safe.

In general, there is no problem if each thread works with its own resources. For example, each thread writes data to its own file. However, this is not always possible or can complicate the solution.

Note: `print` has problems because we write from different threads into one standard output stream but `print` is not thread-safe.

If you have to write from different threads to the same resource, there are two options:

1. Write to the same resource after job in thread is finished. For example, a function has been executed in threads 1, 2 and 3, its result is obtained in turn (consecutively) from each thread, and then written into a file.
2. Use a thread-safe alternative (not always available and/or easy). For example, use a logging module instead of `print` function.

Recommendations when working with threads:

1. Do not write to the same resource from different threads if resource or what you write is not intended for multithreading. It is easy to find out by google something like "python write to file from threads".

- There are nuances to this recommendation. For example, you can write from different threads to the same file if you use a Lock or a thread-safe queue. These options are often difficult to

use and are not covered in the book. It's likely that 95 percent of problems you'll be facing can be solved without them.

- This category includes writing/changing lists/dictionaries/sets from different threads. These objects are inherently thread-safe but there is no guarantee that when you change the same list from different threads, data in the list will be correct. If you want to use a common container for different threads, use queue from Queue module. It's thread-safe and you can work with it from different threads.

2. If there is a possibility, avoid communication between threads in the course of their work. This is not an easy task and it is best to avoid it.
3. Follow the KISS (Keep it simple, stupid) principle - try to make solution as simple as possible.

Note: These recommendations are generally written for those who are just beginning to program on Python. However, they tend to be relevant to most programmers who write applications for users rather than frameworks.

Module concurrent.futures which will be covered further, simplifies implementation of the first principle "Do not write to the same resource from different threads... ". The module interface itself encourages this, but of course it does not prohibit breaking it.

However, before getting to know concurrent.futures, you should read fundamentals of logging module. It will be used instead of `print` function which is not thread-safe.

Module logging

Module logging - a module from Python standard library that allows you to configure logging from the script. Module logging has a lot of features and a lot of configuration options. Only basic configuration option is discussed in this section.

The easiest way to configure logging in script, use logging.basicConfig:

```
import logging


logging.basicConfig(
    format='%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO,
)
```

In this version, the settings are:

- all messages will be displayed on standard output
- messages of INFO level and above will be displayed

- each message will contain thread information, log name, message level, and message itself

Now, to output a log message in this script, you need to write `logging.info("test")`.

Example of script with logging settings (logging_basics.py file):

```
from datetime import datetime
import logging
import netmiko
import yaml


# this string indicates that paramiko log messages will be displayed
# only if they have WARNING level or higher
logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device, show):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device["ip"]
    logging.info(start_msg.format(datetime.now().time(), ip))

    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result =  ssh.send_command(show)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return result


if __name__ == "__main__":
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    for dev in devices:
        print(send_show(dev, 'sh clock'))
```

Result of script execution:

```
$ python logging_basics.py
MainThread root INFO: ===> 12:26:12.767168 Connection: 192.168.100.1
MainThread root INFO: <=== 12:26:18.307017 Received:   192.168.100.1
```

(continues on next page)

(continued from previous page)

```
*12:26:18.137 UTC Wed Jun 5 2019
MainThread root INFO: ===> 12:26:18.413913 Connection: 192.168.100.2
MainThread root INFO: <=== 12:26:23.991715 Received:   192.168.100.2
*12:26:23.819 UTC Wed Jun 5 2019
MainThread root INFO: ===> 12:26:24.095452 Connection: 192.168.100.3
MainThread root INFO: <=== 12:26:29.478553 Received:   192.168.100.3
*12:26:29.308 UTC Wed Jun 5 2019
```

Note: There are still many features in logging module. This section only uses basic configuration option. For more information on features of the module, see Logging HOWTO

Module concurrent.futures

Module `concurrent.futures` provides a high-level interface for working with processes and threads. For both threads and processes the same interface is used which makes it easy to switch between them.

If you compare this module with threading or multiprocessing, it has fewer features but with `concurrent.futures` it's easier to work and interface easier to understand.

Concurrent.futures module allows to solve the problem of starting multiple threads/processes and getting data from them. For this purpose, module uses two classes:

- `ThreadPoolExecutor` - for threads handling
- `ProcessPoolExecutor` - for process handling

Both classes use the same interface, so it is enough to deal with one and then just switch to other if necessary.

Create an Executor object using ThreadPoolExecutor:

```
executor = ThreadPoolExecutor(max_workers=5)
```

After creating an Executor object, it has three methods: `shutdown`, `map`, and `submit`. Method `shutdown` is responsible for the completion of threads/processes, `map` and `submit` methods are responsible for starting functions in different threads/processes.

Note: In fact, map and submit can run not only functions but any callable object. However, only functions will be covered further.

Method `shutdown` indicates that Executor object must be finished. However, if to `shutdown` method pass `wait=True` (default value), it will not return the result until all functions running in threads have

been completed. If `wait=False`, `shutdown` method returns instantly but script itself will not exit until all functions have been completed.

Generally, `shutdown` is not explicitly used because when creating an Executor object in a context manager, `shutdown` is automatically called at the end of a block with `wait=True`.

```
with ThreadPoolExecutor(max_workers=5) as executor:
    ...
```

Since map and submit methods start a function in threads or processes, code must at least have a function that performs one action and must be run in different threads with different arguments of the function.

For example, if you need to ping multiple IP addresses in different threads you need to create a function that pings one IP address and then run this function in different threads for different IP addresses using concurrent.futures.

Method map

Method syntax:

```
map(func, *iterables, timeout=None)
```

Method `map` is similar to built-in map function: applying `func` function to one or more iterable objects. Each call to a function is then started in a separate thread/process. Method `map` returns an iterator with function results for each element of object being iterated. The results are arranged in the same order as elements in iterable object.

When working with thread/process pools, a certain number of threads/processes are created and the code is executed in these threads. For example, if the pool is created with 5 threads and function has to be started for 10 different devices, connection will be performed first to the first five devices and then, as they liberated, to the others.

An example of using a `map` function with ThreadPoolExecutor (netmiko_threads_map_basics.py file):

```
from datetime import datetime
import time
from itertools import repeat
from concurrent.futures import ThreadPoolExecutor
import logging

import netmiko
import yaml


logging.getLogger('paramiko').setLevel(logging.WARNING)
```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device, show):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(show)
        logging.info(received_msg.format(datetime.now().time(), ip))
        return result


with open('devices.yaml') as f:
    devices = yaml.safe_load(f)

with ThreadPoolExecutor(max_workers=3) as executor:
    result = executor.map(send_show, devices, repeat('sh clock'))
    for device, output in zip(devices, result):
        print(device['host'], output)
```

Since function should be passed to map method, send_show function is created which connects to devices, passes specified show command and returns the result with command output.

```
def send_show(device, show):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(show)
```

(continues on next page)

(continued from previous page)

```
        logging.info(received_msg.format(datetime.now().time(), ip))
        return result
```

Function send_show outputs log message at the beginning and at the end of work. This will determine when function has worked for the particular device. Also within function it is specified that when connecting to device with address 192.168.100.1, the pause for 5 seconds is required - thus router with this address will respond longer.

Last 4 lines of code are responsible for connecting to devices in separate threads:

```
with ThreadPoolExecutor(max_workers=3) as executor:
    result = executor.map(send_show, devices, repeat('sh clock'))
    for device, output in zip(devices, result):
        print(device['host'], output)
```

- `with ThreadPoolExecutor(max_workers=3) as executor:` - ThreadPoolExecutor class is initiated in `with` block with indicated number of threads.
- `result = executor.map(send_show, devices, repeat('sh clock'))` - map method is similar to map function, but here the send_show function is called in different threads. However, in different threads the function will be called with different arguments:
  - elements of iterable object devices and the same command “sh clock”.
  - since instead of a list of commands only one command is used, it must be repeated in some way, so that map method will set this command to different devices. It uses repeat function - it repeats command exactly as many times as map requests
- map method returns generator. This generator contains results of functions. Results are in the same order as devices in the list of devices, so zip function is used to combine device IP addresses and command output.

Execution result:

```
$ python netmiko_threads_map_basics.py
ThreadPoolExecutor-0_0 root INFO: ===> 08:28:55.950254 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 08:28:55.963198 Connection: 192.168.100.2
ThreadPoolExecutor-0_2 root INFO: ===> 08:28:55.970269 Connection: 192.168.100.3
ThreadPoolExecutor-0_1 root INFO: <=== 08:29:11.968796 Received:   192.168.100.2
ThreadPoolExecutor-0_2 root INFO: <=== 08:29:15.497324 Received:   192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 08:29:16.854344 Received:   192.168.100.1
192.168.100.1 *08:29:16.663 UTC Thu Jul 4 2019
192.168.100.2 *08:29:11.744 UTC Thu Jul 4 2019
192.168.100.3 *08:29:15.374 UTC Thu Jul 4 2019
```

The first three messages indicate when connection was made and to which device:

```
ThreadPoolExecutor-0_0 root INFO: ===> 08:28:55.950254 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 08:28:55.963198 Connection: 192.168.100.2
ThreadPoolExecutor-0_2 root INFO: ===> 08:28:55.970269 Connection: 192.168.100.3
```

The following three messages show time of receipt of information and completion of the function:

```
ThreadPoolExecutor-0_1 root INFO: <=== 08:29:11.968796 Received:   192.168.100.2
ThreadPoolExecutor-0_2 root INFO: <=== 08:29:15.497324 Received:   192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 08:29:16.854344 Received:   192.168.100.1
```

Since `sleep` was added for the first device for 5 seconds, information from the first router was actually received later. However, since `map` method returns values in the same order as devices in `device` list, the result is:

```
192.168.100.1 *08:29:16.663 UTC Thu Jul 4 2019
192.168.100.2 *08:29:11.744 UTC Thu Jul 4 2019
192.168.100.3 *08:29:15.374 UTC Thu Jul 4 2019
```

Map exception handling

Example of `map` with exception handling:

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger('paramiko').setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
```

(continues on next page)

(continued from previous page)

```
    ip = device_dict['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1': time.sleep(5)

    try:
        with ConnectHandler(**device_dict) as ssh:
            ssh.enable()
            result = ssh.send_command(command)
            logging.info(received_msg.format(datetime.now().time(), ip))
        return result
    except NetMikoAuthenticationException as err:
        logging.warning(err)


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        result = executor.map(send_show, devices, repeat(command))
        for device, output in zip(devices, result):
            data[device['host']] = output
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh ip int br'))
```

Example is generally similar to the previous one but NetMikoAuthenticationException was introduced in send_show function, and the code that started send_show function in threads is now in send_command_to_devices function.

When using map method, exception handling is best done within a function that runs in threads, in this case send_show function.

Method submit and work with futures

Method submit differs from map method:

- submit runs only one function in thread
- submit can run different functions with different unrelated arguments, when map must run with iterable objects as arguments
- submit immediately returns the result without having to wait for function execution

- submit returns special Future object that represents execution of function.
  - submit returns Future in order that the call of submit does not block the code. Once submit has returned Future, code can be executed further. And once all functions in threads are running, you can start requesting Future if results are ready. Or take advantage of special function as_completed, which requests the result itself and code gets it when it's ready
- submit returns results in readiness order, not in argument order
- submit can pass key arguments when map only position arguments

Method submit uses Future object - an object that represents a delayed computation. This object can be requested for status (completed or not), and results or exceptions can be obtained from the job. Future does not need to be created manually, these objects are created by submit.

Example of running a function in threads using submit (netmiko_threads_submit_basics.py file)

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received: {}'
    ip = device_dict['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
```

(continues on next page)

(continued from previous page)

```
    return {ip: result}


with open('devices.yaml') as f:
    devices = yaml.safe_load(f)

with ThreadPoolExecutor(max_workers=2) as executor:
    future_list = []
    for device in devices:
        future = executor.submit(send_show, device, 'sh clock')
        future_list.append(future)
    # the same in the form of list comprehensions:
    # future_list = [executor.submit(send_show, device, 'sh clock') for device in␣
↪devices]
    for f in as_completed(future_list):
        print(f.result())
```

The rest of the code has not changed, so you only need to understand the block which runs send_show function in threads:

```
with ThreadPoolExecutor(max_workers=2) as executor:
    future_list = []
    for device in devices:
        future = executor.submit(send_show, device, 'sh clock')
        future_list.append(future)
    for f in as_completed(future_list):
        print(f.result())
```

The rest of the code has not changed, so only block that runs send_show needs an attention:

```
with ThreadPoolExecutor(max_workers=2) as executor:
    future_list = []
    for device in devices:
        future = executor.submit(send_show, device, 'sh clock')
        future_list.append(future)
    for f in as_completed(future_list):
        print(f.result())
```

Now block with has two cycles:

- future_list - a list of Future objects:
  - submit function is used to create Future object
  - submit expects the name of function to be executed and its arguments

- the next cycle runs through future_list using `as_completed` function. This function returns a Future objects only when they have finished or been cancelled. Future is then returned as soon as work is completed, not in the order of adding to future_list

---

Note: Creation of list with Future can be done with list comprehensions: `future_list = [executor.submit(send_show, device, 'sh clock') for device in devices]`

The result is:

```
$ python netmiko_threads_submit_basics.py
ThreadPoolExecutor-0_0 root INFO: ===> 17:32:59.088025 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 17:32:59.094103 Connection: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: <=== 17:33:11.639672 Received: 192.168.100.2
{'192.168.100.2': '*17:33:11.429 UTC Thu Jul 4 2019'}
ThreadPoolExecutor-0_1 root INFO: ===> 17:33:11.849132 Connection: 192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 17:33:17.735761 Received: 192.168.100.1
{'192.168.100.1': '*17:33:17.694 UTC Thu Jul 4 2019'}
ThreadPoolExecutor-0_1 root INFO: <=== 17:33:23.230123 Received: 192.168.100.3
{'192.168.100.3': '*17:33:23.188 UTC Thu Jul 4 2019'}
```

Please note that the order is not preserved and depends on which function was previously completed.

Future

An example of running `send_show` function with `submit` and displaying information about Future (note the status of Future at different points in time):

```
In [1]: from concurrent.futures import ThreadPoolExecutor

In [2]: from netmiko_threads_submit_futures import send_show

In [3]: executor = ThreadPoolExecutor(max_workers=2)

In [4]: f1 = executor.submit(send_show, r1, 'sh clock')
   ...: f2 = executor.submit(send_show, r2, 'sh clock')
   ...: f3 = executor.submit(send_show, r3, 'sh clock')
   ...:
ThreadPoolExecutor-0_0 root INFO: ===> 17:53:19.656867 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 17:53:19.657252 Connection: 192.168.100.2

In [5]: print(f1, f2, f3, sep='\n')
<Future at 0xb488e2ac state=running>
<Future at 0xb488ef2c state=running>
```

(continues on next page)

(continued from previous page)

```
<Future at 0xb488e72c state=pending>

ThreadPoolExecutor-0_1 root INFO: <=== 17:53:25.757704 Received: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 17:53:25.869368 Connection: 192.168.100.3

In [6]: print(f1, f2, f3, sep='\n')
<Future at 0xb488e2ac state=running>
<Future at 0xb488ef2c state=finished returned dict>
<Future at 0xb488e72c state=running>

ThreadPoolExecutor-0_0 root INFO: <=== 17:53:30.431207 Received: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: <=== 17:53:31.636523 Received: 192.168.100.3

In [7]: print(f1, f2, f3, sep='\n')
<Future at 0xb488e2ac state=finished returned dict>
<Future at 0xb488ef2c state=finished returned dict>
<Future at 0xb488e72c state=finished returned dict>
```

In order to look at Future, several lines with information output are added to the script (netmiko_threads_submit_futures.py):

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received: {}'
    ip = device_dict['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
```

(continues on next page)

(continued from previous page)

```
    if ip == '192.168.100.1':
        time.sleep(5)

    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return {ip: result}


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        future_list = []
        for device in devices:
            future = executor.submit(send_show, device, command)
            future_list.append(future)
            print('Future: {} for device {}'.format(future, device['host']))
        for f in as_completed(future_list):
            result = f.result()
            print('Future done {}'.format(f))
            data.update(result)
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh clock'))
```

The result is:

```
$ python netmiko_threads_submit_futures.py
Future: <Future at 0xb5ed938c state=running> for device 192.168.100.1
ThreadPoolExecutor-0_0 root INFO: ===> 07:14:26.298007 Connection: 192.168.100.1
Future: <Future at 0xb5ed96cc state=running> for device 192.168.100.2
Future: <Future at 0xb5ed986c state=pending> for device 192.168.100.3
ThreadPoolExecutor-0_1 root INFO: ===> 07:14:26.299095 Connection: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: <=== 07:14:32.056003 Received: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 07:14:32.164774 Connection: 192.168.100.3
Future done <Future at 0xb5ed96cc state=finished returned dict>
ThreadPoolExecutor-0_0 root INFO: <=== 07:14:36.714923 Received: 192.168.100.1
Future done <Future at 0xb5ed938c state=finished returned dict>
```

(continues on next page)

(continued from previous page)

```
ThreadPoolExecutor-0_1 root INFO: <=== 07:14:37.577327 Received: 192.168.100.3
Future done <Future at 0xb5ed986c state=finished returned dict>
{'192.168.100.1': '*07:14:36.546 UTC Fri Jul 26 2019',
 '192.168.100.2': '*07:14:31.865 UTC Fri Jul 26 2019',
 '192.168.100.3': '*07:14:37.413 UTC Fri Jul 26 2019'}
```

Since two threads are used by default, only two out of three Future shows running status. The third is in pending state and is waiting for queue to arrive.

Processing of exceptions

If there is an exception in function execution, it will be generated when the result is obtained For example, in device.yaml file the password for device 192.168.100.2 was changed to the wrong one:

```
$ python netmiko_threads_submit.py
===> 06:29:40.871851 Connection to device: 192.168.100.1
===> 06:29:40.872888 Connection to device: 192.168.100.2
===> 06:29:43.571296 Connection to device: 192.168.100.3
<=== 06:29:48.921702 Received result from device: 192.168.100.3
<=== 06:29:56.269284 Received result from device: 192.168.100.1
Traceback (most recent call last):
...
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/netmiko/base_
↪connection.py", line 500, in establish_connection
    raise NetMikoAuthenticationException(msg)
netmiko.ssh_exception.NetMikoAuthenticationException: Authentication failure:␣
↪unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
```

Since an exception occurs when result is obtained, it is easy to add exception processing (netmiko_threads_submit_exception.py file):

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat
import logging

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException
```

(continues on next page)

(continued from previous page)

```
logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)

start_msg = '===> {} Connection: {}'
received_msg = '<=== {} Received: {}'


def send_show(device_dict, command):
    ip = device_dict['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1': time.sleep(5)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return {ip: result}


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        future_ssh = [
            executor.submit(send_show, device, command) for device in devices
        ]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                data.update(result)
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh clock'))
```

The result is:

```
$ python netmiko_threads_submit_exception.py
ThreadPoolExecutor-0_0 root INFO: ===> 07:21:21.190544 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 07:21:21.191429 Connection: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 07:21:23.672425 Connection: 192.168.100.3
Authentication failure: unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
ThreadPoolExecutor-0_1 root INFO: <=== 07:21:29.095289 Received: 192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 07:21:31.607635 Received: 192.168.100.1
{'192.168.100.1': '*07:21:31.436 UTC Fri Jul 26 2019',
 '192.168.100.3': '*07:21:28.930 UTC Fri Jul 26 2019'}
```

Of course, exception handling can be performed within send_show function, but it is just an example of how you can work with exceptions when using a Future.

Using ProcessPoolExecutor

Interface of concurrent.futures module is very convenient because migration from threads to processes is done by replacing ThreadPoolExecutor with ProcessPoolExecutor, so all examples below are completely similar to examples with threads.

Method map

To use processes instead of threads, it is sufficient to change ThreadPoolExecutor to ProcessPoolExecutor:

```
from concurrent.futures import ProcessPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger('paramiko').setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)
```

(continues on next page)

(continued from previous page)

```
def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device_dict['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1': time.sleep(5)

    try:
        with ConnectHandler(**device_dict) as ssh:
            ssh.enable()
            result = ssh.send_command(command)
            logging.info(received_msg.format(datetime.now().time(), ip))
        return result
    except NetMikoAuthenticationException as err:
        logging.warning(err)


def send_command_to_devices(devices, command):
    data = {}
    with ProcessPoolExecutor(max_workers=2) as executor:
        result = executor.map(send_show, devices, repeat(command))
        for device, output in zip(devices, result):
            data[device['host']] = output
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh clock'))
```

Result of execution:

```
$ python netmiko_processes_map.py
MainThread root INFO: ===> 08:35:50.931629 Connection: 192.168.100.2
MainThread root INFO: ===> 08:35:50.931295 Connection: 192.168.100.1
MainThread root INFO: <=== 08:35:56.353774 Received:   192.168.100.2
MainThread root INFO: ===> 08:35:56.469854 Connection: 192.168.100.3
MainThread root INFO: <=== 08:36:01.410230 Received:   192.168.100.1
MainThread root INFO: <=== 08:36:02.067678 Received:   192.168.100.3
{'192.168.100.1': '*08:36:01.242 UTC Fri Jul 26 2019',
```

(continues on next page)

(continued from previous page)

```
 '192.168.100.2': '*08:35:56.185 UTC Fri Jul 26 2019',
 '192.168.100.3': '*08:36:01.900 UTC Fri Jul 26 2019'}
```

Method submit

File netmiko_processes_submit_exception.py:

```
from concurrent.futures import ProcessPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat
import logging

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException

logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)

start_msg = '===> {} Connection: {}'
received_msg = '<=== {} Received: {}'


def send_show(device_dict, command):
    ip = device_dict['host']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1': time.sleep(5)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return {ip: result}


def send_command_to_devices(devices, command):
    data = {}
    with ProcessPoolExecutor(max_workers=2) as executor:
```

(continues on next page)

(continued from previous page)

```
        future_ssh = [
            executor.submit(send_show, device, command) for device in devices
        ]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                data.update(result)
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh clock'))
```

Result of execution:

```
$ python netmiko_processes_submit_exception.py
MainThread root INFO: ===> 08:38:08.780267 Connection: 192.168.100.1
MainThread root INFO: ===> 08:38:08.781355 Connection: 192.168.100.2
MainThread root INFO: <=== 08:38:14.420339 Received: 192.168.100.2
MainThread root INFO: ===> 08:38:14.529405 Connection: 192.168.100.3
MainThread root INFO: <=== 08:38:19.224554 Received: 192.168.100.1
MainThread root INFO: <=== 08:38:20.162920 Received: 192.168.100.3
{'192.168.100.1': '*08:38:19.058 UTC Fri Jul 26 2019',
 '192.168.100.2': '*08:38:14.250 UTC Fri Jul 26 2019',
 '192.168.100.3': '*08:38:19.995 UTC Fri Jul 26 2019'}
```

Further reading

Python documentation:

- concurrent.futures — Launching parallel tasks
- threading
- multiprocessing
- queue
- PEP 3148

- PyMOTW. concurrent.futures — Manage Pools of Concurrent Tasks

GIL

- Can't we get rid of the Global Interpreter Lock?
- Understanding the Python GIL
- Python threads and the GIL

concurrent.futures

- A quick introduction to the concurrent.futures module
- Python - paralellizing CPU-bound tasks with concurrent.futures
- concurrent.futures in Python 3

Useful questions and answers on stackoverflow

- How many processes should I run in parallel?
- How many threads is too many?

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 19.1

Create a ping_ip_addresses function that checks if IP addresses are pingable. Checking IP addresses should be done concurrent in different threads.

Ping_ip_addresses function parameters:

- ip_list - list of IP addresses
- limit - maximum number of parallel threads (default 3)

The function must return a tuple with two lists:

- list of available IP addresses
- list of unavailable IP addresses

You can create any additional functions to complete the task. To check the availability of an IP address, use ping.

Note: A hint about working with concurrent.futures: If you need to ping several IP addresses in different threads, you need to create a function that will ping one IP address, and then run this function in different threads for different IP addresses using concurrent.futures (this last part must be done in the ping_ip_addresses function).

Task 19.2

Create a send_show_command_to_devices function that sends the same show command to different devices in concurrent threads and then writes the output of the commands to a file. The output from the devices in the file can be in any order.

Function parameters:

- devices - a list of dictionaries with parameters for connecting to devices

- command - show command
- filename - is the name of a text file to which the output of all commands will be written
- limit - maximum number of concurrent threads (default 3)

The function returns None.

The output of the commands should be written to a plain text file in this format (before the output of the command, you must write the hostname and the command itself):

```
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                     up
Ethernet0/1                192.168.200.1   YES NVRAM  up                     up
R2#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.2   YES NVRAM  up                     up
Ethernet0/1                10.1.1.1        YES NVRAM  administratively down down
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                     up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
```

You can create any additional functions to complete the task.

Check the operation of the function on devices from the devices.yaml file.

Task 19.3

Create a send_command_to_devices function that sends different show commands to different devices in concurrent threads and then writes the output of the commands to a file. The output from the devices in the file can be in any order.

Function parameters:

- devices - a list of dictionaries with parameters for connecting to devices
- commands_dict - a dictionary that specifies which device to send which command. Dictionary example - commands
- filename is the name of the file to which the output of all commands will be written
- limit - maximum number of concurrent threads (default 3)

The function returns None.

The output of the commands should be written to a plain text file in this format (before the output of the command, you must write the hostname and the command itself):

```
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                     up
Ethernet0/1                192.168.200.1   YES NVRAM  up                     up
R2#sh int desc
Interface                      Status         Protocol Description
Et0/0                          up             up
Et0/1                          up             up
Et0/2                          admin down     down
Et0/3                          admin down     down
Lo9                            up             up
Lo19                           up             up
R3#sh run | s ^router ospf
router ospf 1
 network 0.0.0.0 255.255.255.255 area 0
```

You can create any additional functions to complete the task.

Check the operation of the function on devices from the devices.yaml file.

```
# This dictionary is only needed to check the operation of the code;
# you can change the IP addresses in it.
# The test takes addresses from the devices.yaml file
commands = {
    "192.168.100.3": "sh run | s ^router ospf",
    "192.168.100.1": "sh ip int br",
    "192.168.100.2": "sh int desc",
}
```

Task 19.3a

Create a send_command_to_devices function that sends a list of the specified show commands to different devices in concurrent threads, and then writes the output of the commands to a file. The output from the devices in the file can be in any order.

Function parameters:

- devices - a list of dictionaries with parameters for connecting to devices
- commands_dict - a dictionary that specifies which device to send which commands. Dictionary example - commands

- filename is the name of the file to which the output of all commands will be written
- limit - maximum number of parallel threads (default 3)

The function returns None.

The output of the commands should be written to a plain text file in this format (before the output of the command, you must write the hostname and the command itself):

```
R2#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  192.168.100.1          87   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2           -   aabb.cc00.6600  ARPA   Ethernet0/0
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
R1#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.30.0.1               -   aabb.cc00.6530  ARPA   Ethernet0/3.300
Internet  10.100.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.100
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
R3#sh ip route | ex -

Gateway of last resort is not set

      10.0.0.0/8 is variably subnetted, 4 subnets, 2 masks
O        10.1.1.1/32 [110/11] via 192.168.100.1, 07:12:03, Ethernet0/0
O        10.30.0.0/24 [110/20] via 192.168.100.1, 07:12:03, Ethernet0/0
```

Commands can be written to a file in any order. To complete the task, you can create any additional functions, as well as use the functions created in previous tasks.

Check the operation of the function on devices from the devices.yaml file and the commands dictionary

```
# This dictionary is only needed to check the operation of the code;
# you can change the IP addresses in it.
# The test takes addresses from the devices.yaml file
commands = {
    "192.168.100.3": ["sh ip int br", "sh ip route | ex -"],
```

(continues on next page)

(continued from previous page)

```
    "192.168.100.1": ["sh ip int br", "sh int desc"],
    "192.168.100.2": ["sh int desc"],
}
```

Task 19.4

Create a send_commands_to_devices function that sends a show or config command to different devices in concurrent threads and then writes the output of the commands to a file.

Function parameters:

- devices - a list of dictionaries with parameters for connecting to devices
- filename is the name of the file to which the output of all commands will be written
- show - the show command to be sent (by default, the value is None)
- config - configuration mode commands to be sent (default None)
- limit - maximum number of parallel threads (default 3)

The function returns None.

The show, config and limit arguments should only be passed as keyword arguments. Passing these arguments as positional should raise a TypeError exception.

```
In [4]: send_commands_to_devices(devices, 'result.txt', 'sh clock')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-75adcfb4a005> in <module>
----> 1 send_commands_to_devices(devices, 'result.txt', 'sh clock')

TypeError: send_commands_to_devices() takes 2 positional argument but 3 were given
```

When calling the send_commands_to_devices function, only one of the show, config arguments should always be passed. If both arguments are passed, a ValueError exception should be raised.

The output of the commands should be written to a plain text file in this format (before the output of the command, you must write the hostname and the command itself):

```
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                     up
Ethernet0/1                192.168.200.1   YES NVRAM  up                     up
R2#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
```

(continues on next page)

(continued from previous page)

```
Internet  192.168.100.1          76   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2           -   aabb.cc00.6600  ARPA   Ethernet0/0
Internet  192.168.100.3         173   aabb.cc00.6700  ARPA   Ethernet0/0
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
```

An example of a function call:

```
In [5]: send_commands_to_devices(devices, 'result.txt', show='sh clock')

In [6]: cat result.txt
R1#sh clock
*04:56:34.668 UTC Sat Mar 23 2019
R2#sh clock
*04:56:34.687 UTC Sat Mar 23 2019
R3#sh clock
*04:56:40.354 UTC Sat Mar 23 2019

In [11]: send_commands_to_devices(devices, 'result.txt', config='logging 10.5.5.5
↪')

In [12]: cat result.txt
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#logging 10.5.5.5
R1(config)#end
R1#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R2(config)#logging 10.5.5.5
R2(config)#end
R2#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R3(config)#logging 10.5.5.5
R3(config)#end
R3#

In [13]: commands = ['router ospf 55', 'network 0.0.0.0 255.255.255.255 area 0']
```

(continues on next page)

(continued from previous page)

```
In [13]: send_commands_to_devices(devices, 'result.txt', config=commands)

In [14]: cat result.txt
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#router ospf 55
R1(config-router)#network 0.0.0.0 255.255.255.255 area 0
R1(config-router)#end
R1#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R2(config)#router ospf 55
R2(config-router)#network 0.0.0.0 255.255.255.255 area 0
R2(config-router)#end
R2#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R3(config)#router ospf 55
R3(config-router)#network 0.0.0.0 255.255.255.255 area 0
R3(config-router)#end
R3#
```

You can create any additional functions to complete task.

20. Jinja2 configuration templates

Jinja2 is a template language used in Python. Jinja is not the only template language (template engine) for Python and not the only template language in general.

Jinja2 is used to generate documents based on one or more templates.

Examples of use:

- templates for generating HTML pages
- templates for generating configuration files in Unix/Linux
- templates for generating network device configuration files

Getting started with Jinja2

You can install Jinja2 using pip:

```
pip install jinja2
```

Note: Further, terms Jinja and Jinja2 are used interchangeably.

The main idea of Jinja is to separate data and template. This allows you to use the same template but not the same data. In the simplest case, template is simply a text file that specifies locations of Jinja variables.

Example of Jinja 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
```

Comments to template:

- In Jinja, variables are written in double curly braces.
- When script is executed, these variables are replaced with desired values.

This template can be used to generate configuration of different devices by substituting other sets of variables.

Example of using Jinja

Template templates/router_template.txt is a plain text file:

```
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}}
 xconnect 10.10.{{to_id}}.1 {{to_id}}{{id}}11 encapsulation mpls
  backup peer 10.10.{{to_id}}.2 {{to_id}}{{id}}21
  backup delay 1 1
!
```

(continues on next page)

(continued from previous page)

```
router ospf 10
 router-id 10.10.{{id}}.1
 auto-cost reference-bandwidth 10000
 network 10.0.0.0 0.255.255.255 area 0
 !
```

Data file routers_info.yml

```
- 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
```

Script to generate configurations router_config_generator_ver2.py

```
from jinja2 import Environment, FileSystemLoader
import yaml

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

with open('routers_info.yml') as f:
    routers = yaml.safe_load(f)

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

(continues on next page)

(continued from previous page)

```
        f.write(template.render(router))
```

File router_config_generator.py imports from jinja2 module:

- `FileSystemLoader` - a loader that allows working with a file system
  - path to directory where templates are located is specified here
  - in this case template is in templates directory
- **`Environment` - a class for describing environment parameters. In this** case only loader is specified, but you can specify how to process a template

Note that template is now in `templates` directory.

Jinja2 template syntax

So far, only variable substitution has been used in Jinja2 template examples. This is the simplest and most understandable example of using templates. Syntax of Jinja templates is not limited to this.

In Jinja2 templates you can use :

- variables
- conditions (if/else)
- loops (for)
- filters - special built-in methods that allow to convert variables
- tests - are used to check whether a variable matches a condition

In addition, Jinja supports inheritance between templates and also allows adding the contents of one template to another. This section covers only few possibilities. More information about Jinja2 templates can be found in documentation.

Note: All files used as examples in this subsection are in 3_template_syntax/ directory

---

Script cfg_gen.py will be used to generate templates.

```
from jinja2 import Environment, FileSystemLoader
import yaml
import sys
import os

# python cfg_gen.py templates/for.txt data_files/for.yml
```

(continues on next page)

(continued from previous page)

```
template_dir, template = os.path.split(sys.argv[1])

vars_file = sys.argv[2]

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

with open(vars_file) as f:
    vars_dict = yaml.safe_load(f)

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

In order to see the result, you have to call the script and give it two arguments:

- template
- file with variables in YAML format

The result will be displayed on standard output stream.

Example of script run:

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

Parameters trim_blocks and lstrip_blocks are described in the following subsection.

Control of whitespace symbols

trim_blocks, lstrip_blocks

Parameter `trim_blocks` removes the first empty line after block if its value is True (default False).

Effect of using the flag is showed on example 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 %}
```

If cfg_gen.py script starts without trim_blocks, lstrip_blocks:

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

The output is:

```
$ 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
```

new lines occur because of `for` block.

```
{% for ibgp in bgp.ibgp_neighbors %}
```

By default, the same behavior will be with any other Jinja blocks.

When trim_blocks flag is added:

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

The result will be:

```
$ 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
```

Empty lines after block were removed.

In front of `neighbor ... remote-as` lines two spaces appeared. This is because there is a space in front of `for` block. Once lstrip_blocks has been disabled, spaces and tabs in front of the block are added to the first line of block.

This does not affect the next lines. Therefore, lines with `neighbor ... update-source` are displayed with one space.

Parameter `lstrip_blocks` controls whether spaces and tabs will be removed from the beginning of line to the beginning of block (untill opening curly bracket).

If add `lstrip_blocks=True`:

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

The result will be:

```
$ 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
```

Disabling lstrip_blocks for block

Sometimes you need to disable lstrip_blocks in block.

For example, if `lstrip_blocks` is set to True in an environment, but must be disabled for the second block in template (templates/flagenv_s2.txt file):

```
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 %}
```

The result will be:

```
$ 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
```

(continues on next page)

(continued from previous page)

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

Plus sign after percent sign disables lstrip_blocks for the block, in this case, only in the beginning.

If done this way (plus is added in the end block expression):

```
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 %}
```

It will be disabled for the end of the block:

```
$ 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
```

Removing whitespace from block

Similarly, you can control whitespace removal for a block.

For this example, flags are not set in environment:

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

Template 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 %}
```

Note the minus at the beginning of second block. Minus removes all whitespace characters, in this case, at the beginning of the block.

The result will be:

```
$ 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
```

If you add minus to the end of the block:

```
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 %}
```

(continues on next page)

(continued from previous page)

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

Empty string at the end of the block will be deleted:

```
$ 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
```

Try to add minus at the end of expressions describing the block and look at the result:

```
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 -%}
```

Variables

Variables in template are given in double curly braces:

```
hostname {{ name }}

```

(continues on next page)

(continued from previous page)

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

Variable values are set based on dictionary that is passed to template.

Variable that is passed on in a dictionary may not only be a number or a string, but also for example, a list or a dictionary. Inside template, you can refer to the item by number or key.

Template example templates/variables.txt with usage of different variable variants:

```
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'] }}
```

And corresponding data_files/vars.yml file with variables:

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

Note the use of vlans variable in template: since vlans variable is a list, you can specify which item from list we need

If a dictionary is passed (as in case of ospf variable), you can refer to dictionary objects inside template using one of the variants: `ospf.network or ospf['network']`

The result will be:

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

```

(continues on next page)

(continued from previous page)

```
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
```

Loop for

Loop `for` allows you to walk through sequence of elements.

Loop `for` must be written inside `{% %}`. Furthermore, the end of the loop must be explicitly indicated:

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

Template example templates/for.txt using a loop:

```
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 %}
```

File data_files/for.yml with variables:

```
id: 3
name: R3
```

(continues on next page)

(continued from previous page)

```
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
```

In `for`, it is possible to go through both the list elements (for example, ospf list) and the dictionary (vlans dictionary). And similarly, through any sequence.

The result will be:

```
$ 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` allows you to add a condition to template. For example, you can use `if` to add parts of template depending on the presence of variables in data dictionary.

`if` statement must also be within inside `{% %}`. End of condition must be explicitly stated:

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

Template example 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 ospf` expression works the same way as in Python: if variable exists and is not empty, the result is True. If there is no variable or it is empty, the result is False. That is, in this template the OSPF configuration is generated only if variable `ospf` exists and is not empty. Configuration will be generated with two data variants.

First with data_files/if.yml that does not contain `ospf` variable:

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

The result will be:

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

```

(continues on next page)

(continued from previous page)

```
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
```

Now a similar template but with data_files/if_ospf.yml file:

```
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
```

Now the result will be:

```
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
```

(continues on next page)

(continued from previous page)

```
 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
```

As in Python, Jinja is allowed to make branches in condition.

Template example 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 %}
```

Data file data_files/if_vlans.yml:

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

In this example, different commands are generated depending on value of `action` parameter.

In template you could also use this option to refer to nested dictionaries:

```
{% 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 %}
```

(continues on next page)

(continued from previous page)

```
 switchport trunk allowed vlan {{ trunks[intf]['vlans'] }}
 {% endif %}
{% endfor %}
```

This will result in the following configuration:

```
$ 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
```

Using `if` you can also filter which elements of sequence will be iterated in `for` loop.

Template example templates/if_for.txt with filter in `for` loop:

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

Data file (data_files/if_for.yml):

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

The result will be:

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

Filters

In Jinja, variables can be changed by filters. Filters are separated from variable by a vertical line (pipe |) and may contain additional arguments. In addition, several filters can be applied to variable. In this case, filters are simply written consecutively and each of them is separated by a vertical line.

Jinja supports a large number of built-in filters. We will look at only a few of them. Other filters can be found in documentation.

You can also easily create your own filters. We will not cover this possibility but it is well documented.

default

Filter `default` allows you to set default value for variable. If variable is defined, it will be displayed, if variable is not defined, the value specified in default filter will be displayed.

Template example 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 %}
```

If variable ref_bw is defined in dictionary, its value will be set. If there is no variable, the value of 10000 will be substituted.

Data file (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
```

The result of execution:

```
$ 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
```

By default, if variable is defined and its value is empty, it will be assumed that variable and its value exist.

If you want default value to be set also when variable is empty (i.e., treated as False in Python), you need to specify additional parameter `boolean=true`.

For example, if data file is:

```
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
```

The result will be:

```
$ 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
```

If with the same data file the template will be changed as follows:

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

Note: Instead of `default(10000, boolean=true)` you can write default(10000, true)

The result will be (default value is set):

```
$ 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

Filter `dictsort` allows you to sort the dictionary. By default, sorting is done by keys but by changing filter parameters you can sort by values.

Filter syntax:

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

After `dictsort` sorts the dictionary, it returns a list of tuples, not a dictionary.

Template example templates/filter_dictsort.txt using `dictsort` filter:

```
{% 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 %}
```

Note that filter awaits a dictionary, not a list of tuples or iterator.

Data file (data_files/filter_dictsort.yml):

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

The result of execution will be (interfaces are ordered):

```
$ 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

Filter `join` works just like `join` method in Python.

With `join` filter you can combine sequence of elements into a string with an optional separator between elements.

Template example templates/filter_join.txt using `join` filter:

```
{% 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 %}
```

Data file (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
```

The result of execution:

```
$ 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
```

Tests

Besides filters, Jinja also supports tests. Tests allow variables to be tested for a certain condition.

Jinja supports a large number of built-in tests. We will look at only a few of them. The rest of tests you can find in documentation.

defined

Test `defined` allows you to check if variable is present in the data dictionary.

Template example 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 %}
```

This example is more cumbersome than `default` filter option, but this test may be useful if depending on whether a variable is defined or not, different commands need to be executed.

Data file (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
```

The result of execution:

```
$ 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

Test `iterable` checks whether the object is an iterator. Due to these checks, it is possible to make branches in template which will take into account the type of variable.

Template templates/test_iterable.txt (indents made to make an idea of branches more clear):

```
{% 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' %}
 switchport trunk allowed vlan remove {{ params.vlans }}
   {% else %}
 switchport trunk allowed vlan {{ params.vlans }}
   {% endif %}
 {% endif %}
{% endfor %}
```

Data file (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
```

Note the last line: `vlans: 10`. In this case, 10 is no longer in the list and `join` filter does not work. But, due to `is iterable` test (in this case the result will be false), in this case template goes into `else` branch.

The result of execution:

```
$ 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
```

Such indents appeared because the template uses indents but does not have `lstrip_blocks=True` installed (it removes spaces and tabs at the beginning of the line).

set

You can assign values to variables inside template. These can be new variables or there may be modified values of variables that have been passed to template. In this way you can remember a value that for example was obtained by using several filters. Then use variable name instead of repeating all filters.

Template example templates/set.txt in which `set` expression is used to specify shorter parameter names:

```
{% 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 %}
  {% if action == 'add' %}
 switchport trunk allowed vlan add {{ vlans }}
  {% elif action == 'delete' %}
 switchport trunk allowed vlan remove {{ vlans }}
```

(continues on next page)

(continued from previous page)

```
  {% else %}
 switchport trunk allowed vlan {{ vlans }}
  {% endif %}
 {% endif %}
{% endfor %}
```

Note the second and third lines:

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

In this way new variables are created and these new values are used. It makes template look clearer.

Data file (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
```

The result of execution:

```
$ 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 expression allows you to add one template to another.

Variables that are transmitted as data must contain all data for both the master template and the one that is added through include.

Template templates/vlans.txt:

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

Template templates/ospf.txt:

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

Template 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 %}
```

Template templates/switch.txt uses created templates ospf and vlans:

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

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

Data file for configuration generation (data_files/switch.yml):

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

(continues on next page)

(continued from previous page)

```
  - 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
```

The result of script execution:

```
$ 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
```

The resulting configuration is as if lines from templates ospf.txt and vlans.txt were in switch.txt template.

Template templates/router.txt:

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

{% include 'bgp.txt' %}

logging {{ log_server }}
```

In this case, in addition to `include`, another line in template was added to show that `include` expressions can be mixed with normal template.

Data file (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
```

(continues on next page)

(continued from previous page)

```
    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
```

The result of script execution will be:

```
$ 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
```

Thanks to `include`, template templates/ospf.txt is used both in template templates/switch.txt and in template templates/router.txt, instead of repeating the same thing twice.

Template inheritance

Template inheritance is a very powerful functionality that avoids repetition of the same in different templates.

When using inheritance, there are:

- **`base template` - template that describes template skeleton.**
  - this template may contain any ordinary expressions or text. In addition, special `blocks`

are defined in this template.

- **`child template` - template that extends base template by filling in specified blocks.**

  - child templates can overwrite or supplement blocks defined in base template.

Example of base template 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
 transport input ssh
!
```

Note four blocks that are created in template:

```
{% block services %}
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
```

(continues on next page)

(continued from previous page)

```
service password-encryption
service sequence-numbers
{% endblock %}
!
{% block ospf %}
router ospf 1
 auto-cost reference-bandwidth 10000
{% endblock %}
!
{% block bgp %}
{% endblock %}
!
{% block alias %}
{% endblock %}
```

These are blanks for the corresponding configuration sections. A child template that uses this base template as a base can fill all or only some of the blocks.

Child template 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 %}
```

The first line in template templates/hq_router.txt is very important:

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

It is said that template hq_router.txt will be constructed on the basis of template base_router.txt.

Inside child template, everything happens inside blocks. Due to the blocks that have been defined in base template, child template can extend the parent template.

Note: Note that lines described in child template outside blocks are ignored.

There are four blocks in base template: services, ospf, bgp, alias. In child template only two of them are filled: ospf and alias. That's the convenience of inheritance. You don't have to fill all blocks in every child template.

In this way *ospf* and *alias* blocks are used differently. In base template, *ospf* block already has part of configuration:

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

Therefore, child template has a choice: use this configuration and supplement it or completely rewrite everything in child template.

In this case the configuration is supplemented. That is why in child template templates/hq_router.txt the *ospf* block starts with expression `{{ super() }}`:

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

`{{ super() }}` transfers content of this block from parent template to child template. Because of this, lines from parent are moved to child template.

Note: Expression `super` doesn't have to be at the beginning of the block. It could be anywhere in the block. Content of base template are moved to where `super` expression is located.

`alias` block simply describes the alias. And even if there were some settings in parent template, they would be substituted by content of child template.

Let's recap the rules for working with blocks. If block is created in parent template:

- no content - in child template you can fill this block or ignore it. If block is filled, it will contain only what was written in child template (example - `alias` block)
- with content - in child template you can perform such actions:
  - ignore block - in this case, child template will get content from parent template (example - *services* block)
  - rewrite block - then child template will contain only what it has
  - move content of the block from parent template and supplement it - then child template will contain both the content of the block from parent template and the content from child template. To pass content from parent template the expression `{{ super() }}` is used (example - *ospf* block)

Data file for template configuration generation (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
```

The result will be:

```
$ 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
```

(continues on next page)

(continued from previous page)

```
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
 logging synchronous
 history size 100
line vty 0 4
 logging synchronous
 history size 100
 transport input ssh
!
```

Note that in *ospf* block there are commands from base template and commands from child template.

Further reading

Documentation:

- General documentation Jinja2
- Template syntax

Articles:

- Network Configuration Templates Using Jinja2. Matt Oswalt
- Python And Jinja2 Tutorial. Jeremy Schulman
- Configuration Generator with Python and Jinja2
- Custom filters for a Jinja2 based Config Generator

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 20.1

Create generate_config function.

Function parameters:

- template - path to the template file (for example, "templates/for.txt")
- data_dict - a dictionary with values to be substituted into the template

The function should return the generated configuration string.

Check the operation of the function on the templates/for.txt template and data from the data_files/for.yml file.

```
import yaml


# function call should looks like
if __name__ == "__main__":
    data_file = "data_files/for.yml"
    template_file = "templates/for.txt"
    with open(data_file) as f:
        data = yaml.safe_load(f)
    print(generate_config(template_file, data))
```

Task 20.2

Create template templates/cisco_router_base.txt.

The templates/cisco_router_base.txt template should include the content of the templates:

- templates/cisco_base.txt
- templates/alias.txt

- templates/eem_int_desc.txt

Template text cannot be copied.

Test the template templates/cisco_router_base.txt using the generate_config function from task 20.1. Do not copy the code of the generate_config function.

As data, use the information from the data_files/router_info.yml file.

Task 20.3

Create a template templates/ospf.txt based on the OSPF configuration in the cisco_ospf.txt file. A configuration example is given to show the syntax.

The template must be created manually by copying parts of the config into the corresponding template.

What values should be variables:

- process number. Variable name - process
- router-id. Variable name - router_id
- reference-bandwidth. Variable name - ref_bw
- interfaces on which to enable OSPF. The variable name is ospf_intf. In place of this variable, a list of dictionaries with the following keys is expected:
  - name - interface name, like Fa0/1, Vlan10, Gi0/0
  - ip - interface IP address, like 10.0.1.1
  - area - zone number
  - passive - whether the interface is passive. Valid values: True or False

For all interfaces in the ospf_intf list, you need to generate the following lines:

```
network x.x.x.x 0.0.0.0 area x
```

If the interface is passive, the line must be added for it:

```
passive-interface x
```

For interfaces that are not passive, in interface configuration mode, you need to add the line:

```
ip ospf hello-interval 1
```

All commands must be in the appropriate configuration mode.

Check the resulting template templates/ospf.txt, against the data in the data_files/ospf.yml file, using the generate_config function from task 20.1. Do not copy the code of the generate_config function.

The result should be a configuration of the following type (the commands in router ospf mode do not have to be in this order, the main thing is that they are in the correct config section):

```
router ospf 10
 router-id 10.0.0.1
 auto-cost reference-bandwidth 20000
 network 10.255.0.1 0.0.0.0 area 0
 network 10.255.1.1 0.0.0.0 area 0
 network 10.255.2.1 0.0.0.0 area 0
 network 10.0.10.1 0.0.0.0 area 2
 network 10.0.20.1 0.0.0.0 area 2
 passive-interface Fa0/0.10
 passive-interface Fa0/0.20
interface Fa0/1
 ip ospf hello-interval 1
interface Fa0/1.100
 ip ospf hello-interval 1
interface Fa0/1.200
 ip ospf hello-interval 1
```

Задание 20.4

Create a template templates/add_vlan_to_switch.txt that will be used if you need to add a VLAN to the switch.

The template must support the following features:

- add VLAN and VLAN name
- adding VLANs as access, on the specified interface
- adding VLANs to the list of allowed, on specified trunks

The template must be created manually by copying parts of the config into the corresponding template.

If VLAN needs to be added as access, you need to configure the interface mode and add it to VLAN:

```
interface Gi0/1
 switchport mode access
 switchport access vlan 5
```

For trunks, you only need to add VLANs to the allowed list:

```
interface Gi0/10
 switchport trunk allowed vlan add 5
```

The variable names should be chosen based on the sample data in the data_files/add_vlan_to_switch.yaml file.

Check the templates/add_vlan_to_switch.txt template against the data in data_files/add_vlan_to_switch.yaml using the generate_config function from task 20.1. Do not copy the code of the generate_config function.

Task 20.5

Create templates templates/gre_ipsec_vpn_1.txt and templates/gre_ipsec_vpn_2.txt that generate IPsec over GRE configuration between two routers.

The templates/gre_ipsec_vpn_1.txt template creates the configuration for one side of the tunnel, and templates gre_ipsec_vpn_2.txt for the other.

Examples of the final configuration that should be generated from templates in the files: cisco_vpn_1.txt and cisco_vpn_2.txt.

Templates must be created manually by copying parts of the config into the corresponding templates.

Create a create_vpn_config function that uses these templates to generate a VPN configuration based on the data in the data dictionary.

Function parameters:

- template1 - the name of the template file that creates the configuration for one side of the tunnel
- template2 - the name of the template file that creates the configuration for the second side of the tunnel
- data_dict - a dictionary with values to be substituted into templates

The function must return a tuple with two configurations (strings) that are derived from templates.

Examples of VPN configurations that the create_vpn_config function should return in the cisco_vpn_1.txt and cisco_vpn_2.txt files.

```
data = {
    'tun_num': 10,
    'wan_ip_1': '192.168.100.1',
    'wan_ip_2': '192.168.100.2',
    'tun_ip_1': '10.0.1.1 255.255.255.252',
    'tun_ip_2': '10.0.1.2 255.255.255.252'
}
```

Task 20.5a

Create a configure_vpn function that uses the templates from task 20.5 to configure VPN on routers based on the data in the data dictionary.

Function parameters:

- src_device_params - dictionary with connection parameters for device 1
- dst_device_params - dictionary with connection parameters for device 2
- src_template - a template that creates the configuration for side 1
- dst_template - a template that creates the configuration for side 2
- vpn_data_dict - a dictionary with values to be substituted into templates

The function should configure the VPN based on templates and data on each device using netmiko. The function returns a tuple with the output of commands from two routers (the output returned by the netmiko send_config_set method). The first element of the tuple is the output from the first device (string), the second element of the tuple is the output from the second device.

In this task, the data dictionary does not specify the Tunnel interface number to use. The number must be determined independently based on information from the equipment. If the router does not have Tunnel interfaces, take the number 0, if there is, take the nearest free number, but the same for two routers.

For example, if the src router has the following interfaces: Tunnel1, Tunnel4. And on the dst router are: Tunnel2, Tunnel3, Tunnel8. The first free number that is the same for two routers will be 5. And you will need to configure the Tunnel 5 interface.

For this task, the test verifies that the function works on the first two devices from the devices.yaml file. And it checks that the output contains commands for configuring interfaces, but does not check the configured tunnel numbers and other commands. The tunnels must be configured, but the test has been simplified so that there are fewer constraints on the task.

```
data = {
    'tun_num': None,
    'wan_ip_1': '192.168.100.1',
    'wan_ip_2': '192.168.100.2',
    'tun_ip_1': '10.0.1.1 255.255.255.252',
    'tun_ip_2': '10.0.1.2 255.255.255.252'
}
```

21. Parsing command output with TextFSM

On network devices that does not support any software interface, the output of show commands is returned as a string. And although it's partly structured, it's still just a string. And it has to be processed in some way to get Python objects, like a dictionary or a list.

For example, it is possible to handle the output of a command line by line using regular expressions to get Python objects. But there is a more convenient option: TextFSM

TextFSM - a library created by Google to handle output from network devices. It allows you to create templates to process the output of a command.

Using TextFSM is better than simple line processing, as templates give a better idea of how output will be handled and templates are easier to share. That means it's easier to find templates that have already been created and use them or share your own.

Getting started with TextFSM

First, library must be installed:

```
pip install textfsm
```

To use TextFSM you should create a template to handle the output of command.

Example of traceroute command output:

```
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
  1 10.0.12.1 1 msec 0 msec 0 msec
  2 15.0.0.5  0 msec 5 msec 4 msec
  3 57.0.0.7  4 msec 1 msec 4 msec
  4 79.0.0.9  4 msec *  1 msec
```

For example, you have to get hops that packet went through.

In this case TextFSM template will look like this (traceroute.template file):

```
Value ID (\d+)
Value Hop (\S+)

Start
  ^  ${ID} ${Hop} +\d+ -> Record
```

First two lines define variables:

- `Value ID (\d+)` - this line defines an `ID` variable that describes a regular expression: `(\d+)` - one or more digits, here are the hop numbers
- `Value Hop (\S+)` - line that defines a `Hop` variable that describes an IP address by such regular expression

After `Start` line, the template itself begins. In this case, it's very simple:

- `^ ${ID} ${Hop} -> Record`
- first goes caret sign, then two spaces and `ID` and `Hop` variables
- in TextFSM the variables are written like this: `${variable name}`
- word `Record` at the end means that lines that matches regular expression will be processed and included in the results of TextFSM (we'll look at this in the next section)

Script to process output of traceroute command with TextFSM (parse_traceroute.py):

```
import textfsm

traceroute = '''
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
  1 10.0.12.1 1 msec 0 msec 0 msec
  2 15.0.0.5  0 msec 5 msec 4 msec
  3 57.0.0.7  4 msec 1 msec 4 msec
  4 79.0.0.9  4 msec *  1 msec
'''

with open('traceroute.template') as template:
    fsm = textfsm.TextFSM(template)
    result = fsm.ParseText(traceroute)

print(fsm.header)
print(result)
```

The result of script execution:

```
$ python parse_traceroute.py
['ID', 'Hop']
[['1', '10.0.12.1'], ['2', '15.0.0.5'], ['3', '57.0.0.7'], ['4', '79.0.0.9']]
```

Lines that match the described template are returned as a list of lists. Each element is a list that consists of two elements: hop number and IP address.

Let's sort out the content of script:

- traceroute - a variable that contains traceroute command output
- `template = open('traceroute.template')` - content of TextFSM template file is read into a *template* variable
- `fsm = textfsm.TextFSM(template)` - class that processes a template and creates an object from it in TextFSM
- `result = fsm.ParseText(traceroute)` - method that handles output according to a template and returns a list of lists in which each element is a processed string
- At the end, `print(fsm.header)` header is displayed which contains variable names and processing result

We can work with that output further. For example, periodically execute traceroute command and compare whether the number of hops and their order has changed.

To work with TextFSM you need command output and template:

- different commands need different templates
- TextFSM returns a tabular processing result (as a list of lists)
- this output is easily converted to csv format or to a list of dictionaries

TextFSM template syntax

This section covers template syntax based on TextFSM documentation. The following section shows examples of syntax usage. Therefore, you can move to the next section and return to this section as necessary for those situations for which there is no example and when you need to recall the meaning of parameter.

TextFSM template describes how data should be processed. Each template consists of two parts:

- value definitions (or variable definitions) - these variables describe which columns will be in the table view
- state definitions

Example of traceroute command examination:

```
Value ID (\d+)
Value Hop (\d+(\.\d+){3})

Start
   ^  ${ID} ${Hop} -> Record
```

Value definition

Only value definitions should be used in value section. The only exception there may be comments in this section. This section should not contain empty strings. For TextFSM, an empty string means the end of value definition section.

Format of value description:

```
Value [option[,option...]] name regex
```

Syntax of value description (for each option below we will consider examples):

- `Value` - keyword that indicates that a value is being created. It must be specified
- option - options that define how to work with a variable. If several options are to be specified, they must be separated by a comma, without spaces. These options are supported:
  - `Filldown` - value that previously matched with a regex, remembered until the next processing line (if has not been explicitly cleared or matched again). This means that the last column value that matches regex is stored and used in the following strings if this column is not present.
  - Key - determines that this field contains a unique string identifier
  - `Required` - string that is processed will only be written if this value is present.
  - `List` - value is a list, and each match with a regex will add an item to the list. By default, each next match overwrites the previous one.
  - `Fillup` - works as Filldown but fills empty value up until it finds a match. Not compatible with Required.
- `name` - name of value that will be used as column name. Reserved names should not be used as value names.
- `regex` - regex that describes a value. regex should be in curly braces.

State definition

After defining values, we need to defiine states:

- each state definition must be separated by an empty line (at least one)
- first line - state name
- then follows lines that describe rules. Rules must start with one or two spaces and caret symbol ^

Initial state is always `Start`. Input data is compared to the current state but rule line can specify that you want to go to a different state.

Checking is done line-by-line until E0F (end of file) is reached or the current state goes to End state.

Reserved states

The following states are reserved:

- `Start` - state that must be specified. Without it the template won't work.
- End - state that completes processing of incoming strings and does not execute EOF state.
- EOF - implicit state that always executes when processing reaches the end of the file. It looks like this:

```
EOF
  ^.* -> Record
```

EOF writes down the current string before it is finished. If this behavior needs to be changed you should explicitly write EOF at the end of template:

```
EOF
```

State rules

Each state consists of one or more rules:

- TextFSM handles incoming strings and compares them to rules
- if rule (regex) matches the string, actions in rule are executed and for the next string the process is repeated from the beginning of state.

Rules should be written in this format:

```
^regex [-> action]
```

In rule:

- each rule must start with two spaces and caret symbol ^. Caret symbol ^ means the beginning of a line and must always be clearly indicated
- regex - regex in which variables can be used
  - to specify variable you can use syntax like `$ValueName` or `${ValueName}`(preferred format)
  - in rule, variables are substituted by regex
  - if you want to explicitly specify end of line, use `$$`

Action in rules

After regex, rule may indicate actions:

- there must be -> character between regex and action
- actions can consist of three parts in such format: `L.R S`
  - `L - Line Action` - actions that apply to an input string
  - `R - Record Action` - actions that apply to collected values
  - `S - State Transition` - changing state
- By default `Next.NoRecord` is used

Line Actions

Line Actions:

- `Next` - process the line, read the next line and start checking it from the beginning. This action is used by default unless otherwise specified
- `Continue` - continue to process rules as if there was no match while values are assigned

Record Action

`Record Action` - optional action that can be specified after Line Action. They must be separated by a dot. Types of actions:

- `NoRecord` - do nothing. This is default action when no other is specified
- `Record` - all variables except those with Filldown option are reset.
- `Clear` - reset all variables except those where Filldown option is specified.
- `Clearall` - reset all variables.

You need to split actions with a dot only if you want to specify both Line and Record actions. If you need to specify only one of them, dot is not required.

State Transition

A new state can be specified after action:

- state must be one of reserved or defined in template
- if input line matches:
  - all actions are executed,
  - the next line is read,
  - then the current state changes to a new state and processing continues in new state.

If rule uses `Continue` action, it is not possible to change state inside this rule. This rule is needed to avoid loops in sequence of states.

Error Action

`Error` stops all line processing, discards all lines that have been collected so far and returns an exception.

Syntax of this action is:

```
^regex -> Error [word|"string"]
```

Examples of TextFSM usage

This section discusses examples of templates and TextFSM usage.

Section uses parse_output.py script to process command output by template. It is not tied to a specific template and output: template and command output will be passed as arguments:

```
import sys
import textfsm
from tabulate import tabulate

template = sys.argv[1]
output_file = sys.argv[2]

with open(template) as f, open(output_file) as output:
    re_table = textfsm.TextFSM(f)
    header = re_table.header
    result = re_table.ParseText(output.read())
    print(result)
    print(tabulate(result, headers=header))
```

Example of script run:

```
$ python parse_output.py template command_output
```

Note: Module `tabulate` is used to output data in tabular form (it must be installed if you want to use this script). A similar output could be received with string formatting but with tabulate it is easier to do.

Data processing by template is always done in the same way. Therefore, script will be the same only template and data will be different.

Starting with a simple example we'll figure out how to use TextFSM.

show clock

The first example is a review of "sh clock" command output (output/sh_clock.txt file):

```
15:10:44.867 UTC Sun Nov 13 2016
```

First of all, you have to define variables in template:

- at the beginning of each line there must be a keyword Value
- each variable defines column in table
- next word - variable name
- after name, in parentheses - a regex that describes data

Definition of variables is as follows:

```
Value Time (..:..:..)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)
```

Tips on special symbols:

- `.` - any character
- `+` - one or more repetitions of previous character
- `\S` - all characters except whitespace
- `\w` - any letter or number
- `\d` - any number

Once variables are defined, an empty line and `Start` state must follow, and then the rule follows starting with space and ^ symbol (templates/sh_clock.template file):

```
Value Time (..:..:..)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)

```

(continues on next page)

(continued from previous page)

```
Start
  ^${Time}.* ${Timezone} ${WeekDay} ${Month} ${MonthDay} ${Year} -> Record
```

Because in this case only one line in the output, it is not necessary to write `Record` action in template. But it is better to use it in situations where you have to write values and get used to this syntax and not make mistakes when you need to process multiple lines.

When TextFSM handles output strings it substitutes variable by its values. In the end, rule will look like:

```
^(..:..:..).* (\S+) (\w+) (\w+) (\d+) (\d+)
```

When this regex applies to "show clock" output, each regex group will have a corresponding value:

- 1 group: 15:10:44
- 2 group: UTC
- 3 group: Sun
- 4 group: Nov
- 5 group: 13
- 6 group: 2016

In addition to explicit `Record` action which specifies that record should be placed in final table, the `Next` rule is also used by default. It specifies that you want to go to the next line of text. Since there is only one line in "sh clock" command output, the processing is completed.

The result of script is:

```
$ python parse_output.py templates/sh_clock.template output/sh_clock.txt
Time      Timezone    WeekDay    Month      MonthDay    Year
--------  ----------  ---------  -------  ----------  ------
15:10:44  UTC         Sun        Nov              13    2016
```

show ip interface brief

In case when you need to process data displayed in columns, TextFSM template is the most convenient.

Template for "show ip interface brief" output (templates/sh_ip_int_br.template file):

```
Value INTF (\S+)
Value ADDR (\S+)
Value STATUS (up|down|administratively down)
```

(continues on next page)

(continued from previous page)

```
Value PROTO (up|down)

Start
  ^${INTF}\s+${ADDR}\s+\w+\s+\w+\s+${STATUS}\s+${PROTO} -> Record
```

In this case, the rule can be written in one line. Output command (output/sh_ip_int_br.txt file):

```
R1#show ip interface brief
Interface                  IP-Address      OK? Method Status                ␣
↪Protocol
FastEthernet0/0            15.0.15.1       YES manual up                    up
FastEthernet0/1            10.0.12.1       YES manual up                    up
FastEthernet0/2            10.0.13.1       YES manual up                    up
FastEthernet0/3            unassigned      YES unset  up                    up
Loopback0                  10.1.1.1        YES manual up                    up
Loopback100                100.0.0.1       YES manual up                    up
```

The result will be:

```
$ python parse_output.py templates/sh_ip_int_br.template output/sh_ip_int_br.txt
INT              ADDR        STATUS    PROTO
---------------  ----------  --------  -------
FastEthernet0/0  15.0.15.1   up        up
FastEthernet0/1  10.0.12.1   up        up
FastEthernet0/2  10.0.13.1   up        up
FastEthernet0/3  unassigned  up        up
Loopback0        10.1.1.1    up        up
Loopback100      100.0.0.1   up        up
```

show cdp neighbors detail

Now try to process output of command “show cdp neighbors detail”. Peculiarity of this command is that the data are not in the same line but in different lines.

File output/sh_cdp_n_det.txt contains output of “show cdp neighbors detail”:

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
```

(continues on next page)

(continued from previous page)

```
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9,␣
↪RELEASE SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2

-------------------------
Device ID: R1
Entry address(es):
  IP address: 10.1.1.1
Platform: Cisco 3825,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/22,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1,␣
↪RELEASE SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2009 by Cisco Systems, Inc.
Compiled Fri 19-Jun-09 18:40 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Duplex: full
Management address(es):

-------------------------
Device ID: R2
Entry address(es):
  IP address: 10.2.2.2
Platform: Cisco 2911,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/21,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec
```

(continues on next page)

(continued from previous page)

```
Version :
Cisco IOS Software, 2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1,␣
↪RELEASE SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2009 by Cisco Systems, Inc.
Compiled Fri 19-Jun-09 18:40 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Duplex: full
Management address(es):
```

From command output you need to get such fields:

- LOCAL_HOST - name of device from prompt
- DEST_HOST - neighbor name
- MGMNT_IP - neighbor IP address
- PLATFORM - model of neighbor device
- LOCAL_PORT - local interface that connects to a neighbor
- REMOTE_PORT - neighbor port
- IOS_VERSION - neighbor IOS version

Template looks like this (templates/sh_cdp_n_det.template file):

```
Value LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION},
```

The result of script execution:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM    LOCAL_PORT             REMOTE_
↪PORT         IOS_VERSION
------------  -----------  ----------  ----------  --------------------  --------
↪----------  -------------
SW1           R2           10.2.2.2    Cisco 2911  GigabitEthernet1/0/21 ␣
↪GigabitEthernet0/0  15.2(2)T1
```

Although rules with variables are described in different lines and accordingly work with different lines, TextFSM collects them into one line of the table. That is, variables that are defined at the beginning of template determine the string of resulting table.

Note that sh_cdp_n_det.txt file has three neighbors, but table has only one neighbor, the last one.

Record

This is because Record action is not specified in template. And only the last line left in final table.

Corrected template:

```
Value LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION}, -> Record
```

Now the result is:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM              LOCAL_PORT          ␣
↪  REMOTE_PORT         IOS_VERSION
------------  -----------  ----------  --------------------  --------------------
↪  ------------------  -------------
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/
↪16  GigabitEthernet0/1  12.2(55)SE9
```

(continues on next page)

(continued from previous page)

```
              R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/
↪22  GigabitEthernet0/0  12.4(24)T1
              R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/
↪21  GigabitEthernet0/0  15.2(2)T1
```

Output from all three devices. But LOCAL_HOST variable is not displayed in every line, only in the first one.

Filldown

This is because the prompt from which variable value is taken appears only once. And in order to make it appear in the next lines, use `Filldown` action for LOCAL_HOST variable:

```
Value Filldown LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION}, -> Record
```

Now we get this output:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM              LOCAL_PORT
↪  REMOTE_PORT         IOS_VERSION
------------  -----------  ----------  --------------------  ---------------------
↪  ------------------  -------------
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/
↪16  GigabitEthernet0/1  12.2(55)SE9
SW1           R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/
↪22  GigabitEthernet0/0  12.4(24)T1
SW1           R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/
↪21  GigabitEthernet0/0  15.2(2)T1
SW1
```

LOCAL_HOST now appears in all three lines. But there was another strange effect - the last line in which only LOCAL_HOST column is filled.

Required

The thing is, all variables we've determined are optional. Also, one variable with `Filldown` parameter. And to get rid of the last line, you have to make at least one variable mandatory by using `Required` option:

```
Value Filldown LOCAL_HOST (\S+)
Value Required DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
  ^${LOCAL_HOST}[>#].
  ^Device ID: ${DEST_HOST}
  ^.*IP address: ${MGMNT_IP}
  ^Platform: ${PLATFORM},
  ^Interface: ${LOCAL_PORT},  Port ID \(outgoing port\): ${REMOTE_PORT}
  ^.*Version ${IOS_VERSION}, -> Record
```

Now we get the correct output:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST    DEST_HOST    MGMNT_IP    PLATFORM              LOCAL_PORT
↪  REMOTE_PORT         IOS_VERSION
------------  -----------  ----------  --------------------  ----------------------
↪  ------------------  -------------
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/
↪16  GigabitEthernet0/1  12.2(55)SE9
SW1           R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/
↪22  GigabitEthernet0/0  12.4(24)T1
SW1           R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/
↪21  GigabitEthernet0/0  15.2(2)T1
```

show ip route ospf

Consider the case where we need to process output of "show ip route ospf" command and in routing table there are several routes to the same network.

For routes to the same network, instead of multiple lines where network is repeated, one record will be created in which all available next-hop addresses are in list.

Example of "show ip route ospf" output (output/sh_ip_route_ospf.txt file):

```
R1#sh ip route ospf
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
       D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
       N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
       E1 - OSPF external type 1, E2 - OSPF external type 2
       i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
       ia - IS-IS inter area, * - candidate default, U - per-user static route
       o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
       + - replicated route, % - next hop override

Gateway of last resort is not set

      10.0.0.0/8 is variably subnetted, 10 subnets, 2 masks
O        10.1.1.0/24 [110/20] via 10.0.12.2, 1w2d, Ethernet0/1
O        10.2.2.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
O        10.3.3.3/32 [110/11] via 10.0.12.2, 1w2d, Ethernet0/1
O        10.4.4.4/32 [110/11] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/11] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.5.5.5/32 [110/21] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/21] via 10.0.12.2, 1w2d, Ethernet0/1
                     [110/21] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.6.6.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
```

For this example we simplify the task and assume that routes can only be OSPF and only with "O" designation (i.e., only intra-zone routes).

The first version of template:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
```

The result is:

```
network      mask    distance    metric  nexthop
---------  ------  ----------  --------  ---------
```

(continues on next page)

(continued from previous page)

```
10.1.1.0       24          110         20  10.0.12.2
10.2.2.0       24          110         20  10.0.13.3
10.3.3.3       32          110         11  10.0.12.2
10.4.4.4       32          110         11  10.0.13.3
10.5.5.5       32          110         21  10.0.13.3
10.6.6.0       24          110         20  10.0.13.3
```

All right, but we've lost path options for routes 10.4.4.4/32 and 10.5.5.5/32. This is logical, because there is no rule that would be appropriate for such a line.

Add a rule to template for lines with partial entries:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
```

Now the output is:

```
network    mask      distance    metric  nexthop
---------  ------  ----------  --------  ---------
10.1.1.0   24             110        20  10.0.12.2
10.2.2.0   24             110        20  10.0.13.3
10.3.3.3   32             110        11  10.0.12.2
10.4.4.4   32             110        11  10.0.13.3
                          110        11  10.0.14.4
10.5.5.5   32             110        21  10.0.13.3
                          110        21  10.0.12.2
                          110        21  10.0.14.4
10.6.6.0   24             110        20  10.0.13.3
```

Partial entries are missing networks and masks, but in previous examples we have already covered Filldown and, if desired, it can be applied here. But for this example we will use another option - List.

List

Use `List` option for nexthop variable:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value List nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
```

Now the output is:

```
network    mask      distance    metric  nexthop
---------  ------  ----------  --------  -------------
10.1.1.0   24             110        20  ['10.0.12.2']
10.2.2.0   24             110        20  ['10.0.13.3']
10.3.3.3   32             110        11  ['10.0.12.2']
10.4.4.4   32             110        11  ['10.0.13.3']
                          110        11  ['10.0.14.4']
10.5.5.5   32             110        21  ['10.0.13.3']
                          110        21  ['10.0.12.2']
                          110        21  ['10.0.14.4']
10.6.6.0   24             110        20  ['10.0.13.3']
```

Now nexthop column displays a list but so far with one element. When using `List` the value is a list, and each match with a regex will add an item to the list. By default, each next match overwrites the previous one. If, for example, leave Record action for full lines only:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value List nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop},
```

The result will be:

```
network      mask    distance    metric  nexthop
---------  ------  ----------  --------  ----------------------------------------
10.1.1.0       24         110        20  ['10.0.12.2']
10.2.2.0       24         110        20  ['10.0.13.3']
```

(continues on next page)

(continued from previous page)

```
10.3.3.3       32          110         11   ['10.0.12.2']
10.4.4.4       32          110         11   ['10.0.13.3']
10.5.5.5       32          110         21   ['10.0.14.4', '10.0.13.3']
10.6.6.0       24          110         20   ['10.0.12.2', '10.0.14.4', '10.0.13.3']
```

Now the result is not quite correct, address hops are assigned to wrong routes. This happens because writing is done on full route entry, then hops of incomplete route entries are collected in the list (other variables are overwritten) and when the next full route entry appears, the list is written to it.

```
O        10.4.4.4/32 [110/11] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/11] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.5.5.5/32 [110/21] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/21] via 10.0.12.2, 1w2d, Ethernet0/1
                     [110/21] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.6.6.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
```

In fact, incomplete route entry should really be written when the next full route entry appears, but at the same time they should be written to appropriate route. The following should be done: once full route entry is met, the previous values should be written down and then continue to process the same full route entry to get its information. In TextFSM, you can do this with `Continue.Record`:

```
^O -> Continue.Record
```

Here, `Record` action tells you to write down the current value of variables. Since there are no variables in this rule, what was in the previous values is written.

`Continue` action says to continue working with the current line as if there was no match. So, the next line of template will work. The resulting template (templates/sh_ip_route_ospf.template):

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value List nexthop (\S+)

Start
  ^O -> Continue.Record
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop},
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop},
```

The result is:

```
network      mask    distance    metric  nexthop
---------  ------  ----------  --------  ----------------------------------------
```

(continues on next page)

(continued from previous page)

```
10.1.1.0        24          110         20   ['10.0.12.2']
10.2.2.0        24          110         20   ['10.0.13.3']
10.3.3.3        32          110         11   ['10.0.12.2']
10.4.4.4        32          110         11   ['10.0.13.3', '10.0.14.4']
10.5.5.5        32          110         21   ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.6.6.0        24          110         20   ['10.0.13.3']
```

show etherchannel summary

TextFSM is convenient to use to parse output that is displayed by columns or to process output that is in different lines. Templates are less convenient when it is necessary to get several identical elements from one line.

Example of "show etherchannel summary" output (output/sh_etherchannel_summary.txt file):

```
sw1# sh etherchannel summary
Flags:  D - down        P - bundled in port-channel
        I - stand-alone s - suspended
        H - Hot-standby (LACP only)
        R - Layer3      S - Layer2
        U - in use      f - failed to allocate aggregator

        M - not in use, minimum links not met
        u - unsuitable for bundling
        w - waiting to be aggregated
        d - default port


Number of channel-groups in use: 2
Number of aggregators:           2

Group  Port-channel  Protocol    Ports
------+-------------+-----------+-----------------------------------------------
1      Po1(SU)         LACP      Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3      Po3(SU)          -        Fa0/11(P)   Fa0/12(P)   Fa0/13(P)   Fa0/14(P)
```

In this case, it is necessary to get:

- port-channel name and number. For example, Po1
- list of all the ports in it. For example, `['Fa0/1', 'Fa0/2', 'Fa0/3']`

The difficulty is that ports are in the same line and TextFSM cannot specify the same variable multiple times in line. But it is possible to search multiple times for a match in a line.

The first version of template:

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Record
```

Template has two variables:

- CHANNEL - name and number of aggregated port
- MEMBERS - list of ports that are included in an aggregated port. List – type which is specified for this variable.

The result is:

```
CHANNEL    MEMBERS
---------  ----------
Po1        ['Fa0/1']
Po3        ['Fa0/11']
```

So far, only the first port is in output but we need all ports to hit. In this case after match is found, you should continue processing string with ports. That is, use Continue action and describe the following expression.

The only line in template describes the first port. Add a line that describes the next port.

The next version of template:

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Record
```

The second line describes the same expression, but MEMBERS variable is moved to the next port.

The result is:

```
CHANNEL    MEMBERS
---------  --------------------
Po1        ['Fa0/1', 'Fa0/2']
Po3        ['Fa0/11', 'Fa0/12']
```

Similarly, lines that describe the third and fourth ports should be written to template. But, because the output can have a different number of ports, you have to move Record rule to separate line so that it is not tied to a specific number of ports in string.

For example, if Record is located after the line that describes four ports, for a situation with fewer ports in the line the entry will not be executed.

The resulting template (templates/sh_ether_channelsummary.txt file):

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+.* -> Continue.Record
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){2} +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){3} +${MEMBERS}\( -> Continue
```

The result of processing:

```
CHANNEL    MEMBERS
---------  ----------------------------------------
Po1        ['Fa0/1', 'Fa0/2', 'Fa0/3']
Po3        ['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14']
```

Now all ports are in output.

The template assumes a maximum of four ports in line. If there are more ports, add the corresponding lines to template.

Another version of “sh etherchannel summary” output (output/sh_etherchannel_summary2.txt file):

```
sw1# sh etherchannel summary
Flags:  D - down        P - bundled in port-channel
        I - stand-alone s - suspended
        H - Hot-standby (LACP only)
        R - Layer3      S - Layer2
        U - in use      f - failed to allocate aggregator

        M - not in use, minimum links not met
        u - unsuitable for bundling
        w - waiting to be aggregated
        d - default port


Number of channel-groups in use: 2
Number of aggregators:           2

Group  Port-channel  Protocol    Ports
------+-------------+-----------+-----------------------------------------------
```

(continues on next page)

(continued from previous page)

```
1      Po1(SU)         LACP      Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3      Po3(SU)          -        Fa0/11(P)   Fa0/12(P)   Fa0/13(P)   Fa0/14(P)
                                 Fa0/15(P)   Fa0/16(P)
```

In this output a new version appears - lines containing only ports.

To process this version you should modify template (templates/sh_etherchannel_summary2.txt file):

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+.* -> Continue.Record
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){2} +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){3} +${MEMBERS}\( -> Continue
  ^ +${MEMBERS} -> Continue
  ^ +\S+ +${MEMBERS} -> Continue
  ^ +(\S+ +){2} +${MEMBERS} -> Continue
  ^ +(\S+ +){3} +${MEMBERS} -> Continue
```

The result will be:

```
CHANNEL    MEMBERS
---------  ------------------------------------------------------------
Po1        ['Fa0/1', 'Fa0/2', 'Fa0/3']
Po3        ['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14', 'Fa0/15', 'Fa0/16']
```

This concludes our work with TextFSM templates.

Examples of templates for Cisco and other vendors can be seen in project ntc-ansible.

TextFSM CLI Table

With TextFSM it is possible to process output of commands and get a structured result. However, it is still necessary to manually specify which template will handle show commands each time TextFSM is used.

It would be much more convenient to have some mapping between command and template so that you can write a common script that performs connections to devices, sends commands, chooses template and parse output according to template.

TextFSM has such feature. To use it, you should create a file that describes mapping between commands and templates. In TextFSM it is called `index`.

This file should be in a directory with templates and should have this format:

- first line - column names
- every next line is a pattern match to a command
- mandatory columns with fixed position (mandatory first and last, respectively):
  - first column - names of templates
  - last column - the corresponding command. This column uses a special format to describe that a command may not be fully written
- other columns are optional
  - in example below there are columns Hostname, Vendor. They allow you to refine your device information to determine which template to use. For example, *show version* command may be in Cisco and HP devices. Of course, having only commands are not sufficient to determine which template to use. In this case, you can pass information about the type of equipment used with command and then you can define the correct template.
- all columns except the first column support regular expressions. Regular expressions are not supported inside `[[]]`

Example of index file:

```
Template, Hostname, Vendor, Command
sh_cdp_n_det.template, .*, Cisco, sh[[ow]] cdp ne[[ighbors]] de[[tail]]
sh_clock.template, .*, Cisco, sh[[ow]] clo[[ck]]
sh_ip_int_br.template, .*, Cisco, sh[[ow]] ip int[[erface]] br[[ief]]
sh_ip_route_ospf.template, .*, Cisco, sh[[ow]] ip rou[[te]] o[[spf]]
```

Note how commands are written: `sh[[ow]] ip int[[erface]] br[[ief]]`. Record will be converted to `sh((ow)?)? ip int((erface)?)? br((ief)?)?`. This means that TextFSM will be able to determine which template to use even if command is not fully written. For example, such command variants will work:

- sh ip int br
- show ip inter bri

How to use CLI table

Let's see how to use `clitable` class and index file. `templates` directory contains such templates and index file:

```
sh_cdp_n_det.template
sh_clock.template
sh_ip_int_br.template
```

(continues on next page)

(continued from previous page)

```
sh_ip_route_ospf.template
index
```

First we try to work with CLI Table in ipython to see what features this class has and then we look at the final script.

First, we import `clitable` class:

```
In [1]: from textfsm import clitable
```

Warning: There are different ways to import `clitable` depending on textfsm version:

- `import clitable` for version <= 0.4.1
- `from textfsm import clitable` for version >= 1.1.0

See textfsm version: `pip show textfsm`.

We will check `clitable` on the last example from previous section - "show ip route ospf" command. Read the output that is stored in output/sh_ip_route_ospf.txt file to string:

```
In [2]: with open('output/sh_ip_route_ospf.txt') as f:
   ...:     output_sh_ip_route_ospf = f.read()
   ...:
```

First, you should initialize a class by giving it name of file in which mapping between templates and commands is stored, and specify name of directory in which templates are stored:

```
In [3]: cli_table = clitable.CliTable('index', 'templates')
```

Specify which command should be passed and specify additional attributes that will help to identify template. To do this, you should create a dictionary in which keys are names of columns that are defined in index file. In this case, it is not necessary to specify vendor name, since "sh ip route ospf" command corresponds to only one template.

```
In [4]: attributes = {'Command': 'show ip route ospf' , 'Vendor': 'Cisco'}
```

Command output and dictionary with parameters should be passed to ParseCmd method:

```
In [5]: cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
```

As a result we have processed output of "sh ip route ospf" command in cli_table object.

cli_table methods (to see all methods, call dir(cli_table)):

```
In [6]: cli_table.
cli_table.AddColumn        cli_table.NewRow          cli_table.index
↪cli_table.size
cli_table.AddKeys          cli_table.ParseCmd        cli_table.index_file
↪cli_table.sort
cli_table.Append           cli_table.ReadIndex       cli_table.next
↪cli_table.superkey
cli_table.CsvToTable       cli_table.Remove          cli_table.raw
↪cli_table.synchronised
cli_table.FormattedTable   cli_table.Reset           cli_table.row
↪cli_table.table
cli_table.INDEX            cli_table.RowWith         cli_table.row_class
↪cli_table.template_dir
cli_table.KeyValue         cli_table.extend          cli_table.row_index
cli_table.LabelValueTable  cli_table.header          cli_table.separator
```

For example, if you call `print(cli_table)` you get this:

```
In [7]: print(cli_table)
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']
```

FormattedTable method produces a table output:

```
In [8]: print(cli_table.FormattedTable())
 Network    Mask  Distance  Metric  NextHop
=================================================================
 10.0.24.0  /24   110       20      10.0.12.2
 10.0.34.0  /24   110       20      10.0.13.3
 10.2.2.2   /32   110       11      10.0.12.2
 10.3.3.3   /32   110       11      10.0.13.3
 10.4.4.4   /32   110       21      10.0.13.3, 10.0.12.2, 10.0.14.4
 10.5.35.0  /24   110       20      10.0.13.3
```

This can be useful for displaying information.

To get a structured output from cli_table object, such as a list of lists, you have to refer to object in this way:

```
In [9]: data_rows = [list(row) for row in cli_table]

In [11]: data_rows
Out[11]:
[['10.0.24.0', '/24', '110', '20', ['10.0.12.2']],
 ['10.0.34.0', '/24', '110', '20', ['10.0.13.3']],
 ['10.2.2.2', '/32', '110', '11', ['10.0.12.2']],
 ['10.3.3.3', '/32', '110', '11', ['10.0.13.3']],
 ['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']],
 ['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]]
```

You can get column names separately:

```
In [12]: header = list(cli_table.header)

In [14]: header
Out[14]: ['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
```

The output is now similar to that of the previous section.

Assemble everything into one script (textfsm_clitable.py file):

```
import clitable

output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()

cli_table = clitable.CliTable('index', 'templates')

attributes = {'Command': 'show ip route ospf' , 'Vendor': 'Cisco'}

cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
print('CLI Table output:\n', cli_table)

print('Formatted Table:\n', cli_table.FormattedTable())

data_rows = [list(row) for row in cli_table]
header = list(cli_table.header)

print(header)
for row in data_rows:
    print(row)
```

In exercises to this section there will be a task to combine described procedure into a function and task to get a list of dictionaries.

The output will be:

```
$ python textfsm_clitable.py
CLI Table output:
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']

Formatted Table:
 Network    Mask  Distance  Metric  NextHop
==========================================================================
 10.0.24.0  /24   110       20      10.0.12.2
 10.0.34.0  /24   110       20      10.0.13.3
 10.2.2.2   /32   110       11      10.0.12.2
 10.3.3.3   /32   110       11      10.0.13.3
 10.4.4.4   /32   110       21      10.0.13.3, 10.0.12.2, 10.0.14.4
 10.5.35.0  /24   110       20      10.0.13.3

['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
['10.0.24.0', '/24', '110', '20', ['10.0.12.2']]
['10.0.34.0', '/24', '110', '20', ['10.0.13.3']]
['10.2.2.2', '/32', '110', '11', ['10.0.12.2']]
['10.3.3.3', '/32', '110', '11', ['10.0.13.3']]
['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']]
['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]
```

Now with TextFSM it is possible not only to get a structured output, but also to automatically determine which template to use by command and optional arguments.

Further reading

Documentation:

- TextFSM

Articles:

- Programmatic Access to CLI Devices with TextFSM. Jason Edelman (26.02.2015) - TextFSM basics and development ideas that formed the basis of ntc-ansible module
- Parse CLI outputs with TextFSM. Henry Ölsner (24.08.2015) - an example of using TextFSM to parse a large file with *sh inventory* output. Explains TextFSM syntax in more detail

- Creating Templates for TextFSM and ntc_show_command. Jason Edelman (27.08.2015) - TextFSM syntax is discussed in more detail and examples of ntc-ansible module usage are shown

  (note that syntax of module has already changed slightly)

- TextFSM and Structured Data. Kirk Byers (22.10.2015) - an introductory article on TextFSM. This does not describe syntax but gives an overview of what TextFSM is and an example of its use

Projects that use TextFSM:

- Module ntc-ansible

TextFSM templates (from ntc-ansible module):

- ntc-templates

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 21.1

Create parse_command_output function. Function parameters:

- template - name of the file containing the TextFSM template. For example templates/sh_ip_int_br.template
- command_output - output the corresponding show command (string)

The function should return a list:

- the first element is a list with column names
- the rest of the items are lists, which contain the results of processing the output of the show command

Check the operation of the function on the output of the sh ip int br command from the equipment and on the templates/sh_ip_int_br.template template.

```
from netmiko import ConnectHandler


# this is how a function call should look
if __name__ == "__main__":
    r1_params = {
        "device_type": "cisco_ios",
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "secret": "cisco",
    }
    with ConnectHandler(**r1_params) as r1:
        r1.enable()
        output = r1.send_command("sh ip int br")
```

(continues on next page)

(continued from previous page)

```
    result = parse_command_output("templates/sh_ip_int_br.template", output)
    print(result)
```

Task 21.1a

Create parse_output_to_dict function.

Function parameters:

- template is the name of the file containing the TextFSM template. For example templates/sh_ip_int_br.template
- command_output - output of the corresponding show command (string)

The function should return a list of dictionaries:

- keys - names of variables in the TextFSM template
- values - parts of the output that correspond to variables

Check the operation of the function on the output of the command output/sh_ip_int_br.txt and the template templates/sh_ip_int_br.template.

Task 21.2

Create a TextFSM template to parse the output of the sh ip dhcp snooping binding command and write it to templates/sh_ip_dhcp_snooping.template

The command output is located in the file output/sh_ip_dhcp_snooping.txt.

The template should process and return the values of such columns:

- mac - 00:04:A3:3E:5B:69
- ip - 10.1.10.6
- vlan - 10
- intf - FastEthernet0/10

Check the work of the template using the parse_command_output function from task 21.1.

Task 21.3

Create function parse_command_dynamic.

Function parameters:

- command_output - command output (string)

- attributes_dict - an attribute dict containing the following key-value pairs:
  - 'Command': command
  - 'Vendor': vendor
- index_file is the name of the file where the correspondence between commands and templates is stored. The default is "index"
- templ_path - directory where templates are stored. The default is "templates"

The function should return a list of dicts with the results of parsing the command output (as in task 21.1a):

- keys - names of variables in the TextFSM template
- values - parts of the output that correspond to variables

Check the function on the output of the sh ip int br command.

Task 21.4

Create function send_and_parse_show_command.

Function parameters:

- device_dict - a dict with connectin parameters for one device
- command - the command to be executed
- templates_path - path to the directory with TextFSM templates
- index - file index name, default value "index"

The function should connect to one device, send a show command using netmiko, and then parse the command output using TextFSM.

The function should return a list of dictionaries with the results of parsing the command output (as in task 21.1a):

- keys - names of variables in the TextFSM template
- values - parts of the output that correspond to variables

Check the operation of the function using the output of the sh ip int br command and devices from devices.yaml.

Task 21.5

Create function send_and_parse_command_parallel.

The send_and_parse_command_parallel function must run the send_and_parse_show_command function from task 21.4 in concurrent threads.

Send_and_parse_command_parallel function parameters:

- devices - a list of dicts with connection parameters for devices
- command - command
- templates_path - path to the directory with TextFSM templates
- limit - maximum number of concurrent threads (default 3)

The function should return a dictionary:

- keys - the IP address of the device from which the output was received
- values - a list of dicts (the output returned by the send_and_parse_show_command function)

Dictionary example:

```
{'192.168.100.1': [{'address': '192.168.100.1',
                    'intf': 'Ethernet0/0',
                    'protocol': 'up',
                    'status': 'up'},
                   {'address': '192.168.200.1',
                    'intf': 'Ethernet0/1',
                    'protocol': 'up',
                    'status': 'up'}],
 '192.168.100.2': [{'address': '192.168.100.2',
                    'intf': 'Ethernet0/0',
                    'protocol': 'up',
                    'status': 'up'},
                   {'address': '10.100.23.2',
                    'intf': 'Ethernet0/1',
                    'protocol': 'up',
                    'status': 'up'}]}
```

Check the operation of the function using the output of the sh ip int br command and devices from devices.yaml.

8

VI. Basics of object-oriented programming

Object-oriented programming (OOP) - a programming methodology in which a program consists of objects that interact with each other. Objects are created on basis of class defined in code and typically combine data and actions that can be performed with data into a single whole.

It is possible to write code without using OOP, but at a minimum learning of OOP basics will help to better understand what an object, class, method, variable are. These are things that are used in Python all the time. In addition, knowledge of OOP will be useful in reading someone else's code. For example, it will be easier to understand netmiko code.

22. OOP basics

OOP basics

- Class - an element of a program that describes some data type. Class describes a template for creating objects, typically specifies variables of object and actions that can be performed on object.
- Instance - an object that is a representative of a class.
- Method - a function that is defined within a class and describes an action that class supports
- Instance variable (sometimes instance attribute) - data that refer to an object
- Class variable - data that refer to class and shared by all class instances
- Instance attribute - variables and methods that refer to objects (instances) created on the basis of a class. Every object has its own copy of attributes.

A real-life OOP example:

- Building project - it is a class
- Particular house which was built according to project - instance
- Features such as color of house, number of windows - instance variables (of this particular house)
- House can be sold, repainted, repaired - methods

For example, when working with netmiko, the first thing to do was create connection:

```
from netmiko import ConnectHandler

device = {
    "device_type": "cisco_ios",
    "host": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}

ssh = ConnectHandler(**device)
```

The ssh variable is an object that represents the real connection to equipment. Thanks to the type function, you can find out by an instance what class is the ssh object:

```
In [3]: type(ssh)
Out[3]: netmiko.cisco.cisco_ios.CiscoIosSSH
```

ssh has its own methods and variables that depend on the state the current object. For example, the instance variable `ssh.host` is available for every instance of the class `netmiko.cisco.cisco_ios.CiscoIosSSH` and returns IP address or hostname, whichever is specified in the device dictionary:

```
In [4]: ssh.host
Out[4]: '192.168.100.1'
```

Method `send_command` executes a command on the network device:

```
In [5]: ssh.send_command("sh clock")
Out[5]: '*10:08:50.654 UTC Tue Feb 2 2021'
```

The `enable` method goes into enable mode and the `ssh` object saves state: before and after the transition, a different prompt is visible:

```
In [6]: ssh.find_prompt()
Out[6]: 'R1>'

In [7]: ssh.enable()
Out[7]: 'enable\r\nPassword: \r\nR1#'

In [8]: ssh.find_prompt()
Out[8]: 'R1#'
```

This example illustrates important aspects of OOP: data integration, data handling and state preservation.

Until now, when writing code, data and actions have been separated. Most often, actions are described as functions, and data is passed as arguments to these functions. When a class is created, data and actions are combined. Of course, these data and actions are linked. That is, those actions that are characteristic of an object of this type, and not some arbitrary actions, become class methods.

For example, in an class instance `str`, all methods refer to working with this string:

```
In [10]: s = 'string'

In [11]: s.upper()
Out[11]: 'STRING'

In [12]: s.center(20, '=')
Out[12]: '=======string======='
```

Note: Class does not have to store a state - string is immutable data type and all methods return new strings and do not change the original string.

Above, the following syntax is used when referring to instance attributes (variables and methods): `objectname.attribute`. This entry `s.lower()` means: call `lower` method on `s` object. Calling methods and variables is the same, but to call a method you have to add parentheses and pass all necessary arguments.

Everything described has been used repeatedly in the book but now we will deal with formal terminology.

Class creation

Note: Note that the basis is explained here given that the reader has no experience with OOP. Some examples are not very correct from Python's ideology point of view, but they help to better understand how it works. At the end, an explanation is given of how this should be done in proper way.

Keyword `class` is used in python to create classes. The easiest class you can create in Python:

```
In [1]: class Switch:
   ...:     pass
   ...:
```

Note: Class names: usually class names in Python are written in CamelCase format.

To create a class instance, call class:

```
In [2]: sw1 = Switch()

In [3]: print(sw1)
<__main__.Switch object at 0xb44963ac>
```

Using dot notation, it is possible to get values of instance variables, create new variables and assign a new value to existing ones:

```
In [5]: sw1.hostname = 'sw1'

In [6]: sw1.model = 'Cisco 3850'
```

In another instance of Switch class, the variables may be different:

```
In [7]: sw2 = Switch()
```

(continues on next page)

(continued from previous page)

```
In [8]: sw2.hostname = 'sw2'

In [9]: sw2.model = 'Cisco 3750'
```

You can see value of instance variables using the same dot notation:

```
In [10]: sw1.model
Out[10]: 'Cisco 3850'

In [11]: sw2.model
Out[11]: 'Cisco 3750'
```

Method creation

Before we start dealing with class methods, let's see an example of a function that waits as an argument an instance variable of Switch class and displays information about it using instance variables `hostname` and `model`:

```
In [1]: def info(sw_obj):
   ...:     print(f'Hostname: {sw_obj.hostname}\nModel: {sw_obj.model}')
   ...:

In [2]: sw1 = Switch()

In [3]: sw1.hostname = 'sw1'

In [4]: sw1.model = 'Cisco 3850'

In [5]: info(sw1)
Hostname: sw1
Model: Cisco 3850
```

In `info` function, `sw_obj` awaits an instance of `Switch` class. Most likely, there is nothing new about this example, because in the same way earlier we wrote functions that wait for a string as an argument and then call some methods in this string.

This example will help you to understand `info` method that we will add to Switch class.

To add a method you have to create a function within class:

```
In [15]: class Switch:
    ...:     def info(self):
```

(continues on next page)

(continued from previous page)

```
    ...:         print(f'Hostname: {self.hostname}\nModel: {self.model}')
    ...:
```

If you look closely, `info` method looks exactly like `info` function, only instead of `sw_obj` name the `self` is used. Why there is a strange `self` name here will be explained later and in the meantime we will see how to call `info` method:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'

In [19]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

In example above, first an instance of Switch class is created, then `hostname` and `model` variables are added to instance and then `info` method is called. Method `info` outputs information about switch using values that are stored in instance variables.

Method call is different from the function call: we do not pass a link to an instance of `Switch` class. We don't need that because we call method from instance itself. Another unclear thing - why we wrote `self` then?

Python transforms such a call:

```
In [39]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

To this one:

```
In [38]: Switch.info(sw1)
Hostname: sw1
Model: Cisco 3850
```

In the second case, `self` parameter already makes more sense, it actually accepts the reference to instance and displays information on this basis.

From objects usage point of view, it is more convenient to call methods using the first syntax version, so it is almost always used.

Note: When a class instance method is called the instance reference is passed by the first argu-

ment. In this case, instance is passed implicitly but parameter must be stated explicitly.

This conversion is not a feature of user classes and works for embedded data types in the same way. For example, standard way to call append method in the list is:

```
In [4]: a = [1, 2, 3]

In [5]: a.append(5)

In [6]: a
Out[6]: [1, 2, 3, 5]
```

The same can be done using the second option, calling through a class:

```
In [7]: a = [1, 2, 3]

In [8]: list.append(a, 5)

In [9]: a
Out[9]: [1, 2, 3, 5]
```

Parameter self

Parameter `self` was specified before in method definition, as well as when using instance variables in method. Parameter `self` is a reference to a particular instance of class. Parameter `self` is not a special name but an arrangement. Instead of `self` you can use a different name but you shouldn't do that.

Example of using a different name instead of `self`:

```
In [15]: class Switch:
    ...:     def info(sw_object):
    ...:         print(f'Hostname: {sw_object.hostname}\nModel: {sw_object.model}
↪')
    ...:
```

It will work the same way:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'
```

(continues on next page)

(continued from previous page)

```
In [19]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

Warning: Although technically you can use another name but always use `self`.

In all "usual" methods of class the first parameter will always be `self`. Furthermore, creating an instance variable within a class is also done via `self`.

An example of Switch class with new generate_interfaces method: generate_interfaces method must generate a list with interfaces based on specified type and quantity and create variable in an instance of class. First, an option of creating a usual variable within method:

```
In [5]: class Switch:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = [f'{intf_type}{number}' for number in range(1,␣
↪number_of_intf+1)]
   ...:
```

In this case, class instances will not have interfaces variable:

```
In [6]: sw1 = Switch()

In [7]: sw1.generate_interfaces('Fa', 10)

In [8]: sw1.interfaces
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-e6b457e4e23e> in <module>()
----> 1 sw1.interfaces

AttributeError: 'Switch' object has no attribute 'interfaces'
```

This variable does not exist because it exists only within method and visibility area of method is the same as function. Even other methods of the same class do not see variables in other methods.

For list with interfaces to be available as a variable in instances, you have to assign value in self.interfaces:

```
In [9]: class Switch:
   ...:     def info(self):
   ...:         print(f'Hostname: {self.hostname}\nModel: {self.model}')
```

(continues on next page)

(continued from previous page)

```
   ...:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = [f'{intf_type}{number}' for number in range(1,␣
↪number_of_intf + 1)]
   ...:         self.interfaces = interfaces
   ...:
```

Now, after generate_interfaces method is called interfaces variable is created in instance:

```
In [10]: sw1 = Switch()

In [11]: sw1.generate_interfaces('Fa', 10)

In [12]: sw1.interfaces
Out[12]: ['Fa1', 'Fa2', 'Fa3', 'Fa4', 'Fa5', 'Fa6', 'Fa7', 'Fa8', 'Fa9', 'Fa10']
```

Method __init__

For `info` method to work correctly the instance should have `hostname` and `model` variables. If these variables are not available, an error will occur:

```
In [15]: class Switch:
    ...:     def info(self):
    ...:         print(f'Hostname: {self.hostname}\nModel: {self.model}')
    ...:

In [59]: sw2 = Switch()

In [60]: sw2.info()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-60-5a006dd8aae1> in <module>()
----> 1 sw2.info()

<ipython-input-57-30b05739380d> in info(self)
      1 class Switch:
      2     def info(self):
----> 3         print(f'Hostname: {self.hostname}\nModel: {self.model}')

AttributeError: 'Switch' object has no attribute 'hostname'
```

Almost always, when an object is created it has some initial data. For example, to create a connection to device with netmiko you have to pass connection parameters.

In Python these initial object data are specified in __init__. Method __init__ is executed after Python has created a new instance and __init__ method is passed arguments with which instance was created:

```
In [32]: class Switch:
    ...:     def __init__(self, hostname, model):
    ...:         self.hostname = hostname
    ...:         self.model = model
    ...:
    ...:     def info(self):
    ...:         print(f'Hostname: {self.hostname}\nModel: {self.model}')
    ...:
```

Note that each instance created from this class will have variables: `self.model` and `self.hostname`.

Now, when creating an instance of Switch class you have to specify `hostname` and `model`:

```
In [33]: sw1 = Switch('sw1', 'Cisco 3850')
```

Accordingly, `info` method works without error:

```
In [36]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

Note: __init__ method is sometimes called a class constructor, although technically in Python __new__ method is executed first and then __init__. In most cases there is no necessety to create __new__ method.

An important feature of __init__ method is that it should not return anything. Python will generate an exception if it tries to do this.

Class example

The class that describes the network:

```
class Network:
    def __init__(self, network):
        self.network = network
        self.hosts = tuple(str(ip) for ip in ipaddress.ip_network(network).
↪hosts())
        self.allocated = []
```

(continues on next page)

(continued from previous page)

```
    def allocate(self, ip):
        if ip in self.hosts:
            if ip not in self.allocated:
                self.allocated.append(ip)
            else:
                raise ValueError(f"IP address {ip} is already in the allocated␣
↪list")
        else:
            raise ValueError(f"IP address {ip} does not belong to {self.network}")
```

Using the class:

```
In [2]: net1 = Network("10.1.1.0/29")

In [3]: net1.allocate("10.1.1.1")

In [4]: net1.allocate("10.1.1.2")

In [5]: net1.allocated
Out[5]: ['10.1.1.1', '10.1.1.2']

In [6]: net1.allocate("10.1.1.100")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-9a4157e02c78> in <module>
----> 1 net1.allocate("10.1.1.100")

<ipython-input-1-c5255d37a7fd> in allocate(self, ip)
     12                 raise ValueError(f"IP address {ip} is already in the␣
↪allocated list")
     13         else:
---> 14             raise ValueError(f"IP address {ip} does not belong to {self.
↪network}")
     15

ValueError: IP address 10.1.1.100 does not belong to 10.1.1.0/29

In [7]: net1.hosts
Out[7]: ('10.1.1.1', '10.1.1.2', '10.1.1.3', '10.1.1.4', '10.1.1.5', '10.1.1.6')
```

Class namespace

Each method in class has its own local visibility area. This means that one class method does not see variables of another class method. For variables to be available, you have to assign their instance through `self.name`. Basically, method is a function tied to an object. Therefore, all nuances that concern function apply to methods.

Variable instances are available in another method because instance itself is passed as a first argument to each method. In example below in `__init__` method, `hostname` and `model` variables are assigned to an instance and then used in `info` due to instance being passed as a first argument:

```
class Switch:
    def __init__(self, hostname, model):
        self.hostname = hostname
        self.model = model

    def info(self):
        print(f'Hostname: {self.hostname}\nModel: {self.model}')
```

Class variables

Besides instance variables, there are also class variables. They are are created by specifying variables inside the class itself, not a method:

```
class Network:
    all_allocated_ip = []

    def __init__(self, network):
        self.network = network
        self.hosts = tuple(
            str(ip) for ip in ipaddress.ip_network(network).hosts()
        )
        self.allocated = []


    def allocate(self, ip):
        if ip in self.hosts:
            if ip not in self.allocated:
                self.allocated.append(ip)
                type(self).all_allocated_ip.append(ip)
            else:
                raise ValueError(f"IP address {ip} is already in the allocated␣
↪list")
```

(continues on next page)

(continued from previous page)

```
        else:
            raise ValueError(f"IP address {ip} does not belong to {self.network}")
```

Class variables can be accessed in different ways:

- `self.all_allocated_ip`
- `Network.all_allocated_ip`
- `type(self).all_allocated_ip`

The `self.all_allocated_ip` option allows you to access the value of class variable or add an element if the class variable is a mutable data type. The disadvantage of this option is that if in the method you write `self.all_allocated_ip = ...`, instead of changing the class variable, an instance variable will be created.

The option `Network.all_allocated_ip` will work correctly, but a small drawback this option is that the class name is written manually. You can use the third option `type(self).all_allocated_ip` instead, since `type(self)` returns a class.

Now the class has a variable all_allocated_ip which is written all IP addresses that are allocated on the networks:

```
In [3]: net1 = Network("10.1.1.0/29")

In [4]: net1.allocate("10.1.1.1")
   ...: net1.allocate("10.1.1.2")
   ...: net1.allocate("10.1.1.3")
   ...:

In [5]: net1.allocated
Out[5]: ['10.1.1.1', '10.1.1.2', '10.1.1.3']

In [6]: net2 = Network("10.2.2.0/29")

In [7]: net2.allocate("10.2.2.1")
   ...: net2.allocate("10.2.2.2")
   ...:

In [9]: net2.allocated
Out[9]: ['10.2.2.1', '10.2.2.2']

In [10]: Network.all_allocated_ip
Out[10]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']
```

The variable is accessible not only through the class, but also through the instances:

```
In [40]: Network.all_allocated_ip
Out[40]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']

In [41]: net1.all_allocated_ip
Out[41]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']

In [42]: net2.all_allocated_ip
Out[42]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']
```

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 22.1

Create a Topology class that represents the topology of the network.

When creating an instance of a class, a dictionary that describes the topology is passed as an argument. The dictionary may contain "duplicate" connections. "Duplicate" connections are a situation when there are two connections in the dictionary:

```
("R1", "Eth0/0"): ("SW1", "Eth0/1")
("SW1", "Eth0/1"): ("R1", "Eth0/0")
```

The task is to leave only one of these links in the final dictionary, no matter which one.

In each instance, a topology instance variable must be created, which contains the topology dictionary, but already without "duplicates". The topology instance variable should contain a dict without "duplicates" immediately after instance creation.

An example of creating an instance of a class:

```
In [2]: top = Topology(topology_example)
```

After that, the topology variable should be available:

```
In [3]: top.topology
Out[3]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

```
topology_example = {('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
                    ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
```

(continues on next page)

(continued from previous page)

```
                    ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
                    ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
                    ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
                    ('R3', 'Eth0/2'): ('R5', 'Eth0/0'),
                    ('SW1', 'Eth0/1'): ('R1', 'Eth0/0'),
                    ('SW1', 'Eth0/2'): ('R2', 'Eth0/0'),
                    ('SW1', 'Eth0/3'): ('R3', 'Eth0/0')}
```

Task 22.1a

Copy the Topology class from task 22.1 and modify it.

Transfer the functionality of removing "duplicates" to the _normalize method. In this case, the __init__ method should look like this:

```
class Topology:
    def __init__(self, topology_dict):
        self.topology = self._normalize(topology_dict)
```

Task 22.1b

Copy the Topology class from either task 22.1a or 22.1 and modify it.

Add a delete_link method that deletes the specified connection. The method should also remove the "reverse" connection, if any (an example is given below).

If there is no such link, the message "There is no such link" should be printed.

Topology creation:

```
In [7]: t = Topology(topology_example)

In [8]: t.topology
Out[8]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Removing a link:

```
In [9]: t.delete_link(('R3', 'Eth0/1'), ('R4', 'Eth0/0'))

In [10]: t.topology
Out[10]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Deleting the "reverse" link: the dictionary contains an entry `('R3', 'Eth0/2'): ('R5', 'Eth0/0')`, but calling the delete_link method specifying the key and value in reverse order `('R5', 'Eth0/0'): ('R3', 'Eth0/2')` should delete the connection:

```
In [11]: t.delete_link(('R5', 'Eth0/0'), ('R3', 'Eth0/2'))

In [12]: t.topology
Out[12]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3')}
```

If there is no such connection, the following message is printed:

```
In [13]: t.delete_link(('R5', 'Eth0/0'), ('R3', 'Eth0/2'))
There is no such link
```

Task 22.1c

Copy the Topology class from task 22.1b and modify it.

Add a delete_node method that deletes all connections to the specified device. If there is no such device, the message "There is no such device" is printed.

Topology creation:

```
In [1]: t = Topology(topology_example)

In [2]: t.topology
Out[2]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
```

(continues on next page)

(continued from previous page)

```
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Removing a device:

```
In [3]: t.delete_node('SW1')

In [4]: t.topology
Out[4]:
{('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

If there is no such device, the following message is printed:

```
In [5]: t.delete_node('SW1')
There is no such device
```

Task 22.1d

Copy the Topology class from task 22.1c and modify it.

Add the add_link method, which adds the specified link if it is not already in the topology. If the connection exists, print the message “Such a connection already exists”, If one of the sides is in the topology, display the message “A link to one of the ports exists”.

Topology creation:

```
In [7]: t = Topology(topology_example)

In [8]: t.topology
Out[8]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [9]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/0'))

In [10]: t.topology
```

(continues on next page)

(continued from previous page)

```
Out[10]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [11]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/0'))
Such a connection already exists

In [12]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/5'))
A link to one of the ports exists
```

Task 22.2

Create a CiscoTelnet class that connects via Telnet to Cisco equipment.

When instantiating the class, a Telnet connection should be created, as well as the transition to enable mode. The class must use the telnetlib module to connect via Telnet.

The CiscoTelnet class, in addition to __init__, must have at least two methods:

- _write_line - takes a string as an argument and sends the string converted to bytes to the hardware and adds a line end character at the end. The _write_line method must be used inside the class.
- send_show_command - takes the show command as an argument and returns the output received from the device

__init__ method parameters:

- ip - IP address
- username - username
- password - password
- secret - enable password

An example of creating an instance of a class:

```
In [2]: from task_22_2 import CiscoTelnet

In [3]: r1_params = {
   ...:     'ip': '192.168.100.1',
```

(continues on next page)

(continued from previous page)

```
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}
   ...:

In [4]: r1 = CiscoTelnet(**r1_params)

In [5]: r1.send_show_command("sh ip int br")
Out[5]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method␣
↪Status                Protocol\r\nEthernet0/0               192.168.100.1  ␣
↪YES NVRAM  up                    up      \r\nEthernet0/1               192.168.
↪200.1   YES NVRAM  up                    up      \r\nEthernet0/2                ␣
↪unassigned      YES manual up                    up      \r\nEthernet0/3       ␣
↪        192.168.130.1   YES NVRAM  up                    up      \r\nR1#'
```

Note: The _write_line method is needed in order to be able to shorten a line: `self.telnet.write(line.encode("ascii") + b"\n")`

to this: `self._write_line(line)`

He shouldn't do anything else.

Task 22.2a

Copy the CiscoTelnet class from job 22.2 and modify the send_show_command method by adding three parameters:

- parse - controls what will be returned: normal command output or a list of dicts received after parsing command output using TextFSM. If parse=True, a list of dicts should be returned, and parse=False normal output. The default is True.
- templates - path to the directory with templates. The default is "templates"
- index is the name of the file where the correspondence between commands and templates is stored. The default is "index"

An example of creating an instance of a class:

```
In [1]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}
```

(continues on next page)

(continued from previous page)

```
In [2]: from task_22_2a import CiscoTelnet

In [3]: r1 = CiscoTelnet(**r1_params)
```

Using the send_show_command method:

```
In [4]: r1.send_show_command("sh ip int br", parse=True)
Out[4]:
[{'intf': 'Ethernet0/0',
  'address': '192.168.100.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/1',
  'address': '192.168.200.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/2',
  'address': '192.168.130.1',
  'status': 'up',
  'protocol': 'up'}]

In [5]: r1.send_show_command("sh ip int br", parse=False)
Out[5]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method␣
↪Status
Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up
up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up...'
```

Task 22.2b

Copy the CiscoTelnet class from task 22.2a and add the send_config_commands method.

The send_config_commands method must be able to send one configuration mode command and a list of commands. The method should return output similar to the send_config_set method of netmiko (example output below).

An example of creating an instance of a class:

```
In [1]: from task_22_2b import CiscoTelnet

In [2]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
```

(continues on next page)

(continued from previous page)

```
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [3]: r1 = CiscoTelnet(**r1_params)
```

Using the send_config_commands method:

```
In [5]: r1.send_config_commands('logging 10.1.1.1')
Out[5]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#logging 10.1.1.1\r\nR1(config)#end\r\nR1#'

In [6]: r1.send_config_commands(['interface loop55', 'ip address 5.5.5.5 255.255.
↪255.255'])
Out[6]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#interface loop55\r\nR1(config-if)#ip address 5.5.5.5 255.255.255.
↪255\r\nR1(config-if)#end\r\nR1#'
```

Task 22.2c

Copy the CiscoTelnet class from task 22.2b and modify the send_config_commands method to check for errors.

The send_config_commands method must have an additional strict parameter:

- strict=True means that when an error is encountered, a ValueError must be raised (default)
- strict=False means that when an error is found, you only need to print the error message to the stdout

The method should return output similar to the send_config_set method of netmiko (example output below). The text of the exception and error in the example below.

An example of creating an instance of a class:

```
In [1]: from task_22_2c import CiscoTelnet

In [2]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [3]: r1 = CiscoTelnet(**r1_params)

In [4]: commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
```

(continues on next page)

(continued from previous page)

```
In [5]: correct_commands = ['logging buffered 20010', 'ip http server']
In [6]: commands = commands_with_errors+correct_commands
```

Using the send_config_commands method:

```
In [7]: print(r1.send_config_commands(commands, strict=False))
When executing the command "logging 0255.255.1" on device 192.168.100.1, an error␣
↪occurred -> Invalid input detected at '^' marker.
When executing the command "logging" on device 192.168.100.1, an error occurred ->
↪ Incomplete command.
When executing the command "a" on device 192.168.100.1, an error occurred ->␣
↪Ambiguous command:  "a"
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#logging 0255.255.1
                   ^
% Invalid input detected at '^' marker.

R1(config)#logging
% Incomplete command.

R1(config)#a
% Ambiguous command:  "a"
R1(config)#logging buffered 20010
R1(config)#ip http server
R1(config)#end
R1#

In [8]: print(r1.send_config_commands(commands, strict=True))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-0abc1ed8602e> in <module>
----> 1 print(r1.send_config_commands(commands, strict=True))

...

ValueError: When executing the command "logging 0255.255.1" on device 192.168.100.
↪1, an error occurred -> Invalid input detected at '^' marker.
```

23. Special methods

Special methods in Python - methods that are responsible for “standard” possibilities of objects and are called automatically when these possibilities are used. For example, expression `a + b` where a and b are numbers that is converted to such a call `a.__add__(b)`. That is, special method `__add__` is responsible for the addition operation. All special methods start and end with double underscore, therefore in English they are often called dunder methods, shortened from “double underscore”.

Note: Special methods are often called magic methods.

Special methods are responsible for such features as working in context managers, creating iterators and iterable objects, addition operations, multiplication and others. By adding special methods to objects that are created by user, we make them look like built in Python objects.

Underscore in names

In Python, underscore at the beginning or at the end of a name indicates special names. Most often it’s just an arrangement, but sometimes it actually affects object behavior.

One underscore before name

One underscore before method name indicates that method is an internal feature of implementation and it should not be used directly.

For example, CiscoSSH class uses paramiko to connect to equipment:

```
import time
import paramiko


class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        self.client = paramiko.SSHClient()
        self.client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        self.client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)
```

(continues on next page)

(continued from previous page)

```
        self.ssh = self.client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

After creating an instance of the class, not only send_show_command method is available but also client and ssh attributes (3rd line is tab tips in ipython):

```
In [2]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [3]: r1.
            client
            send_show_command()
            ssh
```

If you want to specify that client and ssh are internal attributes that are needed for class operation but are not intended for user, you need to underscore name below:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        self._client = paramiko.SSHClient()
        self._client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        self._client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = self._client.invoke_shell()
        self._ssh.send('enable\n')
        self._ssh.send(enable + '\n')
        if disable_paging:
```

(continues on next page)

(continued from previous page)

```
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(1000)

    def send_show_command(self, command):
        self._ssh.send(command + '\n')
        time.sleep(2)
        result = self._ssh.recv(5000).decode('ascii')
        return result
```

Note: Often such methods and attributes are called private but this does not mean that methods and variables are not available to user.

Two underscores before name

Two underscores before method name are not used simply as an agreement. Such names are transformed into format "name of class + name of method". This allows the creation of unique methods and attributes of classes.

This conversion is only performed if less than two underscores endings or no underscores.

```
In [14]: class Switch(object):
    ...:     __quantity = 0
    ...:
    ...:     def __configure(self):
    ...:         pass
    ...:

In [15]: dir(Switch)
Out[15]:
['_Switch__configure', '_Switch__quantity', ...]
```

Although methods were created without _Switch, it was added.

If you create a subclass then __configure method will not rewrite parent class method Switch:

```
In [16]: class CiscoSwitch(Switch):
    ...:     __quantity = 0
    ...:     def __configure(self):
    ...:         pass
    ...:
```

(continues on next page)

(continued from previous page)

```
In [17]: dir(CiscoSwitch)
Out[17]:
['_CiscoSwitch__configure', '_CiscoSwitch__quantity', '_Switch__configure', '_
↪Switch__quantity', ...]
```

Two underscores before and after name

Thus, special variables and methods are denoted.

For example, Python module has such special variables:

- __name__ - this variable is equal to __main__ when the script runs directly and is equal to module name when imported
- __file__ - this variable is equal to name of the script that was run directly and equals to complete path to module when it is imported

Variable __name__ is most commonly used to indicate that a certain part of code must be executed only when module is called directly:

```
def multiply(a, b):

    return a * b

if __name__ == '__main__':
    print(multiply(3, 5))
```

And __file__ variable can be useful in determining the current path to script file:

```
import os

print('__file__', __file__)
print(os.path.abspath(__file__))
```

The output will be:

```
__file__ example2.py
/home/vagrant/repos/tests/example2.py
```

Python also denotes special methods. These methods are called when using Python functions and operators and allow to implement a certain functionality.

As a rule, such methods need not be called directly. But for example, when creating your own class it may be necessary to describe such method in order to object can support some operations in Python.

For example, in order to get length of an object it must support __len__ method.

Methods __str__, __repr__

Special methods __str__ and __repr__ are responsible for string representation of the object. They are used in different places.

Consider example of IPAddress class that is responsible for representing IPv4 address:

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip
```

After creating class instances, they have a default string view that looks like this (the same output is displayed when print is used):

```
In [2]: ip1 = IPAddress('10.1.1.1')

In [3]: ip2 = IPAddress('10.2.2.2')

In [4]: str(ip1)
Out[4]: '<__main__.IPAddress object at 0xb4e4e76c>'

In [5]: str(ip2)
Out[5]: '<__main__.IPAddress object at 0xb1bd376c>'
```

Unfortunately, this presentation is not very informative. It would be better to show information about which address this instance represents. Special method __str__ is responsible for displaying information when using str function. As an argument this method expects only instance and must return string.

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip

    def __str__(self):
        return f"IPAddress: {self.ip}"


In [7]: ip1 = IPAddress('10.1.1.1')

In [8]: ip2 = IPAddress('10.2.2.2')

In [9]: str(ip1)
```

(continues on next page)

(continued from previous page)

```
Out[9]: 'IPAddress: 10.1.1.1'

In [10]: str(ip2)
Out[10]: 'IPAddress: 10.2.2.2'
```

A second string view which is used in Python objects is displayed when using `repr` function and when adding objects to containers such as lists:

```
In [11]: ip_addresses = [ip1, ip2]

In [12]: ip_addresses
Out[12]: [<__main__.IPAddress at 0xb4e40c8c>, <__main__.IPAddress at 0xb1bc46ac>]

In [13]: repr(ip1)
Out[13]: '<__main__.IPAddress object at 0xb4e40c8c>'
```

Method `__repr__` is responsible for this output and it should also return a string, but it would return a string by copying which you can get an instance of a class:

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip

    def __str__(self):
        return f"IPAddress: {self.ip}"

    def __repr__(self):
        return f"IPAddress('{self.ip}')"


In [15]: ip1 = IPAddress('10.1.1.1')

In [16]: ip2 = IPAddress('10.2.2.2')

In [17]: ip_addresses = [ip1, ip2]

In [18]: ip_addresses
Out[18]: [IPAddress('10.1.1.1'), IPAddress('10.2.2.2')]

In [19]: repr(ip1)
Out[19]: "IPAddress('10.1.1.1')"
```

Arithmetic operator support

Special methods are also responsible for arithmetic operations support, for example, __add__ method is responsible for addition operation:

```
__add__(self, other)
```

Let's add to IPAddress class the support of summing with numbers, but in order not to complicate method implementation we will take an advantage of ipaddress module possibilities.

```
In [1]: import ipaddress

In [2]: ipaddress1 = ipaddress.ip_address('10.1.1.1')

In [3]: int(ipaddress1)
Out[3]: 167837953

In [4]: ipaddress.ip_address(167837953)
Out[4]: IPv4Address('10.1.1.1')
```

IPAddress class with __add__:

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip

    def __str__(self):
        return f"IPAddress: {self.ip}"

    def __repr__(self):
        return f"IPAddress('{self.ip}')"

    def __add__(self, other):
        ip_int = int(ipaddress.ip_address(self.ip))
        sum_ip_str = str(ipaddress.ip_address(ip_int + other))
        return IPAddress(sum_ip_str)
```

ip_int variable refers to source address value in decimal format. And sum_ip_str is a string with IP address obtained by adding two numbers. In general, it is desirable that the summation operation returns an instance of the same class, so in the last line of method an instance of IPAddress class is created and a string with resulting address is passed to it as an argument.

Now IPAddress class instances support addition with number. As a result we get a new instance of IPAddress class.

```
In [6]: ip1 = IPAddress('10.1.1.1')

In [7]: ip1 + 5
Out[7]: IPAddress('10.1.1.6')
```

Since ipaddress module is used within method and it supports creating IP address only from a decimal number, it is necessary to limit method to work only with `int` data type. If the second element was an object of another type, an exception must be generated. The exception and error message are taken from a similar error in the `ipaddress.ip_address` function:

```
In [8]: a1 = ipaddress.ip_address('10.1.1.1')

In [9]: a1 + 4
Out[9]: IPv4Address('10.1.1.5')

In [10]: a1 + 4.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-a0a045adedc5> in <module>
----> 1 a1 + 4.0

TypeError: unsupported operand type(s) for +: 'IPv4Address' and 'float'
```

Now IPAddress class looks like:

```
class IPAddress:
    def __init__(self, ip):
        self.ip = ip

    def __str__(self):
        return f"IPAddress: {self.ip}"

    def __repr__(self):
        return f"IPAddress('{self.ip}')"

    def __add__(self, other):
        if not isinstance(other, int):
            raise TypeError(f"unsupported operand type(s) for +:"
                            f" 'IPAddress' and '{type(other).__name__}'")

        ip_int = int(ipaddress.ip_address(self.ip))
        sum_ip_str = str(ipaddress.ip_address(ip_int + other))
        return IPAddress(sum_ip_str)
```

If the second operand is not an instanse of `int` class, a TypeError exception is generated. In excep-

tion, information is displayed that summation is not supported between IPAddress class instances and operand class instance. Class name is derived from class itself, after calling type: `type(other).__name__`.

Check for summation with decimal number and error generation:

```
In [12]: ip1 = IPAddress('10.1.1.1')

In [13]: ip1 + 5
Out[13]: IPAddress('10.1.1.6')

In [14]: ip1 + 5.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-5e619f8dc37a> in <module>
----> 1 ip1 + 5.0

<ipython-input-11-77b43bc64757> in __add__(self, other)
     11     def __add__(self, other):
     12         if not isinstance(other, int):
---> 13             raise TypeError(f"unsupported operand type(s) for +:"
     14                             f" 'IPAddress' and '{type(other).__name__}'")
     15

TypeError: unsupported operand type(s) for +: 'IPAddress' and 'float'

In [15]: ip1 + '1'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-c5ce818f55d8> in <module>
----> 1 ip1 + '1'

<ipython-input-11-77b43bc64757> in __add__(self, other)
     11     def __add__(self, other):
     12         if not isinstance(other, int):
---> 13             raise TypeError(f"unsupported operand type(s) for +:"
     14                             f" 'IPAddress' and '{type(other).__name__}'")
     15

TypeError: unsupported operand type(s) for +: 'IPAddress' and 'str'
```

See also:

Manual of special methods Numeric magic methods

Protocols

Special methods are responsible not only for support of operations like addition and comparison, but also for protocol support. Protocol - set of methods that must be implemented in object to make object support a certain behavior. For example, Python has protocols like iteration, context manager, containers and others. After creating certain methods in the object, it will behave as built-in and use an interface understood by all who write on Python.

Note: A table with abstract classes describing which methods an object should have to make it support a certain protocol

---

Iteration protocol

`iterable` - object that can return elements one at a time. For Python, it is any object that has `__iter__` or `__getitem__` method. If an object has `__iter__` method, the iterable becomes an iterator by calling `iter(name)` where `name` - name of iterable. If `__iter__` method is not present, Python iterates elements using `__getitem__` (also by calling `iter` function).

```
class Items:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]


In [2]: iterable_1 = Items([1, 2, 3, 4])

In [3]: iterable_1[0]
Calling __getitem__
Out[3]: 1

In [4]: for i in iterable_1:
   ...:     print('>>>>', i)
   ...:
Calling __getitem__
>>>> 1
Calling __getitem__
>>>> 2
Calling __getitem__
>>>> 3
```

(continues on next page)

(continued from previous page)

```
Calling __getitem__
>>>> 4
Calling __getitem__

In [5]: list(map(str, iterable_1))
Calling __getitem__
Calling __getitem__
Calling __getitem__
Calling __getitem__
Calling __getitem__
Out[5]: ['1', '2', '3', '4']
```

If object has `__iter__` method (which must return iterator), it is used for values iteration:

```
class Items:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]

    def __iter__(self):
        print('Вызываю __iter__')
        return iter(self.items)


In [12]: iterable_1 = Items([1, 2, 3, 4])

In [13]: for i in iterable_1:
     ...:     print('>>>>', i)
     ...:
Calling __iter__
>>>> 1
>>>> 2
>>>> 3
>>>> 4

In [14]: list(map(str, iterable_1))
Calling __iter__
Out[14]: ['1', '2', '3', '4']
```

In Python, `iter` function is responsible for getting an iterator:

```
In [1]: lista = [1, 2, 3]

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

`iter` function will work on any object that has `__iter__` or `__getitem__` method. Method `__iter__` returns an iterator. If this method is not available, `iter` function checks availability of `__getitem__` method that can get elements by index. If `__getitem__` method exists, elements will be iterated through index (starting with 0).

`iterator` - object that returns its elements one at a time. From Python point of view, it is any object that has `__next__` method. This method returns the next item if any or raises `Stopiteration` exception when items are ended. In addition, iterator remembers which object it stopped at in the last iteration. Each iterator also has `__iter__` method - that is, every iterator is an iterable object. This method returns iterator itself.

An example of creating iterator from list:

```
In [3]: lista = [1, 2, 3]

In [4]: i = iter(lista)
```

Now you can use `next` function that calls `__next__` method to take the next element:

```
In [5]: next(i)
Out[5]: 1

In [6]: next(i)
Out[6]: 2

In [7]: next(i)
Out[7]: 3

In [8]: next(i)
---------------------------------------------------------------
StopIteration                Traceback (most recent call last)
<ipython-input-8-bed2471d02c1> in <module>()
----> 1 next(i)

StopIteration:
```

After elements are ended, `Stopiteration` exception is raised. In order for iterator to return elements again, it has to be re-created. Similar steps are performed when `for` loop iterates items in the list:

```
In [9]: for item in lista:
```

(continues on next page)

(continued from previous page)

```
   ...:     print(item)
   ...:
1
2
3
```

When we iterate list items, `iter` function is first applied to the list to create an iterator and then `__next__` method is called until `Stopiteration` exception raised.

An example of `my_for` function that works with any iterable and loosely imitates built-in function `for` (actually gititem are iterated over by iter function):

```
def my_for(iterable):
    if getattr(iterable, "__iter__", None):
        print('Есть __iter__')
        iterator = iter(iterable)
        while True:
            try:
                print(next(iterator))
            except StopIteration:
                break
    elif getattr(iterable, "__getitem__", None):
        print('Нет __iter__, но есть __getitem__')
        index = 0
        while True:
            try:
                print(iterable[index])
                index += 1
            except IndexError:
                break
```

Check function on object that has `__iter__`:

```
In [18]: my_for([1, 2, 3, 4])
Есть __iter__
1
2
3
4
```

Check function on object that does not have `__iter__` but has `__getitem__`:

```
class Items:
    def __init__(self, items):
```

(continues on next page)

(continued from previous page)

```
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]


In [20]: iterable_1 = Items([1, 2, 3, 4, 5])

In [21]: my_for(iterable_1)
Нет __iter__, но есть __getitem__
Calling __getitem__
1
Calling __getitem__
2
Calling __getitem__
3
Calling __getitem__
4
Calling __getitem__
5
Calling __getitem__
```

Iterator creation

Example of Network class:

```
In [10]: import ipaddress
    ...:
    ...: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
```

Example of Network class instance creation:

```
In [14]: net1 = Network('10.1.1.192/30')

In [15]: net1
Out[15]: <__main__.Network at 0xb3124a6c>
```

(continues on next page)

(continued from previous page)

```
In [16]: net1.addresses
Out[16]: ['10.1.1.193', '10.1.1.194']

In [17]: net1.network
Out[17]: '10.1.1.192/30'
```

Create an iterator from Network class:

```
class Network:
    def __init__(self, network):
        self.network = network
        subnet = ipaddress.ip_network(self.network)
        self.addresses = [str(ip) for ip in subnet.hosts()]
        self._index = 0

    def __iter__(self):
        print('Вызываю __iter__')
        return self

    def __next__(self):
        print('Вызываю __next__')
        if self._index < len(self.addresses):
            current_address = self.addresses[self._index]
            self._index += 1
            return current_address
        else:
            raise StopIteration
```

Method `__iter__` in iterator must return object itself, therefore `return self` is specified in method and `__next__` method returns elements one at a time and raises `StopIteration` exception when elements have run out.

```
In [14]: net1 = Network('10.1.1.192/30')

In [15]: for ip in net1:
    ...:     print(ip)
    ...:
Calling __iter__
Calling __next__
10.1.1.193
Calling __next__
10.1.1.194
Calling __next__
```

Most of the time, iterator is a disposable object and once we've iterated elements, we can't do it again:

```
In [16]: for ip in net1:
    ...:     print(ip)
    ...:
Calling __iter__
Calling __next__
```

Creation of iterable

Very often it is sufficient for class to be an iterable and not necessarily an iterator. If an object is iterable, it can be used in `for` loop, `map` functions, `filter`, `sorted`, `enumerate` and others. It is also generally easier to make an iterable than an iterator.

In order for Network class to be iterable, class must have `__iter__` (`__next__` is not needed) and method must return iterator. Since in this case, Network iterates addresses that are in `self.addresses` list, the easiest option to return iterator is to return `iter(self.addresses)`:

```
class Network:
    def __init__(self, network):
        self.network = network
        subnet = ipaddress.ip_network(self.network)
        self.addresses = [str(ip) for ip in subnet.hosts()]

    def __iter__(self):
        return iter(self.addresses)
```

Now all Network class instances will be iterable objects:

```
In [18]: net1 = Network('10.1.1.192/30')

In [19]: for ip in net1:
    ...:     print(ip)
    ...:
10.1.1.193
10.1.1.194
```

Sequence protocol

In the most basic version, sequence protocol (sequence) includes two methods: `__len__` and `__getitem__`. In more complete version also methods: `__contains__`, `__iter__`, `__reversed__`, `index` and `count`. If sequence is mutable, several other methods are added.

Add __len__ and __getitem__ methods to Network class:

```
In [1]: class Network:
   ...:     def __init__(self, network):
   ...:         self.network = network
   ...:         subnet = ipaddress.ip_network(self.network)
   ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
   ...:
   ...:     def __iter__(self):
   ...:         return iter(self.addresses)
   ...:
   ...:     def __len__(self):
   ...:         return len(self.addresses)
   ...:
   ...:     def __getitem__(self, index):
   ...:         return self.addresses[index]
   ...:
```

Method __len__ is called by len function:

```
In [2]: net1 = Network('10.1.1.192/30')

In [3]: len(net1)
Out[3]: 2
```

And __getitem__ method is called when you acess item by index:

```
In [4]: net1[0]
Out[4]: '10.1.1.193'

In [5]: net1[1]
Out[5]: '10.1.1.194'

In [6]: net1[-1]
Out[6]: '10.1.1.194'
```

__getitem__ method is responsible not only for access by index, but also for slices:

```
In [7]: net1 = Network('10.1.1.192/28')

In [8]: net1[0]
Out[8]: '10.1.1.193'

In [9]: net1[3:7]
Out[9]: ['10.1.1.196', '10.1.1.197', '10.1.1.198', '10.1.1.199']
```

(continues on next page)

(continued from previous page)

```
In [10]: net1[3:]
Out[10]:
['10.1.1.196',
 '10.1.1.197',
 '10.1.1.198',
 '10.1.1.199',
 '10.1.1.200',
 '10.1.1.201',
 '10.1.1.202',
 '10.1.1.203',
 '10.1.1.204',
 '10.1.1.205',
 '10.1.1.206']
```

In this case, because __getitem__ method uses a `list`, errors are processed correctly automatically:

```
In [11]: net1[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-09ca84e34cb6> in <module>
----> 1 net1[100]

<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
---> 14         return self.addresses[index]
     15

IndexError: list index out of range

In [12]: net1['a']
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-facd90673864> in <module>
----> 1 net1['a']

<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
---> 14         return self.addresses[index]
     15
```

(continues on next page)

(continued from previous page)

```
TypeError: list indices must be integers or slices, not str
```

Remaining methods of sequence protocol:

- `__contains__` - this method is responsible for checking the presence of element in sequence `'10.1.1.198' in net1`. If object does not define this method, the presence of element is checked by iteration of elements using `__iter__` and if this method is also unavailable, then by index iteration with `__getitem__`.
- `__reversed__` - is used by built-in `reversed` function. This method is usually best not to create and rely on the fact that `reversed` function in absence of `__reversed__` method will use methods `__len__` and `__getitem__`.
- `index` - returns index of element. Works exactly the same as `index` method in lists and tuples.
- `count` - returns number of values. Works exactly the same as `count` method in lists and tuples.

Context manager

Context manager allows specified actions to be performed at the beginning and end of `with` block. Two methods are responsible for context manager:

- `__enter__(self)` - indicates what should be done at the beginning of `with` block. Value that returns method is assigned to variable after `as`.
- `__exit__(self, exc_type, exc_value, traceback)` - indicates what should be done at the end of `with` block or when it is interrupted. If there is an exception within block, then exc_type, exc_value, traceback will contain exception information, if there is no exception they will be equal to None.

Examples of context manager usage:

- file opening/closing
- opening/closing of SSH/Telnet session
- transactions handling in database

CiscoSSH class uses paramiko to connect to the equipment:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
```

(continues on next page)

(continued from previous page)

```
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self.ssh = client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

Example of class usage:

```
In [9]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [10]: r1.send_show_command('sh clock')
Out[10]: 'sh clock\r\n*12:58:47.523 UTC Sun Jul 28 2019\r\nR1#'

In [11]: r1.send_show_command('sh ip int br')
Out[11]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method␣
↪Status                Protocol\r\nEthernet0/0               192.168.100.1  ␣
↪YES NVRAM  up                    up      \r\nEthernet0/1               192.168.
↪200.1   YES NVRAM  up                    up      \r\nEthernet0/2                ␣
↪19.1.1.1        YES NVRAM  up                    up      \r\nEthernet0/3       ␣
↪        192.168.230.1   YES NVRAM  up                    up      \r\nLoopback0 ␣
↪               4.4.4.4         YES NVRAM  up                    up          ␣
↪\r\nLoopback90                90.1.1.1        YES manual up                    ␣
↪up      \r\nR1#'
```

In order for the class to support work in context manager, it is necessary to add methods __enter__ and __exit__:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        print('Метод __init__')
        client = paramiko.SSHClient()
```

(continues on next page)

(continued from previous page)

```
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self.ssh = client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def __enter__(self):
        print('Метод __enter__')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print('Метод __exit__')
        self.ssh.close()

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

Example of class usage in context manager:

```
In [14]: with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:
Метод __init__
Метод __enter__
sh clock
*13:05:50.677 UTC Sun Jul 28 2019
R1#
Метод __exit__
```

Even if an exception occurs within block, __exit__ method is executed:

```
In [18]: with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
    ...:     result = r1.send_show_command('sh clock')
    ...:     result / 2
    ...:
Метод __init__
Метод __enter__
Метод __exit__
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-b9ff1fa74be2> in <module>
      1 with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
      2     result = r1.send_show_command('sh clock')
----> 3     result / 2
      4

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 23.1

In this task, you must create an IPAddress class.

When creating an instance of a class, the IP address and mask are passed as an argument, and the correctness of the address and mask must be checked:

- The address is considered to be correctly specified if it:
  - consists of 4 numbers separated by a dot
  - every number in the range from 0 to 255
- the mask is considered correct if the mask is a number and a number in the range from 8 to 32 inclusiveA

If the mask or address fails validation, you must raise a ValueError with the appropriate text (output below).

Also, when creating a class, two instance variables must be created: ip and mask, which contain the address and mask, respectively.

An example of creating an instance of a class:

```
In [1]: ip1 = IPAddress('10.1.1.1/24')
```

ip and mask attributes

```
In [2]: ip1 = IPAddress('10.1.1.1/24')

In [3]: ip1.ip
Out[3]: '10.1.1.1'

In [4]: ip1.mask
Out[4]: 24
```

Checking the correctness of the address (traceback is shortened)

```
In [5]: ip1 = IPAddress('10.1.1/24')
-------------------------------------------------------------------------
...
ValueError: Incorrect IPv4 address

Checking the correctness of the mask (traceback is shortened)
In [6]: ip1 = IPAddress('10.1.1.1/240')
-------------------------------------------------------------------------
...
ValueError: Incorrect mask
```

Task 23.1a

Copy and modify the IPAddress class from task 23.1.

Add two string views for instances of the IPAddress class. How string representations should look like should be determined from the output below.

An example of creating an instance of a class:

```
In [5]: ip1 = IPAddress('10.1.1.1/24')

In [6]: str(ip1)
Out[6]: 'IP address 10.1.1.1/24'

In [7]: print(ip1)
IP address 10.1.1.1/24

In [8]: ip1
Out[8]: IPAddress('10.1.1.1/24')

In [9]: ip_list = []

In [10]: ip_list.append(ip1)

In [11]: ip_list
Out[11]: [IPAddress('10.1.1.1/24')]

In [12]: print(ip_list)
[IPAddress('10.1.1.1/24')]
```

Task 23.2

Copy the CiscoTelnet class from any 22.2x task and add context manager support to the class. When exiting the context manager block, the connection should be closed.

Example of work:

```
In [14]: r1_params = {
    ...:     'ip': '192.168.100.1',
    ...:     'username': 'cisco',
    ...:     'password': 'cisco',
    ...:     'secret': 'cisco'}

In [15]: from task_23_2 import CiscoTelnet

In [16]: with CiscoTelnet(**r1_params) as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:
sh clock
*19:17:20.244 UTC Sat Apr 6 2019
R1#
```

Task 23.3

Copy and modify the Topology class from job 22.1x.

In this task, you need to add a method that will allow you to add two instances of the Topology class. The addition should return a new instance of the Topology class.

Creating two instances of the Topology class:

```
In [1]: t1 = Topology(topology_example)

In [2]: t1.topology
Out[2]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [3]: topology_example2 = {('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
                             ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

(continues on next page)

(continued from previous page)

```
In [4]: t2 = Topology(topology_example2)

In [5]: t2.topology
Out[5]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Summing instances of the Topology class:

```
In [6]: t3 = t1 + t2

In [7]: t3.topology
Out[7]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
 ('R1', 'Eth0/6'): ('R9', 'Eth0/0'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Checking that the original instances haven't changed:

```
In [9]: t1.topology
Out[9]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [10]: t2.topology
Out[10]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Task 23.3a

In this task, you need to make sure that instances of the Topology class are iterables. The base of the Topology class can be taken from either task 22.1x or task 23.3.

After creating an instance of a class, the instance should act like an iterable object. Each iteration should return a tuple that describes one connection. The order of output of connections can be any.

An example of how the class works:

```
In [1]: top = Topology(topology_example)

In [2]: for link in top:
   ...:     print(link)
   ...:
(('R1', 'Eth0/0'), ('SW1', 'Eth0/1'))
(('R2', 'Eth0/0'), ('SW1', 'Eth0/2'))
(('R2', 'Eth0/1'), ('SW2', 'Eth0/11'))
(('R3', 'Eth0/0'), ('SW1', 'Eth0/3'))
(('R3', 'Eth0/1'), ('R4', 'Eth0/0'))
(('R3', 'Eth0/2'), ('R5', 'Eth0/0'))
```

24. Inheritance

Inheritance basics

Inheritance allows creation of new classes based on existing ones. There are child and parents classes: child class inherits parent class. In inheritance, child class inherits all methods and attributes of parent class.

Example of ConnectSSH class that performs SSH connection using paramiko:

```
import paramiko
import time


class ConnectSSH:
    def __init__(self, ip, username, password):
        self.ip = ip
        self.username = username
        self.password = password
        self._MAX_READ = 10000

        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = client.invoke_shell()
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._ssh.close()

    def close(self):
        self._ssh.close()
```

(continues on next page)

(continued from previous page)

```
    def send_show_command(self, command):
        self._ssh.send(command + '\n')
        time.sleep(2)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        if isinstance(commands, str):
            commands = [commands]
        for command in commands:
            self._ssh.send(command + '\n')
            time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result
```

This class will be used as the basis for classes that are responsible for connecting to devices of different vendors. For example, CiscoSSH class will be responsible for connecting to Cisco devices and will inherit ConnectSSH class.

Inheritance syntax:

```
class CiscoSSH(ConnectSSH):
    pass
```

After that, all ConnectSSH methods and attributes are available in CiscoSSH class:

```
In [3]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco')

In [4]: r1.ip
Out[4]: '192.168.100.1'

In [5]: r1._MAX_READ
Out[5]: 10000

In [6]: r1.send_show_command('sh ip int br')
Out[6]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method␣
↪Status                Protocol\r\nEthernet0/0               192.168.100.1  ␣
↪YES NVRAM  up                    up      \r\nEthernet0/1               192.168.
↪200.1   YES NVRAM  up                    up      \r\nEthernet0/2                ␣
↪19.1.1.1        YES NVRAM  up                    up      \r\nEthernet0/3       ␣
↪        192.168.230.1   YES NVRAM  up                    up      \r\nLoopback0 ␣
↪               4.4.4.4         YES NVRAM  up                    up          ␣
↪\r\nLoopback33               3.3.3.3         YES manual up                    ␣
↪up      \r\nLoopback90                90.1.1.1        YES manual up            ␣
↪        up      \r\nR1#'
```

(continues on next page)

(continued from previous page)

```
In [7]: r1.send_show_command('enable')
Out[7]: 'enable\r\nPassword: '

In [8]: r1.send_show_command('cisco')
Out[8]: '\r\nR1#'

In [9]: r1.send_config_commands(['conf t', 'int loopback 33',
   ...:                          'ip address 3.3.3.3 255.255.255.255', 'end'])
Out[9]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#int loopback 33\r\nR1(config-if)#ip address 3.3.3.3 255.255.255.
↪255\r\nR1(config-if)#end\r\nR1#'
```

After inheriting all methods of parent class, child class can:

- leave them unchanged
- rewrite them completely
- supplement method
- add your methods

In CiscoSSH class you have to create `__init__` method and add parameters to it:

- `enable_password` - enable password
- `disable_paging` - is responsible for paging turning on/off

Method `__init__` can be created entirely from scratch but basic SSH connection logic is the same in ConnectSSH and CiscoSSH, so it is better to add necessary parameters and call `__init__` method of ConnectSSH class for connection. There are several options for calling parent method, for example, all of these options will call `send_show_command` method of parent class from child class CiscoSSH:

```
command_result = ConnectSSH.send_show_command(self, command)
command_result = super(CiscoSSH, self).send_show_command(command)
command_result = super().send_show_command(command)
```

The first version of `ConnectSSH.send_show_command` explicitly specifies the name of parent class - this is the most understandable version for perception, but its disadvantage is that when a parent class name is changed the name will have to be changed in all places where parent class methods were called. This option also has disadvantages when using multiple inheritance. The second and third options are essentially equivalent but the third option is shorter, so we will use it.

CiscoSSH class with `__init__` method:

```
class CiscoSSH(ConnectSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
```

Method __init__ in CiscoSSH class added enable_password and disable_paging parameters and uses them accordingly to enter enable mode and disable paging. Example:

```
In [10]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [11]: r1.send_show_command('sh clock')
Out[11]: 'sh clock\r\n*11:30:50.280 UTC Mon Aug 5 2019\r\nR1#'
```

Now when connecting, switch enters enable mode and paging is disabled by default, so you can try to run a long command like sh run.

Another method that should be further developed is send_config_commands method: since CiscoSSH class is designed to work with Cisco, you can add switching to configuration mode before commands and exit after.

```
class CiscoSSH(ConnectSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
```

(continues on next page)

(continued from previous page)

```
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result
```

Example of send_config_commands method use:

```
In [12]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [13]: r1.send_config_commands(['interface loopback 33',
    ...:                          'ip address 3.3.3.3 255.255.255.255'])
Out[13]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.
↪\r\nR1(config)#interface loopback 33\r\nR1(config-if)#ip address 3.3.3.3 255.
↪255.255.255\r\nR1(config-if)#end\r\nR1#'
```

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 24.1

Create a CiscoSSH class that inherits the BaseSSH class from the base_connect_class.py file.

Create an __init__ method in the CiscoSSH class so that after connecting via SSH, it switches to enable mode.

To do this, the __init__ method must first call the __init__ method of the BaseSSH class, and then switch to enable mode.

```
In [2]: from task_24_1 import CiscoSSH

In [3]: r1 = CiscoSSH(**device_params)

In [4]: r1.send_show_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status
↪   Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up
↪         up      \nEthernet0/1                192.168.200.1   YES NVRAM  up
↪               up      \nEthernet0/2                190.16.200.1    YES NVRAM
↪up                    up      \nEthernet0/3                192.168.230.1   YES
↪NVRAM  up                    up      \nEthernet0/3.100            10.100.0.1
↪  YES NVRAM  up                    up      \nEthernet0/3.200            10.200.
↪0.1      YES NVRAM  up                    up      \nEthernet0/3.300
↪10.30.0.1       YES NVRAM  up                    up      '
```

Task 24.1a

Copy and update the CiscoSSH class from task 24.1.

Before connecting via SSH, you need to check if the dictionary with the connection parameters contains the following parameters: username, password, secret. If any parameter is missing, ask the user for a value and then connect. If all parameters are present, connect.

```
In [1]: from task_24_1a import CiscoSSH

In [2]: device_params = {
   ...:         'device_type': 'cisco_ios',
   ...:         'host': '192.168.100.1',
   ...: }

In [3]: r1 = CiscoSSH(**device_params)
Enter username: cisco
Enter password: cisco
Enter enable passwod: cisco

In [4]: r1.send_show_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status
↪   Protocol\nEthernet0/0               192.168.100.1   YES NVRAM  up
↪         up      \nEthernet0/1               192.168.200.1   YES NVRAM  up
↪               up      \nEthernet0/2               190.16.200.1    YES NVRAM
↪up                   up      \nEthernet0/3               192.168.230.1   YES
↪NVRAM  up                    up      \nEthernet0/3.100           10.100.0.1
↪  YES NVRAM  up                    up      \nEthernet0/3.200           10.200.
↪0.1      YES NVRAM  up                    up      \nEthernet0/3.300
↪10.30.0.1       YES NVRAM  up                    up      '
```

Task 24.2

Create a MyNetmiko class that inherits the CiscoIosSSH class from netmiko. Write the __init__ method in the MyNetmiko class so that after connecting via SSH, it switches to enable mode.

To do this, the __init__ method must first call the __init__ method of the CiscoIosSSH class, and then switch to enable mode.

Check that the send_command and send_config_set methods are available in the MyNetmiko class (they are inherited automatically, this is just for checking).

```
In [2]: from task_24_2 import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status
↪   Protocol\nEthernet0/0               192.168.100.1   YES NVRAM  up
↪         up      \nEthernet0/1               192.168.200.1   YES NVRAM  up
↪               up      \nEthernet0/2               190.16.200.1    YES NVRAM
↪up                   up      \nEthernet0/3               192.168.230.1   YES
↪NVRAM  up                    up      \nEthernet0/3.100           10.100.0.1
↪  YES NVRAM  up                    up      \nEthernet0/3.200           10.200.
↪0.1      YES NVRAM  up                    up      \nEthernet0/3.300
↪10.30.0.1       YES NVRAM  up                    up      '
```

(continues on next page)

(continued from previous page)

Task 24.2a

Copy and update the MyNetmiko class from task 24.2.

Add the _check_error_in_command method that checks for such errors: Invalid input detected, Incomplete command, Ambiguous command

The method expects a command and command output as an argument. If no error is found in the output, the method returns nothing. If an error is found in the output, the method should raise an ErrorInCommand exception with a message about which error was detected, on which device, and in which command.

An ErrorInCommand exception is created in the task file.

Rewrite send_command method to include error checking.

```
In [2]: from task_24_2a import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status
↪   Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up
↪         up      \nEthernet0/1                192.168.200.1   YES NVRAM  up
↪               up      \nEthernet0/2                190.16.200.1    YES NVRAM
↪up                    up      \nEthernet0/3                192.168.230.1   YES
↪NVRAM  up                    up      \nEthernet0/3.100            10.100.0.1
↪  YES NVRAM  up                    up      \nEthernet0/3.200            10.200.
↪0.1      YES NVRAM  up                    up      \nEthernet0/3.300
↪10.30.0.1       YES NVRAM  up                    up      '

In [5]: r1.send_command('sh ip br')
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-2-1c60b31812fd> in <module>()
----> 1 r1.send_command('sh ip br')
...
ErrorInCommand: When executing the command "sh ip br" on device 192.168.100.1, an
↪error occurred "Invalid input detected at '^' marker."
```

Task 24.2b

Copy the class MyNetmiko from task 24.2a.

Add error checking to the send_config_set method using the _check_error_in_command method.

The send_config_set method should send commands one at a time and check each for errors. If no errors are encountered while executing the commands, the send_config_set method returns the output of the commands.

```
In [2]: from task_24_2b import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_config_set('lo')
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-2-8e491f78b235> in <module>()
----> 1 r1.send_config_set('lo')

...
ErrorInCommand: When executing the command "lo" on device 192.168.100.1, an error␣
↪occurred "Incomplete command."
```

Task 24.2c

Copy the class MyNetmiko from task 24.2b. Check that the send_command method, in addition to a command, also accepts additional arguments, for example, strip_command.

If an error occurs, rewrite the method to accept any arguments that netmiko supports.

```
In [2]: from task_24_2c import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_command('sh ip int br', strip_command=False)
Out[4]: 'sh ip int br\nInterface                  IP-Address      OK? Method␣
↪Status                Protocol\nEthernet0/0                192.168.100.1   YES␣
↪NVRAM  up                    up      \nEthernet0/1                192.168.200.1␣
↪  YES NVRAM  up                    up      \nEthernet0/2                190.16.
↪200.1    YES NVRAM  up                    up      \nEthernet0/3                ␣
↪192.168.230.1   YES NVRAM  up                    up      \nEthernet0/3.100     ␣
↪      10.100.0.1      YES NVRAM  up                    up      \nEthernet0/3.
↪200            10.200.0.1      YES NVRAM  up                    up      ␣
↪\nEthernet0/3.300            10.30.0.1       YES NVRAM  up                    ␣
↪up      '
```

(continues on next page)

(continued from previous page)

```
In [5]: r1.send_command('sh ip int br', strip_command=True)
Out[5]: 'Interface                  IP-Address      OK? Method Status
↪   Protocol\nEthernet0/0               192.168.100.1   YES NVRAM  up
↪         up      \nEthernet0/1               192.168.200.1   YES NVRAM  up
↪               up      \nEthernet0/2               190.16.200.1    YES NVRAM
↪up                    up      \nEthernet0/3               192.168.230.1   YES
↪NVRAM  up                    up      \nEthernet0/3.100           10.100.0.1
↪  YES NVRAM  up                    up      \nEthernet0/3.200           10.200.
↪0.1      YES NVRAM  up                    up      \nEthernet0/3.300
↪10.30.0.1       YES NVRAM  up                    up      '
```

Task 24.2d

Copy class MyNetmiko from task 24.2c or task 24.2b.

Add the ignore_errors parameter to the send_config_set method. If ignore_errors=True, no error checking is needed and the method should work exactly like the send_config_set method in netmiko. If ignore_errors=False, errors should be checked.

By default, errors should be ignored.

```
In [2]: from task_24_2d import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [6]: r1.send_config_set('lo')
Out[6]: 'config term\nEnter configuration commands, one per line.  End with CNTL/
↪Z.\nR1(config)#lo\n% Incomplete command.\n\nR1(config)#end\nR1#'

In [7]: r1.send_config_set('lo', ignore_errors=True)
Out[7]: 'config term\nEnter configuration commands, one per line.  End with CNTL/
↪Z.\nR1(config)#lo\n% Incomplete command.\n\nR1(config)#end\nR1#'

In [8]: r1.send_config_set('lo', ignore_errors=False)
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-8-704f2e8d1886> in <module>()
----> 1 r1.send_config_set('lo', ignore_errors=False)

...
ErrorInCommand: When executing the command "lo" on device 192.168.100.1, an error
↪occurred "Incomplete command."
```

9

VII. Working with databases

25. Database operations

The use of databases is another way of storing information. Databases are useful not only in storing information. Using DBMS it is possible to make information slices according to different parameters.

Database (DB) - data stored according to a certain scheme. This scheme describes relationships between data. **DB language (language tools)** - used to describe database structure, manage data (add, edit, delete, receive), manage access rights to database and its objects, and manage transactions.

Database Management System (DBMS) - a software tool that enables management of DB. DBMS must support appropriate language(s) for DB management.

SQL

SQL (structured query language) - used to describe database structure, manage data (add, edit, delete, receive), manage access rights to database and its objects, and manage transactions.

SQL language is divided into the following categories:

- DDL (Data Definition Language)
- DML (Data Manipulation Language)
- DCL (Data Control Language)
- TCL (Transaction Control Language)

Each category has its own operators (not all operators are listed):

- DDL
  - CREATE - create new table, DBMS, schemas
  - ALTER - change of existing table, columns
  - DROP - removing existing objects from DBMS
- DML
  - SELECT - data selection
  - INSERT - adding new data
  - UPDATE - updating existing data
  - DELETE - deleting data
- DCL
  - GRANT - Allow users to read/write certain objects to DBMS
  - REVOKE - - withdrawal of prior authorizations

- TCL
  - COMMIT - committing of transaction
  - ROLLBACK - rollback of all changes made in the current transaction

SQL and Python

Two approaches can be used to work with a relational DBMS in Python:

- work with a library that corresponds to a specific database and use SQL language to work with database. For example, sqlite uses sqlite3 module
- work with ORM which uses an object-oriented approach to work with database. For example, Sqlalchemy

SQLite

SQLite — a built-in SQL machine implementation. Sqlite is often used as an embedded DBMS in applications.

Note: The word SQL server is not used here because server is not needed there - all functionality that is embedded in SQL server is implemented inside a library (and therefore within program that uses it).

SQLite CLI

SQLite package also includes a command line utility for working with SQLite. Utility is presented as a sqlite3 executable file (sqlite3.exe for Windows) and can be used to execute SQL commands manually.

With this utility it is very convenient to check the correctness of SQL commands as well as to get acquainted with SQL language in general.

Let's try to use this utility to figure out basic SQL commands that will be needed to work with database.

We'll figure out how to build a database first.

Note: If you are using Linux or Mac OS, it is likely that sqlite3 is installed. If you are using Windows you can download sqlite3 here.

To create a database (or open an already created database), you simply call sqlite3:

```
$ sqlite3 testDB.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite>
```

Inside sqlite3 you can execute SQL commands or so-called metacommands (or dot commands).

Metacommands include several special commands to work with SQLite. They refer only to sqlite3 utility, not to SQL language. There is no need to put ; at the end of command.

Examples of metacommands:

- `.help` - a prompt with a list of all metacommands
- `.exit` or `.quit` - exit sqlite3 session
- `.databases` - shows connected databases
- `.tables` - shows available tables

Examples:

```
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail ON|OFF           Stop after hitting an error.  Default OFF
.databases             List names and files of attached databases
...

sqlite> .databases
seq  name      file
---  --------  ----------------------------------
0    main      /home/nata/py_for_ne/db/db_article/testDB.db
```

litecli

The standard Sqlite CLI interface has several disadvantages:

- no autocomplete commands
- no tips
- all content of a column is not always displayed

All these deficiencies are fixed in litecli. So it's best to use it.

Installation of litecli:

```
$ pip install litecli
```

Open database in litecli:

```
$ litecli example.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
example.db>
```

SQL basics (in sqlite3 CLI)

This section covers with SQL syntax.

If you are familiar with basic SQL syntax you can skip this section and move to section *Sqlite3 module*

CREATE

CREATE TABLE statement allows you to create tables.

First connect to database or create it with litecli:

```
$ litecli new_db.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
new_db.db>
```

Create a *switch* table which stores information about switches:

```
new_db.db> create table switch (mac text not NULL primary key, hostname text,␣
↪model text, location text);
Query OK, 0 rows affected
Time: 0.010s
```

In this example, we described *switch* table: we defined which fields would be in the table and which types of values would be in them.

Additionally, *mac* field is primary key. That automatically means that:

- field must be unique
- field cannot have null value (in SQLite this must be stated explicitly)

In this example this is quite logical as MAC address must be unique.

There are no entries in the table at the moment, only a definition. You can view definition with this command:

```
new_db.db> .schema switch
+-------------------------------------------------------------------------------
↪--------------+
| sql
↪              |
+-------------------------------------------------------------------------------
↪--------------+
| CREATE TABLE switch (mac text not NULL primary key, hostname text, model text,␣
↪location text) |
+-------------------------------------------------------------------------------
↪--------------+
Time: 0.037s
```

DROP

DROP operator removes table along with schema and all data.

You can delete table like this:

```
new_db.db> DROP table switch;
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 0 rows affected
Time: 0.009s
```

INSERT

INSERT operator is used to add data to the table.

Note: If table was deleted in previous step, create it:

```
new_db.db> create table switch (mac text not NULL primary key, hostname text,␣
↪model text, location text);
Query OK, 0 rows affected
Time: 0.010s
```

There are several options for adding entries, depending on whether all fields are filled and whether or not they follow the field order.

If values for all fields are specified you can add an entry in this way (the order of fields must be respected):

```
new_db.db> INSERT into switch values ('0010.A1AA.C1CC', 'sw1', 'Cisco 3750',
↪'London, Green Str');
Query OK, 1 row affected
Time: 0.008s
```

If you want to specify not all fields or specify them randomly, this entry is used:

```
new_db.db> INSERT into switch (mac, model, location, hostname) values ('0020.A2AA.
↪C2CC', 'Cisco 3850', 'London, Green Str', 'sw2');
Query OK, 1 row affected
Time: 0.009s
```

SELECT

SELECT operator allows you to query information from the table.

For example:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
+----------------+----------+------------+--------------------+
2 rows in set
Time: 0.033s
```

SELECT * means that all fields in the table must be displayed. Then indicates from which table data is requested: `from switch`.

Thus, it is possible to specify specific columns to be derived and in what order:

```
new_db.db> SELECT hostname, mac, model from switch;
+----------+----------------+------------+
| hostname | mac            | model      |
+----------+----------------+------------+
| sw1      | 0010.A1AA.C1CC | Cisco 3750 |
| sw2      | 0020.A2AA.C2CC | Cisco 3850 |
+----------+----------------+------------+
2 rows in set
Time: 0.033s
```

WHERE

WHERE operator is used to specify a query. With the help of this operator it is possible to specify certain conditions under which data is selected. If condition is met the corresponding value is returned from table, if not - it is not returned.

Now there are only two enties in *switch* table:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
+----------------+----------+------------+--------------------+
2 rows in set
Time: 0.033s
```

To create more entries in table you need to create more rows. Litecli has a **source** command that lets you upload SQL commands from a file.

File add_rows_to_testdb.txt is prepared to add entries:

```
INSERT into switch values ('0030.A3AA.C1CC', 'sw3', 'Cisco 3750', 'London, Green␣
↪Str');
INSERT into switch values ('0040.A4AA.C2CC', 'sw4', 'Cisco 3850', 'London, Green␣
↪Str');
INSERT into switch values ('0050.A5AA.C3CC', 'sw5', 'Cisco 3850', 'London, Green␣
↪Str');
INSERT into switch values ('0060.A6AA.C4CC', 'sw6', 'C3750', 'London, Green Str');
INSERT into switch values ('0070.A7AA.C5CC', 'sw7', 'Cisco 3650', 'London, Green␣
↪Str');
```

To upload commands from a file you should execute the command:

```
new_db.db> source add_rows_to_testdb.txt
Query OK, 1 row affected
Time: 0.023s

Query OK, 1 row affected
Time: 0.002s

Query OK, 1 row affected
Time: 0.003s

Query OK, 1 row affected
```

(continues on next page)

(continued from previous page)

```
Time: 0.002s

Query OK, 1 row affected
Time: 0.002s
```

Now *switch* table looks like:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str |
+----------------+----------+------------+--------------------+
7 rows in set
Time: 0.040s
```

Using the WHERE clause, you can show only those switches whose model is 3850:

```
new_db.db> SELECT * from switch WHERE model = 'Cisco 3850';
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str |
+----------------+----------+------------+--------------------+
3 rows in set
Time: 0.033s
```

WHERE operator allows you to specify more than a specific field value. If you add LIKE operator to it you can specify a field template.

Like with characters _ and % indicates what the value should look like:

- _ - denotes one character or number
- % - denotes zero, one or many characters

For example, if *model* field is written in different formats the previous query will not be able to extract needed switches. For sw6 switch the model field is written in this format: C3750, but for sw1 and

sw3 switches: Cisco 3750.

In this version, WHERE query does not show sw6:

```
new_db.db> SELECT * from switch WHERE model = 'Cisco 3750';
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str |
+----------------+----------+------------+--------------------+
2 rows in set
Time: 0.037s
```

If with WHERE operator use LIKE operator:

```
new_db.db> SELECT * from switch WHERE model LIKE '%3750';
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str |
+----------------+----------+------------+--------------------+
3 rows in set
Time: 0.040s
```

ALTER

ALTER TABKE statement allows you to change an existing table: add new columns or rename the table.

Add new fields to the table:

- mngmt_ip - switch IP address in management VLAN
- mngmt_vid - VLAN ID of management VLAN

Adding entries using ALTER TABLE:

```
new_db.db> ALTER table switch ADD COLUMN mngmt_ip text;
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 0 rows affected
Time: 0.009s
```

(continues on next page)

(continued from previous page)

```
new_db.db> ALTER table switch ADD COLUMN mngmt_vid integer;
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 0 rows affected
Time: 0.010s
```

Now table looks like this (new fields are set to NULL):

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+----------+---------
↪--+
| mac            | hostname | model      | location           | mngmt_ip | mngmt_
↪vid |
+----------------+----------+------------+--------------------+----------+---------
↪--+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | <null>   | <null> ␣
↪  |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | <null>   | <null> ␣
↪  |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>   | <null> ␣
↪  |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>   | <null> ␣
↪  |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>   | <null> ␣
↪  |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>   | <null> ␣
↪  |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>   | <null> ␣
↪  |
+----------------+----------+------------+--------------------+----------+---------
↪--+
7 rows in set
Time: 0.034s
```

UPDATE

UPDATE operator is used to change an existing table entry.

Usually, UPDATE is used with WHERE operator to specify which entry to change.

With UPDATE you can fill in new columns in the table.

For example, add an IP address for sw1 switch:

```
new_db.db> UPDATE switch set mngmt_ip = '10.255.1.1' WHERE hostname = 'sw1';
Query OK, 1 row affected
Time: 0.009s
```

Now table is like this:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | <null>
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>     | <null>
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>     | <null>
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>     | <null>
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.035s
```

VLAN number can be changed in the same way:

```
new_db.db> UPDATE switch set mngmt_vid = 255 WHERE hostname = 'sw1';
Query OK, 1 row affected
Time: 0.009s

new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
```

(continues on next page)

(continued from previous page)

```
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>     | <null>
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>     | <null>
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>     | <null>
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.037s
```

You can change several fields at a time:

```
new_db.db> UPDATE switch set mngmt_ip = '10.255.1.2', mngmt_vid = 255 WHERE␣
↪hostname = 'sw2'
Query OK, 1 row affected
Time: 0.009s

new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>     | <null>
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
```

(continues on next page)

(continued from previous page)

```
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>     | <null>
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>     | <null>
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>     | <null>
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.033s
```

To avoid filling fields mngmt_ip and mngmt_vid manually, fill in the rest from update_fields_in_testdb.txt file (command `source update_fields_in_testdb.txt`):

```
UPDATE switch set mngmt_ip = '10.255.1.3', mngmt_vid = 255 WHERE hostname = 'sw3';
UPDATE switch set mngmt_ip = '10.255.1.4', mngmt_vid = 255 WHERE hostname = 'sw4';
UPDATE switch set mngmt_ip = '10.255.1.5', mngmt_vid = 255 WHERE hostname = 'sw5';
UPDATE switch set mngmt_ip = '10.255.1.6', mngmt_vid = 255 WHERE hostname = 'sw6';
UPDATE switch set mngmt_ip = '10.255.1.7', mngmt_vid = 255 WHERE hostname = 'sw7';
```

After commands upload table is as follows:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | 10.255.1.3 | 255  ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255  ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255  ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255  ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255  ␣
↪    |
```

(continues on next page)

(continued from previous page)

```
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.038s
```

Now suppose that sw1 was replaced from 3750 model to 3850. Accordingly, not only model field but also MAC address field was changed.

Making changes:

```
new_db.db> UPDATE switch set model = 'Cisco 3850', mac = '0010.D1DD.E1EE' WHERE␣
↪hostname = 'sw1';
Query OK, 1 row affected
Time: 0.009s
```

The result will be:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | 10.255.1.3 | 255  ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255  ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255  ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255  ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255  ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.049s
```

REPLACE

REPLACE operator is used to add or replace data in the table.

Note: REPLACE operator may not be supported in all DBMS.

When a field uniqueness condition is violated, an expression with REPLACE operator:

- deletes the existing string that caused the violation
- adds a new line

An example of uniqueness condition rule violation:

```
new_db.db> INSERT INTO switch VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850',
↪'London, Green Str', '10.255.1.3', 255);
UNIQUE constraint failed: switch.mac
```

There are two types of REPLACE expression:

```
new_db.db> INSERT OR REPLACE INTO switch VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco␣
↪3850', 'London, Green Str', '10.255.1.3', 255);
Query OK, 1 row affected
Time: 0.010s
```

Or a shorter version:

```
new_db.db> REPLACE INTO switch VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850',
↪'London, Green Str', '10.255.1.3', 255);
Query OK, 1 row affected
Time: 0.009s
```

The result of any of these commands is to replace sw3 switch model:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+------------+-------
↪----+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+--------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
```

(continues on next page)

(continued from previous page)

```
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255 ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255 ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255 ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255 ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255 ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
```

In this case, MAC address in new entry is the same as in existing one, so the replacement occurs.

Note: If not all fields have been specified, the new entry will contain only those fields that have been specified. This is because REPLACE first removes an existing entry.

For entry which was added without uniqueness violation, REPLACE functions as a normal INSERT:

```
new_db.db> REPLACE INTO switch VALUES ('0080.A8AA.C8CC', 'sw8', 'Cisco 3850',
↪'London, Green Str', '10.255.1.8', 255);
Query OK, 1 row affected
Time: 0.009s

new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255 ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255 ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255 ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255 ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255 ␣
↪    |
```

(continues on next page)

(continued from previous page)

```
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255
↪    |
| 0080.A8AA.C8CC | sw8      | Cisco 3850 | London, Green Str | 10.255.1.8 | 255
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
8 rows in set
Time: 0.034s
```

DELETE

DELETE operator is used to delete enties. It is commonly used together with WHERE operator.

For example, *switch* table is:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255
↪    |
| 0080.A8AA.C8CC | sw8      | Cisco 3850 | London, Green Str | 10.255.1.8 | 255
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
```

(continues on next page)

(continued from previous page)

```
8 rows in set
Time: 0.033s
```

Deleting information about sw8 switch is performed as follows:

```
new_db.db> DELETE from switch where hostname = 'sw8';
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 1 row affected
Time: 0.008s
```

No line with sw8 switch in the table now:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+------------+-------
↪----+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+--------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255  ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255  ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255  ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255  ␣
↪    |
+----------------+----------+------------+--------------------+------------+-------
↪----+
7 rows in set
Time: 0.039s
```

ORDER BY

ORDER BY operator is used to sort the output by a certain field, ascending or descending. To do this it should be added to SELECT operator.

If you perform a simple SELECT query, the output is:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+------------+-------
↪----+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+--------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255  ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255  ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255  ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255  ␣
↪    |
+----------------+----------+------------+--------------------+------------+-------
↪----+
7 rows in set
Time: 0.039s
```

With help of ORDER BY operator you can get entries from *switch* table by sorting them by switch name:

```
new_db.db> SELECT * from switch ORDER BY hostname ASC;
+----------------+----------+------------+--------------------+------------+-------
↪----+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+--------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
```

(continues on next page)

(continued from previous page)

```
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255 ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255 ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255 ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255 ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255 ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255 ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.034s
```

By default, sorting is ascending, so query could be without ASC parameter:

```
new_db.db> SELECT * from switch ORDER BY hostname;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255 ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255 ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255 ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255 ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255 ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255 ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255 ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
```

(continues on next page)

(continued from previous page)

```
Time: 0.034s
```

Sorting by IP address descending:

```
SELECT * from switch ORDER BY mngmt_ip DESC;
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255  ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255  ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255  ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
7 rows in set
Time: 0.034s
```

AND

AND operator allows grouping of several conditions:

```
new_db.db> select * from switch where model = 'Cisco 3850' and mngmt_ip LIKE '10.
↪255.%';
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
```

(continues on next page)

(continued from previous page)

```
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255 ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255 ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255 ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255 ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255 ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
5 rows in set
Time: 0.034s
```

OR

Operator OR:

```
new_db.db> select * from switch where model LIKE '%3750' or model LIKE '%3850';
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255 ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255 ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255 ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255 ␣
↪    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255 ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255 ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
6 rows in set
Time: 0.046s
```

IN

Operator IN:

```
new_db.db> select * from switch where model in ('Cisco 3750', 'C3750');
+----------------+----------+-------+-------------------+------------+-----------+
| mac            | hostname | model | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+-------+-------------------+------------+-----------+
| 0060.A6AA.C4CC | sw6      | C3750 | London, Green Str | 10.255.1.6 | 255       |
+----------------+----------+-------+-------------------+------------+-----------+
1 row in set
Time: 0.034s
```

NOT

Operator NOT:

```
new_db.db> select * from switch where model not in ('Cisco 3750', 'C3750');
+----------------+----------+------------+-------------------+------------+-------
↪----+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_
↪vid |
+----------------+----------+------------+-------------------+------------+-------
↪----+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255  ␣
↪    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255  ␣
↪    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255  ␣
↪    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255  ␣
↪    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255  ␣
↪    |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255  ␣
↪    |
+----------------+----------+------------+-------------------+------------+-------
↪----+
6 rows in set
Time: 0.037s
```

Sqlite3 module

Python uses sqlite3 module to work with SQLite.

`Connection` object - this object can be said to represent a database.

Example of creating a connection:

```
import sqlite3

connection = sqlite3.connect('dhcp_snooping.db')
```

Once you have created a connection you should create a Cursor object which is the main way to work with database.

Cursor is created from DB connection:

```
connection = sqlite3.connect('dhcp_snooping.db')
cursor = connection.cursor()
```

Executing SQL commands

There are several methods for execution of SQL commands in module:

- `execute` - method for executing one SQL expression
- `executemany` - method allows to execute one SQL expression for a sequence of parameters (or for iterator)
- `executescript` - method allows to execute multiple SQL expressions at once

Method execute

Method `execute` allows one SQL command to be executed. First, create connection and cursor:

```
In [1]: import sqlite3

In [2]: connection = sqlite3.connect('sw_inventory.db')

In [3]: cursor = connection.cursor()
```

Creates a switch table using `execute`:

```
In [4]: cursor.execute("create table switch (mac text not NULL primary key,␣
↪hostname text, model text, location text)")
Out[4]: <sqlite3.Cursor at 0x1085be880>
```

SQL expressions can be parameterized - data can be substituted by special values. Due to this you can use the same SQL command to pass different data.

For example, switch table needs to be filled with data from data list:

```
In [5]: data = [
   ...: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
   ...: ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
   ...: ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

You can use this query:

```
In [6]: query = "INSERT into switch values (?, ?, ?, ?)"
```

Question marks in command are used to fill in the data that will be passed to execute.

Data can now be passed as follows:

```
In [7]: for row in data:
   ...:     cursor.execute(query, row)
   ...:
```

The second argument that is passed to execute must be a tuple. If you want to pass a tuple with one element, `(value, )` entry is used.

For changes to be applied, commit must be executed (note that commit method is called at the connection):

```
In [8]: connection.commit()
```

Now, when querying from sqlite3 command line you can see these rows in switch table:

```
$ litecli sw_inventory.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
sw_inventory.db> SELECT * from switch;
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0000.AAAA.CCCC | sw1      | Cisco 3750 | London, Green Str |
| 0000.BBBB.CCCC | sw2      | Cisco 3780 | London, Green Str |
| 0000.AAAA.DDDD | sw3      | Cisco 2960 | London, Green Str |
| 0011.AAAA.CCCC | sw4      | Cisco 3750 | London, Green Str |
+----------------+----------+------------+-------------------+
4 rows in set
```

(continues on next page)

(continued from previous page)

```
Time: 0.039s
sw_inventory.db>
```

Method executemany

Method executemany allows one SQL command to be executed for parameter sequence (or for iteratoAr). Using executemany method you can add a similar data list to switch table by a single command.

For example, you should add data from data2 list to switch table:

```
In [9]: data2 = [
   ...: ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

To do this, use a similar request:

```
In [10]: query = "INSERT into switch values (?, ?, ?, ?)"
```

Now you can pass data to executemany():

```
In [11]: cursor.executemany(query, data2)
Out[11]: <sqlite3.Cursor at 0x10ee5e810>

In [12]: connection.commit()
```

After commit, data is available in the table:

```
$ litecli sw_inventory.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
sw_inventory.db> SELECT * from switch;
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0000.AAAA.CCCC | sw1      | Cisco 3750 | London, Green Str |
| 0000.BBBB.CCCC | sw2      | Cisco 3780 | London, Green Str |
| 0000.AAAA.DDDD | sw3      | Cisco 2960 | London, Green Str |
| 0011.AAAA.CCCC | sw4      | Cisco 3750 | London, Green Str |
| 0000.1111.0001 | sw5      | Cisco 3750 | London, Green Str |
```

(continues on next page)

(continued from previous page)

```
| 0000.1111.0002 | sw6      | Cisco 3750 | London, Green Str |
| 0000.1111.0003 | sw7      | Cisco 3750 | London, Green Str |
| 0000.1111.0004 | sw8      | Cisco 3750 | London, Green Str |
+----------------+----------+------------+-------------------+
8 rows in set
Time: 0.034s
```

Method executemany placed corresponding tuples to SQL command and all data was added to the table.

Method executescript

Method executescript allows multiple SQL expressions to be executed at once.

This method is particularly useful when creating tables:

```
In [13]: connection = sqlite3.connect('new_db.db')

In [14]: cursor = connection.cursor()

In [15]: cursor.executescript('''
    ...:     create table switches(
    ...:         hostname     text not NULL primary key,
    ...:         location     text
    ...:     );
    ...:
    ...:     create table dhcp(
    ...:         mac          text not NULL primary key,
    ...:         ip           text,
    ...:         vlan         text,
    ...:         interface    text,
    ...:         switch       text not null references switches(hostname)
    ...:     );
    ...: ''')
Out[15]: <sqlite3.Cursor at 0x10efd67a0>
```

Fetching query results

There are several ways to get query results in sqlite3:

- using `fetch...` - depending on the method one, more or all rows are returned
- using cursor as an iterator - iterator returns

Method fetchone

Method fetchone returns one data row. Example of fetching information from sw_inventory.db database:

```
In [16]: import sqlite3

In [17]: connection = sqlite3.connect('sw_inventory.db')

In [18]: cursor = connection.cursor()

In [19]: cursor.execute('select * from switch')
Out[19]: <sqlite3.Cursor at 0x104eda810>

In [20]: cursor.fetchone()
Out[20]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
```

Note that while the SQL query requests all table content, fetchone returned only one row. If you re-call method, it returns the next row:

```
In [21]: print(cursor.fetchone())
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
```

Similarly, method will return the next rows. After processing all rows, method starts returning None.

In this way, method can be used in the loop, for example:

```
In [22]: cursor.execute('select * from switch')
Out[22]: <sqlite3.Cursor at 0x104eda810>

In [23]: while True:
   ...:     next_row = cursor.fetchone()
   ...:     if next_row:
   ...:         print(next_row)
   ...:     else:
   ...:         break
   ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

Method fetchmany

Method `fetchmany` returns a list of data rows.

Method syntax:

```
cursor.fetchmany([size=cursor.arraysize])
```

Size parameter allows you to specify how many rows are returned. By default the size parameter is `cursor.arraysize`:

```
In [24]: print(cursor.arraysize)
1
```

For example, you can return three rows at a time from query:

```
In [25]: cursor.execute('select * from switch')
Out[25]: <sqlite3.Cursor at 0x104eda810>

In [26]: from pprint import pprint

In [27]: while True:
    ...:     three_rows = cursor.fetchmany(3)
    ...:     if three_rows:
    ...:         pprint(three_rows)
    ...:     else:
    ...:         break
    ...:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')]
[('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')]
[('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Method displays required number of rows and if amount of rows are less than the size parameter, it returns remaining rows.

Method fetchall

Method `fetchall` returns all rows as a list:

```
In [28]: cursor.execute('select * from switch')
Out[28]: <sqlite3.Cursor at 0x104eda810>

In [29]: cursor.fetchall()
Out[29]:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

An important aspect of method - it returns all remaining rows.

That is, if `fetchone` method was used before `fetchall`, then `fetchall` would return remaining query rows:

```
In [30]: cursor.execute('select * from switch')
Out[30]: <sqlite3.Cursor at 0x104eda810>

In [31]: cursor.fetchone()
Out[31]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')

In [32]: cursor.fetchone()
Out[32]: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')

In [33]: cursor.fetchall()
Out[33]:
[('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Method `fetchmany` works similarly in this aspect.

Cursor as iterator

If you want to process the resulting strings, use cursor as an iterator. It is not necessary to use fetch methods.

If you use execute methods, the cursor is returned. Since cursor can be used as an iterator you can use it, for example, in `for` loop:

```
In [34]: result = cursor.execute('select * from switch')

In [35]: for row in result:
    ...:     print(row)
    ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

The same option will work without assigning a variable:

```
In [36]: for row in cursor.execute('select * from switch'):
    ...:     print(row)
    ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

Using sqlite3 module without explicit cursor creation

Execute methods are available in Connection object and in Cursor object but `fetch` methods are only available in Cursor object.

When using execute methods with Connection object, cursor is returned as a result of execute method. It can be used as an iterator and receive data without `fetch` methods. This allows you not to create cursor when working with sqlite3 module.

Example of the resulting script (create_sw_inventory_ver1.py):

```
import sqlite3
```

(continues on next page)

(continued from previous page)

```
data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


con = sqlite3.connect('sw_inventory2.db')

con.execute('''create table switch
            (mac text not NULL primary key, hostname text, model text, location␣
↪text)'''
            )

query = 'INSERT into switch values (?, ?, ?, ?)'
con.executemany(query, data)
con.commit()

for row in con.execute('select * from switch'):
    print(row)

con.close()
```

The result will be:

```
$ python create_sw_inventory_ver1.py
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
```

Exception Handling

Let's see an example of how to use execute method when an error occurs.

In switch table the mac field must be unique. If you try to write an overlapping MAC address, there is an error:

```
In [37]: con = sqlite3.connect('sw_inventory2.db')

In [38]: query = "INSERT into switch values ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960
↪', 'London, Green Str')"
```

(continues on next page)

(continued from previous page)

```
In [39]: con.execute(query)
---------------------------------------------------------------
IntegrityError              Traceback (most recent call last)
<ipython-input-56-ad34d83a8a84> in <module>()
----> 1 con.execute(query)

IntegrityError: UNIQUE constraint failed: switch.mac
```

Accordingly, you can catch the exception:

```
In [40]: try:
    ...:     con.execute(query)
    ...: except sqlite3.IntegrityError as e:
    ...:     print("Error occurred: ", e)
    ...:
Error occurred:  UNIQUE constraint failed: switch.mac
```

Note that you should catch `sqlite3.IntegrityError` exception, not `IntegrityError`.

Connection as context manager

After operations are completed the changes must be saved (apply `commit`), and then you can close connection if it is no longer needed.

Python allows you to use Connection object as a context manager. In that case, you don't have to explicitly commit.

At the same time:

- If an exception occurs the transaction automatically rolls back
- if no exception, commit applies automatically

Example of using a database connection as a context manager (create_sw_inventory_ver2.py):

```
import sqlite3


data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


con = sqlite3.connect('sw_inventory3.db')
```

(continues on next page)

(continued from previous page)

```
con.execute('''create table switch
               (mac text not NULL primary key, hostname text, model text,␣
↪location text)'''
            )

try:
    with con:
        query = 'INSERT into switch values (?, ?, ?, ?)'
        con.executemany(query, data)

except sqlite3.IntegrityError as e:
    print('Error occured: ', e)

for row in con.execute('select * from switch'):
    print(row)

con.close()
```

Note that although a transaction will be rolled back when an exception occurs, the exception itself must still be intercepted.

To check this functionality you should write to the table the data in which MAC address is repeated. But before, in order to not repeat parts of the code, it is better to split the code by functions in create_sw_inventory_ver2.py file:

```
from pprint import pprint
import sqlite3


data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


def create_connection(db_name):
    '''
    Function creates a connection to db_name database and returns it
    '''
    connection = sqlite3.connect(db_name)
    return connection
```

(continues on next page)

(continued from previous page)

```
def write_data_to_db(connection, query, data):
    '''
    Function awaits arguments:
     * connection - connection to DB
     * query - query to execute
     * data - data to be passed as a list of tuples

    Function attempts to write all data from *data* list.
    If data is saved successfully, changes are saved to database and returns True.

    If an error occurs during the writing process, transaction rolls back and␣
↪function returns False.

    '''
    try:
        with connection:
            connection.executemany(query, data)
    except sqlite3.IntegrityError as e:
        print('Error occured: ', e)
        return False
    else:
        print('Data writing was successful')
        return True


def get_all_from_db(connection, query):
    '''
    Function awaits arguments:
     * connection - connection to DB
     * query - query to execute

    Function returns data from database
    '''
    result = [row for row in connection.execute(query)]
    return result


if __name__ == '__main__':
    con = create_connection('sw_inventory3.db')

    print('DB creation...')
    schema = '''create table switch
                (mac text primary key, hostname text, model text, location text)''
↪'
```

(continues on next page)

(continued from previous page)

```
    con.execute(schema)

    query_insert = 'INSERT into switch values (?, ?, ?, ?)'
    query_get_all = 'SELECT * from switch'

    print('Write data to DB:')
    pprint(data)
    write_data_to_db(con, query_insert, data)
    print('\nChecking DB content')
    pprint(get_all_from_db(con, query_get_all))

    con.close()
```

The result of script execution is:

```
$ python create_sw_inventory_ver2_functions.py
Table creation...
Data writing to DB:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
Data writing was successful

Checking DB content
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

Now let's check how `write_data_to_db` function will work when there are identical MAC addresses in the data.

File create_sw_inventory_ver3.py uses functions from create_sw_inventory_ver2_functions.py file and implies that the script will run after the previous data is written:

```
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC address of sw7 matches sw3 switch MAC in *data* list
data2 = [('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
         ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
         ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960',
```

(continues on next page)

(continued from previous page)

```
          'London, Green Str'), ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750',
                                 'London, Green Str')]

con = dbf.create_connection('sw_inventory3.db')

query_insert = "INSERT into switch values (?, ?, ?, ?)"
query_get_all = "SELECT * from switch"

print("\nChecking current content of DB")
pprint(dbf.get_all_from_db(con, query_get_all))

print('-' * 60)
print("Trying to write data with a duplicate MAC address:")
pprint(data2)
dbf.write_data_to_db(con, query_insert, data2)
print("\nChecking DB content")
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

In data2 list, sw7 switch has the same MAC address as sw3 switch already existing in database.

Result of script execution:

```
$ python create_sw_inventory_ver3.py

Cheking current DB content
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
------------------------------------------------------------
Attempt to write data with repeating MAC address:
[('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
Error occurred:  UNIQUE constraint failed: switch.mac

Cheking DB content
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

Note that the content of switch table before and after adding of information is the same. This means that no line from data2 list has been written. This is because executemany method is used and within the same transaction we try to write all four lines. If an error occurs with one of them, all changes are reversed.

Sometimes it's exactly the kind of behavior you need. If you want to ignore only row with errors you should use execute method and write each row separately.

File create_sw_inventory_ver4.py has write_rows_to_db function which writes data in turn and if there is an error, only changes for specific data are rolled back:

```
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC address of sw7 matches sw3 switch MAC in *data* list
data2 = [('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
         ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
         ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
         ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]


def write_rows_to_db(connection, query, data, verbose=False):
    '''
    Function awaits arguments:
     * connection - connection to DB
     * query - query to execute
     * data - data to be passed as a list of tuples

    Function attempts to write a tuples in turn from *data* list.
    If tuple can be written successfully, changes are saved to database.
    If an error occurs while writing the tuple, transaction rolls back.


    Flag *verbose* controls whether messages about successful or unsuccessful␣
↪tuple
    writing attempt.

    '''
    for row in data:
        try:
            with connection:
                connection.execute(query, row)
        except sqlite3.IntegrityError as e:
            if verbose:
```

(continues on next page)

(continued from previous page)

```
                print("Error occurred while writing data '{}'".format(', '.
↪join(row), e))
        else:
            if verbose:
                print("Data writing  was successful '{}'".format(
                    ', '.join(row)))


con = dbf.create_connection('sw_inventory3.db')

query_insert = 'INSERT into switch values (?, ?, ?, ?)'
query_get_all = 'SELECT * from switch'

print('\nChecking current content of DB')
pprint(dbf.get_all_from_db(con, query_get_all))

print('-' * 60)
print('Trying to write data with a duplicate MAC address:')
pprint(data2)
write_rows_to_db(con, query_insert, data2, verbose=True)
print('\nChecking DB content')
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

The execution result is (missing only sw7):

```
$ python create_sw_inventory_ver4.py

Cheking current DB content
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
------------------------------------------------------------
Attempt to write data with repeating MAC address:
[('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
Data "0055.AAAA.CCCC, sw5, Cisco 3750, London, Green Str" writing was successful
Data "0066.BBBB.CCCC, sw6, Cisco 3780, London, Green Str" writing was successful
While writing data "0000.AAAA.DDDD, sw7, Cisco 2960, London, Green Str" the error␣
↪occured
```

(continues on next page)

(continued from previous page)

```
Data "0088.AAAA.CCCC, sw8, Cisco 3750, London, Green Str" writing was successful

Cheking DB content
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

SQLite use example

In section 15 there was an example of reviewing the output of command “show ip dhcp snooping binding”. In the output we received information about parameters of connected devices (interface, IP, MAC, VLAN).

In this version you can only see all devices connected to switch. If you want to find out others based on one of the parameters, it’s not convenient in this way.

For example, if you want to get information based on IP address about to which interface the host is connected, which MAC address it has and in which VLAN it is, then script is not very simple and more importantly, not convenient.

Let’s write information obtained from the output “sh ip dhcp snooping binding” to SQLite. This will allow do queries based on any parameter and get missing ones. For this example, it is sufficient to create a single table where information will be stored.

Table is defined in a separate dhcp_snooping_schema.sql file:

```
create table if not exists dhcp (
    mac          text not NULL primary key,
    ip           text,
    vlan         text,
    interface    text
);
```

For all fields the data type is “text”.

MAC address is the primary key of our table which is logical because MAC address must be unique.

Additionally, by using expression `create table if not exists` - SQLite will only create a table if it does not exist.

Now you have to create a database file, connect to database and create a table (create_sqlite_ver1.py file):

```
import sqlite3

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print("Done")

conn.close()
```

Comments to file:

- during execution of `conn = sqlite3.connect('dhcp_snooping.db')`:
  - file dhcp_snooping.db is created if it does not exist
  - Connection object is created
- table is created in database (if it does not exist) based on commands specified in dhcp_snooping_schema.sql file:
  - dhcp_snooping_schema.sql file opens
  - `schema = f.read()` - whole file is read in one string
  - `conn.executescript(schema)` - executescript() method allows SQL to execute commands that are written in the file

Execution of script:

```
$ python create_sqlite_ver1.py
Creating schema...
Done
```

The result should be a database file and a dhcp table.

You can check that table has been created with sqlite3 utility which allows you to execute queries directly in command line.

List of tables created is shown as follows:

```
$ sqlite3 dhcp_snooping.db "SELECT name FROM sqlite_master WHERE type='table'"
dhcp
```

Now it is necessary to write information from the output of “sh ip dhcp snooping binding” command to the table (dhcp_snooping.txt file):

```
MacAddress           IpAddress        Lease(sec)  Type           VLAN  Interface
------------------   ---------------  ----------  -------------  ----  ------------
↪--------
00:09:BB:3D:D6:58    10.1.10.2        86250       dhcp-snooping   10   ␣
↪FastEthernet0/1
00:04:A3:3E:5B:69    10.1.5.2         63951       dhcp-snooping   5    ␣
↪FastEthernet0/10
00:05:B3:7E:9B:60    10.1.5.4         63253       dhcp-snooping   5    ␣
↪FastEthernet0/9
00:09:BC:3F:A6:50    10.1.10.6        76260       dhcp-snooping   10   ␣
↪FastEthernet0/3
Total number of bindings: 4
```

In the second version of the script, the output in dhcp_snooping.txt file is processed with regular expressions and then entries are added to database (create_sqlite_ver2.py file):

```
import sqlite3
import re

regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print('Done')

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = '''insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)'''
```

(continues on next page)

(continued from previous page)

```
            conn.execute(query, row)
    except sqlite3.IntegrityError as e:
        print('Error occured: ', e)

conn.close()
```

Note: For now, you should delete database file every time because script tries to create it every time you start.

Comments to the script:

- in regular expression that processes the output of "sh ip dhcp snooping binding", numbered groups are used instead of named groups as it was in example of section **'https://pyneng.readthedocs.io/en/latest/book/14_regex/4a_group_example.html>'__**
  - groups were created only for those elements we are interested in
- result - a list that stores the result of processing the command output
  - but now there is no dictionaries but tuples with results
  - this is necessary to enable them to be immediately written to database
- Scroll elements in received list of tuples
- This script uses another version of database entry
  - `query` string containes a query. But instead of values, question marks are given. This query type allows dynamicly substite field values.
  - then `execute` method is passed a query string and `row` tuple where values are

Execute the script:

```
$ python create_sqlite_ver2.py
Creating schema...
Done
Inserting DHCP Snooping data
```

Let's check if data has been written:

```
$ sqlite3 dhcp_snooping.db "select * from dhcp"
-- Loading resources from /home/vagrant/.sqliterc

mac                ip          vlan        interface
-----------------  ----------  ----------  ---------------
00:09:BB:3D:D6:58  10.1.10.2   10          FastEthernet0/1
```

(continues on next page)

(continued from previous page)

```
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/1
00:05:B3:7E:9B:60  10.1.5.4    5           FastEthernet0/9
00:09:BC:3F:A6:50  10.1.10.6   10          FastEthernet0/3
```

Now let's try to ask by a certain parameter:

```
$ sqlite3 dhcp_snooping.db "select * from dhcp where ip = '10.1.5.2'"
-- Loading resources from /home/vagrant/.sqliterc

mac                ip          vlan        interface
-----------------  ----------  ----------  ----------------
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/10
```

That is, it is now possible to get others parameters based on one parameter.

Let's modify the script to make it check for the presence of dhcp_snooping.db. If you have a database file you don't need to create a table, we believe it has already been created.

File create_sqlite_ver3.py:

```
import os
import sqlite3
import re

data_filename = 'dhcp_snooping.txt'
db_filename = 'dhcp_snooping.db'
schema_filename = 'dhcp_snooping_schema.sql'

regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

db_exists = os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if not db_exists:
    print('Creating schema...')
```

(continues on next page)

(continued from previous page)

```
    with open(schema_filename, 'r') as f:
        schema = f.read()
    conn.executescript(schema)
    print('Done')
else:
    print('Database exists, assume dhcp table does, too.')

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = '''insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)'''
            conn.execute(query, row)
    except sqlite3.IntegrityError as e:
        print('Error occured: ', e)

conn.close()
```

Now there is a verification of the presence of database file and dhcp_snooping.db file will only be created if it does not exist. Data is also written only if dhcp_snooping.db file is not created.

Note: Separating the process of creating a table and completing it with the data is specified in tasks to the section.

If no file (delete it first):

```
$ rm dhcp_snooping.db
$ python create_sqlite_ver3.py
Creating schema...
Done
Inserting DHCP Snooping data
```

Let's check. In case the file already exists but the data is not written:

```
$ rm dhcp_snooping.db

$ python create_sqlite_ver1.py
Creating schema...
Done
$ python create_sqlite_ver3.py
```

(continues on next page)

(continued from previous page)

```
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
```

If both DB and data are exist:

```
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
Error occurred:  UNIQUE constraint failed: dhcp.mac
Error occurred:  UNIQUE constraint failed: dhcp.mac
Error occurred:  UNIQUE constraint failed: dhcp.mac
Error occurred:  UNIQUE constraint failed: dhcp.mac
```

Now we make a separate script that sends queries to database and displays results. It should:

- expect parameters from user:
  - parameter name
  - parameter value
- provide normal output on request

File get_data_ver1.py:

```
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

key, value = sys.argv[1:]
keys = ['mac', 'ip', 'vlan', 'interface']
keys.remove(key)

conn = sqlite3.connect(db_filename)

#Allows to further access data in columns by column name
conn.row_factory = sqlite3.Row

print('\nDetailed information for host(s) with', key, value)
print('-' * 40)

query = 'select * from dhcp where {} = ?'.format(key)
result = conn.execute(query, (value, ))

for row in result:
```

(continues on next page)

(continued from previous page)

```
    for k in keys:
        print('{:12}: {}'.format(k, row[k]))
    print('-' * 40)
```

Comments to the script:

- key, value are read from arguments that passed to script
  - selected key is removed from keys list. Thus, only parameters that you want to display are left in the list
- connecting to DB
  - `conn.row_factory = sqlite3.Row` - allows further access data in column based on column names
- Select rows from database where key is equal to specified value
  - in SQL the values can be set by a question mark but you cannot give a column name. Therefore, the column name is substituted by row formatting and the value by SQL tool.
  - Pay attention to `(value,)` - tuple with one element is passed
- The resulting information is displayed to standard output stream:
  - iterate over the results obtained and show only those fields that are in keys list

Let's check the script.

Show host parameters with IP 10.1.10.2:

```
$ python get_data_ver1.py ip 10.1.10.2

Detailed information for host(s) with ip 10.1.10.2
----------------------------------------
mac         : 00:09:BB:3D:D6:58
vlan        : 10
interface   : FastEthernet0/1
----------------------------------------
```

Show hosts in VLAN 10:

```
$ python get_data_ver1.py vlan 10

Detailed information for host(s) with vlan 10
----------------------------------------
mac         : 00:09:BB:3D:D6:58
ip          : 10.1.10.2
interface   : FastEthernet0/1
```

(continues on next page)

(continued from previous page)

```
----------------------------------------
mac         : 00:07:BC:3F:A6:50
ip          : 10.1.10.6
interface   : FastEthernet0/3
----------------------------------------
```

The second version of the script to get data with minor improvements:

- Instead of rows formatting, a dictionary that contains queries corresponding to each key is used.
- Checking the key that was selected
- Method keys is used to get all columns that match the query

File get_data_ver2.py:

```
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

query_dict = {
    'vlan': 'select mac, ip, interface from dhcp where vlan = ?',
    'mac': 'select vlan, ip, interface from dhcp where mac = ?',
    'ip': 'select vlan, mac, interface from dhcp where ip = ?',
    'interface': 'select vlan, mac, ip from dhcp where interface = ?'
}

key, value = sys.argv[1:]
keys = query_dict.keys()

if not key in keys:
    print('Enter key from {}'.format(', '.join(keys)))
else:
    conn = sqlite3.connect(db_filename)
    conn.row_factory = sqlite3.Row

    print('\nDetailed information for host(s) with', key, value)
    print('-' * 40)

    query = query_dict[key]
    result = conn.execute(query, (value, ))

    for row in result:
```

(continues on next page)

(continued from previous page)

```
        for row_name in row.keys():
            print('{:12}: {}'.format(row_name, row[row_name]))
        print('-' * 40)
```

There are several drawbacks to this script:

- does not check number of arguments that are passed to the script
- It would be good to collect information from different switches. To do this, you should add a field that indicates on which switch the entry was found

In addition, a lot of work needs to be done in the script that creates database and writes the data.

All improvements will be done in tasks of this section.

Further reading

Documentation:

- SQLite Tutorial - SQLite detailed description
- Module documentation sqlite3
- sqlite3 на сайте PyMOTW

Articles:

- A thorough guide to SQLite database operations in Python

Tasks

All tasks and additional files can be downloaded from repository.

Warning: Starting from section "4. Data types in Python" there are automated tests for testing tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests. Testing is done using the pyneng utility. *Learn more about how to work with the pyneng utility*.

Task 25.1

There are no tests for the tasks of the 25th chapter!

You need to create two scripts:

1. create_db.py
2. add_data.py

The code in scripts should be broken down into functions. It is up to you to decide which functions and how to split the code. Some of the code can be global.

create_db.py - this script should contain the functionality for creating a database:

- check for the presence of a database file
- if the file does not exist, according to the description of the database schema in the dhcp_snooping_schema.sql file, the database must be created
- the name of the database file is dhcp_snooping.db

The database should contain two tables (the schema is described in the dhcp_snooping_schema.sql file):

- switches - it contains data about switches
- dhcp - information obtained from the output of sh ip dhcp snooping binding is stored here

An example of script execution when there is no dhcp_snooping.db file:

```
$ python create_db.py
Database creation...
```

After creating the file:

```
$ python create_db.py
Database exists
```

add_data.py script adds data to the database. This script should add data from sh ip dhcp snooping binding output and switch information

Accordingly, there should be two parts in the add_data.py file:

- information about switches is added to the switches table. Switch data are in the switches.yml file
- information based on the output of sh ip dhcp snooping binding is added to the dhcp table
  - output from three switches in files: files sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt
  - since the dhcp table has changed, and now there is a switch field, it must also be filled. The switch name is derived from the data file name

An example of script execution when the database has not yet been created:

```
$ python add_data.py
The database does not exist. Before adding data, you need to create it
```

An example of executing the script for the first time after creating the database:

```
$ python add_data.py
Adding data to the switches table...
Add data to dhcp table...
```

An example of script execution after the data has been added to the table (the order of adding data can be arbitrary, but messages must output similarly to the output below):

```
$ python add_data.py
Adding data to the switches table...
While adding data: ('sw1', 'London, 21 New Globe Walk') An error occurred: UNIQUE␣
↪constraint failed: switches.hostname
While adding data: ('sw2', 'London, 21 New Globe Walk') An error occurred: UNIQUE␣
↪constraint failed: switches.hostname
While adding data: ('sw3', 'London, 21 New Globe Walk') An error occurred: UNIQUE␣
↪constraint failed: switches.hostname
Adding data to the dhcp table...
While adding data: ('00:09:BB:3D:D6:58', '10.1.10.2', '10', 'FastEthernet0/1',
↪'sw1') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:04:A3:3E:5B:69', '10.1.5.2', '5', 'FastEthernet0/10', 'sw1
↪') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:05:B3:7E:9B:60', '10.1.5.4', '5', 'FastEthernet0/9', 'sw1
↪') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:07:BC:3F:A6:50', '10.1.10.6', '10', 'FastEthernet0/3',
↪'sw1') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7
↪', 'sw1') An error occurred: UNIQUE constraint failed: dhcp.mac
```

(continues on next page)

(continued from previous page)

```
While adding data: ('00:E9:BC:3F:A6:50', '100.1.1.6', '3', 'FastEthernet0/20',
↪'sw3') An error occurred: UNIQUE constraint failed: dhcp.mac
...
```

At this stage, both scripts are called with no arguments.

The code in scripts should be broken down into functions. It is up to you to decide which functions and how to split the code. Some of the code can be global.

Task 25.2

There are no tests for the tasks of the 25th chapter!

In this task, you need to create the get_data.py script.

The code in the script should be broken down into functions. It is up to you to decide which functions and how to split the code. Some of the code can be global.

The script can be passed arguments and, depending on the arguments, you need to display different information. If the script is called:

- with no arguments, print the entire contents of the dhcp table
- with two arguments, display information from the dhcp table that matches the field and value
- with any other number of arguments, print a message that the script only supports two or zero arguments

The database file can be copied from task 25.1.

Examples of output for different numbers and values of arguments:

```
$ python get_data.py
The dhcp table has the following entries:
-----------------  ---------------  --  ----------------  ---
00:09:BB:3D:D6:58  10.1.10.2        10  FastEthernet0/1   sw1
00:04:A3:3E:5B:69  10.1.5.2          5  FastEthernet0/10  sw1
00:05:B3:7E:9B:60  10.1.5.4          5  FastEthernet0/9   sw1
00:07:BC:3F:A6:50  10.1.10.6        10  FastEthernet0/3   sw1
00:09:BC:3F:A6:50  192.168.100.100   1  FastEthernet0/7   sw1
00:E9:BC:3F:A6:50  100.1.1.6         3  FastEthernet0/20  sw3
00:E9:22:11:A6:50  100.1.1.7         3  FastEthernet0/21  sw3
00:A9:BB:3D:D6:58  10.1.10.20       10  FastEthernet0/7   sw2
00:B4:A3:3E:5B:69  10.1.5.20         5  FastEthernet0/5   sw2
00:C5:B3:7E:9B:60  10.1.5.40         5  FastEthernet0/9   sw2
00:A9:BC:3F:A6:50  10.1.10.60       20  FastEthernet0/2   sw2
-----------------  ---------------  --  ----------------  ---
```

(continues on next page)

(continued from previous page)

```
$ python get_data.py vlan 10

Information about devices with the following parameters: vlan 10
-----------------  ----------  --  ---------------  ---
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1  sw1
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/3  sw1
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7  sw2
-----------------  ----------  --  ---------------  ---

$ python get_data.py ip 10.1.10.2

Information about devices with the following parameters: ip 10.1.10.2
-----------------  ---------  --  ---------------  ---
00:09:BB:3D:D6:58  10.1.10.2  10  FastEthernet0/1  sw1
-----------------  ---------  --  ---------------  ---

$ python get_data.py vln 10
This parameter is not supported.
Valid parameter values: mac, ip, vlan, interface, switch

$ python get_data.py ip vlan 10
Please enter two or zero arguments
```

Task 25.3

There are no tests for the tasks of the 25th chapter!

In previous tasks, information was added to an empty database. In this task, the script should work correctly even in a situation where the database already contains information.

Copy the add_data.py script from task 25.1 and try running it again on the existing database. The output should be like this:

```
$ python add_data.py
Adding data to the switches table...
While adding data: ('sw1', 'London, 21 New Globe Walk') An error occurred: UNIQUE␣
↪constraint failed: switches.hostname
While adding data: ('sw2', 'London, 21 New Globe Walk') An error occurred: UNIQUE␣
↪constraint failed: switches.hostname
While adding data: ('sw3', 'London, 21 New Globe Walk') An error occurred: UNIQUE␣
↪constraint failed: switches.hostname
Adding data to the dhcp table...
```

(continues on next page)

(continued from previous page)

```
While adding data: ('00:09:BB:3D:D6:58', '10.1.10.2', '10', 'FastEthernet0/1',
↪'sw1') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:04:A3:3E:5B:69', '10.1.5.2', '5', 'FastEthernet0/10', 'sw1
↪') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:05:B3:7E:9B:60', '10.1.5.4', '5', 'FastEthernet0/9', 'sw1
↪') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:07:BC:3F:A6:50', '10.1.10.6', '10', 'FastEthernet0/3',
↪'sw1') An error occurred: UNIQUE constraint failed: dhcp.mac
While adding data: ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7
↪', 'sw1') An error occurred: UNIQUE constraint failed: dhcp.mac
... (the command output is abbreviated)
```

When creating the database schema, it was explicitly stated that the MAC address field must be unique. Therefore, when adding an entry with the same MAC address, an exception (error) is raised. Task 25.1 handles the exception and writes a message to stdout.

In this task, it is assumed that information is periodically read from the switches and written to files. After that, the information from the files must be transferred to the database. At the same time, there may be changes in the new data: MAC disappeared, MAC switched to another port/vlan, a new MAC appeared, etc.

In this task, in the dhcp table, you need to create a new field, active, which will indicate whether the record is up-to-date. The new database schema is located in the dhcp_snooping_schema.sql file.

The active field can have the following values:

- 0 - means False. Used to mark an entry as inactive
- 1 - True. Used to indicate that a record is active

Every time information from files with DHCP snooping output is added again, all existing entries (for this switch) must be marked as inactive (active = 0). You can then update the information and mark the new records as active (active = 1).

Thus, the old records will remain in the database for MAC addresses that are currently inactive, and updated information for active addresses will appear.

For example, the dhcp table contains the following entries:

```
mac               ip          vlan        interface         switch      active
----------------  ----------  ----------  ----------------  ----------  ---------
↪-
00:09:BB:3D:D6:58  10.1.10.2   10          FastEthernet0/1   sw1         1
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/10  sw1         1
00:05:B3:7E:9B:60  10.1.5.4    5           FastEthernet0/9   sw1         1
00:07:BC:3F:A6:50  10.1.10.6   10          FastEthernet0/3   sw1         1
00:09:BC:3F:A6:50  192.168.10  1           FastEthernet0/7   sw1         1
```

And you need to add the following information from the file:

```
MacAddress           IpAddress        Lease(sec)  Type            VLAN  Interface
------------------   ---------------  ----------  -------------   ----  ------------
↪--------
00:09:BB:3D:D6:58    10.1.10.2        86250       dhcp-snooping   10    ␣
↪FastEthernet0/1
00:04:A3:3E:5B:69    10.1.15.2        63951       dhcp-snooping   15    ␣
↪FastEthernet0/15
00:05:B3:7E:9B:60    10.1.5.4         63253       dhcp-snooping   5     ␣
↪FastEthernet0/9
00:07:BC:3F:A6:50    10.1.10.6        76260       dhcp-snooping   10    ␣
↪FastEthernet0/5
```

After adding the data, the table should look like this:

```
mac                ip               vlan        interface          switch      ␣
↪active
-----------------  ---------------  ----------  ---------------    ----------  ----
↪------
00:09:BC:3F:A6:50  192.168.100.100  1           FastEthernet0/7    sw1          0
00:09:BB:3D:D6:58  10.1.10.2        10          FastEthernet0/1    sw1          1
00:04:A3:3E:5B:69  10.1.15.2        15          FastEthernet0/15   sw1          1
00:05:B3:7E:9B:60  10.1.5.4         5           FastEthernet0/9    sw1          1
00:07:BC:3F:A6:50  10.1.10.6        10          FastEthernet0/5    sw1          1
```

The new information should overwrite the previous one:

- MAC 00:04:A3:3E:5B:69 went to a different port and got into a different interface and got a different IP address
- MAC 00:07:BC:3F:A6:50 switched to another port

If some MAC address is not in the new file, it must be left in the database with the value active = 0: MAC addresses 00:09:BC:3F:A6:50 not in new information (turned off the computer)

Modify the add_data.py script so that the new conditions are met and the active field is populated.

The code in the script should be broken down into functions. It is up to you to decide which functions and how to split the code. Some of the code can be global.

To check task and operation of a new field, first add information to the database from files sw*_dhcp_snooping.txt, and then add information from files new_data/sw*_dhcp_snooping.txt.

The data should look like this (the lines can be in any order)

```
-----------------  ---------------  --  ----------------  ---  -
00:09:BC:3F:A6:50  192.168.100.100   1  FastEthernet0/7   sw1  0
```

(continues on next page)

(continued from previous page)

```
00:C5:B3:7E:9B:60  10.1.5.40          5  FastEthernet0/9   sw2  0
00:09:BB:3D:D6:58  10.1.10.2         10  FastEthernet0/1   sw1  1
00:04:A3:3E:5B:69  10.1.15.2         15  FastEthernet0/15  sw1  1
00:05:B3:7E:9B:60  10.1.5.4           5  FastEthernet0/9   sw1  1
00:07:BC:3F:A6:50  10.1.10.6         10  FastEthernet0/5   sw1  1
00:E9:BC:3F:A6:50  100.1.1.6          3  FastEthernet0/20  sw3  1
00:E9:22:11:A6:50  100.1.1.7          3  FastEthernet0/21  sw3  1
00:A9:BB:3D:D6:58  10.1.10.20        10  FastEthernet0/7   sw2  1
00:B4:A3:3E:5B:69  10.1.5.20          5  FastEthernet0/5   sw2  1
00:A9:BC:3F:A6:50  10.1.10.65        20  FastEthernet0/2   sw2  1
00:A9:33:44:A6:50  10.1.10.77        10  FastEthernet0/4   sw2  1
-----------------  ---------------  --  ----------------  ---  -
```

Task 25.4

There are no tests for the tasks of the 25th chapter!

Copy file get_data.py from task 25.2. Add to the script support for the active column, which we added in task 25.3.

Now, when requesting information, active records should be displayed first, and then, inactive ones. If there are no inactive records, do not display the “Inactive records” header.

Examples of script execution:

```
$ python get_data.py
The dhcp table has the following entries:

Active entries:
-----------------  ----------  --  ----------------  ---  -
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1   sw1  1
00:04:A3:3E:5B:69  10.1.15.2   15  FastEthernet0/15  sw1  1
00:05:B3:7E:9B:60  10.1.5.4     5  FastEthernet0/9   sw1  1
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/5   sw1  1
00:E9:BC:3F:A6:50  100.1.1.6    3  FastEthernet0/20  sw3  1
00:E9:22:11:A6:50  100.1.1.7    3  FastEthernet0/21  sw3  1
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7   sw2  1
00:B4:A3:3E:5B:69  10.1.5.20    5  FastEthernet0/5   sw2  1
00:A9:BC:3F:A6:50  10.1.10.65  20  FastEthernet0/2   sw2  1
00:A9:33:44:A6:50  10.1.10.77  10  FastEthernet0/4   sw2  1
-----------------  ----------  --  ----------------  ---  -

Inactive entries:
```

(continues on next page)

(continued from previous page)

```
----------------  ---------------  -  ---------------  ---  -
00:09:BC:3F:A6:50  192.168.100.100  1  FastEthernet0/7  sw1  0
00:C5:B3:7E:9B:60  10.1.5.40        5  FastEthernet0/9  sw2  0
----------------  ---------------  -  ---------------  ---  -

$ python get_data.py vlan 5

Information about devices with the following parameters: vlan 5

Active entries:
----------------  ---------  -  ---------------  ---  -
00:05:B3:7E:9B:60  10.1.5.4   5  FastEthernet0/9  sw1  1
00:B4:A3:3E:5B:69  10.1.5.20  5  FastEthernet0/5  sw2  1
----------------  ---------  -  ---------------  ---  -

Inactive entries:
----------------  ---------  -  ---------------  ---  -
00:C5:B3:7E:9B:60  10.1.5.40  5  FastEthernet0/9  sw2  0
----------------  ---------  -  ---------------  ---  -


$ python get_data.py vlan 10

Information about devices with the following parameters: vlan 10

Active entries:
----------------  ----------  --  ---------------  ---  -
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1  sw1  1
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/5  sw1  1
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7  sw2  1
00:A9:33:44:A6:50  10.1.10.77  10  FastEthernet0/4  sw2  1
----------------  ----------  --  ---------------  ---  -
```

Task 25.5

There are no tests for the tasks of the 25th chapter!

After completing tasks 25.1 - 25.5, information about inactive records remains in the database. And, if some MAC address did not appear in new records, the record with it may remain in the database forever.

And while it can be useful to see where the MAC address was last located, it is not very useful to keep this information permanently. Instead, you can delete a record if, for example, it has been

stored in the database for more than a month.

In order to be able to delete records by date, you need to enter a new field in which the last time the record was added will be recorded.

The new field is called last_active and must contain a string in the format: YYYY-MM-DD HH:MM:SS.

In this task you need to:

- change the dhcp table accordingly and add a new field. The table can be changed from cli sqlite, but the dhcp_snooping_schema.sql file must also be changed
- modify the add_data.py script to add time to each entry

You can get a string with time and date in the specified format using the datetime function in an SQL query. The syntax is as follows:

```
sqlite> insert into dhcp (mac, ip, vlan, interface, switch, active, last_active)
   ...> values ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7',
↪'sw1', '0', datetime('now'));
```

That is, instead of the value that is written to the database, you must specify datetime('now').

After this command, the following entry will appear in the database:

```
mac                ip              vlan  interface        switch  active  last_
↪active
-----------------  --------------  ----  ---------------  ------  ------  -------
↪------------
00:09:BC:3F:A6:50  192.168.100.100  1     FastEthernet0/7  sw1     0       2021-
↪03-09 07:46:31
```

Task 25.5a

There are no tests for the tasks of the 25th chapter!

After completing task 25.5, the dhcp table has a new last_active field.

Update the add_data.py script so that it removes all records that were active more than 7 days ago. In order to get such records, you can manually update the last_active field in some records and set the time to 7 or more days.

The task file shows an example of working with objects of the datetime module. Shows how to get the date 7 days ago. It will be necessary to compare the last_active time with this date.

Please note that date strings that are written to the database can be compared with each other.

```
from datetime import timedelta, datetime

```

(continues on next page)

(continued from previous page)

```
now = datetime.today().replace(microsecond=0)
week_ago = now - timedelta(days=7)

#print(now)
#print(week_ago)
#print(now > week_ago)
#print(str(now) > str(week_ago))
```

Task 25.6

There are no tests for the tasks of the 25th chapter!

This task contains the parse_dhcp_snooping.py file. You cannot change anything in the parse_dhcp_snooping.py file.

The file creates several functions and describes the command line arguments that the file takes. There is support for arguments to perform all the actions that, in the previous tasks, were performed in the files create_db.py, add_data.py and get_data.py.

The parse_dhcp_snooping.py file contains this line: import parse_dhcp_snooping_functions as pds

And the goal of this task is to create all the necessary functions in the parse_dhcp_snooping_functions.py file based on the information in the parse_dhcp_snooping.py file.

From the parse_dhcp_snooping.py file, you need to decide:

- what functions should be in the parse_dhcp_snooping_functions.py file
- what parameters to create in these functions

It is necessary to create the corresponding functions and transfer to them the functionality that is described in the previous tasks. All the necessary information is present in the create, add, get functions, in the parse_dhcp_snooping.py file.

In principle, to complete the task, it is not necessary to understand the argparse module, but, you can read about it in this section

To make it easier to get started, try creating the required functions in parse_dhcp_snooping_functions.py and just print the function arguments. Then, you can create functions that request information from the database (the database can be copied from previous jobs).

You can create any helper functions in parse_dhcp_snooping_functions.py, not just those called from parse_dhcp_snooping.py.

Check all operations:

- creating a database

- adding information about switches
- adding information based on the output of sh ip dhcp snooping binding from files
- selection of information from the database (by parameter and all information)

To make it easier to understand what the script call will look like, here are some examples. The examples show the option when the database has active and last_active fields, but you can also use the option without these fields.

```
$ python parse_dhcp_snooping.py get -h
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]

optional arguments:
  -h, --help            show this help message and exit
  --db DB_FILE          database name
  -k {mac,ip,vlan,interface,switch}
                        parameter for searching records
  -v VALUE              parameter value
  -a                    show all database content


$ python parse_dhcp_snooping.py add -h
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                  filename [filename ...]

positional arguments:
  filename      file(s) to add

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  database name
  -s            if the flag is set, add switch data, otherwise add DHCP
                records


$ python parse_dhcp_snooping.py create_db
Creating a dhcp_snooping.db database with dhcp_snooping_schema.sql schema
Creating database...


$ python parse_dhcp_snooping.py add sw[1-3]_dhcp_snooping.txt
Adding information from files
sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt
```

(continues on next page)

(continued from previous page)

```
Adding data on DHCP records to dhcp_snooping.db


$ python parse_dhcp_snooping.py add -s switches.yml
Adding switch data

$ python parse_dhcp_snooping.py get
The dhcp table has the following entries:

Active entries:

-----------------  ---------------  --  ----------------  ---  -  ----------------
↪---
00:09:BB:3D:D6:58  10.1.10.2        10  FastEthernet0/1   sw1  1  2019-03-08␣
↪16:47:52
00:04:A3:3E:5B:69  10.1.5.2          5  FastEthernet0/10  sw1  1  2019-03-08␣
↪16:47:52
00:05:B3:7E:9B:60  10.1.5.4          5  FastEthernet0/9   sw1  1  2019-03-08␣
↪16:47:52
00:07:BC:3F:A6:50  10.1.10.6        10  FastEthernet0/3   sw1  1  2019-03-08␣
↪16:47:52
00:09:BC:3F:A6:50  192.168.100.100   1  FastEthernet0/7   sw1  1  2019-03-08␣
↪16:47:52
00:A9:BB:3D:D6:58  10.1.10.20       10  FastEthernet0/7   sw2  1  2019-03-08␣
↪16:47:52
00:B4:A3:3E:5B:69  10.1.5.20         5  FastEthernet0/5   sw2  1  2019-03-08␣
↪16:47:52
00:C5:B3:7E:9B:60  10.1.5.40         5  FastEthernet0/9   sw2  1  2019-03-08␣
↪16:47:52
00:A9:BC:3F:A6:50  10.1.10.60       20  FastEthernet0/2   sw2  1  2019-03-08␣
↪16:47:52
00:E9:BC:3F:A6:50  100.1.1.6         3  FastEthernet0/20  sw3  1  2019-03-08␣
↪16:47:52
-----------------  ---------------  --  ----------------  ---  -  ----------------
↪---


$ python parse_dhcp_snooping.py get -k vlan -v 10
Data from the database: dhcp_snooping.db
Information about devices with the following parameters: vlan 10

Active entries:
```

(continues on next page)

(continued from previous page)

```
-----------------  ----------  --  ---------------  ---  -  -------------------
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1  sw1  1  2019-03-08 16:47:52
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/3  sw1  1  2019-03-08 16:47:52
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7  sw2  1  2019-03-08 16:47:52
-----------------  ----------  --  ---------------  ---  -  -------------------


$ python parse_dhcp_snooping.py get -k vln -v 10
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]
parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'vln' (choose␣
↪from 'mac', 'ip', 'vlan', 'interface', 'switch')
```

10

VIII. Additional information

This section covers information that is not included in main sections of the book, but which can still be useful.

Testing tasks with the pyneng utility

Starting with “4. Python Data Types”, automated tests are used to validate tasks. They help to check whether everything matches the task, and also give feedback on what does not correspond to the task. As a rule, after the first period of adaptation to tests, it becomes easier to do tasks with tests.

In addition to the positive points listed above, in the tests you can also see what the final result is needed: to clarify the data structure and little things that can affect the result.

To run the tests, `pyneng.py` is used - a script that is located in the job repository.

Where to solve tasks

Tasks must be done in prepared files. For example, section 04_data_structures contains task 4.3. Open the exercises/04_data_structures/task_4_3.py file and write code for the task directly in this file after the task description.

This is important because tests are tied to the fact that solution for the tasks is written in specific files and in a specific directory structure. In addition to the fact that the tasks must be solved in the prepared files, be sure to copy the entire exercises directory (or even better, the entire pyneng-examples-exercises-en repository), since tests depend on files in the exercises directory, not only on files in the specific tasks directory.

Installing the pyneng script

First, you need to install it so that you don’t have to write `python pyneng.py` every time.

To install the script, the pyneng.py and setup.py files must be in the repository. These files are located in the root of the repository.

You need to go to your repository, for example (write the name of your repository):

```
cd my_repo/
```

Then, inside the repository, give the command

```
pip install .
```

This will install the module and make it possible to call it in any directory using the word pyneng.

pyneng utility

Stages of work with tasks:

1. Completion of tasks
2. Checking that the task is working as needed `python task_4_2.py` or running the script in the editor/IDE
3. Checking assignments with tests `pyneng 1-5`
4. If the tests pass, look at the solutions `pyneng 1-5 -a`

Note: The second step is very important because it is much easier to find syntax errors and similar problems with the script at this stage than when running the code through a test in step 3.

The script makes it easier to run tests, since you do not need to specify any parameters, by default the output is set to verbose and runs with the pytest-clarity plugin, which improves diff when the solution is different from correct result. Also, some things are hidden, for example, the warning that pytest shows, so as not to distract from the task.

Tests can still be run with *pytest* if you are used to it or have used it before. The pyneng script is just a wrapper around running pytest.

The second part of the script's work is copying the answers to the tasks. This part is done for convenience, so that you do not have to look for answers and is made in such a way that first the task must pass the test and only after that `pyneng -a` will work and show the answers (copy them to the current directory). To copy responses, the script clones the asnwers repository into the user's home directory, copies the task(s) answer, and deletes the repository with answers.

Checking tasks with tests

After completing the task, it must be checked using tests. To run the tests, you need to call pyneng in the task directory. For example, if you are doing section 4 of the tasks, you need to run pyneng from the exercises/04_data_structures directory

Testing all tasks of the current section:

```
pyneng
```

Running tests for task 4.1:

```
pyneng 1
```

Running tests for task 4.1, 4.2, 4.3:

```
pyneng 1-3
```

If there are tasks with letters, for example, in section 7, you can run it in such a way to start checking for tasks 7.2a, 7.2b (you must be in the 07_files directory):

```
pyneng 2a-b
```

or so to run all 7.2x tasks with and without letters:

```
pyneng 2*
```

How to get answers

If the tasks pass the tests, you can see the answers to the tasks.

To do this, add `-a` to the previous versions of the command. Such a call means running tests for tasks 1 and 2 and copying the answers if the tests passed:

```
pyneng 1-2 -a
```

For the specified tasks, tests will run, and for those tasks that passed the tests, the answers will be copied to the answer_task_x.py files in the current directory.

pyneng output

Warning

At the end of the test output, “1 warning” is often written. This can be ignored, warnings are mainly related to the operation of some modules and are hidden so as not to distract from the tasks.

Tests passed

```
collected 11 items

test_task_6_2a.py::test_task_correct_ip[10.1.1.1-unicast] PASSED                                    [  9%]
test_task_6_2a.py::test_task_correct_ip[230.1.1.1-multicast] PASSED                                 [ 18%]
test_task_6_2a.py::test_task_correct_ip[255.255.255.255-local broadcast] PASSED                     [ 27%]
test_task_6_2a.py::test_task_correct_ip[0.0.0.0-unassigned] PASSED                                  [ 36%]
test_task_6_2a.py::test_task_correct_ip[250.1.1.1-unused] PASSED                                    [ 45%]
test_task_6_2a.py::test_task_wrong_ip[10.1.1-неправильный] PASSED                                   [ 54%]
test_task_6_2a.py::test_task_wrong_ip[10.a.2.a-неправильный] PASSED                                 [ 63%]
test_task_6_2a.py::test_task_wrong_ip[10.1.1.1.1-неправильный] PASSED                               [ 72%]
test_task_6_2a.py::test_task_wrong_ip[10.1.1.-неправильный] PASSED                                  [ 81%]
test_task_6_2a.py::test_task_wrong_ip[300.1.1.1-неправильный] PASSED                                [ 90%]
test_task_6_2a.py::test_task_wrong_ip[30,1.1.1.1-неправильный] PASSED                               [100%]

---------------------------------------------------- JSON report -----------------------------------------------------
no JSON report written.
============================================ 11 passed, 1 warning in 0.12s =============================================
(pyneng-py3-8-0)
```

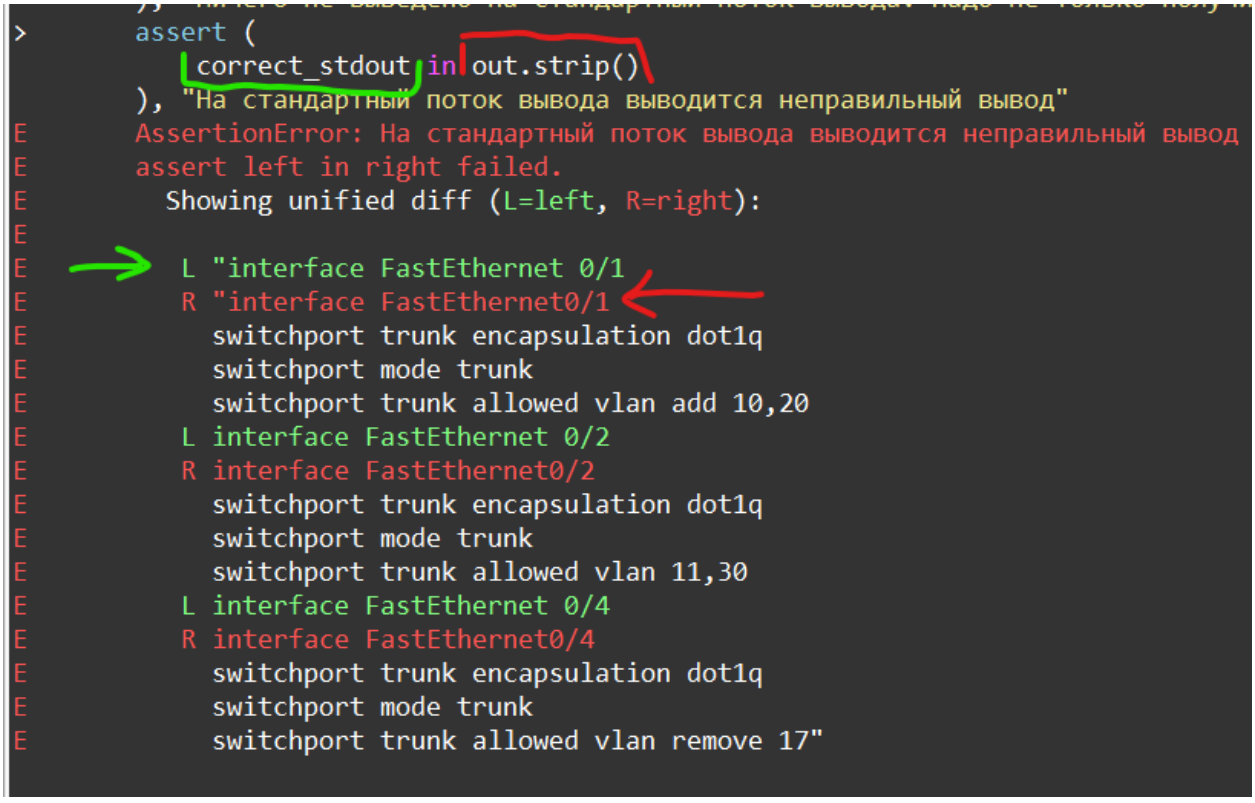
Test failed

When some tests fail, the output shows the difference between what the output should look like and what output was received.

The differences are shown as Left and Right, unfortunately there is no such thing that the correct option is highlighted in green, and the wrong one is highlighted in red, you need to look at the situation. Every time you display differences, there is a line in front of them like this:

```
assert correct_stdout in out.strip()
```

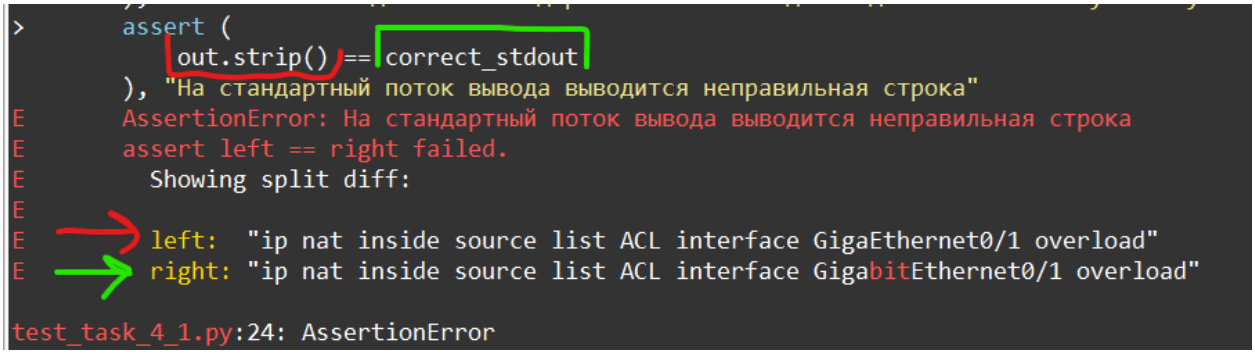
In this case, Left is the correct output, right is the output of the task:



another example:

```
return_value == correct_return_value
```

In this case, Right is the correct output, Left is the output of the task:


assert (
out.strip() == correct_stdout
), "На стандартный поток вывода выводится неправильная строка"
AssertionError: На стандартный поток вывода выводится неправильная строка
assert left == right failed.
Showing split diff:
left: "ip nat inside source list ACL interface GigaEthernet0/1 overload"
right: "ip nat inside source list ACL interface GigabitEthernet0/1 overload"
test_task_4_1.py:24: AssertionError

Tasks checking with tests

Starting with section “9. Functions” automatic tests are used to check tasks. They help to check that everything conforms to the task and also provide feedback on what is not up to task. Usually, after the first period of adaptation it becomes easier to do tasks with tests.

In addition to above-mentioned positive features, tests can also show what result is expected: clarify structure of data and details that may affect the result.

Pytest is used to run tests - a framework for writing tests.

Note: Record of lecture on using pytest for test verification

---

Pytest basics

First, you need to install pytest and pyyaml:

```
pip install pytest
pip install pyyaml
```

Although you don’t have to write tests code but to understand it you should look at an example of a test. For example, there is the following code with check_ip function:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

Function check_ip checks whether the argument given to it is an IP address. An example of calling a function with different arguments:

```
In [1]: import ipaddress
   ...:
```

(continues on next page)

(continued from previous page)

```
   ...:
   ...: def check_ip(ip):
   ...:     try:
   ...:         ipaddress.ip_address(ip)
   ...:         return True
   ...:     except ValueError as err:
   ...:         return False
   ...:

In [2]: check_ip('10.1.1.1')
Out[2]: True

In [3]: check_ip('10.1.')
Out[3]: False

In [4]: check_ip('a.a.a.a')
Out[4]: False

In [5]: check_ip('500.1.1.1')
Out[5]: False
```

Now it is necessary to write a test for check_ip function. Test must check that function returns True when correct address is passed and False when wrong argument is passed.

To simplify task, test can be written in the same file. In pytest, test can be a normal function with a name that starts with `test_`. Inside function you have to write conditions that are checked. In pytest this is done with `assert`.

assert

`assert` does nothing if expression is True and generates an exception if expression is False:

```
In [6]: assert 5 > 1

In [7]: a = 4

In [8]: assert a in [1,2,3,4]

In [9]: assert a not in [1,2,3,4]
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-9-1956288e2d8e> in <module>
----> 1 assert a not in [1,2,3,4]
```

(continues on next page)

(continued from previous page)

```
AssertionError:

In [10]: assert 5 < 1
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-b224d03aab2f> in <module>
----> 1 assert 5 < 1

AssertionError:
```

After `assert` and expression you can write a message. If there is a message, it is displayed in exception:

```
In [11]: assert a not in [1,2,3,4], "a not in a list"
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-11-7a8f87272a54> in <module>
----> 1 assert a not in [1,2,3,4], "a not in a list"

AssertionError: a not in a list
```

Test example

pytest uses `assert` to specify which conditions must be met in order for test to be considered passed.

In pytest, you can write test as a normal function but function name must start with `test_`. Below is `test_check_ip` test which verify `check_ip` function by passing two values to it: correct address and wrong one, and after each check the message is written:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


def test_check_ip():
    assert check_ip('10.1.1.1') == True, 'If IP is correct, the fucntion returns␣
↪True'
```

(continues on next page)

(continued from previous page)

```
    assert check_ip('500.1.1.1') == False, 'If IP is wrong, the fucntion returns␣
↪False'


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

Code is written in check_ip_functions.py. Now you have to figure out how to call tests. The easiest option is to write `pytest` word. In this case, pytest will automatically detect tests in the current directory. However, pytest has certain rules, not only by name of function but also by name of test files - file names should also start with `test_`. If rules are respected, pytest will automatically find tests, if not - you have to specify a test file.

In the case of example above, you have to call a command:

```
$ pytest check_ip_functions.py
========================= test session starts ==========================
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

check_ip_functions.py .                                              [100%]

======================== 1 passed in 0.02 seconds ======================
```

By default if tests pass, each test (test_check_ip function) is marked with a dot. Since in this case there is only one test - `test_check_ip` function, there is a dot after name check_ip_functions.py and it is also written below that 1 test has passed.

Now, suppose the function does not work correctly and always returns False (write return False at the beginning of function). In this case, test execution will look like:

```
$ pytest check_ip_functions.py
========================= test session starts ==========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

check_ip_functions.py F                                              [100%]

=============================== FAILURES ===============================
_______________________________ test_check_ip ________________________________

```

(continues on next page)

(continued from previous page)

```
    def test_check_ip():
>       assert check_ip('10.1.1.1') == True, 'If IP is correct, the fucntion␣
↪returns True'
E       AssertionError: If IP is correct, the fucntion returns True
E       assert False == True
E        +  where False = check_ip('10.1.1.1')

check_ip_functions.py:14: AssertionError
======================== 1 failed in 0.06 seconds ========================
```

If test fails, pytest displays more information and shows where things went wrong. In this case, after execution of `assert check_ip('10.1.1.1') == True` string, the expression did not return True result, so an exception was generated.

Below, pytest shows what it has compared: `assert False == True` and specifies that False is `check_ip('10.1.1.1')`. Looking at the output, one suspects that something is wrong with `check_ip` function because it returns False to correct address.

Most tests are written in separate files. For this example, test is only one but it is still in a separate file.

File test_check_ip_function.py:

```
from check_ip_functions import check_ip


def test_check_ip():
    assert check_ip('10.1.1.1') == True, 'If IP is correct, the fucntion returns␣
↪True'
    assert check_ip('500.1.1.1') == False, 'If IP is wrong, the fucntion returns␣
↪False'
```

File check_ip_functions.py:

```
import ipaddress


def check_ip(ip):
    #return False
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

In that case, test can be run without specifying a file:

```
$ pytest
================= test session starts ========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

test_check_ip_function.py .                                  [100%]

================= 1 passed in 0.02 seconds ====================
```

Specifics of using pytest to check tasks

Pytest in course is primarily used for self-tests of tasks. However, this test is not optional - task is considered done when it complies with all specified points and passes tests. For my part, I also check tasks with automatic tests and then look at the code, write comments if necessary and show a solution option.

At first, tests require effort but through a couple of sections they will help solve tasks.

> **Warning:** Tests that are written for course are not a benchmark or best practice of test writing. Tests are written with maximum emphasis on clarity and many things are done differently.

When solving tasks especially when there are doubts about the final format of data to be obtained, it is better to look into test. For example, if task_9_1.py the corresponding test will be in test/test_task_9_1.py.

Test example tests/test_task_9_1.py:

```
import pytest
import task_9_1
import sys
sys.path.append('..')

from common_functions import check_function_exists, check_function_params
```

(continues on next page)

(continued from previous page)

```
# Checks is function generate_access_config is created in task task_9_1
def test_function_created():
    check_function_exists(task_9_1, 'generate_access_config')

# Cheks fucntion parameters
def test_function_params():
    check_function_params(function=task_9_1.generate_access_config,
                          param_count=2, param_names=['intf_vlan_mapping',
↪'access_template'])


def test_function_return_value():
    access_vlans_mapping = {
        'FastEthernet0/12': 10,
        'FastEthernet0/14': 11,
        'FastEthernet0/16': 17
    }
    template_access_mode = [
        'switchport mode access', 'switchport access vlan',
        'switchport nonegotiate', 'spanning-tree portfast',
        'spanning-tree bpduguard enable'
    ]
    correct_return_value = ['interface FastEthernet0/12',
                            'switchport mode access',
                            'switchport access vlan 10',
                            'switchport nonegotiate',
                            'spanning-tree portfast',
                            'spanning-tree bpduguard enable',
                            'interface FastEthernet0/14',
                            'switchport mode access',
                            'switchport access vlan 11',
                            'switchport nonegotiate',
                            'spanning-tree portfast',
                            'spanning-tree bpduguard enable',
                            'interface FastEthernet0/16',
                            'switchport mode access',
                            'switchport access vlan 17',
                            'switchport nonegotiate',
                            'spanning-tree portfast',
                            'spanning-tree bpduguard enable']

    return_value = task_9_1.generate_access_config(access_vlans_mapping, template_
↪access_mode)
```

(continues on next page)

(continued from previous page)

```
    assert return_value != None, "Functon returns nothing"
    assert type(return_value) == list, "Function has to return a list"
    assert return_value == correct_return_value, "Function return wrong value"
```

Note correct_return_value variable - this variable contains the resulting list that should return generate_access_config function. Therefore for example, if question has arisen of whether to add spaces before commands or a new line at the end, you can look at what the result requires. Also check your output against the output in variable_return_value.

How to run tests for tasks verification

The most important thing is where to run tests: all tests must be run from a directory with section tasks, not from a test directory. For example, in section 09_functions such a directory structure with tasks:

```
[~/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions]
vagrant: [master|✓]
$ tree
.

├── config_r1.txt
├── config_sw1.txt
├── config_sw2.txt
├── conftest.py
├── task_9_1a.py
├── task_9_1.py
├── task_9_2a.py
├── task_9_2.py
├── task_9_3a.py
├── task_9_3.py
├── task_9_4.py
└── tests
    ├── test_task_9_1a.py
    ├── test_task_9_1.py
    ├── test_task_9_2a.py
    ├── test_task_9_2.py
    ├── test_task_9_3a.py
    ├── test_task_9_3.py
    └── test_task_9_4.py
```

In this case, you have to run tests from 09_functions directory:

```
[~/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions]
vagrant: [master|✓]
$ pytest tests/test_task_9_1.py
========================= test session starts ==========================
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_
↪functions
collected 3 items

tests/test_task_9_1.py ...                                           [100%]
...

If you run tests from tests directory, errors will appear.
```

conftest.py

In addition to test directory there is a conftest.py file - special file in which you can write functions (more precisely fixtures) common to different tests. For example, this file contains functions that connect via SSH/Telnet to equipment.

Useful commands

Run one test:

```
$ pytest tests/test_task_9_1.py
```

Run one test with more detailed output (shows diff between data in test and what is received from function):

```
$ pytest tests/test_task_9_1.py -vv
```

Start all tests of one section:

```
[~/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions]
vagrant: [master|✓]
$ pytest
======================== test session starts ========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_
↪functions
collected 21 items

```

(continues on next page)

(continued from previous page)

```
tests/test_task_9_1.py ..F                                    [ 14%]
tests/test_task_9_1a.py FFF                                   [ 28%]
tests/test_task_9_2.py FFF                                    [ 42%]
tests/test_task_9_2a.py FFF                                   [ 57%]
tests/test_task_9_3.py FFF                                    [ 71%]
tests/test_task_9_3a.py FFF                                   [ 85%]
tests/test_task_9_4.py FFF                                    [100%]

...
```

Starts all tests of the same section with error messages displayed in one line:

```
$ pytest --tb=line
```

argparse

argparse is a module for handling command line arguments. Examples of what a module does:

- create arguments and options with which script can be called
- specify argument types, default values
- indicate which actions correspond to arguments
- call functions when argument is specified
- show messages with hints of script usage

argparse is not the only module for handling command line arguments. And not even the only one in standard library.

This book covers only argparse, but in addition it is worth looking at modules that are not part of standard Python library. For example, click.

Note: A good article, compares different command line argument processing modules (covered argparse, click and docopt).

Example of ping_function.py script:

```
import subprocess
import argparse


def ping_ip(ip_address, count):
```

(continues on next page)

(continued from previous page)

```
    '''
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    '''
    reply = subprocess.run(
        'ping -c {count} -n {ip}'.format(count=count, ip=ip_address),
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding='utf-8'
    )
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stdout+reply.stderr



parser = argparse.ArgumentParser(description='Ping script')

parser.add_argument('-a', action="store", dest="ip")
parser.add_argument('-c', action="store", dest="count", default=2, type=int)

args = parser.parse_args()
print(args)

rc, message = ping_ip(args.ip, args.count)
print(message)
```

Creation of a parser:

- `parser = argparse.ArgumentParser(description='Ping script')`

Adding arguments:

- `parser.add_argument('-a', action="store", dest="ip")`
  - rgument that is passed after `-a` option is saved to variable `ip`
- `parser.add_argument('-c', action="store", dest="count", default=2, type=int)`
  - argument that is passed after `-c` option will be saved to variable `count`, but will be converted to a number first. If no argument was specified, the default is 2

String `args = parser.parse_args()` is specified after all arguments have been defined. After running it, variable `args` contains all arguments that were passed to the script. They can be accessed

using `args.ip` syntax.

Let's try a script with different arguments. If both arguments are passed:

```
$ python ping_function.py -a 8.8.8.8 -c 5
Namespace(count=5, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.673 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.902 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=48.696 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=48 time=50.040 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=48 time=48.831 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.673/49.228/50.040/0.610 ms

Namespace is an object that returns parse\_args() method
```

Pass only IP address:

```
$ python ping_function.py -a 8.8.8.8
Namespace(count=2, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.563 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.616 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.563/49.090/49.616/0.526 ms
```

Call script without arguments:

```
$ python ping_function.py
Namespace(count=2, ip=None)
Traceback (most recent call last):
  File "ping_function.py", line 31, in <module>
    rc, message = ping_ip( args.ip, args.count )
  File "ping_function.py", line 16, in ping_ip
    stderr=temp)
  File "/usr/local/lib/python3.6/subprocess.py", line 336, in check_output
    ````kwargs).stdout
  File "/usr/local/lib/python3.6/subprocess.py", line 403, in run
    with Popen(*popenargs, ````kwargs) as process:
  File "/usr/local/lib/python3.6/subprocess.py", line 707, in __init__
```

(continues on next page)
```

(continued from previous page)

```
    restore_signals, start_new_session)
  File "/usr/local/lib/python3.6/subprocess.py", line 1260, in _execute_child
    restore_signals, start_new_session, preexec_fn)
TypeError: expected str, bytes or os.PathLike object, not NoneType
```

If function was called without arguments when `argparse` is not used, an error would occur that not all arguments are specified.

Because of `argparse` the argument is actually passed, but it has `None` value. You can see this in `Namespace(count=2, ip=None)` string.

In such a script the IP address must be specified at all times. And in `argparse` you can specify that argument is mandatory. To do this, change `-a` option: add `required=True` at the end:

```
parser.add_argument('-a', action="store", dest="ip", required=True)
```

Now, if you call a script without arguments, the output is:

```
$ python ping_function.py
usage: ping_function.py [-h] -a IP [-c COUNT]
ping_function.py: error: the following arguments are required: -a
```

Now you see a clear message that you need to specify a mandatory argument and a usage hint.

Also, thanks to `argparse`, `help` is available:

```
$ python ping_function.py -h
usage: ping_function.py [-h] -a IP [-c COUNT]

Ping script

optional arguments:
  -h, --help  show this help message and exit
  -a IP
  -c COUNT
```

Note that in message all options are in `optional arguments` section. `argparse` itself determines that options are specified because they start with `-` and only one letter in name.

Set IP address as a positional argument (ping_function_ver2.py file):

```
import subprocess
from tempfile import TemporaryFile

import argparse

```

(continues on next page)

(continued from previous page)

```
def ping_ip(ip_address, count):
    '''
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    '''
    reply = subprocess.run(
        'ping -c {count} -n {ip}' .format(count=count, ip=ip_address),
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding='utf-8',
    )
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stdout+reply.stderr



parser = argparse.ArgumentParser(description='Ping script')

parser.add_argument('host', action="store", help="IP or name to ping")
parser.add_argument('-c', action="store", dest="count", default=2, type=int,
                    help="Number of packets")

args = parser.parse_args()
print(args)

rc, message = ping_ip( args.host, args.count )
print(message)
```

Now instead of giving `-a` option you can simply pass IP address. It will be automatically saved in `host` variable. And it's automatically considered as a mandatory. That is, it is no longer necessary to specify `required=True` and `dest="ip"`.

In addition, script specifies messages that will be displayed when you call `help`. Now script call looks like this:

```
$ python ping_function_ver2.py 8.8.8.8 -c 2
Namespace(host='8.8.8.8', count=2)
PING 8.8.8.8 (8.8.8.8): 56 data bytes
```

(continues on next page)

(continued from previous page)

```
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=49.203 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=51.764 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 49.203/50.484/51.764/1.280 ms
```

help message:

```
$ python ping_function_ver2.py -h
usage: ping_function_ver2.py [-h] [-c COUNT] host

Ping script

positional arguments:
  host        IP or name to ping

optional arguments:
  -h, --help  show this help message and exit
  -c COUNT    Number of packets
```

Nested parsers

Consider one of the methods to organize a more complex hierarchy of arguments.

Note: This example will show more features of argparse but they are not limited to that, so if you use argparse you should check module documentation or article on PyMOTW.

File parse_dhcp_snooping.py:

```
# -*- coding: utf-8 -*-
import argparse

# Default values:
DFLT_DB_NAME = 'dhcp_snooping.db'
DFLT_DB_SCHEMA = 'dhcp_snooping_schema.sql'


def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))
```

(continues on next page)

(continued from previous page)

```
def add(args):
    if args.sw_true:
        print("Adding switch data to database")
    else:
        print("Reading info from file(s) \n{}".format(', '.join(args.filename)))
        print("\nAdding data to db {}".format(args.db_file))


def get(args):
    if args.key and args.value:
        print("Geting data from DB: {}".format(args.db_file))
        print("Request data for host(s) with {} {}".format((args.key, args.
↪value)))
    elif args.key or args.value:
        print("Please give two or zero args\n")
        print(show_subparser_help('get'))
    else:
        print("Showing {} content...".format(args.db_file))


parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers(title='subcommands',
                                   description='valid subcommands',
                                   help='description')


create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults(func=create)


add_parser = subparsers.add_parser('add', help='add data to db')
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
add_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db␣
↪name')
add_parser.add_argument('-s', dest='sw_true', action='store_true',
                        help='add switch data if set, else add normal data')
add_parser.set_defaults(func=add)
```

(continues on next page)

(continued from previous page)

```
get_parser = subparsers.add_parser('get', help='get data from db')
get_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db␣
↪name')
get_parser.add_argument('-k', dest="key",
                        choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                        help='host key (parameter) to search')
get_parser.add_argument('-v', dest="value", help='value of key')
get_parser.add_argument('-a', action='store_true', help='show db content')
get_parser.set_defaults(func=get)



if __name__ == '__main__':
    args = parser.parse_args()
    if not vars(args):
        parser.print_usage()
    else:
        args.func(args)
```

Now not only a parser is created as in previous example, but also nested parsers. Nested parsers will be displayed as commands. In fact, they will be used as mandatory arguments.

With help of nested parsers a hierarchy of arguments and options is created. Arguments that are added to nested parser will be available as arguments for this parser. For example, this part creates a nested `create_db` parser and adds `-n` option:

```
create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', dest='name', default=DFLT_DB_NAME,
                           help='db filename')
```

Syntax for creating nested parsers and adding arguments to them is the same:

```
create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults(func=create)
```

Method `add_argument` adds an argument. Here, syntax is exactly the same as without nested parsers.

String `create_parser.set_defaults(func=create)` specifies that the `create` function will be called when calling the `create_parser` parser.

Function create receives as an argument all arguments that have been passed. And within function you can access to necessary arguments:

```
def create(args):
    print("Creating DB {} with DB schema {}".format(args.name, args.schema))
```

If you call help for this script, the output is:

```
$ python parse_dhcp_snooping.py -h
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...

optional arguments:
  -h, --help            show this help message and exit

subcommands:
  valid subcommands

  {create_db,add,get}  description
    create_db          create new db
    add                add data to db
    get                get data from db
```

Note that each nested parser that is created in the script is displayed as a command in usage hint:

```
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...
```

Each nested parser now has its own help:

```
$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n db-filename] [-s SCHEMA]

optional arguments:
  -h, --help      show this help message and exit
  -n db-filename  db filename
  -s SCHEMA       db schema filename
```

In addition to nested parsers, there are also several new features of argparse in this example.

metavar

Parser create_parser uses a new argument - metavar:

```
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
```

(continues on next page)

(continued from previous page)

```
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
```

Argument `metavar` allows you to specify argument name to show it in usage message and `help`:

```
$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n db-filename] [-s SCHEMA]

optional arguments:
  -h, --help      show this help message and exit
  -n db-filename  db filename
  -s SCHEMA       db schema filename
```

Look at the difference between `-n` and `-s` options:

- after `-n` option in both `usage` and `help` the name is specified in the `metavar` parameter
- after `-s` option the name is specified to which value is saved

nargs

Parser `add_parser` uses `nargs`:

```
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
```

Parameter `nargs` allows to specify a certain number of elements that must be entered into this argument. In this case, all arguments that have been passed to the script after `filename` argument will be included in `nargs` list, but at least one argument must be passed.

In this case, `help` message looks like:

```
$ python parse_dhcp_snooping.py add -h
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                  filename [filename ...]

positional arguments:
  filename      file(s) to add to db

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  db name
  -s            add switch data if set, else add normal data
```

If you pass several files, they'll be in the list. And since add function simply displays file names, the output is:

```
$ python parse_dhcp_snooping.py add filename test1.txt test2.txt
Reading info from file(s)
filename, test1.txt, test2.txt

Adding data to db dhcp_snooping.db
```

nargs supports such values as:

- N - - number of arguments should be specified. Arguments will be in list (even if only one is specified)
- ? - 0 or 1 argument
- * - all arguments will be in list
- + - all arguments will be in list, but at least one argument has to be passed

choices

Parser get_parser uses choices:

```
get_parser.add_argument('-k', dest="key",
                        choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                        help='host key (parameter) to search')
```

For some arguments it is important that the value is selected only from certain options. In such cases you can specify choices.

For this parser help looks like this:

```
$ python parse_dhcp_snooping.py get -h
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]

optional arguments:
  -h, --help            show this help message and exit
  --db DB_FILE          db name
  -k {mac,ip,vlan,interface,switch}
                        host key (parameter) to search
  -v VALUE              value of key
  -a                    show db content
```

And if you choose the wrong option:

```
$ python parse_dhcp_snooping.py get -k test
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]
parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'test' (choose␣
↪from 'mac', 'ip', 'vlan', 'interface', 'switch')
```

In this example it is important to specify allowed options that could be chosen because based on chosen option the SQL-query is generated. And thanks to choices there is no pissibility to specify parameter that is not allowed.

Parser import

In parse_dhcp_snooping.py, the last two lines will only be executed if script has been called as a main script.

```
if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

Therefore, if you import a file these lines will not be called.

Trying to import parser into another file (call_pds.py file):

```
from parse_dhcp_snooping import parser

args = parser.parse_args()
args.func(args)
```

Call help message:

```
$ python call_pds.py -h
usage: call_pds.py [-h] {create_db,add,get} ...

optional arguments:
  -h, --help           show this help message and exit

subcommands:
  valid subcommands

  {create_db,add,get}  description
    create_db          create new db
    add                add data to db
    get                get data from db
```

Invoking the argument:

```
$ python call_pds.py add test.txt test2.txt
Reading info from file(s)
test.txt, test2.txt

Adding data to db dhcp_snooping.db
```

Everything works without a problem.

Passing of arguments manually

The last feature of `argparse` is the ability to pass arguments manually.

Arguments can be passed as a list when calling `parse_args` method (call_pds2.py file):

```
from parse_dhcp_snooping import parser, get

args = parser.parse_args('add test.txt test2.txt'.split())
args.func(args)
```

It is necessary to use `split` method since `parse_args` method expects list of arguments.

The result will be the same as if script was called with arguments:

```
$ python call_pds2.py
Reading info from file(s)
test.txt, test2.txt

Adding data to db dhcp_snooping.db
```

String formatting with % operator

Example of `%` operator use:

```
In [2]: "interface FastEthernet0/%s" % '1'
Out[2]: 'interface FastEthernet0/1'
```

Old string format syntax uses these symbols:

- `%s` - string or any other object with a string type
- `%d` - integer
- `%f` - float

Output data columns of equal width of 15 characters with right side alignment:

```
In [3]: vlan, mac, intf = ['100', 'aabb.cc80.7000', 'Gi0/1']

In [4]: print("%15s %15s %15s" % (vlan, mac, intf))
            100  aabb.cc80.7000           Gi0/1
```

Left side alignment:

```
In [6]: print("%-15s %-15s %-15s" % (vlan, mac, intf))
100             aabb.cc80.7000  Gi0/1
```

You can also use string formatting to influence the appearance of numbers.

For example, you can specify how many digits to show after comma:

```
In [8]: print("%.3f" % (10.0/3))
3.333
```

Note: String formatting still has many possibilities. Good examples and explanations of two string formatting options can be found here.

Naming convention

Python has certain objects naming convention

In general, it is better to adhere to this convention. However, if a particular library or module uses different convention, it is worth following the style used in them.

> Not all rules are described in this section. More information can be found in PEP8 in English or Russian.

Variable names

Variable names should not overlap with operators and names of modules or other reserved values. Variable names are usually written entirely in large or small letters. It is better to stick to one of option within a script/module/package.

If variables are constants for module, it is better to use names written in capital letters:

```
DB_NAME = 'dhcp_snooping.db'
TESTING = True
```

For ordinary variables it is better to use lower case names:

```
db_name = 'dhcp_snooping.db'
testing = True
```

Module and package names

Names of modules and packages are given in small letters.

Modules can use underscores to make names more understandable. For packages it is better to select short names.

Function names

Function names are given in small letters with underscores between words.

```
def ignore_command(command, ignore):

    ignore_command = False

    for word in ignore:
        if word in command:
            return True
    return ignore_command
```

Class names

Class names are given with capital letters, no spaces.

```
class CiscoSwitch:

    def __init__(self, name, vendor='cisco', model='3750'):
        self.name = name
        self.vendor = vendor
        self.model = model
```

Underscore in names

In Python, underscores at the beginning or at the end of a name indicates special names. Most often it's just an arrangement but sometimes it actually affects object behavior.

Underscore in name

In Python, one underscore is used to simply indicate that data is discarded.

For example, if you want to get MAC address, IP address, VLAN and interface from *line* string and discard the rest of fields, you can use this option:

```
In [1]: line = '00:09:BB:3D:D6:58  10.1.10.2 86250   dhcp-snooping   10 ␣
↪FastEthernet0/1'

In [2]: mac, ip, _, _, vlan, intf = line.split()

In [3]: print(mac, ip, vlan, intf)
00:09:BB:3D:D6:58 10.1.10.2 10 FastEthernet0/1
```

This record indicates that we do not need the third and fourth elements.

You can do this:

```
In [4]: mac, ip, lease, entry_type, vlan, intf = line.split()
```

But then it may be unclear why *lease* and *entry_type* variables are not used any further. It is better to call variable names like *ignored*.

A similar technique can be used when a loop variable is not needed:

```
In [5]: [0 for _ in range(10)]
Out[5]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Underscore in interpreter

In the python and ipython interpreter undesrcore is used to get result of the last experision.

```
In [6]: [0 for _ in range(10)]
Out[6]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [7]: _
Out[7]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [8]: a = _

In [9]: a
Out[9]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Single underscore

One underscore before name

One underscore before name indicates that the name is used as an internal name.

For example, if one underscore is specified in name of function or method, this means that the object is an internal implementation and should not be used directly.

But also, when importing `from module import *` the objects that start with underscore will not be imported.

For instanse, example.py file contains these variables and functions:

```
db_name = 'dhcp_snooping.db'
_path = '/home/nata/pyneng/'


def func1(arg):
    print arg


def _func2(arg):
    print arg
```

If you import all objects from module, those that start with underscore will not be imported:

```
In [7]: from example import *

In [8]: db_name
Out[8]: 'dhcp_snooping.db'

In [9]: _path
...
NameError: name '_path' is not defined

In [10]: func1(1)
1

In [11]: _func2(1)
...
NameError: name '_func2' is not defined
```

One underscore after name

One underscore after name is used when the name of object or parameter overlaps with the embedded names.

Example:

```
In [12]: line = '00:09:BB:3D:D6:58  10.1.10.2 86250   dhcp-snooping   10 ␣
↪FastEthernet0/1'

In [13]: mac, ip, lease, type_, vlan, intf = line.split()
```

Two underscores

Two underscores before name

Two underscores before method name are not used simply as an agreement. Such names are transformed into format “class name + method name”. This allows the creation of unique methods and attributes of classes.

Note: This transformation is only performed if less than two underscore endings or no underscores.

```
In [14]: class Switch(object):
    ...:     __quantity = 0
    ...:     def __configure(self):
    ...:         pass
    ...:

In [15]: dir(Switch)
Out[15]:
['_Switch__configure', '_Switch__quantity', ...]
```

Although methods were created without _Switch, it was added.

If you create a subclass, then __configure method will not rewrite method of parent Switch class:

```
In [16]: class CiscoSwitch(Switch):
    ...:     __quantity = 0
    ...:     def __configure(self):
    ...:         pass
    ...:
```

(continues on next page)

(continued from previous page)

```
In [17]: dir(CiscoSwitch)
Out[17]:
['_CiscoSwitch__configure', '_CiscoSwitch__quantity', '_Switch__configure', '_
↪Switch__quantity', ...]
```

Two underscores before and after name

Thus, special variables and methods are denoted.

For example, Python module has such special variables:

- __name__ - this variable is equal to __main__ when script runs directly, and it is equal to module name when imported
- __file__ - this variable is equal to script name that was run directly, and equals to complete path to the module when it is imported

__name__ variable is most commonly used to indicate that a certain part of the code must be executed only when module is executed directly:

```
def multiply(a, b):

    return a * b

if __name__ == '__main__':
    print(multiply(3, 5))
```

__file__ variable can be useful in determining the current path to script file:

```
import os

print('__file__', __file__)
print(os.path.abspath(__file__))
```

The output will be:

```
__file__ example2.py
/home/vagrant/repos/tests/example2.py
```

Python also denotes special methods in this way. These methods are called when using Python functions and operators and allow for implementation of a certain functionality.

As a rule, such methods need not be called directly. But for example, when creating your own class it may be necessary to describe such method in order to make object support some operations in Python.

For example, in order to get object length, it must support `__len__` method.

Another special method `__str__` is called when `print` operator is used or `str` function is called. If it is necessary to get a certain output, you have to create this method in the class:

```
In [10]: class Switch(object):
    ...:
    ...:     def set_name(self, name):
    ...:         self.name = name
    ...:
    ...:     def __configure(self):
    ...:         pass
    ...:
    ...:     def __str__(self):
    ...:         return 'Switch {}'.format(self.name)
    ...:

In [11]: sw1 = Switch()

In [12]: sw1.set_name('sw1')

In [13]: print sw1
Switch sw1

In [14]: str(sw1)
Out[14]: 'Switch sw1'
```

There are many such special methods in Python. Some useful links where you can read about a particular method:

- documentation
- Dive Into Python 3

Python 2.7 and Python 3.6 distinctions

Unicode

Python 2.7 has two string types: `str` and `unicode`:

```
In [1]: line = 'test'

In [2]: line2 = u'test'
```

In Python 3, string is `str` type but in addition `bytes` type appeared in Python 3:

```
In [3]: line = 'test'

In [4]: line.encode('utf-8')
Out[4]: b'\xd1\x82\xd0\xb5\xd1\x81\xd1\x82'

In [5]: byte_str = b'test'
```

print function

In Python 2.7 `print` was an operator:

```
In [6]: print 1, 'test'
1 test
```

In Python 3 print - function:

```
In [7]: print(1, 'test')
1 test
```

In Python 2.7 it is possible to put arguments in parentheses, but it doesn't make `print` a function and `print` returns another result (tuple):

```
In [8]: print(1, 'test')
(1, 'test')
```

In Python 3, using Python 2.7 syntax will result in an error:

```
In [9]: print 1, 'test'
  File "<ipython-input-2-328abb6b105d>", line 1
    print 1, 'test'
          ^
SyntaxError: Missing parentheses in call to 'print'
```

input instead of raw_input

In Python 2.7, `raw_input` function was used to get information from user as a string:

```
In [10]: number = raw_input('Number: ')
Number: 55

In [11]: number
Out[11]: '55'
```

Python 3 uses `input`:

```
In [12]: number = input('Number: ')
Number: 55

In [13]: number
Out[13]: '55'
```

range instead of xrange

Python 2.7 had two functions

- `range` - returns list
- `xrange` - returns iterator

Example `range` and `xrange` in Python 2.7:

```
In [14]: range(5)
Out[14]: [0, 1, 2, 3, 4]

In [15]: xrange(5)
Out[15]: xrange(5)

In [16]: list(xrange(5))
Out[16]: [0, 1, 2, 3, 4]
```

Python 3 has only a `range` function and it returns an iterator:

```
In [17]: range(5)
Out[17]: range(0, 5)

In [18]: list(range(5))
Out[18]: [0, 1, 2, 3, 4]
```

Dictionary methods

Several changes have occurred in dictionary methods.

dict.keys, values, items

Methods `keys`, `values`, `items` in Python 3 return `views` instead of lists. The peculiarity of view is that they change with the change of dictionary. And in fact, they just give you a way to look at corresponding objects but they don't make a copy of them.

Python 3 has no methods:

- viewitems, viewkeys, viewvalues
- iteritems, iterkeys, itervalues

For comparison, dictionary methods in Python 2.7:

```
In [19]: d = {1:100, 2:200, 3:300}

In [20]: d.
    d.clear      d.get        d.iteritems  d.keys       d.setdefault d.viewitems
    d.copy       d.has_key    d.iterkeys   d.pop        d.update     d.viewkeys
    d.fromkeys   d.items      d.itervalues d.popitem    d.values     d.viewvalues
```

And in Python 3:

```
In [21]: d = {1:100, 2:200, 3:300}

In [22]: d.
           clear()      get()        pop()        update()
           copy()       items()      popitem()    values()
           fromkeys()   keys()       setdefault()
```

Variables unpacking

In Python 3 it is possible to use * when unpacking variables:

```
In [23]: a, *b, c = [1,2,3,4,5]

In [24]: a
Out[24]: 1

In [25]: b
Out[25]: [2, 3, 4]

In [26]: c
Out[26]: 5
```

Python 2.7 does not support this syntax:

```
In [27]: a, *b, c = [1,2,3,4,5]
  File "<ipython-input-10-e3f57143ffb4>", line 1
    a, *b, c = [1,2,3,4,5]
```

(continues on next page)

(continued from previous page)

```
      ^
SyntaxError: invalid syntax
```

Iterator instead of list

In Python 2.7 map, filter and zip returned a list:

```
In [28]: map(str, [1,2,3,4,5])
Out[28]: ['1', '2', '3', '4', '5']

In [29]: filter(lambda x: x>3, [1,2,3,4,5])
Out[29]: [4, 5]

In [30]: zip([1,2,3], [100,200,300])
Out[30]: [(1, 100), (2, 200), (3, 300)]
```

In Python 3, they return an iterator:

```
In [31]: map(str, [1,2,3,4,5])
Out[31]: <map at 0xb4ee3fec>

In [32]: filter(lambda x: x>3, [1,2,3,4,5])
Out[32]: <filter at 0xb448c68c>

In [33]: zip([1,2,3], [100,200,300])
Out[33]: <zip at 0xb4efc1ec>
```

subprocess.run

Python 3.5 introduced the new run function in subprocess module. It provides a more user-friendly interface for working with module and getting output of commands.

Accordingly, `run` function is used instead of `call` and `check_output` functions. But `call` and `check_output` functions remain.

Jinja2

In Jinja2 module it is no longer necessary to use such code, since the default encoding is utf-8:

```
import sys
reload(sys)
sys.setdefaultencoding('utf-8')
```

In the templates themselves as in Python, dictionary methods have changed. Here, you should use `items` instead of `iteritems`.

Modules pexpect, telnetlib, paramiko

Modules pexpect, telnetlib, paramiko send and receive bytes, so you have to make encode/decode accordingly.

In netmiko this conversion is performed automatically.

Trivia

- Name of Queue module changed to queue
- Starting from Python 3.6, csv.DictReader returns OrderedDict instead of a regular dictionary.

Additional information

Below are links to resources with information about changes in Python 3.

Documentation:

- What's New In Python 3.0
- Should I use Python 2 or Python 3 for my development activity?

Articles:

- The key differences between Python 2.7.x and Python 3.x with examples
- Supporting Python 3: An in-depth guide

Preparing Windows

In order to do tasks on Windows, you need to install Python and Cmder.

Installing Python 3.7

Download and install Python 3.7. Be sure to check the "Add Python 3.7 to PATH" checkbox.

After installation, check:

```
python --version
```

The output should be: `Python 3.7.7`

Note: If you are not going to use the Mu editor, you can install 3.8 as well, but it is better to look at Mu first. For the basic topics covered in the course, there are practically no changes in 3.8, so you can safely use Python 3.7.

Cmder

Since we work with git on the course, you need to install Cmder to work on the course. To do this, select "Download Full".

After installing Cmder, git is immediately available in it and you need to figure out how it works and configure it to work with Github.

You can also make additional Cmder settings.

Installing Mu

The only caveat with installing Mu on windows is that it is better to install it via pip so that the modules that are installed in pip are visible. That is, you need to install it like this:

```
pip install mu-editor
```

Not through the Windows Installer.

Tips for completing tasks on windows

The pexpect module does not work on Windows, and since it is not needed to perform tasks, this only affects the fact that it will not be possible to repeat the examples from the book.

All other modules work, but with some there are nuances.

graphviz

To complete the tasks in sections 11 and 17, you will need graphviz. And you will need to install the Python module:

```
pip install graphviz
```

And app graphviz.

After installation add graphviz to PATH.

csv

When working with csv on Windows, you always need to specify `newline =""` when opening a file:

```
with open(output, "w", newline="") as dest:
    writer = csv.writer(dest)
```

textfsm

Some of the modules that textfsm uses are not available for Windows. And at the same time, they are not needed for our use of textfsm. For textfsm to work correctly on Windows, you need to comment out some lines in the terminal.py file in the textfsm directory.

How to find the directory textfsm. First, we look at where the site-packages directory is located:

```
In [2]: import sys

In [3]: sys.path
Out[3]:
 ...
 'c:\\users\\nata\\appdata\\local\\programs\\python\\python37\\lib\\site-packages
↪',
 ...
```

Then we go to this directory and inside it we look for the textfsm directory. In the textfsm directory, open the terminal.py file and comment out the lines in this way:

```
# import fcntl
import getopt
import os
```

(continues on next page)

(continued from previous page)

```
import re
import struct
import sys
# import termios
import time
# import tty
```

After that, the code using textfsm should work on Windows.

Working with network equipment

In the last third of the course, you will need network equipment to complete the assignments. You can use real or virtual network equipment and any equipment control system: GNS3, UNL or other.

Where to download GNS3

Option to connect to virtual devices via Loopback

- right click on start and select Device manager
- press Action, select Add legacy hardware
- Click Next
- select “Install the hardware that i manually select from a list”. Click Next
- select Network adapters. Click Next
- select Microsoft in the left column and Microsoft KM-TEST Loopback in the right. Click Next
- click Next
- click Finish
- Be sure to restart the machine afterwards

Then in GNS3 select this loopback interface for connecting the “cloud”.

Preparing Linux

Installing Python 3.7 on Debian 9

If you are installing on a clean OS, it is best to install these packages:

```
sudo apt-get install build-essential checkinstall
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev
sudo apt-get install libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev␣
↪libffi-dev
```

Installing Python 3.7

```
wget https://www.python.org/ftp/python/3.7.3/Python-3.7.3.tgz
tar xvf Python-3.7.3.tgz
cd Python-3.7.3
./configure --enable-optimizations --enable-loadable-sqlite-extensions
sudo make altinstall
```

After that, you can create a virtual environment.

Virtual environment

Installing virtualenvwrapper with pip:

```
python3.7 -m pip install virtualenvwrapper
```

After installation, in ~/.bashrc file in current user's home folder, you need to add several lines:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.7
export WORKON_HOME=~/venv

. /usr/local/bin/virtualenvwrapper.sh
```

Restart command interpreter:

```
exec bash
```

Create a virtual environment using Python 3.7 (the same command will take you to a virtual environment):

```
mkvirtualenv --python=/usr/local/bin/python3.7 pyneng-py3
```

List of modules that need to be installed to complete tasks

```
pip install pytest pytest-clarity pyyaml tabulate jinja2 textfsm pexpect netmiko
```

You also need to install graphviz on the OS (example for debian):

```
apt-get install graphviz
```

11

What's next

Information is usually hard to grasp from the first time. Especially new information.

If you do your homework and make notes during your study, you learn a lot more information than if you just read a book. But most likely, in some way you'll have to read about the same information several times.

Book provides only basics of Python and therefore it is necessary to continue to learn and to repeat already completed topics and to learn new ones. And there are a lot of options:

- automate something at work
- learn more Python for network automation
- learn Python without binding to network equipment

These resources are listed selectively, considering you've already read the book. But in addition, I've made a compilation of resources where other materials can be found.

You have ideas for the scripts you want to write

Most likely, after reading the book there will be ideas what you can automate at work. It's a great option, because it's always easier to learn on a real problem. But it is better to go beyond work tasks and study Python further.

Python allows you to do quite a lot with only basic knowledge. Therefore, with work tasks it is not always possible to increase level of knowledge, but knowing Python better you can usually solve the same problems much more easily. So it's best not to stop and learn.

The following resources are connected to network equipment and generally Python. Depending on from what materials you learn best you can select a book or video course from list

Python for network equipment automation

Books:

- Network Programmability and Automation: Skills for the Next-Generation Network Engineer
- Mastering Python Networking (Eric Chou) - is partly similar to what was discussed in this book but there are many new themes. Plus, examples are considered not only on Cisco equipment but on Juniper and Arista as well.

Blogs - will let you know news in this field:

- Kirk Byers
- Jason Edelman
- Matt Oswalt
- Michael Kashin
- Henry Ölsner
- Mat Wood

Packet Pushers often have podcasts about automation:

- Show 176 – Intro To Python & Automation For Network Engineers
- Show 198 – Kirk Byers On Network Automation With Python & Ansible
- Show 270: Design & Build 9: Automation With Python And Netmiko
- Show 332: Don't Believe The Programming Hype
- Show 333: Automation & Orchestration In Networking
- PQ Show 99: Netmiko & NAPALM For Network Automation

Projects:

- CiscoConfParse - library that parses Cisco IOS configurations. It can: check existing router/switch configurations, get a certain part of configuration, change configuration
- NAPALM - NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) - library that allows working with network equipment of different vendors using a unified API
- NOC Project - NOC is scalable, high-performance and open-source OSS system for ISP, service and content providers
- Requests - library for working with HTTP
- SaltStack - Ansible analogue
- Scapy - network utility that allows you to manipulate network packages

- StackStorm - event-driven automation commonly used for auto-remediation, security responses, facilitated troubleshooting, complex deployments and more
- netdev
- Nornir
- eNMS

General Python

Books

Basic level:

- Think Python - good book on Python basics. There are tasks in the book.
- Python Crash Course: A Hands-On, Project-Based Introduction to Programming - a book on Python basics. Half of the book is dedicated to "standard" description of Python basics and in the second half these bases are used for projects. There are tasks in the book.
- Automate the Boring Stuff with Python - in this book you can find many ideas on automation of daily work. These topics are: working with PDF, Excel, Word, sending letters, working with pictures, working with the web

Medium/advanced level:

- Python Tricks - excellent for 2-3 book on Python. Book describes various aspects of Python and how to use it correctly. The book is fairly new (late 2017) and covers Python 3.
- Effective Python: 59 Specific Ways to Write Better Python (Effective Software Development Series) - book of useful advice on how best to write code
- Dive Into Python 3 - briefly covers fundamentals of Python and then more advanced topics: closure, generators, tests and so on. Book written in 2009 but covers Python 3 and 99% of topics remained unchanged.
- Problem Solving with Algorithms and Data Structures using Python - excellent book on data structures and algorithms. Many examples and homework.
- Fluent Python - excellent book on more advanced topics. Even topics that are obsolete in the current version of Python (asyncio) are worth reading for a perfect explanation of topic.
- Python Cookbook - great recipe book. A huge number of scenarios are considered with solutions and explanations.

Courses

- MITx - 6.00.1x Introduction to Computer Science and Programming Using Python - a very good course in Python. It's a great way to continue your study after

  book. In it you will repeat material on Python basics but from a different angle and learn a lot of new things. There's a lot of practical tasks and it's pretty intense.

- Python от Computer Science Center - an excellent video lecture on Python. There are some basics and more advanced topics
- Talk Python courses

Coding challenges

- Bites of Py
- HackerRank - on this resource tasks are broken down by fields: algorithms, regular expressions, databases and others. But there are basic tasks as well
- CheckIO - online game for Python and JavaScript coders

Podcasts

Podcasts will generally broaden the horizon and give an idea of various Python projects, modules and libraries:

- Talk Python To Me
- Best Python Podcasts

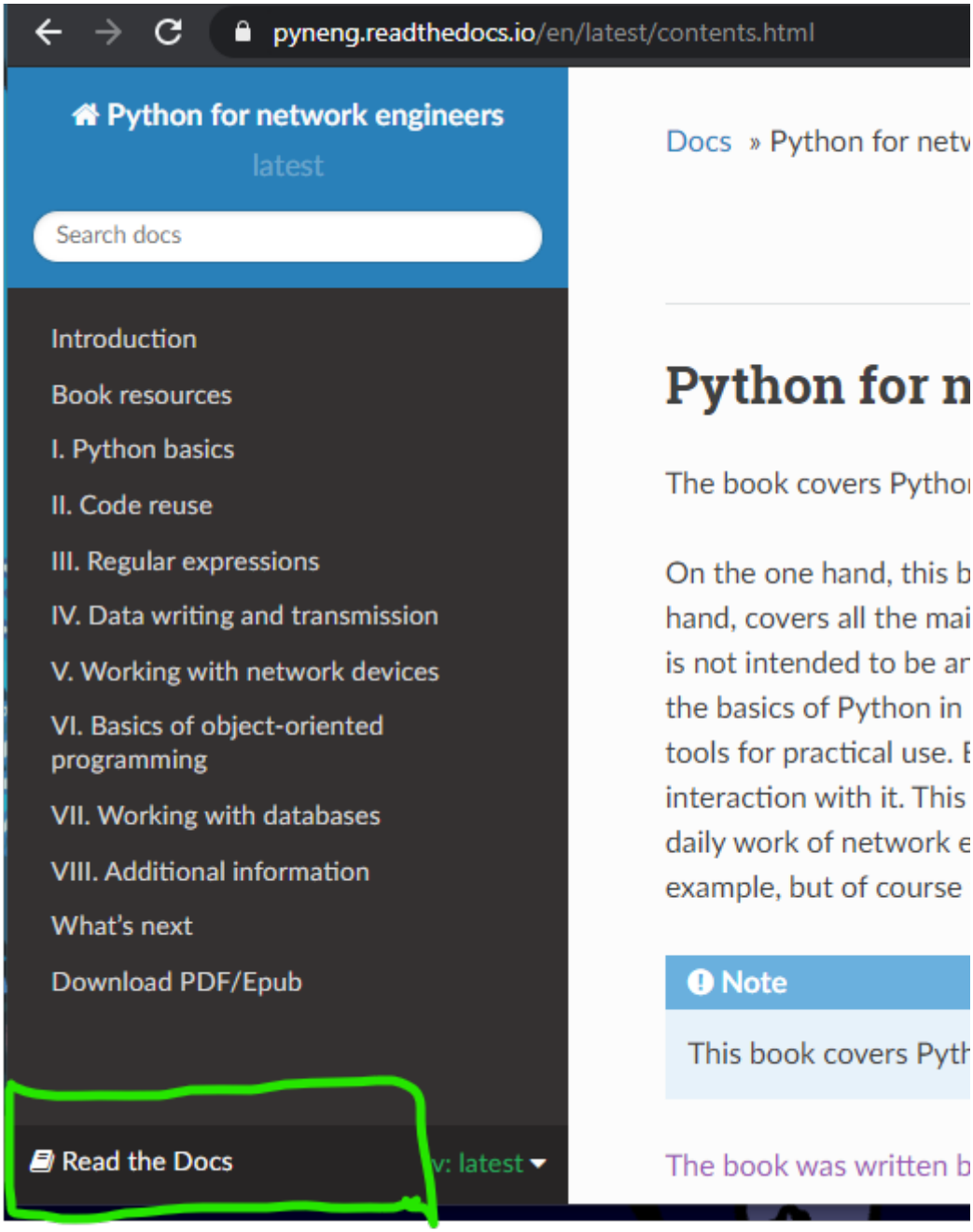
Documentation

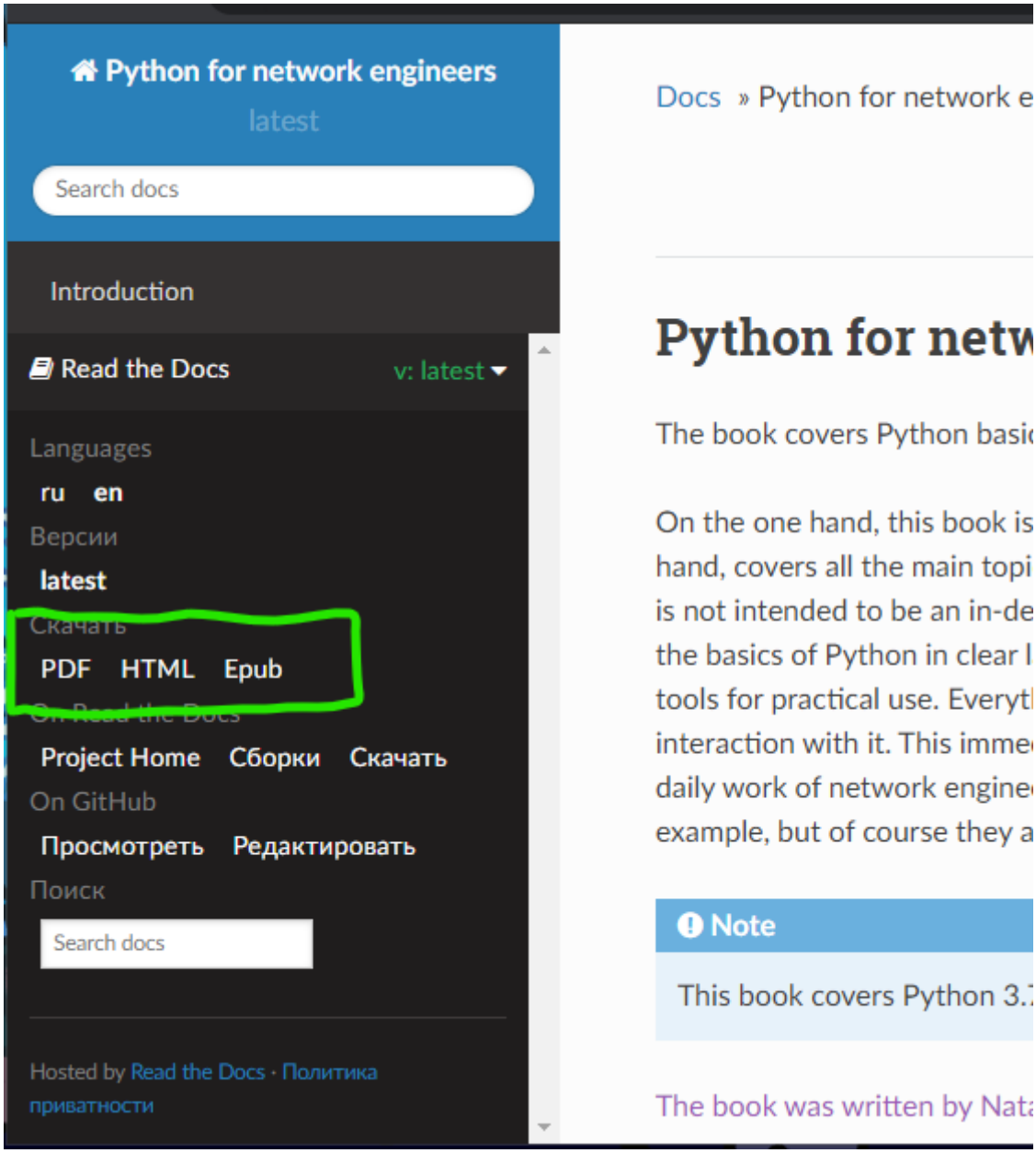
- Official Python documentation
- Python Module of the Week

12

Download PDF/Epub

Click on “Read the Docs” label at the bottom left of the page. In the panel “PDF” link navigates to the PDF file. The build system of readthedocs automatically keeps this file up to date.


pyneng.readthedocs.io/en/latest/contents.html
Python for network engineers
latest
Search docs
Introduction
Book resources
I. Python basics
II. Code reuse
III. Regular expressions
IV. Data writing and transmission
V. Working with network devices
VI. Basics of object-oriented programming
VII. Working with databases
VIII. Additional information
What's next
Download PDF/Epub
Read the Docs
v: latest
Docs » Python for netv
Python for n
The book covers Pytho
On the one hand, this b
hand, covers all the mai
is not intended to be ar
the basics of Python in
tools for practical use. E
interaction with it. This
daily work of network e
example, but of course
Note
This book covers Pyth
The book was written b


Python for network engineers
latest
Search docs
Introduction
Read the Docs
v: latest
Languages
ru en
Версии
latest
Скачать
PDF HTML Epub
Project Home Сборки Скачать
On GitHub
Просмотреть Редактировать
Поиск
Search docs
Hosted by Read the Docs · Политика приватности
Docs » Python for network e
Python for netw
The book covers Python basi
On the one hand, this book is
hand, covers all the main topi
is not intended to be an in-de
the basics of Python in clear l
tools for practical use. Everyt
interaction with it. This imme
daily work of network engine
example, but of course they a
Note
This book covers Python 3.
The book was written by Nata