Python для сетевых инженеров

Release 3.0

Sep 28, 2020

Оглавление

В книге рассматриваются основы Python с примерами и заданиями построенными на сетевой тематике.

С одной стороны, книга достаточно базовая, чтобы её мог одолеть любой желающий, а с другой стороны, в книге рассматриваются все основные темы, которые позволят дальше расти самостоятельно. Книга не ставит своей целью глубокое рассмотрение Python. Задача книги – объяснить понятным языком основы Python и дать понимание необходимых инструментов для его практического использования. Всё, что рассматривается в книге, ориентировано на сетевое оборудование и работу с ним. Это даёт возможность сразу использовать в работе сетевого инженера то, что было изучено на курсе. Все примеры показываются на примере оборудования Cisco, но, конечно же, они применимы и для любого другого оборудования.

Note: В книге используется Python 3.7.

При желании, вы можете сказать “спасибо” автору книги.

1

Introduction

About book

In nutshell, this book is like CCNA but for python. From the one hand, the book is basic enough, so everyone can handle it, from the other hand, the book considers 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.

Why you need to learn programming?

Programming knowledge for network engineer could be compared with necessity of English knowledge. When you know English at least on level which allows to read technical documentation you expand your opportunities at once:

- Much more literature, forums, blogs are available;
- Easier to find solution for almost every question or issue if you ask Google.

Knowledge of programming is very similar in this. For instance, If you know Python at least on basic level you open plenty of new opportunities. Also analogy to English fits here because you can be capable specialist without knowledge of English language. English gives you opportunity but it's not a mandatory requirement.

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 3.5. 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. If task name consist of letter (for ex. 5.2a) it's better to complete this task after tasks without letters. Usually, tasks with letter are more complex and they continue the idea of task without letter. If possible it's better to do tasks one by one. There are no answers in the book because, unfortunately, when answers are present there is a great temptation to look at them instead of solving complex task on your own. Of course, sometimes it's difficult to find a solution - try to set this task aside, ask question in Slack and do another task.

Note: Answers to almost all questions can be found in Stack Overflow. So, if you see this website in Google search results it means with high probability the answer is found. Of course, it's better to ask Google in English - there are a lot of materials on Python and in general, it's easy to find a tip.

Answers can show how to solve task in another way or how to solve it in better way. But no need to worry about it because in the next chapters you will likely meet an example with proper code.

Quiz

Some chapters have additional questions:

- Data types. Part 1
- Data types. Part 2
- Compound statements. Part 1
- Compound statements. Part 2
- Functions and modules. Part 1
- Functions and modules. Part 2
- Regular expressions. Part 1
- Regular expressions. Part 2
- Data bases

These quiz can be considered an evaluation test or as a task. It's useful to answer to these questions after reading of corresponding chapter. They will help you recall chapter's material and also see different aspects of Python usage in practice. First, try answer on your own and only then check answers in IPython on questions which you are doubting.

Presentations

There are presentations for each book chapter in repository. It's convenient way to repeat and go through the information. If you know basics of Python it worth getting through it.

All presentations can be downloaded from special repository.

Book formats

Book is available in PDF and Epub formats. Both of them are being updated automatically, therefore the content is equal.

Discussion

Discussions of book, tasks and other related topics are taken place in Slack. Also write to Slack in case of questions, suggestions, comments and observations on book.

Resources for study course

Here are the links to all resources which will be helpful during study process:

- Variants of virtual machines for this course;
- Repositories with examples and tasks
- Quizzes;
- Chat PyNEng in Slack;

Almost every book chapter has subchapter “Additional materials” 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.

Frequently Asked Questions (FAQ)

Here are some of the most frequently asked questions in reading books.

How does this differ from the regular Python introductory course?

The main differences are three:

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

I’m a network engineer. What do I need this book for?

First of all, to automate routine tasks. Automation provides several advantages:

- High-level thinking - it’s easier to rise above everything when you free of routine work. You’ll have time and opportunity to think of improvements;
- Trust - you won’t be afraid to make changes that are often risky because the network is the backbone of every applications and the cost of error is high;
- A coherent configuration - you will able to automatically create network configuration files, from users and interface descriptions to security functionality, and you’ll be less worried about whether you have forgotten something.

Of course, it won't be that after reading the book you "automate everything and happiness will come" but this is a step in this direction. I am in no way encouraging for all automation to be done via bunch of scripts. If there is some software that solves your needs, that's great, use it. But if there isn't or if you are just haven't thought about it yet, try to start with a simple - Ansible, for example, allows to perfrom many tasks almost "out of the box".

Why then learn Python? The fact is that the same Ansible won't solve everything. And you may need to add some functionality independently. In addition, apart of equipment configuration adjustment, there are daily routine tasks that can be automated by Python. Let's just say that if you don't want to deal with Python, but want to automate setup and operation processes, please turn attention on Ansible. Even "out of the box" it will be very useful. Later, if you get taste for it and you want to add something that missed in Ansible, come back :-)

And yes, this course is not only about how to use Python for network equipment configuration and connecton to it. It's also about how to solve tasks that are not connected to the equipment. For example, change something in multiple configuration files or parse log-file - Python will help you solve these tasks.

Why is this book specifically for network engineers?

There are several reasons:

- Network engineers already have experience in IT, and some of the 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 the 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 the subject area in the 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 the 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, for example, 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.

The module I want does not support Python 3

There are several options:

- Try to find an alternative module that supports Python 3 (not necessarily the latest version of the language);
- Try to find a community version of this module for Python 3. There may not be an official version but the community could translate it independently to version 3, especially if this module is popular;
- Use Python 2.7, nothing terrible will happen. If you're not going to write a huge application but you're just using Python to automate your problems, Python 2.7 will definitely work.

I don't know if I need this.

I, of course, think you need this :-) Otherwise I wouldn't be writing this book. You don't necessarily want to go into all this stuff, so you might want to start with Ansible. Perhaps you'll have enough of it for a long time. Start with simple "show" commands, try to connect first to test equipment (virtual machines), then try to execute "show" command on real network, on 2-3 devices, then on more. If that's enough for you, you can stop there. The next step is to try using Ansible to generate configuration patterns.

Why would a network engineer need programming?

In my opinion, programming is very important for a network engineer, not because everybody's talking about it right now or because everybody's scaring with SDN, job loss or something like that, but because the network engineer is constantly facing with:

- Routine tasks

- Problems and solutions to be tested;
- Large quantity of monotonous and repetitive tasks;
- Large quantity of equipment;

At present, a large amount of equipment still offers us only the command line interface and unstructured output of commands. The software is often limited to a vendor, expensive and has reduced possibilities - we end up doing the same thing over and over again by hand. Even banal things like sending the same show command to 20 devices are not always easy to do. Suppose your SSH client supports this feature. And what if you now need to analyze the output? We are limited by the means we have been given and knowledge of programming, even the most basic, allows us to expand our means and even create new ones. I don't think everyone should be rushing to learn programming but for an engineer that's a very important skill. It's for the engineer, not everyone.

Now clearly there is a tendency that can be described by the phrase « everybody is learning to code» and it is, in general, good. But programming is not something elementary, it's difficult, it's time-consuming, especially if you've never had relation to technology world. It might give an impression that it's enough to pass "these courses" and after 3 months you are great programmer with high salary. No, this book is not about that :-) We don't talk about programming as a profession in it and we don't set such a goal, we're talking about programming as a tool such as knowing CLI Linux. It's not that engineers are anything special but, in general:

- They already have technical education;
- Many work with the command line, in one way or another;
- They have encountered at least one programming language;
- They have an «engineering mindset».

This does not mean that everybody else is «not allowed». It will just be easier for the engineers.

Will the book ever be charged with fee?

No, this book will always be free. I read a paid online course «Python for network engineers» (in Russian), but this will not affect this book - it will always be free.

Gratitude

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 the 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 the 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 the book - thanks to you now the book not only has fewer typographical errors and typos, but also the contents of the book have improved.

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

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

2

I. Python basics

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

- Python data types;
- How to create basic scripts;
- Compound statements;
- Working with files;

1. Preparation for work

OS and editor

You can choose any OS and any editor but it is desirable to use Python version 3.7 because the book uses this version.

All of the examples in the book were run on Debian, but other operating systems may have a slightly different output. You can use Linux, macOS or Windows to perform tasks from a book. However, it is worth considering that, for example, Ansible can only be installed on Linux/macOS.

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

Mu editor

It is worth mentioning that Mu editor it is an editor for beginners to learn Python (it supports only Python).

On the one hand, there's nothing superfluous about it that can initially be very distracting and confusing. At the same time, it has important features such as checking code against PEP 8 and debugger. Plus, Mu runs on different operating systems (macOS, Windows, Linux).

Note: Video tutorials on Mu: Basics of Mu, Using Debugger in Mu

---

IDE PyCharm

PyCharm — is an integrated development environment for Python. For beginners it may be difficult because of the plethora of settings but it depends on personal preferences. Pycharm supports a huge number of features, even in the 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 the 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: The 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 the 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 the command failed, the pip is not installed. Pip installation is described in documentation

Module installation

The command to install modules `pip install`:

```
$ pip install tabulate
```

Removing the package is done as follows:

```
$ pip uninstall tabulate
```

In addition, it is sometimes necessary to update the package:

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

pip or pip3

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

A variant 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 variant 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 the system uses pip3, then every time a Python module is installed in the 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 - if, for example, one project requires a 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 one of them. To start with, you can use virtualenvwrapper and then eventually you can figure out which options are still available.

---

virtualenvwrapper

Virtual environments are created with virtualenvwrapper.

Installing virtualenvwrapper with pip:

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

After installation, in the . bashrc file in the 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 the virtualenvwrapper documentation. The environment variable VIRTUALENVWRAPPER_PYTHON points to the Python command line binary file, WORKON_HOME – points to the location of virtual environments. The third line indicates location of the script installed with the virtualenvwrapper package. To start virtualenvwrapper.sh script work with virtual environments, bash must be restarted.

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

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

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

To move to the created virtual environment, you must run the "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 the 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
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 the specified directory and all necessary subdirectories within it if they have not been created.

The 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 must execute the command:

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

To exit the virtual environment, use command “deactivate”:

```
$ deactivate
```

More about the venv module in documentation.

Package installation

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

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

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

Note: The book will use ipython instead of the standard Python interpreter

---

Additional material

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:

- Define the OS you want to use:
- since all examples in the book are Linux-oriented (Debian), it is desirable to use it
- it is desirable to use a new virtual machine, so you can safely experiment
- Install Python 3.7. Verify that Python and pip are installed
- Create a virtual environment
- Choose the editor

2. Using Git and Github

There are a lot of tasks in the 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 the future. Tasks and examples from the book are in a separate repository on Github. They can be downloaded as a zip archive but it is better to work with the repository using Git, then you can see the changes made and easily update the 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 the practical side of the question and it says:

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

There will be no much theory in this subsection but references to useful resources are given. Try doing all the basic settings for the tasks and then continue reading the book. And at the end, when the 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 the documentation and all its versions;
- to store schemes and all its versions;
- to store the code and its versions.

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

- hosting 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 the GNU GPL v2 license. It can:

- track changes in files;
- store multiple versions of the same file;
- cancel the changes made;

- record who made the changes and when.

Git stores the changes as a snapshot of the 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 the user name and e-mail that will be used to synchronize the local repository with the Github repository:

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

See the Git settings:

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

Repository initialization

The repository is initialized using the "git init" command:

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

After executing this command, the current directory creates .git folder containing the 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 the status of the current repository. To do this, you need to install a special utility that will show the status of the repository. To install the utility, copy its repository to the user's home directory under which you work:

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

(continues on next page)

(continued from previous page)

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 the changes, restart bash:

```
exec bash
```

In my configuration the 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 the repository.

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

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

If you go to the Git repository:

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

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

Working with Git

Various commands are used to control Git, the meaning of which is explained below.

git status

When working with Git it is important to understand the current status of the 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 the 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 the “git add” command to start Git tracking them.

Create README file and add “test” line to it

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

After that, the invitation looks like this

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

The invitation shows that there are two files that Git is not following

```
[~/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 cancel changes not only in the current file session but also in the past. Note that Git reports there are files that it does not follow and tells you using which command you can start following.

File .gitignore

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

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

```
*.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 the repository .gitignore, the files that are listed in it are being ignored.

git add

The "git add" command is used to start Git following files.

You can specify that you want to follow a particular file

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

Or all the 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 the files are in a section called “Changes to be committed”.

git commit

After all the necessary files have been added in staging, you can commit the changes. Staging is a collection of files that will be added to the next commit. The “git commit” command has only one obligatory parameter - the 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
```

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

Additional facilities

git diff

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

The “git status” command 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")
```

The “git diff” command shows what changes have been made since the 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 the 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 the 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

The “git log” command shows when the 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, the command displays all commits starting from the nearest time. With the help of additional parameters it is possible not only to look at the information about commits but also what changes have been made.

The `-p` flag 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

To start working with Github you must 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"
```

All questions need to be pressed Enter (it is more secure to use the passphrase key but it is possible without it, if you press Enter when asked then passphrase will not be requested from you permanently during operations with the 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 to copy it:

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

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

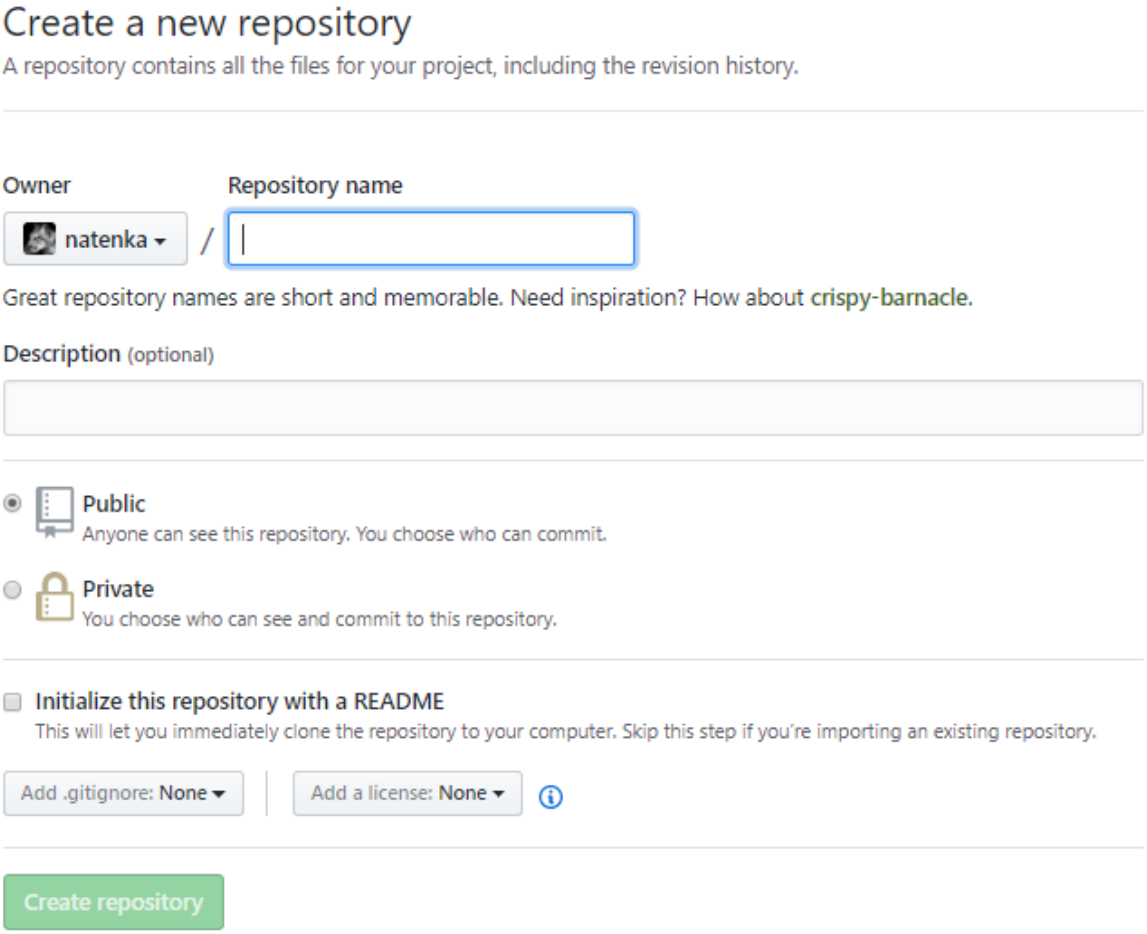
This chapter describes how to work with a repository on your local machine.

Creating a Github repository

To create a Github repository you need:

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

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



Cloning a Github repository

To work locally with the repository, it must 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:

- The pyneng user name for your Github user name;

- The online-2-natasha-samoylenko repository name for your Github repository.

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

Working with the repository

The previous command not only copied the 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 the repository
- Github repository is configured as a remote for local repository

Now you have a complete local Git repository where you can work. Typically, the 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 the Github repository with “git push” command

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

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

Synchronizing local repository with remote repository

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

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

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

If there were changes, the 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 the current directory are added by `git add .` command.

Commit

You must specify a message when you are running a commit. It is better if the 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

The “git push” command 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 the examples and tasks of the 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 the 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, the output would be:

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

If there were updates, the 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: Наташа Самойленко <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 the R1 router.
```

In this command `-p` flag indicates that the output of the Linux diff utility should be displayed for changes, not just the 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 the number of commands but this is not always convenient. Before executing “git pull” you can see what changes have been made since the last synchronization.

The following command shall be used:

```
$ git log -p ..origin/master
commit 4c1821030d20b3682b67caf362fd777d098d9126
```

(continues on next page)

(continued from previous page)

```
Author: Наташа Самойленко <nataliya.samoylenko@gmail.com>
Date:   Mon May 29 07:53:45 2017 +0300

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 the PDF versions of configuration manuals of the tools that␣
↪are used on the course.
```

In this case, the changes were in only one file. This command will be very useful to see what changes have been made to the 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 the local copy of the repository

---

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

Additional material

Documentation:

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

About Git/GitHub:

- GitHowTo - interactive howto in Russian;
- git/github guide. a minimal tutorial - minimum knowledge required to work with Git и GitHub;
- Pro Git book. The same book in Russian;
- Version control system (GIT) (course on Hexlet).

Tasks

All tasks and auxiliary files can be downloaded from repository. If you have tasks with letters (for example, 5.2a) in a section, it is better to do tasks without letters and then with letters. Tasks with letter tend to be slightly more complex than letter-free tasks and they develop or complicate the idea in the respective task without letter.

Note: For example, in the section there are tasks 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. First it is better to complete 5.1, 5.2, 5.3 and then 5.2a, 5.2b, 5.3a

---

If you can do a task with letters right away, it is better to do it in order.

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 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 examines:

- Python syntax
- Work in interactive mode
- Python variables

Python syntax

The first thing that meets the eye when it comes to Python syntax is that indentation matters:

- It determines which code enters 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 the if/else construction has not yet been considered, it is likely that the meaning of the code will be understood.

Python understands which lines refer to “if” on the indentation basis. The execution of a block `if a > b` ends when another string with the same indent as the string `if a > b` appears. Similarly to the 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 the editor so that the tab is 4 spaces - then when using the tab key, 4 spaces will be placed instead of 1 tab sign).
- The number of spaces must be the same in one block (it is better to have the same number of spaces in the whole code - the popular option is to use 2-4 spaces, for example, this book uses 4 spaces).

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

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

Comments

When writing code you often need to leave a comment, for example, to describe the features of the 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 the pound sign. Note that the comment can be in the line where the code itself is or in a separate line.

If it is necessary to write several lines with comments in order to not put pound 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 the code and to exclude the execution of a particular line or block of code (i.e., to comment it).

Python interpreter. Ipython

The interpreter makes it possible to receive an instant response to the executed actions. You can say that the interpreter works as the 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 the previous section, a standard interpreter was called to verify the installation of Python. There is also an improved interpreter IPython. Ipython allows much more than the 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 variant;
- More structured and understandable output of commands;
- Automatic indentation in cycles and other objects;
- You can either walk through the command execution history or watch it with the %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]:
```

The “quit” command is used to exit. The following is how IPython will be used.

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

Example of string output by function print():

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

When a loop is created in the interpreter, for example, the invitation changes to ellipsis inside the loop. To complete the 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 on you can view the 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 the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the 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 the 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 the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the 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 the string S with leading and trailing
whitespace removed.
If chars is given and not None, remove characters in chars instead.
Type:      method_descriptor
```

print()

The `print()` function displays information on a standard output (the current terminal screen). If you want to get a string, you must place it in quotation marks (double or single). If you want to derive, 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 line feed. If it is necessary that after the output of each expression there would be no line feed, an additional “end” argument should be specified as the last expression in print().

See also:

Additional parameters of print function *Print*

dir()

The `dir()` function can be used to see what attributes (variables tied to the object) and methods (functions tied to the 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 the 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 do dir() with no value, it shows the existing methods, attributes, and variables defined in the current session of the interpreter:

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

For example, after creating the 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

The %time command 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, the 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 parentheses, 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 the '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:

- The name of the variable can consist only of letters, digits and an underscore;
- The 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 the memory area. This can be demonstrated by using id() which shows the 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 the three references “a”, “b” and “c” point. Concerning numbers Python has one feature that can be slightly misunderstood: 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 the number in the generated array.

Note: This feature is specific to the implementation of Cpython which is discussed in the 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, the identifiers are all the same (in this variant 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 the 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 auxiliary files can be downloaded from repository. If you have tasks with letters (for example, 5.2a) in a section, it is better to do tasks without letters and then with letters. Tasks with letter tend to be slightly more complex than letter-free tasks and they develop or complicate the idea in the respective task without letter.

Note: For example, in the section there are tasks 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. First it is better to complete 5.1, 5.2, 5.3 and then 5.2a, 5.2b, 5.3a

---

If you can do a task with letters right away, it is better to do it in order.

Task 3.1

Install IPython in a virtual environment or globally in system if virtual environments are not used. After installation, by *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 standard data types:

- Numbers
- Strings
- Lists
- Dictionaries
- Tuples
- Sets
- Boolean (logical data type)

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

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

Content of section:

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

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

The round() function can round the numbers to the required number of characters:

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

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

Remainder of the 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 the number system:

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

(continues on next page)

(continued from previous page)

```
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, the 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
```

(continues on next page)

(continued from previous page)

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

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

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

Strings

The string in Python is:

- sequence of characters enclosed in quotation marks
- 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
```

(continues on next page)

(continued from previous page)

```
 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, the string repeats the 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. The slicing starts with the 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, the slice is until the end of the string:

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

Slice the last three character of string:

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

You can also specify a step in the 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: The entries `a[::]` and `a[:]` give the same result but the 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 the number of characters in a string:

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

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

Note: The function and method differ in that the method is tied to a particular type of object and the function is generally more universal and can be applied to objects of different types. For example, the `len` function can be applied to strings, lists, dictionaries and so on, but the `startswith` method only applies to strings.

Полезные методы для работы со строками

При автоматизации очень часто надо будет работать со строками, так как конфигурационный файл, вывод команд и отправляемые команды - это строки.

Знание различных методов (действий), которые можно применять к строкам, помогает более эффективно работать с ними.

Строки неизменяемый тип данных, поэтому все методы, которые преобразуют строку возвращают новую строку, а исходная строка остается неизменной.

Методы upper, lower, swapcase, capitalize

Методы `upper()`, `lower()`, `swapcase()`, `capitalize()` выполняют преобразование регистра строки:

```
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'
```

Очень важно обращать внимание на то, что часто методы возвращают преобразованную строку. И, значит, надо не забыть присвоить ее какой-то переменной (можно той же).

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

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

Метод count

Метод `count()` используется для подсчета того, сколько раз символ или подстрока встречаются в строке:

```
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
```

Метод find

Методу `find()` можно передать подстроку или символ, и он покажет, на какой позиции находится первый символ подстроки (для первого совпадения):

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

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

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

Если совпадение не найдено, метод find возвращает `-1`.

Методы startswith, endswith

Проверка на то, начинается или заканчивается ли строка на определенные символы (методы `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

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

Метод replace

Замена последовательности символов в строке на другую последовательность (метод `replace()`):

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

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

Метод strip

Часто при обработке файла файл открывается построчно. Но в конце каждой строки, как правило, есть какие-то спецсимволы (а могут быть и в начале). Например, перевод строки.

Для того, чтобы избавиться от них, очень удобно использовать метод `strip()`:

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

In [48]: print(string1)

    interface FastEthernet0/1

```

(continues on next page)

(continued from previous page)

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

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

По умолчанию метод strip() убирает пробельные символы. В этот набор символов входят: \t\n\r\f\v

Методу strip можно передать как аргумент любые символы. Тогда в начале и в конце строки будут удалены все символы, которые были указаны в строке:

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

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

Метод strip() убирает спецсимволы и в начале, и в конце строки. Если необходимо убрать символы только слева или только справа, можно использовать, соответственно, методы `lstrip()` и `rstrip()`.

Метод split

Метод `split()` разбивает строку на части, используя как разделитель какой-то символ (или символы) и возвращает список строк:

```
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']
```

В примере выше `string1.split()` разбивает строку по пробельным символам и возвращает список строк. Список записан в переменную commands.

По умолчанию в качестве разделителя используются пробельные символы (пробелы, табы, перевод строки), но в скобках можно указать любой разделитель:

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

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

В списке commands последний элемент это строка с вланами, поэтому используется индекс -1. Затем строка разбивается на части с помощью split `commands[-1].split(',')`. Так как, как разделитель указана запятая, получен такой список `['10', '20', '30', '100-200']`.

Полезная особенность метода split с разделителем по умолчанию — строка не только разделяется в список строк по пробельным символам, но пробельные символы также удаляются в начале и в конце строки:

```
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']
```

У метода `split()` есть ещё одна хорошая особенность: по умолчанию метод разбивает строку не по одному пробельному символу, а по любому количеству. Это будет, например, очень полезным при обработке команд show:

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

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

А вот так выглядит разделение той же строки, когда один пробел используется как разделитель:

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

Форматирование строк

При работе со строками часто возникают ситуации, когда в шаблон строки надо подставить разные данные.

Это можно делать объединяя, части строки и данные, но в Python есть более удобный способ — форматирование строк.

Форматирование строк может помочь, например, в таких ситуациях:

- необходимо подставить значения в строку по определенному шаблону
- необходимо отформатировать вывод столбцами
- надо конвертировать числа в двоичный формат

Существует несколько вариантов форматирования строк:

- с оператором % — более старый вариант
- метод format() — относительно новый вариант
- f-строки — новый вариант, который появился в Python 3.6.

Несмотря на то, что рекомендуется использовать метод format, часто можно встретить форматирование строк и через оператор %.

Форматирование строк с методом format

Пример использования метода format:

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

Специальный символ {} указывает, что сюда подставится значение, которое передается методу format. При этом каждая пара фигурных скобок обозначает одно место для подстановки.

Значения, которые подставляются в фигурные скобки, могут быть разного типа. Например, это может быть строка, число или список:

```
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]
```

С помощью форматирования строк можно выводить результат столбцами. В форматировании строк можно указывать, какое количество символов выделено на данные. Если количество символов в данных меньше, чем выделенное количество символов, недостающие символы заполняются пробелами.

Например, таким образом можно вывести данные столбцами одинаковой ширины по 15 символов с выравниванием по правой стороне:

```
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
```

Выравнивание по левой стороне:

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

Шаблон для вывода может быть и многострочным:

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

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

IP address:
10.1.1.1
```

С помощью форматирования строк можно также влиять на отображение чисел.

Например, можно указать, сколько цифр после запятой выводить:

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

С помощью форматирования строк можно конвертировать числа в двоичный формат:

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

При этом по-прежнему можно указывать дополнительные параметры, например, ширину столбца:

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

А также можно указать, что надо дополнить числа нулями, вместо пробелов:

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

В фигурных скобках можно указывать имена. Это позволяет передавать аргументы в любом порядке, а также делает шаблон более понятным:

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

Еще одна полезная возможность форматирования строк - указание номера аргумента:

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

За счет этого, например, можно избавиться от повторной передачи одних и тех же значений:

```
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
```

В примере выше октеты адреса приходится передавать два раза - один для отображения в десятичном формате, а второй - для двоичного.

Указав индексы значений, которые передаются методу format, можно избавиться от дублирования:

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

Форматирование строк с помощью f-строк

В Python 3.6 добавился новый вариант форматирования строк - f-строки или интерполяция строк. F-строки позволяют не только подставлять какие-то значения в шаблон, но и позволяют выполнять вызовы функций, методов и т.п.

Во многих ситуациях f-строки удобней и проще использовать, чем format, кроме того, f-строки работают быстрее, чем format и другие методы форматирования строк.

Синтаксис

F-строки — это литерал строки с буквой f перед ним. Внутри f-строки в паре фигурных скобок указываются имена переменных, которые надо подставить:

```
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'

Аналогичный результат с format можно получить так:
``"IP: {ip}, mask: {mask}".format(ip=ip, mask=mask)``.
```

Очень важное отличие f-строк от format: f-строки — это выражение, которое выполняется, а не просто строка. То есть, в случае с ipython, как только мы написали выражение и нажали Enter, оно выполнилось и вместо выражений {ip} и {mask} подставились значения переменных.

Поэтому, например, нельзя сначала написать шаблон, а затем определить переменные, которые используются в шаблоне:

```
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
```

Кроме подстановки значений переменных, в фигурных скобках можно писать выражения:

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

In [6]: second_name = 'Shakespeare'

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

После двоеточия в f-строках можно указывать те же значения, что и при использовании format:

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

```

(continues on next page)

(continued from previous page)

```
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: Так как для полноценного объяснения f-строк, надо показывать примеры с циклами и работой с объектами, которые еще не рассматривались, это тема также есть в разделе *Formatting lines with f-strings* с дополнительными примерами и пояснениями.

Объединение литералов строк

В Python есть очень удобная функциональность — объединение литералов строк.

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

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

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

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

Можно даже переносить составляющие строки на разные строки, но только если они в скобках:

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

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

Этим очень удобно пользоваться в регулярных выражениях:

```
regex = ('(\S+) +(\S+) +'
         '\w+ +\w+ +'
```

(continues on next page)

(continued from previous page)

```
        '(up|down|administratively down) +'
        '(\w+)')
```

Так регулярное выражение можно разбивать на части и его будет проще понять. Плюс можно добавлять поясняющие комментарии в строках.

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

Также этим приемом удобно пользоваться, когда надо написать длинное сообщение:

```
In [7]: message = ('При выполнении команды "{}" '
   ...: 'возникла такая ошибка "{}".\n'
   ...: 'Исключить эту команду из списка? [y/n]')

In [8]: message
Out[8]: 'При выполнении команды "{}" возникла такая ошибка "{}".\nИсключить эту␣
↪команду из списка? [y/n]'
```

List

The 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']
```

Creation of a list by means of a literal:

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

Note: The literal is an expression that creates the object.

Create a list using the **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 the 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 mutual, the 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 the nested lists:

```
In [16]: interfaces = [['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up
↪'],
```

(continues on next page)

(continued from previous page)

```
   ....: ['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 the number of items in the list:

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

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

And the sorted() function sorts list items in ascending order and returns a new list with sorted items:

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

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

Useful methods for working with lists

The list is a mutable data type, so it is important to note that most methods for working with lists change the list on the spot without returning anything.

join()

The **join()** method collects a list of strings into one string with the separator specified before join:

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

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

Note: The join method actually relates to strings but since the value must be given as a list, it is

considered in this section.

append()

The **append()** method adds the specified item to the end of the 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']
```

The append() method changes the list on the spot and does not return anything.

extend()

If you want to combine two lists you can use two methods: the **extend()** method and the addition operation.

These methods have an important difference: extend() changes the list to which the method is applied and addition returns a new list that consists of two.

The extend() method:

```
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 line Out appeared. This means that the result of the summation can be assigned to the variable:

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

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

pop()

The **pop()** method removes the item that corresponds to the specified number. But, importantly, the 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 the list is deleted.

remove()

The **remove()** method removes the specified item.

remove() does not return the 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() method you must specify the item to be deleted, not its number in the list. If item number is specified, error occurs:

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

(continues on next page)

(continued from previous page)

```
----> 1 vlans.remove(-1)

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

index()

The **index()** method is used to check under which number the item is stored in the list:

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

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

insert()

The **insert()** method allows you to insert an item into a specific place in the 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()

The sort() method sorts on the spot:

```
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 the dictionary are pairs `key: value`
- values are accessible by key, not by number as in lists
- the entries in the dictionary display in the order they were defined.
- since dictionaries are mutable, the dictionary items can be changed, added, removed
- the 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,
    'port':'G1/0/11'
}
```

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

In the 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',
        '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 the 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'
```

The sorted() function sorts the 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']
```

Useful methods for working with dictionaries

clear()

The **clear()** method allows to clear the dictionary:

```
In [1]: london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco',
↪'model': '4451', 'ios': '15.4'}

In [2]: london.clear()

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

copy()

The **copy()** method allows to create a full copy of the dictionary.

If one dictionary is equal to the 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 the dictionary. And when you change the “london” dictionary the “london2” dictionary changes as well because it’s a link to the same object.

Therefore, if you want to make a copy of the 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 the 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'
```

The **get()** method query for the 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
```

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

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

setdefault()

The **setdefault()** method searches for the 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 the key is present, setdefault() returns the value that corresponds to it:

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

The second argument allows to specify which value should correspond to the 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'}
```

The setdefault() method replaces this construction:

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

(continues on next page)

(continued from previous page)

```
    ...: 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 display keys, values, and key-value pairs of the 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 the objects, but they don't make a copy of them.

Using the example of 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 the keys variable corresponds to view dict_keys, in which three keys: name, location and vendor.

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

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

```

(continues on next page)

(continued from previous page)

```
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 the dictionary changes, it is enough to convert view to the 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

The update() method allows you to add the 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',
```

(continues on next page)

(continued from previous page)

```
 'location': 'London Str',
 'vendor': 'Cisco',
 'ios': '15.4'}
```

Dictionary creation options

Literal

A dictionary can be created with the help of a literal:

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

dict

The constructor **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), the **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
```

(continues on next page)

(continued from previous page)

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

By default fromkeys() sets None value. But you can also give your own version of the 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 the 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': []}

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 the lists, the others are updated.

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

Tuple

The tuple in Python is:

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

Roughly speaking, a tuple is a list that can't be changed. I mean, the tuple only has reading rights. 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 the list:

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

In [5]: tuple_keys = tuple(list_keys)

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

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

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

But since the 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
```

The sorted() function sorts the 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

A set is a mutable unordered data type. The set always contains only unique elements.

A set in Python is a sequence of elements that are separated by a comma and placed in curly brackets.

A 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}
```

Useful methods for working with sets

add()

The `add()` method adds an item to the set:

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

In [2]: set1.add(50)

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

discard()

The `discard()` method allows deleting elements without showing an error if there is no element in the 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()

The method `clear()` empties the 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 us 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 constants have the same 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 the 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

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>()
```

(continues on next page)

(continued from previous page)

```
----> 1 int('a')

ValueError: invalid literal for int() with base 10: 'a'
```

The error is perfectly logical. We're trying to convert string 'a' into decimal format.

And if the example here is probably stupid, however, 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 it, it would be nice to be able to check what we're working with.

isdigit()

Python has such methods. For example, the `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()

The `isalpha()` method 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()

The `isalnum()` method 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 output different types of objects. For example, if an there is one object a string is returned, if several a tuple is returned.

We have to construct the program in different ways, depending on whether a string or a tuple has been returned.

The `type()` function can help:

```
In [13]: type("string")
Out[13]: str

In [14]: type("string") is 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)) is tuple
Out[16]: True

In [17]: type((1,2,3)) is list
Out[17]: False
```

Additional material

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:

- Примеры использования форматирования строк
- Документация по форматированию строк
- Python 3's f-Strings: An Improved String Formatting Syntax (Guide)
- Python String Formatting Best Practices

Tasks

All tasks and auxiliary files can be downloaded from repository. If you have tasks with letters (for example, 5.2a) in a section, it is better to do tasks without letters and then with letters. Tasks with letter tend to be slightly more complex than letter-free tasks and they develop or complicate the idea in the respective task without letter.

Note: For example, in the section there are tasks 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. First it is better to complete 5.1, 5.2, 5.3 and then 5.2a, 5.2b, 5.3a

---

If you can do a task with letters right away, it is better to do it in order.

Task 4.1

Using the prepared *nat* string, get a new string that has GigabitEthernet in interface name instead of FastEthernet. Restriction: All tasks must be performed using only covered topics.

```
nat = "ip nat inside source list ACL interface FastEthernet0/1 overload"
```

Task 4.2

Convert *mac* string from XXXXXX:XXXX format to XXXXXX.XXXXX.XX format. Restriction: All tasks must be performed using only covered topics.

```
mac = "AAAA:BBBB:CCCC"
```

Task 4.3

Get from *config* string such Vlan list:

```
["1", "3", "10", "20", "30", "100"]
```

Restriction: All tasks must be performed using only covered topics.

```
config = "switchport trunk allowed vlan 1,3,10,20,30,100"
```

Task 4.4

List *vlans* is a list of VLANs collected from all network devices, so list has duplicate VLAN numbers. From list you need to get a unique list of VLANs sorted in ascending order. You cannot remove specific vlans manually to get the final list.

Restriction: All tasks must be performed using only covered topics.

```
vlans = [10, 20, 30, 1, 2, 100, 10, 30, 3, 4, 10]
```

Task 4.5

From *command1* and *command2* strings get list of VLANs that are both in *command1* and in *command2* (intersection).

The result should be a list: `["1", "3", "8"]`

Restriction: All tasks must be performed using only covered topics.

```
command1 = "switchport trunk allowed vlan 1,2,3,5,8"
command2 = "switchport trunk allowed vlan 1,3,8,9"
```

Task 4.6

Process ospf_route string and display information to standard output stream as:

```
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 performed using only covered topics.

```
ospf_route = "       10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0"
```

Task 4.7

Convert *mac* MAC-address to a binary string of this type: `101010101010101010111011101110111001100110011001100`

Restriction: All tasks must be performed using only covered topics.

```
mac = "AAAA:BBBB:CCCC"
```

Task 4.8

Convert IP address in *ip* variable to a binary format and display output in columns in this way:

- first string should be decimal bytes valuesв

- second string binary values

The output should be ordered as in example:

- by columns
- column width of 10 characters (in binary format, you have to add two spaces between columns

Example of output for address 10.1.1.1:

```
10        1         1         1
00001010  00000001  00000001  00000001
```

Restriction: All tasks must be performed using only covered topics.

```
ip = "192.168.3.1"
```

5. Basic scripts creation

Generally speaking, the script is a regular file. This file stores the sequence of commands that you want to execute.

Let's start with basic script and display several strings on the 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 the list are combined into a string that is separated by \n and the VLAN number is inserted into the string using string formatting.

After this you must save the file and go to the command line.

This is the execution of the script:

```
$ python access_template.py
switchport mode access
switchport access vlan 5
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

It is not necessary to specify extension .py for a file.

But if you are using Windows it is better to do so because Windows uses a file extension to determine how to process a file.

All the scripts that will be created in this course have an extension. 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 the file executable (for Linux)
- the first line of the 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',
                   '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
```

Transferring argument to the script (argv)

Very often the script solves some common problem. For example, the 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 the script.

It will be much better to pass the file name as the 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']
```

(continues on next page)

(continued from previous page)

```
print('interface {}'.format(interface))
print('\n'.join(access_template).format(vlan))
```

Script test:

```
$ python access_template_argv.py Gi0/7 4
interface Gi0/7
switchport mode access
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 the template.

Several points need to be clarified:

- argv is a list
- all arguments are in the list and represented as strings
- argv contains not only arguments that passed to the script but also the name of script itself

In this case, the argv list contains the following elements:

```
['access_template_argv.py', 'Gi0/7', '4']
```

First comes the name of script itself, then the arguments in the same order.

User input

Sometimes it is necessary to get information from user, for example, to request a password.

The `input()` function is used to obtain information from user:

```
In [1]: print(input('What is your faivorite routing protocol? '))
What is your faivorite routing protocol? OSPF
OSPF
```

In this case the information is immediately displayed to user, but in addition, the information entered by user can be stored in a variable and can be used later in the script.

```
In [2]: protocol = input('What is your faivorite routing protocol? ')
What is your faivorite routing protocol? OSPF

```

(continues on next page)

(continued from previous page)

```
In [3]: print(protocol)
OSPF
```

In brackets, a question is usually written that specifies what information to enter.

Request information from script (file access_template_input.py):

```
interface = input('Enter interface type and number: ')
vlan = input('Enter VLAN number: ')

access_template = ['switchport mode access',
                   '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))
```

The first two lines request information from user.

The `print('\n' + '-' * 30)` line is used to visually separate the information request from the output.

Execution of the script:

```
$ 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 auxiliary files can be downloaded from repository. If you have tasks with letters (for example, 5.2a) in a section, it is better to do tasks without letters and then with letters. Tasks with letter tend to be slightly more complex than letter-free tasks and they develop or complicate the idea in the respective task without letter.

Note: For example, in the section there are tasks 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. First it is better to complete 5.1, 5.2, 5.3 and then 5.2a, 5.2b, 5.3a

---

If you can do a task with letters right away, it is better to do it in order.

Task 5.1

A dictionary with information about different devices is created in the task.

You should ask user to enter device name (r1, r2 or sw1). And display information about corresponding device on standard output stream (information will be in form of a dictionary).

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 change london_co dictionary.

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *if* condition and other topics to be discussed later.

```
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",
```

(continues on next page)

(continued from previous page)

```
        "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 5.1a

Modify script from Task 5.1 so that in addition to device name the device parameter that you want to display is also requested.

Display information about corresponding parameter of specified device.

Example of script execution:

```
$ python task_5_1a.py
Enter device name : r1
Enter parameter name: ios
15.4
```

Restriction: You cannot change london_co dictionary.

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *if* condition and other topics to be discussed later.

```
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 5.1b

Modify script from task 5.1so that a list of possible parameters is displayed when you ask for parameter. List of parameters should be obtained from dictionary, not written manually.

Display information about corresponding parameter of specified device.

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 change london_co dictionary.

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *if* condition and other topics to be discussed later.

```
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",
```

(continues on next page)

(continued from previous page)

```
        "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 5.1c

Modify script from task 5.1b so that when you ask for a parameter that is not present in device dictionary, the message “No such parameter” is displayed.

Note: Try typing an invalid parameter name or a nonexistent parameter to see what the result is. And then do the task.

If an existing parameter is selected display information about corresponding parameter of specified device.

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 change london_co dictionary.

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *if* condition and other topics to be discussed later.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
```

(continues on next page)

(continued from previous page)

```
        "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"
    },
    "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 script from task 5.1c so that when you ask for parameter, user can enter name of parameter in any register.

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 change london_co dictionary.

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *if* condition and other topics to be discussed later.

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
```

(continues on next page)

(continued from previous page)

```
        "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"
    },
    "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

Request user to enter an IP network in format: `10.1.1.0/24`

Then display network and mask information in this format:

```
Network:
10        1         1         0
00001010  00000001  00000001  00000000

Mask:
/24
255       255       255       0
11111111  11111111  11111111  00000000
```

Check script on different combinations of network/mask.

Hint: Get a mask in binary format:

```
In [1]: "1" * 28 + "0" * 4
Out[1]: "11111111111111111111111111110000"
```

Restriction: All tasks must be performed using only covered topics.

Task 5.2a

It's like task 5.2 but if user entered host address instead of network address, you have to convert host address to network address and display network address and mask as in task 5.2.

Example of network address (all host bits are zero):

- 10.0.1.0/24
- 190.1.0.0/16

Example of host address:

- 10.0.1.1/24 - хост из сети 10.0.1.0/24
- 10.0.5.1/30 - хост из сети 10.0.5.0/30

If user entered address 10.0.1.1/24, , the output should be:

```
Network:
10        0         1         0
00001010  00000000  00000001  00000000

Mask:
/24
255       255       255       0
11111111  11111111  11111111  00000000
```

Check script on different host/mask combinations, for example: 10.0.5.195/28, 10.0.1.1/24

Hint:

There is a binary host address and a network mask 28. Network address is the first 28 bits of host address + 4 zeros. That is, for example, host address 10.1.1.195/28 in binary format will be `bin_ip = "00001010000000010000000111000011"`.

And network address will be the first 28 symbols from bin_ip + 0000 (4 because total address can be 32 bits and 32 - 28 = 4): `00001010000000010000000111000000`

Restriction: All tasks must be performed using only covered topics.

Task 5.2b

Modify script from task 5.2a so that the network/mask is not requested from user, but is passed as script argument.

Restriction: All tasks must be performed using only covered topics.

Task 5.3

Script must request from user:

- interface mode (access/trunk)
- interface number (type and number, like Gi0/3)
- Vlan number (Vlan list will be entered for trunk mode)

Depending on selected mode, the appropriate access or trunk configuration should be displayed (command templates are in access_template and trunk_template lists).

First, interface string goes and interface number is substituted and then goes the corresponding template into which Vlan number (or Vlan list) is substituted.

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *if* condition and *for/while* loops.

Hint: Leading to this task was task 5.1. To make this task easier, you can look at task 5.1 and figure out how it was possible to extract different information depending on user's input.

The following are examples of how to execute a script to make task easier to understand.

Example of script execution when you select access mode:

```
$ 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
```

Example of script execution if 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
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

Complete script from task 5.3 in such a way that depending on selected mode the different questions are asked in request for Vlan number or Vlan list:

- for access: “Enter VLAN number:”
- for trunk: “Enter allowed VLANs:”

Restriction: All tasks must be performed using only covered topics. That is, it is possible to solve this task without using *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. Compound statements

So far, all the code has been executed sequentially - all lines of the script have been executed in the order in which they are written in the file. This section discusses how Python can manage the program:

- branching with the help of `if/elif/else` construction
- repetition of actions in the cycle using `for` and `while` constructions
- error handling with `try/except` construction

if/elif/else

The `if/elif/else` construction allows make branches during program implementation. The program goes into the branch when a certain condition is met.

In this construction only **if** is mandatory, **elif** and **else** are optional:

- **If** condition is always checked first.
- After **If** operator there must be some condition: if this condition is met (returns true), then the 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 “what if ...”
- There can be many **elif** blocks.
- **else** block is executed if none of the conditions **if** or **elif** were true.

Example of construction:

```
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 less than 10')
   ...:
a less than 10
```

Condition

If construction is based on conditions: conditions are always written after **if** and **elif**. Blocks if/elif are executed only when the 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 the 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 the list is empty:

```
In [12]: list_to_test = [1, 2, 3]

In [13]: if list_to_test:
   ....:     print("The list has objects")
   ....:
The 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")
   ....:
The 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 the equality is checked by double ==.

Example of the 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 an element in a sequence (for example, an 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 the **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

The 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 the and operator returns not a boolean value but a value of one of the operands.

If both operands are true, the result is a last value:

```
In [24]: 'string1' and 'string2'
Out[24]: 'string2'

In [25]: 'string1' and 'string2' and 'string3'
Out[25]: 'string3'
```

If one of the operators is a false, the result of the 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 the value of one of the operands.

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 construction use

An example of a check_password.py script that checks length of the password and whether the 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 action should be performed for a set of the same data type. For example, convert all strings in the list to uppercase. Python uses `for` loop for such purposes.

Loop **for** iterates elements of specified sequence and performs the actions specified for each element.

Examples of sequences of elements that can be iterated by **for**:

- string
- list
- dictionary

- *Range*
- Any *Iterable object*

An example of converting strings in a list to uppercase without a loop **for**:

```
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

Same actions with loop **for**:

```
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']
```

The expression `for word in words: upper_words.append(word.upper())` means “for each word

in the **words** list to perform actions in the block **for**". Note, that **word** is the name of variable that refers to different values for each iteration of the loop.

Note: The ‘ pythontutor <http://www.pythontutor.com/>‘__ project can help to understand the loops. There is a special visualization of the code that allows you to see what happens at every stage of the code execution, which is especially useful in the first steps of learning loops. The pythontutor allows you to upload your code but, for instance, you can see example above.

The **for** loop can work with any sequence of elements. For example, the list was used above and the loop iterates through the list elements. Similarly, **for** works 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: The loop uses a variable named **letter**. Although the name can be any name, it is convenient when the name tells you which objects go through a loop.

Sometimes it is necessary to use sequence of numbers in the loop. In this case, it is best to use *Range*

Example of a 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
```

(continues on next page)

(continued from previous page)

```
interface FastEthernet0/4
interface FastEthernet0/5
interface FastEthernet0/6
interface FastEthernet0/7
interface FastEthernet0/8
interface FastEthernet0/9
```

This loop uses range(10). The range() function generates numbers in range from zero to the specified number (in this example, up to 10) not including it.

In this example, the loop runs through the Vlans list, so the 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 by the 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
```

(continues on next page)

(continued from previous page)

```
hostname
location
model
vendor
```

If you want to print key-value pairs in the 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 the items() method which allows you to run the 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
```

The items() method 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, the **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']
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 the list of commands.

Combination for and if

Consider the 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():
```

(continues on next page)

(continued from previous page)

```
    print('interface FastEthernet' + intf)
    for command in access_template:
        if command.endswith('access vlan'):
            print(' {} {}'.format(command, vlan))
        else:
            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 the loop the current key is stored in **intf** variable
- At this moment of the loop the current value is stored in **vlan** variable
- The string "interface Fastethernet" is displayed with interface number added
- The second cycle **for** iterates commands from the access_template list
- Since **switchport access to vlan** command requires a VLAN number:
  - within the second loop **for** commands are checked
  - if command ends with "access vlan"
    - command is displayed and a VLAN number is added to it
  - in all other cases the 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
```

(continues on next page)

(continued from previous page)

```
switchport mode access
switchport access vlan 150
spanning-tree portfast
spanning-tree bpduguard enable
```

while

A **while** loop is another type of loop in Python.

Unlike **if**, after executing 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 the loop is false will be reached.

Consider a simple 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, create a variable with a value of 5.

Then, in the **while** loop the condition a > 0 is specified. That is, as long as the value is greater than 0, actions in the body of the loop will be performed. In this case, the value of variable **a** will be displayed.

In addition, in the body of the loop, after each pass the value of **a** becomes one less.

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`.

As the value **a** decreases, the loop will not be infinite, and at some point the expression a > 0 becomes false.

The following example is based on the example about password from section which describes **if** construction use *Example of if/elif/else construction use*. In that example you had to restart the script if the password did not meet the requirements.

With a **while** loop you can make sure that the script itself requests the password again if it does not meet the requirements.

Check_password_with_while.py file:

```
# -*- 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, the **while** loop is useful because it returns the script back to the beginning of the checks and allows the password to be typed again but does not require the script to restart.

Now the 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

The **break operator** allows early termination of the loop:

- **break** breaks the current loop and continues executing the next expressions
- if multiple nested loops are used the **break** interrupts internal loop and continues to execute expressions following the block. **Break** can be used in loops **for** and **while**

Example of a 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
```

Use break in an example with a password request (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:
        print('Password contains username\n')
    else:
        print('Password for user {} is set'.format(username))
        # finish while loop
        break
    password = input('Enter password once again: ')
```

Now it is possible not to repeat the string `password = input('Enter password once again: ')` in each branch, it is enough to move it to the end of the loop.

And as soon as the correct password is entered, **break** will take the program out of loop **while**.

Continue operator

The **continue** operator returns the control to the beginning of the loop. That is, **continue** allows to «jump» the remaining expressions in the 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")
```

(continues on next page)

(continued from previous page)

```
   ....:         continue
   ....:         print("No one will see it")
   ....:     else:
   ....:         print("Current value: ", i)
   ....:
Current value:  1
Current value:  2
Skip 3
Current value:  4
Current value:  5
Current value:  6
```

Use of **continue** in the example with a 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 the loop by checking the password_correct flag. When the correct password is entered, the flag is set to True, and with **continue** the jump to the beginning of the loop is occurred by skipping the last line with the 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
```

(continues on next page)

(continued from previous page)

```
Enter password once again: asdfsujljhdflaskjdfh
Password for user nata is set
```

Pass operator

The pass operator does nothing. In fact, it is a null statement.

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 the execution of the 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 the loops **for** and **while** you may optionally use **else** block.

for/else

In the loop **for**:

- block **else** is executed if the loop has completed the iteration of the list
- but it *does not execute* if **break** was applied in the loop.

Example of a 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
```

(continues on next page)

(continued from previous page)

```
1
2
3
4
Run out of numbers
```

An example of a loop **for** with **else** and **break** in the loop (because of **break** the block **else** is not applied):

```
In [2]: for num in range(5):
   ....:     if num == 3:
   ....:         break
   ....:     else:
   ....:         print(num)
   ....: else:
   ....:     print("Run out of numbers")
   ....:
0
1
2
```

Example of the loop **for** with **else** and **continue** in the loop (**continue** does not affect the **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 the loop **while**:

- block **else** is executed if the loop has completed the iteration of the list

- but it *does not execute* if **break** was applied in the loop.

Example of a loop **while** with **else** (the block **else** runs after the loop **while**):

```
In [4]: i = 0
In [5]: while i < 5:
   ....:     print(i)
   ....:     i += 1
   ....: else:
   ....:     print("Конец")
   ....:
0
1
2
3
4
Конец
```

An example of a loop **while** with **else** and **break** in a loop (because of **break** the 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("Конец")
   ....:
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 there was a lack of colon, for example.

Python generally reacts quite understandably to such errors and they can easily be corrected.

However, even if the code is written correctly, 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
```

There are two exceptions: **ZeroDivisionError** and **TypeError**.

Most often, it is possible to predict what kind of exceptions will occur during the execution of the program.

For example, if the program expects two numbers on the input and at the output returns their sum, and the user has entered a string instead of one of the numbers a TypeError error will appear as in the 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, the program is immediately interrupted.

In order to work with exceptions the `try/except` construction is used:

```
In [3]: try:
   ...:     2/0
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero")
   ...:
You can't divide by zero
```

The **try** construction works as follows:

- first execute the expressions that are written in the **try** block
- if there are no exceptions during the execution of the **try** block, the block **except** is skipped and the following code is executed
- if there is an exception within the **try** block, the rest part of the **try** block is skipped * if **except** block contains an exception which has been occurred, the code in **except** block is executed * if the exception that has raised is not specified in **except** block, the program execution is interrupted and an error is generated

Note that the Cool! string in the **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
```

Construction try/except may have many **except** if different actions are needed depending on the type of error.

For example, the divide.py script divides two numbers entered by the 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
Результат:  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, the ValueError exception occurs when the user has entered a string instead of a number.

The ZeroDivisionError exception occurs if the second number is 0.

If you do not need to display 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 the data that the user entered, you can write them in the **else** block (divide_ver3.py file):

```
# -*- coding: utf-8 -*-

try:
    a = input("Enter first number: ")
    b = input("Enter second number: ")
```

(continues on next page)

(continued from previous page)

```
    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
Enter second number: 3
Something went wrong...
```

try/except/finally

The **finally** block is another optional block in **try** construction. 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, the same code can be written with or without exceptions.

For example, this version of the 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():
```

(continues on next page)

(continued from previous page)

```
        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 the code is more comprehensible, compact and universal - with or without exceptions.

If you've used some other programming language before, it's possible that the use of exceptions was considered as 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.

Additional material

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 auxiliary files can be downloaded from repository. If you have tasks with letters (for example, 5.2a) in a section, it is better to do tasks without letters and then with letters. Tasks with letter tend to be slightly more complex than letter-free tasks and they develop or complicate the idea in the respective task without letter.

Note: For example, in the section there are tasks 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. First it is better to complete 5.1, 5.2, 5.3 and then 5.2a, 5.2b, 5.3a

---

If you can do a task with letters right away, it is better to do it in order.

Task 6.1

Mac list contains MAC addresses in XXXXXX:XXXX:XXXX format. However, in cisco hardware, MAC addresses are used in XXXXXX.XXX.XXXX format.

Write code that converts MAC addresses to cisco format and adds them to the new list mac_cisco

Restriction: All tasks must be performed using only covered topics.

```
mac = ["aabb:cc80:7000", "aabb:dd80:7340", "aabb:ee80:7000", "aabb:ff80:7000"]
```

Task 6.2

1. Request user input of IP addresses in 10.0.1.1 format
2. Depending on address type (described below), print to standard output stream:
    - “unicast” - if first byte in range 1-223
    - “multicast” - if first byte in range 224-239
    - “local broadcast” - if IP address is 255.255.255.255
    - “unassigned” - if IP address is 0.0.0.0
    - “unused” - in all other cases

Restriction: All tasks must be performed using only covered topics.

Task 6.2a

Make a copy of script from task 6.2.

Add a check of entered IP address. An address is considered correct if it:

- consists of 4 numbers (not letters or other symbols)
- numbers separated by a dot
- each number in range 0 to 255

If address is not set correctly, display message: “Wrong IP address”. Message must be displayed only once.

Restriction: All tasks must be performed using only covered topics.

Task 6.2b

Make a copy of script from task 6.2a.

Complete script: If address was entered incorrectly, ask for address again.

Restriction: All tasks must be performed using only covered topics.

Task 6.3

Script has a configuration generator for access ports.

Make a similar configuration generator for trunk ports.

The situation with trunk ports is complicated by the fact that there could be many Vlans and you have to know what to do with it.

Therefore, according to each port there is a list and the first (zero) item of the list indicates how to perceive VLAN numbers that go further.

Example of 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

Tasks for ports 0/1, 0/2, 0/4:

- generate configuration based on template trunk_template
- based on keywords add, del, only

The code should not be tied to specific port numbers. That is, if there are other interface numbers in *trunk* dictionary, the code should work.

Restriction: All tasks must be performed using only covered topics.

```
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 considered 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 deals with simple text files. For example, Cisco configuration file.

There are several aspects to working with files:

- opening/closing
- reading
- writing

This section only deals with 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()`

The `open()` function is most often used to open files:

```
file = open('file_name.txt', 'r')
```

In the 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

The `open()` function 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 the file exists, its content is removed
- if the file does not exist, a new one is created
- `w+` - open file for reading and writing
- if the file exists, its content is removed
- if the file does not exist, a new one is created
- `a` - open file to add a data. Data is added to end of file
- `a+` - open file for reading and writing. Data is added to end of file

---

Note: `r` - read; `a` - append; `w` - write

File reading

Python has several file reading methods:

- `read()` - reads the contents of the file to the string
- `readline()` - reads file line by line
- `readlines()` - reads the 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
```

(continues on next page)

(continued from previous page)

```
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

read()

The `read()` method reads the entire file to one string.

Example of the use of `read()`:

```
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 the `read()` method is called. And after the file has been read the cursor stays at the end of the file. The cursor position can be controlled by means of `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)
   ...:
!

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 line feed character at the end, you can use `split()` method and specify the 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 the list is an empty string.

If you use `split()` before `rstrip()`, the 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, the file had to be reopened to read it again. This is because after reading methods the 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 the empty string returns:

```
In [17]: print(f.read())
```

But with the seek method you can go to the beginning of the file (0 means the beginning of the file):

```
In [18]: f.seek(0)
```

Once the cursor has been set to the beginning of the 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 it is very important to decide how to open a file in order 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 added to the end of the file

Both modes create a file if it does not exist.

These 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()

The `write` method 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 the *cat* command, you can easily see the content of the 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()

The `writelines()` method 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 the lines.

You can add line feed character in different ways. For example, you can simply process the list in the loop:

```
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 write the resulting list into the file, it already contains line feed characters:

```
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
```

(continues on next page)

(continued from previous page)

```
!
```

File closing

Note: In real life, the most common way to close files is use of `with` construction. It's much more convenient way than to close file explicitly. But since you can also find the `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 the file itself. But it's best not to count on it and close the file explicitly.

close()

The close() method met in File writing section. It was there to make sure that the content of the file was written on disk.

For this, Python has a separate `flush()` method. But since in the example with the file writing there was no need to perform any more operations, the 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 the file is open, it returns `False`:

```
In [3]: f.closed
Out[3]: False
```

Now close the file and check `closed` again:

```
In [4]: f.close()

In [5]: f.closed
Out[5]: True
```

If you try to read the file an exception occurs:

```
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
```

Use `try/finally` to work with files

By processing exceptions, you can:

- intercept exceptions that occur when trying to read a non-existent file
- close file after all operations in `finally` block

If you try to read a file that does not exist this exception will occur:

```
In [7]: f = open('r3.txt', 'r')
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-54-1a33581ca641> in <module>()
----> 1 f = open('r3.txt', 'r')

IOError: [Errno 2] No such file or directory: 'r3.txt'
```

Using `try/except` construction you can capture this exception and print your message:

```
In [8]: try:
  ....:     f = open('r3.txt', 'r')
  ....: except IOError:
  ....:     print('No such file')
```

(continues on next page)

(continued from previous page)

```
  ....:
No such file
```

And with `finally` you can close the file after all operations:

```
In [9]: try:
  ....:     f = open('r1.txt', 'r')
  ....:     print(f.read())
  ....: except IOError:
  ....:     print('No such file')
  ....: finally:
  ....:     f.close()
  ....:
!
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 [10]: f.closed
Out[10]: True
```

Construction with

The construction **with** is a context manager.

Python has a more convenient way of working with files than the ones used so far - the construction `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

```

(continues on next page)

(continued from previous page)

```
service timestamps log datetime msec localtime show-timezone year

service password-encryption

service sequence-numbers

!

no ip domain lookup

!

ip ssh version 2

!
```

In addition, the construction `with` guarantees file closure automatically.

Pay attention to how the lines of the file are read:

```
for line in f:
    print(line)
```

When the file needs to be run line by line, it is best to use this option.

In the previous output there were extra empty lines between the lines of the file because **print** adds another line feed 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
!
```

(continues on next page)

(continued from previous page)

```
In [3]: f.closed
Out[3]: True
```

And of course, with construction can be used not only as a line-by-line reader, all methods that have been considered 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
!
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:
```

(continues on next page)

(continued from previous page)

```
...:         for line in src:
...:             if line.startswith('service'):
...:                 dest.write(line)
...:
```

Additional material

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 auxiliary files can be downloaded from repository. If you have tasks with letters (for example, 5.2a) in a section, it is better to do tasks without letters and then with letters. Tasks with letter tend to be slightly more complex than letter-free tasks and they develop or complicate the idea in the respective task without letter.

Note: For example, in the section there are tasks 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. First it is better to complete 5.1, 5.2, 5.3 and then 5.2a, 5.2b, 5.3a

---

If you can do a task with letters right away, it is better to do it in order.

Task 7.1

Process lines from ospf.txt file and display information for each line as follows:

```
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 performed using only covered topics.

Task 7.2

Create a script that will process configuration file config_sw1.txt. The file name is passed as a script argument.

Script should return commands from passed configuration file, excluding lines that start with `!`.

Output should be without empty lines.

Restriction: All tasks must be performed using only covered topics.

Task 7.2a

Make a copy of script from task 7.2.

Complete script: Script should not display commands containing words that are specified in *ignore* list.

Restriction: All tasks must be performed using only covered topics.

```
ignore = ["duplex", "alias", "Current configuration"]
```

Task 7.2b

Complete script from task 7.2a: instead of displaying to standard output stream, script should write received lines to config_sw1_cleared.txt file

You have to filter lines from *ignore* list. Lines that start with ! should not be filtered.

Restriction: All tasks must be performed using only covered topics.

```
ignore = ["duplex", "alias", "Current configuration"]
```

Task 7.2c

Redo script from task 7.2b: pass to script as arguments:

- source configuration file name
- resulting configuration file name

Inside, script should filter those lines in original configuration file that contain words from *ignore* list. And write the rest of lines to resulting file.

Check script with config_sw1.txt.

Restriction: All tasks must be performed using only covered topics.

```
ignore = ["duplex", "alias", "Current configuration"]
```

Task 7.3

Script should process entries in CAM_table.txt file. Every line with MAC address should be processed in a way that such view table is displayed on standard output stream (not all lines from the file are shown):

```
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 performed using only covered topics.

Task 7.3a

Make a copy of script from task 7.3.

Complete script: Sort output by VLAN number.

The result should be like this:

```
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
```

Note, vlan 1000 should be the last to be displayed. The correct sort can be achieved if vlan is a number rather than a string.

Restriction: All tasks must be performed using only covered topics.

Task 7.3b

Make a copy of script from task 7.3a.

Redo script:

- Ask user to enter VLAN number.
- Display information only for specified VLAN.

Restriction: All tasks must be performed using only covered topics.

8. Python basic examples

This section collects topics that were not included in the previous sections and also provides examples of using the 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 the file will be replaced to get output from the 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 the 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 the f- string, in figure brackets there are names of the 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 the case of ipython, as soon as we wrote the expression and pressed Enter, it was performed and instead of the expressions {ip} and {mask} the values of the variables were substituted.

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

```
In [1]: f"IP: {ip}, mask: {mask}"
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<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 brackets:

```
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 the 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 the f-string must be written in the 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'
```

Alignment with columns:

```
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
```

Work with dictionary

```
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 the len() function inside the 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 the template looks more understandable and compact. However, there are cases when the 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 the script. F-string will execute the first time and will set the current values of the variables and to use the template again it has to be rewritten. This means that the 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

The unpacking of variables is a special syntax that allows to assign elements of an iterated object to variables.

Note: This functionality is often referred to as tuple unpacking but the 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 the list falls into the corresponding variable. It is important to keep in mind that the variables on the left should be exactly as many 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 the elements of an iterated object are needed. The unpacking syntax requires that exactly as many variables as the 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 record type:

```
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 the item_type variable with underline character:

```
In [12]: vlan, mac, _, intf = line.split()
```

This clearly indicates that this element is not needed.

The 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 *

The unpacking of variables supports a special syntax that allows unpacking of several elements into one. If you put * in front of the variable name all elements except those that are explicitly assigned will be written into it.

For example, you can get the first element in the *first* variable and the rest in the *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]
```

The 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 could be only one variable with an asterisk in terms of unpacking.

```
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 the 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

An 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.

The terms are as follows:

- List comprehensions
- Dict comprehensions
- Set comprehensions

Unfortunately, the official translation into Russian sounds like abstraction of lists or list inclusion which does not help to understand the essence of the object.

The book used the term «list generator» which unfortunately is also not the best version because in Python there is a separate concept of generator and generator expressions, but it better reflects the essence of expression.

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 generators)

List generator is an expression like:

```
In [1]: vlans = ['vlan {}'.format(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.

The expression above is similar to this loop:

```
In [3]: vlans = []

In [4]: for num in range(10,16):
   ...:     vlans.append('vlan {}'.format(num))
   ...:

In [5]: print(vlans)
['vlan 10', 'vlan 11', 'vlan 12', 'vlan 13', 'vlan 14', 'vlan 15']
```

In the 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']
```

(continues on next page)

(continued from previous page)

```
In [7]: only_digits = []

In [8]: for item in items:
   ...:     if item.isdigit():
   ...:         only_digits.append(int(item))
   ...:

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 generator but when it is possible to do so without making the expression more complex, it is better to use the list generators.

Note: In Python, list generators can also replace filter and map functions and are considered as more understandable solutions.

With the help of the list generator it is also convenient to obtain 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',
```

(continues on next page)

(continued from previous page)

```
    ...:     '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 [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, the syntax of the list generator 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 the expression.

For example, the *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]
```

Similar to the list generator:

```
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 = ['vlan {}\n name {}'.format(vlan, name) for vlan, 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 (dictionary generators)

Dictionary generators are similar to list generators 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}
```

You can replace it with a dictionary generator:

```
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 where you need to convert an existing dictionary and transfer all keys to a lower register. First, a solution without a dictionary generator:

```
In [32]: r1 = {'IOS': '15.4',
    ...:       'IP': '10.255.0.1',
    ...:       '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[str.lower(key)] = 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'}
```

A similar variant with a dictionary generator:

```
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 = {str.lower(key): value for key, value in r1.items()}

In [38]: lower_r1
```

(continues on next page)

(continued from previous page)

```
Out[38]:
{'hostname': 'london_r1',
 'ios': '15.4',
 'ip': '10.255.0.1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

Like the 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():
```

(continues on next page)

(continued from previous page)

```
    ...:         lower_london_co[device][str.lower(key)] = 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'},
 '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: {str.lower(key):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'},
```

(continues on next page)

(continued from previous page)

```
 '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 generators)

Set generators are generally similar to list generators.

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}
```

Working with dictionary

When processing output of commands or configuration, often it will be necessary to write the 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 of output with columns

This example will deal with the output of *sh ip int br* command. From the output of command we need to get the interface name - IP address. So the interface name is the dictionary key and the IP address is the value. At the same time, the match must be made only for those interfaces with the 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)
```

The command *sh ip int br* displays the output with columns. So the desired fields are in the same line. The 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 the IP address is configured, the first character of the second column is checked: if the first character is a number the address is assigned to the interface and the string has to be processed.

In `interface, address, *other = line` - the variables are unpacked. The *interface* variable will have the interface name, the *address* will have the IP address and *other* - all other fields.

Since each line has a key-value pair, they are assigned to the dictionary: `result[interface] = address`.

The result of the script execution will be a dictionary (here it is split into key-value pairs for convenience, in the real script output the dictionary will be displayed in one line):

```
{'FastEthernet0/0': '15.0.15.1',
 'FastEthernet0/1': '10.0.12.1',
 'FastEthernet0/2': '10.0.13.1',
 '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 the key and the value are in different lines. And you have to figure out how to process the output to get the right match.

For example, from the output of the *sh ip int br* command you need to get the 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
  ...
```

The interface name is in `Ethernet0/0 is up, line protocol is up` line and MTU in the `MTU is 1500 bytes` line.

For example, try to remember the 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:
```

(continues on next page)

(continued from previous page)

```
   ...:         if 'line protocol' in line:
   ...:             interface = line.split()[0]
   ...:         elif 'MTU is' in line:
   ...:             mtu = line.split()[-2]
   ...:             print('{:15}{}'.format(interface, mtu))
   ...:
Ethernet0/0    1500
Ethernet0/1    1500
Ethernet0/2    1500
Ethernet0/3    1500
Loopback0      1514
```

The 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 the interface every time it appears and at the time when line meets with 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 the values when the 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 the script execution will be a dictionary (here it is split into key-value pairs for convenience, in the real script output the dictionary 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 the 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. The 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
```

(continues on next page)

(continued from previous page)

```
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 the 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, the dictionary `result` creates a key with the name of the interface that corresponds to an empty dictionary. This blank is used so that at the time when the IP address or MTU is detected, the parameter can be written into the nested dictionary of the corresponding interface.

The result of the script execution will be a dictionary (here it is split into key-value pairs for convenience, in the real script output the dictionary 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 the case of output `sh ip interface`, interfaces may look like:

```
Ethernet0/1 is up, line protocol is up
  Internet protocol processing disabled
```

(continues on next page)

(continued from previous page)

```
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 the previous script for a file with such interfaces, the result is this (output for the 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 the dictionary only when an IP address is assigned to the interface, you need to move the creation of the key with interface name to the moment when the 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'}}
```

Additional material

Documentation:

- PEP 3132 – Extended Iterable Unpacking

Articles:

- List, Dict And Set Comprehensions By Example - a good article. And at the end of the article there are several exercises (with answers)
- Python List Comprehensions: Explained Visually - a great explanation of the list comprehensions, plus video

Stack Overflow:

- Answer with many unpacking options

3

II. Code reuse

When the code is written you will find that some of the actions are often repeated. It can be a small block of 3-5 lines or it can be a rather large sequence of actions.

Copying code is a bad idea. Because if you have to update one of the copies later, you have to update the others.

Instead, you create a special code block with the name - function. And every time the code has to be repeated, you just call a function. The function allows not only to name a block of code but also to make it more abstract through parameters. The parameters make it possible to transfer different source data for the execution of the function. And, correspondingly, get different results depending on the input parameters.

Section *9. Functions* deals with the creation of functions. In addition, section *10. Useful functions* considers useful embedded functions.

Once the code is divided into functions, there comes a time when you need to use the 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* is considered useful modules from the standard Python library.

The last section *13. Iterators, iterable objects and generators* is dedicated to iterable objects, iterators and generators.

9. Functions

A function is a block of code that performs certain actions:

- 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
  - hence, the function code will be executed according to the stated arguments

What are the functions for?

Typically, the 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 the exclamation mark (for Cisco)
- deleting (or skipping) empty lines
- deleting line feed 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 the code you have to update it in all the 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 round brackets
- parameters that the function accepts inside brackets

- after round brackets goes colon and from a new line with indent there is a block of code that the function executes
- optionally, the first line may be a comment, so-called **docstring**
- function can use **return** operator
  - it is used to terminate and exit a function
  - most often **return** operator returns some value

Note: The function code used in this subsection can be copied from the create_func file.

Example of a 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 the specified name and IP address. Function has three parameters: intf_name, ip, mask. When function is called the real data will enter these parameters.

Note: When a 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.

The configure_intf() function 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
```

(continues on next page)

(continued from previous page)

```
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
```

Current configure_intf() function prints commands to a standard output, commands can be seen but the result of the function cannot be saved to a variable.

For example, the sorted() function does not simply print the sorting result to the standard output stream but *returns* it, so it can be saved to the 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 the string `Out[5]` in ipython: thus ipython shows that the function/method returns something and shows what it returns.

If you try to write the result of configure_intf() function to a variable, the variable will have None:

```
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

The **return** operator is used to return a value while it completes the function. Function can return any Python object. By default, function always returns None.

In order for the 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
    ...:

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. However, it is possible to use print() to add some messages.

Another important aspect of the **return** operator is that after **return** the function closes, meaning that the expressions that follow **return** are not executed.

For example, in the 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'
```

The function can return multiple values. In this case, they are separated by a comma after **return** operator. In fact, the function returns the 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')
```

(continues on next page)

(continued from previous page)

```
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

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 the 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 not to be lazy to write short comments that describe the function. For example, describe what the 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 the function you wrote is doing.

Namespace. Scope of variables

Variables in Python have a scope. Depending on the location in the code where variable has been defined, the 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. The namespace that is available at each moment depends on the area in which the 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 the local area of outer functions (these are the 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 the 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
```

(continues on next page)

(continued from previous page)

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

Note that the intf_config variable is not available outside of the 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 the 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** - the variables that are used when creating a function.
- **arguments** - the actual values (data) that are passed to the 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.

The function checks the 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:
```

(continues on next page)

(continued from previous page)

```
   ...:         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
   ...:
```

The check_passwd function expects two arguments: username and password.

The function checks the 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, the min_length option is specified with the 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
```

(continues on next page)

(continued from previous page)

```
...:     else:
...:         print(f'Password for user {username} has passed all checks')
...:         return True
...:
```

Since the min_length parameter has a default value the corresponding argument can be omitted when a function is called if the default value fits you:

```
In [7]: check_passwd('nata', '12345')
Password is too short
Out[7]: False
```

If you want to change the 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 the creation of the function. That is, the order of argument transfer determines what value each argument will get.
- as **keyword** - passed with the argument name and its value. In such a case, the 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 the 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:
```

(continues on next page)

(continued from previous page)

```
...:         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

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 the 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, the 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
```

(continues on next page)

(continued from previous page)

```
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

The 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)
  ....:
```

The sum_arg function 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
```

(continues on next page)

(continued from previous page)

```
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)
  ....:

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

The parameter that accepts keyword variable length arguments is created by adding two asterisk in front of the parameter name. Name of parameter can be any but by agreement most commonly use the name **kwargs (from keyword arguments).

Example of a function:

```
In [8]: def sum_arg(a,**kwargs):
  ....:     print(a, kwargs)
  ....:     return a + sum(kwargs.values())
  ....:
```

The sum_arg function 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 the relevant arguments.

In reality, it is usually necessary to transfer data to the 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 the list or tuple. To transfer them to the 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}'
```

(continues on next page)

(continued from previous page)

```
    ..:     no_shut = 'no shutdown'
    ..:     ip_addr = f'ip address {ip_address} {mask}'
    ..:     result = [interface, no_shut, ip_addr]
    ..:     return result
    ..:
```

The function expects such arguments:

- intf_name - interface name
- ip_address - IP address
- mask - subnet mask

Function returns a list of strings to configure the 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 the 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 the list in the loop and pass the 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))
```

(continues on next page)

(continued from previous page)

```
      3

TypeError: config_interface() missing 2 required positional arguments: 'ip_address
↪' and 'mask'
```

The error is quite logical: the function expects three arguments and it is given 1 argument - a list.

In such a situation it is necessary to unpack the 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']
['interface Lo0', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']
```

Python will unpack the *info* list itself and transfer list elements to the 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

TypeError: check_passwd() missing 1 required positional argument: 'password'
```

The error is because the function has taken dictionary as one argument and believes that it lacks only password argument.

If you add ** пbefore passing a dictionary to function, the 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 the 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 transfer arguments between functions. Let me give you an example.

check_passwd function (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:
   ...:         print('Password contains username')
   ...:         return False
   ...:     else:
   ...:         print(f'Password for user {username} has passed all checks')
   ...:         return True
   ...:
```

The 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 the specified user, checks it and requests it again if password has not been checked or writes user and password to the file if password has been verified

```
In [7]: def add_user_to_users_file(user, users_filename='users.txt'):
```

(continues on next page)

(continued from previous page)

```
   ...:     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
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 variant 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 the password. In the following variant 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 the check_passwd function in defining the 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 the internal function (in this case it is check_passwd) will be taken in **kwargs. Then, when calling the 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')
   ...:

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 variant you can add arguments to the check_passwd function without having to duplicate them in the add_user_to_users_file function.

Additional material

Documenation:

- Defining Functions
- Built-in Functions
- Sorting HOW TO
- Functional Programming HOWTO
- Range function

Tasks

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 9.1

Create a function that generates configuration for access ports.

Function expects such arguments:

1. dictionary with interface-VLAN mapping:

```
{"FastEthernet0/12": 10,
 "FastEthernet0/14": 11,
 "FastEthernet0/16": 17}
```

2. access port configuration template in the form of a list of commands (access_mode_template list)

Function should return a list of all ports in access mode with configuration based on template access_mode_template. There should be no line feed at the end of lines in the list.

In this task, the blank for function is already done and only body of function need to be written.

Example of resulting list (line feed after each item is made for ease of reading):

```
[
"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 function with access_config dictionary.

Restriction: All tasks must be performed using only covered topics.

```
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
}


def generate_access_config(intf_vlan_mapping, access_template):
    """
    intf_vlan_mapping - dictionary with interface-VLAN mapping:
        {"FastEthernet0/12": 10,
         "FastEthernet0/14": 11,
         "FastEthernet0/16": 17}
    access_template - list of commands for port in access mode

    Return list of all ports in access mode with configuration based on temlate
    """
```

Task 9.1a

Make a copy of generate_access_config() function from task 9.1.

Complete script: enter an additional parameter that controls whether port-security will be configured:

- parameter name “psecurity”
- default is None
- to configure port-security, list of *port-security* commands should be passed as a value (in port_security_template list)

Function should return a list of all ports in access mode with configuration based on access_mode_template template and template port_security_template template if it has been passed. There should be no line feed at the end of lines in the list.

Check function with access_config dictionary, with and without port-security configuration generation.

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 performed using only covered topics.

```
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 a generate_trunk_config() function that generates configuration for trunk ports.

Function should have such parameters:

1. intf_vlan_mapping: expects a dictionary with interface-VLAN mapping:

```
{"FastEthernet0/1": [10, 20],
 "FastEthernet0/2": [11, 30],
 "FastEthernet0/4": [17]}
```

2. trunk_template: expects trunk port configuration template as command list (trunk_mode_template list)

Function should return a list of commands with configuration based on specified ports and trunk_mode_template template. There should be no line feed at the end of lines in the list.

Check function with trunk_config dictionary and list of commands trunk_mode_template. If this check is successful, check function again with trunk_config_2 dictionary and make sure that the resulting list contains correct interface and vlan numbers.

Example of resulting list (line feed after each item is made for ease of reading):

```
[
"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 performed using only covered topics.

```
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 9.2a

Make a copy of generate_trunk_config() function from task 9.2.

Change function to return a dictionary rather than a list of commands:

- keys: interface names like “FastEthernet0/1”
- values: list of commands to execute on this interface

Check function with trunk_config dictionary and trunk_mode_template template.

Restriction: All tasks must be performed using only covered topics.

```
trunk_mode_template = [
    "switchport mode trunk", "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
]

trunk_config = {
```

(continues on next page)

(continued from previous page)

```
    "FastEthernet0/1": [10, 20, 30],
    "FastEthernet0/2": [11, 30],
    "FastEthernet0/4": [17]
}
```

Task 9.3

Create get_int_vlan_map() function that processes switch configuration file and returns a tuple with two dictionaries:

1. dictionary of ports in access mode, where keys are port numbers and values is access VLAN (numbers):

```
{"FastEthernet0/12": 10,
 "FastEthernet0/14": 11,
 "FastEthernet0/16": 17}
```

2. dictionary of ports in trunk mode, where keys are port number and values are list of allowed VLANs (list of numbers):

Function should have one parameter - config_filename, that expects as an argument the name of configuration file.

Check function with config_sw1.txt file

Restriction: All tasks must be performed using only covered topics.

Task 9.3a

Copy get_int_vlan_map() function from task 9.3.

Complete function: add configuration support when access port configuration is like:

```
interface FastEthernet0/20
 switchport mode access
 duplex auto
```

That is, port is in VLAN 1

In this case, port dictionary should add information that port in VLAN 1

Function should have one parameter - config_filename, that expects as an argument the name of configuration file.

Check function with config_sw2.txt

Restriction: All tasks must be performed using only covered topics.

Task 9.4

Create convert_config_to_dict() function that processes switch configuration file and returns dictionary:

- All top-level commands (global configuration mode) will be the keys.
- If top-level command has a sub-command, it must be in value of corresponding key as a list (spaces at the beginning of line should be removed).
- If top level command does not have a sub-command, the value is an empty list

Function should have one parameter - config_filename, that expects as an argument the name of configuration file.

When processing a configuration file, you should ignore lines that start with "!" as well as the lines that contain words from *ignore* list. To check whether to ignore a line, use ignore_command() function.

Check function with config_sw1.txt file

Restriction: All tasks must be performed using only covered topics.

```
ignore = ["duplex", "alias", "Current configuration"]


def ignore_command(command, ignore):
    """
    Function checks whether command words from *ignore* list.

    command - string. Command that should be checked.
    ignore - list. List of words.

    Returns
    * True, if command contains a word from *ignore* list.
    * False - if not
    """
    return any(word in command for word in ignore)
```

10. Useful functions

This section discusses the following functions:

- print
- range
- sorted
- enumerate
- zip
- all, any
- lambda
- map, filter

Print

The `print()` function has been used many times in the book but its full syntax has not yet been considered:

```
print(*items, sep=' ', end='\n', file=sys.stdout, flush=False)
```

The print() function outputs all elements by separating them by their **sep** value and finishes output with the **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

The sep parameter controls which separator will be used between elements.

By default, the 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

In [10]: print(1, 2, 3, sep='\n')
1
2
3

In [11]: print(1, 2, 3, sep='\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

The **end** parameter controls which value will be displayed after all elements are printed. By default, line feed 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

The **file** parameter 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. The print() function 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):
```

(continues on next page)

(continued from previous page)

```
    print(num)
    time.sleep(1)
```

Try running the 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 the function. The numbers does not appear one per second but all appear after 10 seconds.

This is because when output is displayed on standard output the **flush** is performed after line feed character.

In order to make script work properly the **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

The range() function 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

The range function stores only **start**, **stop** and **step** values and calculates values as necessary. This means that regardless of the size of range that describes the 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 the 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]
```

The 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 obtain a descending sequence use a negative step and specify **start** by a greater number and **stop** by a smaller number.

In the descending sequence the steps can also be different:

```
In [8]: list(range(10, 0, -2))
Out[8]: [10, 8, 6, 4, 2]
```

The 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]
```

(continues on next page)

(continued from previous page)

```
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
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 the 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

The `sorted()` function returns a new sorted list that is obtained from an iterable object that has been passed as an argument. The function also supports additional options that allow you to control sorting.

The first aspect that is important to pay attention to - **sorted** returns the list.

If you sort the 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 the tuple also the list returns:

```
In [3]: tuple_of_words = ('one', 'two', 'list', '', 'dict')

In [4]: sorted(tuple_of_words)
Out[4]: ['', 'dict', 'list', 'one', 'two']
```

Sorting the set:

```
In [5]: set_of_words = {'one', 'two', 'list', '', 'dict'}

In [6]: sorted(set_of_words)
Out[6]: ['', 'dict', 'list', 'one', 'two']
```

Sorting the 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',
   ...:          '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]:
['IT_VLAN',
 'Mngmt_VLAN',
 'User_VLAN',
 'id',
 'name',
 'port',
 'to_id',
 'to_name']
```

reverse

The **reverse** flag allows you to control the sorting order. By default, the sorting will be incremental.

The **reverse** flag changes the order:

```
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 the **key** option you can specify how to perform sorting. The **key** parameter expects the 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']

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 the anonymous lambda() function.

Using the **key** option you can sort objects by any element. However, this requires either lambda() or special functions from the **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))
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 the 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)
    ...:
```

(continues on next page)

(continued from previous page)

```
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 the 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.

The 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 the 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 the EEM.

A simple Python script can generate EEM commands based on the 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

The `zip()` function:

- sequences are passed to the function
- zip() returns an iterator with tuples in which the n-tuple consists of n-elements of sequences that have been passed as arguments
- for example, the 10th tuple will contain the 10th element of each of the 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 display 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]

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',
```

(continues on next page)

(continued from previous page)

```
 '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 the 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'},
```

(continues on next page)

(continued from previous page)

```
 '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

The `all()` function returns True if all elements are true (or the 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

The any() function 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):
    '''
    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 the 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.

The lambda function 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 the 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)]
```

The lambda function is also useful in map() and filter() functions which will be discussed in the following sections.

Map

The map() function 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(). Most often, list comprehension option is more understandable 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 the list comprehension variant.

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]

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

The `filter()` function applies the function to all sequence elements and returns the iterator with those objects for which the 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 the list comprehension variant.

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',
 ...
```

(continues on next page)

(continued from previous page)

```
 'exit',
 '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 invoke some function or method from the **os** module you should specify `os.` and then the object name:

```
In [4]: os.getlogin()
Out[4]: 'natasha'
```

This import method is good because the module objects do not enter the namespace of the current program. That is, if you create a function named getlogin() it will not conflict with the same function of the **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 one or two functions are needed from the 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 the 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 the **os** module, so the output is shortened. At the end, the 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 the **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))
```

The check_ip_function.py file has created check_ip function which checks that the argument is an IP address. This is done by using the **ipaddress** module which will be discussed in the next section.

The ipaddress.ip_address function itself checks the correctness of the IP address and generates ValueError exception if the address is not validated.

The check_ip function returns True if address is validated 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 the check_ip function and uses it to select from the 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 the get_correct_ip.py script is displayed, but also information from the check_ip_function.py. This is because any type of import executes the entire script. That is, even when the import looks like `from check_ip_function import check_ip`, the entire check_ip_function.py script is executed, not just check_ip function. As a result, all messages of the imported script will be displayed.

Messages from the imported script are not scary, they are just confusing. Worse when script performed some kind of connection to the hardware and when importing a function from it, we will have to wait for the connection to take place.

Python can specify that some strings should not be executed when importing. This is discussed in the following subsection.

Note: The return_correct_ip function can be replaced by a filter() or a list generator. 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 the 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 the 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 the `if __name__ == '__main__'` block are not executed at import.

The variable __name__ is a special variable that will be equal to "__main__" only if the file is run as the main program and is set equal to the module name when importing the module. That is, the `if __name__ == '__main__'` condition checks whether the file was run directly.

As a rule, the `if __name__ == '__main__'` block includes all function calls and information output on the standard output stream. That is, in the check_ip_function.py script this block conytains everything except import and the 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 the check_ip_function.py script directly, all lines are executed, because the variable __name__ in this case is equal to '__main__'.

The get_correct_ip.py script 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 the 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 the script getcorrect__ip.py.

In general, it is better to write all the code that calls functions and outputs something to the standard output stream inside the block `if __name__ == '__main__'`.

Warning: Starting with Section 9, there are program tests for tasks that can be used to check whether the tasks are properly executed. To work correctly with tests you have to always write a function call in the job file within the block `if __name__ == '__main__'`. The absence of this block will cause errors, not in all tasks, but it will still avoid problems.

Tasks

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 11.1

Create a parse_cdp_neighbors() function that handles the output of show cdp neighbors command.

Function should have one parameter - command_output, that expects as an argument the command output as a single string (not file name). To do this, you should read the entire contents of file into a string and then pass string as function argument (how to pass command output is shown in code below).

Function should return a dictionary that describes connections between devices.

For example, if such an output is given 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 such dictionary:

```
{("R4", "Fa0/1"): ("R5", "Fa0/1"),
 ("R4", "Fa0/2"): ("R6", "Fa0/0")}
```

In dictionary, interfaces should be written without space between type and name. That is Fa0/0, not Fa 0/0.

Check function with contents of sh_cdp_n_sw1.txt file. Function also should work on other files (test checks function operation on output from sh_cdp_n_sw1.txt and sh_cdp_n_r3.txt).

Restriction: All tasks must be performed using only covered topics.

```
def parse_cdp_neighbors(command_output):
    """
    Here we pass command output with one string because in this form we will␣
↪receive command output from equipment.
    Taking command output as argument, instead of a file name, we make function␣
↪more universal:
```

(continues on next page)

(continued from previous page)

```
    it can work with both files and output from equipment. Plus, we learn to work␣
↪with that 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 handles the output of show cdp neighbors command from multiple files and integrates it into one common topology.

Function should have one parameter – filenames, that expects as an argument a list of file names in which show cdp neighbors output is found.

Function should return a dictionary that describes connections between devices. Structure of dictionary is the same as in task 11.1:

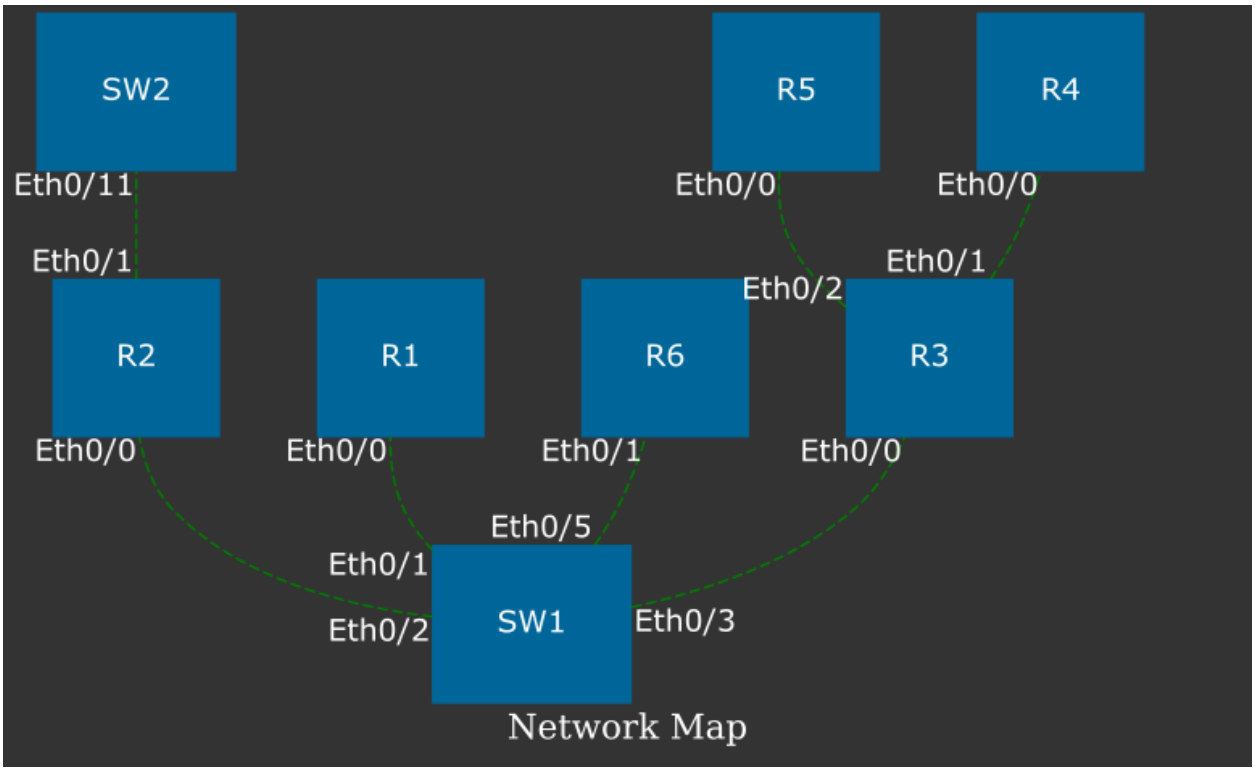
```
{("R4", "Fa0/1"): ("R5", "Fa0/1"),
 ("R4", "Fa0/2"): ("R6", "Fa0/0")}
```

Generate a topology that matches the output from files:

- sh_cdp_n_sw1.txt
- sh_cdp_n_r1.txt
- sh_cdp_n_r2.txt
- sh_cdp_n_r3.txt

There should be no duplicates in dictionary that returns create_network_map() function.

Using draw_topology() function from draw_network_graph.py file, draw a diagram based on topology received with create_network_map() function. The result should look the same as scheme in task_11_2_topology.svg file



At the same time:

- The arrangement of devices on diagram may be different
- Connections should follow the diagram

Do not copy code of functions parse_cdp_neighbors() and draw_topology().

Restriction: All tasks must be performed using only covered topics.

Note: To complete this task, graphviz must be installed: : `apt-get install graphviz`

And a python module for working with graphviz: `pip install graphviz`

```
# These blanks are written to show at what moment
# a topology should be drawn (after function call)
def create_network_map(filenames):
    pass


if __name__ == "__main__":
    infiles = [
        "sh_cdp_n_sw1.txt",
        "sh_cdp_n_r1.txt",
```

(continues on next page)

(continued from previous page)

```
        "sh_cdp_n_r2.txt",
        "sh_cdp_n_r3.txt",
    ]

    topology = create_network_map(infiles)
    # draw topology:
    # draw_topology(topology)
```

12. Useful modules

This section describes the modules:

- subprocess
- os
- argparse
- ipaddress
- pprint
- tabulate

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 the script. And depending on the 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 the 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 the execution of the process, such as the return code:

```
In [3]: result
Out[3]: CompletedProcess(args='ls', returncode=0)

```

(continues on next page)

(continued from previous page)

```
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
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 a **shell** argument and call the 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 the `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, the 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
```

(continues on next page)

(continued from previous page)

```
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 the 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
```

Примеры использования модуля

Example of subprocess module use (subprocess_run_basic.py file):

```
import subprocess

reply = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])

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.

The ping_ip function checks the availability of the 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],
```

(continues on next page)

(continued from previous page)

```
                          stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE,
                          encoding='utf-8')
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        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 the list of IP addresses and return as a result two lists: accessible and inaccessible addresses.

Note: You will find it in tasks of section

If the number of IP addresses to check is large, you can use threading or multiprocessing modules to speed up verification.

Os

The os module allows working with the filesystem, environment and managing processes.

This subsection addresses only several useful features. For a more complete description of the capabilities of the 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')
```

(continues on next page)

(continued from previous page)

```
In [3]: ls -ls
total 0
0 drwxr-xr-x  2 nata  nata  68 Jan 23 18:58 test/
```

In addition, the 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 the 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'

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 the standard Python library.

---

ipaddress.ip_address()

The 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

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()

Function ipaddress.ip_network() allows the creation of an object that describes a network (Ipv4 or Ipv6):

```
In [20]: subnet1 = ipaddress.ip_network('80.0.1.0/28')
```

As with an address the 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
```

The hosts() method returns generator, so to view all hosts you should apply the 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')]
```

The subnets() method 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]:
[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 the **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
```

(continues on next page)

(continued from previous page)

```
80.0.1.15
```

And it is possible to get a specific address:

```
In [32]: subnet1[0]
Out[32]: IPv4Address('80.0.1.0')

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 creating 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 the module has built-in address correctness checks, you can use them, for example, to check whether the address is a network or host address:

```
In [40]: IP1 = '10.0.1.1/24'

```

(continues on next page)

(continued from previous page)

```
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
   ....:     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 the 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

The tabulate() function is used to generate the 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))
---------------  ---------  --  --
```

(continues on next page)

(continued from previous page)

```
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

The **headers** parameter 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 the 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 the 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',
  '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
```

Table style

tabulate supports different table display 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         |
```

(continues on next page)

(continued from previous page)

```
+-----------------+-----------+----------+------------+
| 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>
<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 the 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

The **pprint** module allows you to display Python objects beautifully. This saves the structure of the object. You can use the result that produces **pprint** to create object. The **pprint** module is part of the standard Python library.

The simplest use of module is the `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'}}
```

(continues on next page)

(continued from previous page)

```
   ...:

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'],
 ['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'
```

(continues on next page)

(continued from previous page)

```
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

The pprint() function 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': [],
   ...:   ' 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',
```

(continues on next page)

(continued from previous page)

```
                      ' 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

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 the arguments
- invoke functions when the argument is specified
- display messages with hints of script usage

argparse is not the only module for handling command line arguments. And not even the only one in the standard library.

This book deals only with **argparse**, but in addition it is worth looking at modules that are not part of the standard Python library. For example, click.

Note: A good article, compares different command line argument processing modules (considered argparse, click and docopt).

Example of ping_function.py script:

```
import subprocess
import argparse


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('-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 the 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__
    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 the 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 the 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
```

(continues on next page)

(continued from previous page)

```
  -a IP
  -c COUNT
```

Note that in the message all options are in `optional arguments` section. **argparse** itself determines that options are specified because they start with `-` and only one letter in the name.

Set the IP address as a positional argument (ping_function_ver2.py file):

```
import subprocess
from tempfile import TemporaryFile

import argparse


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 the 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, the script specifies messages that will be displayed when you call *help*. Now the 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
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

```

(continues on next page)

(continued from previous page)

```
# 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)))


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)
```

(continues on next page)

(continued from previous page)

```
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)


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 the 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. The arguments that are added to the 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')
```

The 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')
```

(continues on next page)

(continued from previous page)

```
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults(func=create)
```

The add_argument method adds an argument. Here the 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.

The create() function receives as an argument all the arguments that have been passed. And within the 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 the 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
```

(continues on next page)

(continued from previous page)

```
  -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

The *create_parser* parser uses a new argument - `metavar`:

```
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')
```

The **metavar** argument allows you to specify the argument name to display 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 the 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 the **nargs** list, but at least one argument must be passed.

In this case the *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 on the list. And since the 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 the 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 the *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 the 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 the 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 the script was called with arguments:

```
$ python call_pds2.py
Reading info from file(s)
test.txt, test2.txt
```

(continues on next page)

(continued from previous page)

```
Adding data to db dhcp_snooping.db
```

Tasks

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 12.1

Create a ping_ip_addresses() function that checks if IP addresses are pingable. Function expects as argument a list of IP addresses.

Function should return a tuple with two lists:

- list of reachable IP addresses
- list of unreachable IP addresses

To check availability of IP address, use ping command.

Restriction: All tasks must be performed using only covered topics.

Task 12.2

Function ping_ip_addresses() from task 12.1 accepts only list of addresses, but it would be convenient to be able to specify addresses using a range such as 192.168.100.1-10.

In this task, you need to create a convert_ranges_to_ip_list() function that converts the list of IP addresses in different formats to a list where each IP address is specified separately.

Function expects as argument a list of IP addresses and/or IP address ranges.

List elements can be in the following format:

- 10.1.1.1
- 10.1.1.1-10.1.1.10
- 10.1.1.1-10

If address is specified as a range, you should expand range to separate addresses, including the last address of range. To simplify the task, it can be assumed that only the last octet of address changes in range.

Function returns a list of IP addresses.

For example, if you pass to convert_ranges_to_ip_list() function such a list:

```
['8.8.4.4', '1.1.1.1-3', '172.21.41.128-172.21.41.132']
```

Function should return the list:

```
['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 print_ip_table() function that displays a table of reachable and unreachable IP addresses.

Function expects as arguments two lists:

- list of reachable IP addresses
- list of unreachable IP addresses

The result of function is a table displayed on standard output:

```
Reachable    Unreachable
-----------  -------------
10.1.1.1     10.1.1.7
10.1.1.2     10.1.1.8
             10.1.1.9
```

Function should not change lists passed to it as arguments. That is, lists should look the same before and after function execution.

There are no tests for this task.

13. Iterators, iterable objects and generators

This section discusses:

- iterable objects
- iterators
- generator expressions

Iterable object

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 object 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 iterable objects:

- all sequences: list, string, tuple
- dictionaries
- files

In Python the iter() function is responsible for iterator deriving.

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

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

The `iter()` function will work on any object that has `__iter__` or `__getitem__` method.

The `__iter__` method returns the iterator. If this method is not available, the iter() function checks if there is `__getitem__` method that allows getting elements by index.

If method `__getitem__` is present the 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 the *StopIteration* exception when the 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 object. This method simply returns the iterator itself.

An example of creating an iterator from the list:

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

In [4]: i = iter(numbers)
```

Now you can use the 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 the elements are finished, *StopIteration* exception is returned.

To make iterator to return elements again, it has to be re-created.

Similar actions are performed when loop **for** processes the list:

```
In [9]: for item in numbers:
   ...:     print(item)
   ...:
1
2
3
```

When we search list elements, the iter() function is first applied to the list to create an iterator, and then __next__ method is called until the *StopIteration* exception occurs.

Iterators are useful because they give elements one at a time. For example, when working with a file, it is useful that the 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.

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 we open the file with the normal open() function, we get the object 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 the 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
!
ip ssh version 2
!
```

When working with files, using a file as an iterator does not simply allow iterate the 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 construction 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 can easily create their own iterators. Unlike normal functions, the generator does not just return the value and finish the work, but returns the iterator which gives the elements one by one.

The usual function ends if:

- `return` expression is met
- function code is ended (this works as `return None` expression)
- exception has arisen

After function execution is finished, the control is returned and program execution goes further. All the arguments that were passed to the function, the local variables, all of this is lost. Only the result that returned the function remains.

A function can return a list of elements, multiple objects or different results depending on the arguments, but it always returns a single result.

The generator generates values. The values are then returned on demand and after the return of one value the function-generator is suspended until the next value is requested. Between requests, the generator retains its state.

Python allows generators to be created in two ways:

- generator expression
- generator function

The following is an example of a generator expression and a separate note for generator functions

generator expression

The generator expression uses the same syntax as the list comprehensions, but returns the iterator, not the list.

The generator expression looks exactly the same as the list comprehensions, but the brackets are used:

```
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
```

Note that this is not a tuple comprehensions but a generator expression.

It is useful when working with a large iterable object or infinite iterator.

Additional material

Documentation Python:

- Sequence types
- Iterator types
- Functional Programming HOWTO

Articles:

- Iterables vs. Iterators vs. Generators

4

III. Regular expressions

A regular expression is a sequence of ordinary and special characters. This sequence specifies the 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 the lines from the 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 the log file the lines that correspond to the template.
- get from the 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, the use of regular expressions will involve getting part of the 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 the methods that apply to strings. And if the problem 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 the problems or may make the problem much harder to solve. Regular expressions can help in this case.

14. Regular expression syntax

Regular expression syntax

Python uses **re** module to work with regular expressions. Accordingly, you have to import it to start working with regular expressions.

This section will use the search() function for all examples. And in the next subsection, the rest of the functions of **re** will be considered.

Syntax of the search() function is:

```
match = re.search(regex, string)
```

The search() function has two prerequisites:

- regex - regular expression
- string - string in which search pattern is searched

If a match is found the function will return special object Match. If there is no match, the function will return None.

The feature of the 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 regular expression, search() will return only the first match found.

To get an idea of regular expressions, consider a few examples.

The simplest example of a regular expression 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 the example of string - int_line
- – and in line 3 the 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, the *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 the received match. For example, the group() method shows that the string matches the expression described.

In this case, it's just a 'MTU' substring:

```
In [5]: match.group()
Out[5]: 'MTU'
```

If there was no match, the *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
```

Regular expressions are fully enabled when special characters are used. For example, the symbol \d means a digit, but + means repetition of the 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 the string that describes the 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 the 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 through the regular expression:

```
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')
```

The group() method returns only those parts of the original string that are in brackets. Thus, by

placing a part of the expression in brackets, you can specify which parts of the line you want to remember.

The expression \d+ has been used before - it describes one or more digits. And the 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 the 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 the necessary characters.

For example, get time from the 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'
```

The 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'
```

The 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.

The symbol groups are very convenient, but for now it is necessary to manually specify the character repetition. The following subsection deals with repetition symbols which will simplify the description of expressions.

Repeating characters

- regex+ - one or more repetitions of the preceding element
- regex* - zero or more repetitions of the preceding element
- regex? – zero or one repetition of the preceding element
- regex{n} - exactly 'n' repetitions of the preceding element
- regex{n,m} - from 'n' to 'm' repetitions of the preceding element
- regex{n, } - 'n' or more repetitions of the 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 the 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'
```

(continues on next page)

(continued from previous page)

```
The expresson ``(a1)+`` uses brackets to specify that repetition is related to␣
↪sequence of symbols 'a1'.
```

IP address can be described by \d+\.\d+\.\d+\.\d+. This plus is used to indicate that there can be several digits. And there's also an expression \..

It is required because the point is a special symbol (it denotes any symbol). And in order to indicate that we are interested in a point as a symbol, you have to screen it - put a backslash in front of the point.

Using this expression, you can get an IP address from the sh_ip_int_br string:

```
In [5]: sh_ip_int_br = 'Ethernet0/1    192.168.200.1   YES NVRAM  up          up'

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 the string which has digits first, then whitespace characters, and then the not whitespace characters (all except the 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'
```

*****

The 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.

The 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'
```

In *line* string, if 'b' meets before 'baaa' substring, then the match is 'b':

```
In [11]: line = '100     ab11.baaa.a5d3    FastEthernet0/1'

```

(continues on next page)

(continued from previous page)

```
In [12]: re.search('ba*', line).group()
Out[12]: 'b'
```

Suppose you write a regular expression that describes the email addresses in two formats: user@example.com and user.test@example.com. That is, the left side of the address can have either one word or two words separated by a dot.

The first variant 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'
```

Regular expression 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 regular expression 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, the regular expression 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 the regular expression 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())
    ...:
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 the curly brackets.

For example, the 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 brackets. For example, try to get all VLAN numbers from the 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
```

(continues on next page)

(continued from previous page)

```
    ...:  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, the expression `\d{1,4}` will have the 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
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

The expression `\d{1,4}` describes one to four digits.

Note that the output of the command from equipment does not have a VLAN with number 1. The regular expression got a number 1 from somewhere. The number 1 was in the output from the hostname in the line `sw1#sh mac address-table`.

To correct this, it suffices to complete the 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
```

(continues on next page)

(continued from previous page)

```
VLAN:  1300
```

Special symbols

- . - any character except line feed character
- ^ - beginning of line
- $ - end of line
- [abc] - any symbol in brackets
- [^abc] - any symbol except those in brackets
- a|b - element a or b
- (regex) - expression is treated as one element. In addition, the substring that matches the 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 the subsection “Greedy symbols”.

^

Character ^ means the beginning of line. The expression ^\d+ corresponds to the 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.

The expression \S+$ describes any characters except whitespace at the end of the 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, the expression ^.+[>#] describes characters from the beginning of a line to # or > sign (including them). This expression can be used to derive the name of the 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, it can be stated that we are interested in 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'
```

Similarly, letters can be indicated:

```
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'
```

The expression [a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+ describes three groups of symbols separated by a dot. The characters in each group can be letters a-f or digits 0-9. This expression describes MAC address.

Another feature of the square brackets is that the special symbols within the square brackets lose their special meaning and are simply a symbol. For example, a dot inside the square brackets will denote a dot, not any symbol.

The 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 ^, the 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, the 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, the expression means that we're looking for Fast or 0/1.

()

Brackets are used to group expressions. As in mathematical expressions, brackets can be used to indicate which elements the operation is applied to.

For example, the 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'
```

Brackets 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'
```

Brackets not only allow you to group expressions. The string that matches the bracketed expression is memorized. It can be obtained separately by special methods groups() and group(n). This is considered in the subsection “Grouping of expressions”.

Greedy symbols

By default, repetition symbols in regular expressions are greedy. This means that the resulting substring which corresponds to the template will have the longest match.

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>'
```

That is, in this case, the expression captured the maximum possible piece of symbols contained in <>.

If greedy behavior need to be disabled, it is sufficient to 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 the greed of the last plus, the expression \d+\s+\S+ describes such a 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, the expression \S+ with the greedy repetition symbol describes the maximal 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'
```

Expressions grouping

Expressions grouping indicates that the 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 the string that has been described by the expression.

For example, from a log file you should select strings in which “%SW_MATM-4-MACFLAP_NOTIF” meets and then from each such string get MAC address, VLAN and interfaces. In this case, the regular expression simply has to describe the string and all the parts of the string to be obtained are simply placed in brackets.

Python has two options for using groups:

- Numbered groups
- Named groups

Numbered groups

The group is defined by placing the expression in brackets ().

Inside the expression, group are numbered from left to right starting with 1. Groups can then be approached by numbers and receive text that corresponds to the 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:

- the first group - any characters other than whitespaces
- the second group - any letter or digit (symbol \w) or dot

The second group could be described as the first. The other version is just for example.

You can now access the group by number. Group 0 is a string that corresponds to the entire template:

```
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'

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 display all substrings that correspond to the specified groups:

```
In [18]: match.groups()
Out[18]: ('FastEthernet0/1', '10.0.12.1')
```

Named groups

When the expression is complex, it is not very convenient to determine the number of the 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 the groups.

The named groups allow you to give a name to the group.

Syntax of the 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>[\d.]+)\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 the groupdict() method you can get a dictionary where the keys are the names of groups and the 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 the regular expression and rely on their name instead of order:

```
In [24]: match = re.search('(?P<intf>\S+)\s+(?P<address>[\d\.]+)\s+\w+\s+\w+\s+(?P
↪<status>up|down|administratively 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
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 the regular expression:

- `(?P<mac>\S+) +` - group with name 'mac' matches any characters except whitespace characters. So the expression describes the sequence of any characters before the space
- `(?P<ip>\S+) +` - the same here: a sequence of any non-whitespace characters up to the 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, the 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 the regular expression has worked well, you can create a script. In the script all lines of dhcp_snooping.txt file are iterated and information about the devices is displayed on the standard output stream.

File parse_dhcp_snooping.py:

```
# -*- coding: utf-8 -*-
import re

#'00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10   ␣
↪FastEthernet0/1'
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('{} devices connected to switch'.format(len(result)))

for num, comp in enumerate(result, 1):
    print('Parameters of device {}:'.format(num))
    for key in comp:
        print('{:10}: {:10}'.format(key,comp[key]))
```

Result of implementation:

```
$ 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
```

(continues on next page)

(continued from previous page)

```
    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 brackets are needed to indicate the part of the expression that repeats. And, in doing so, you don't need to remember the expression.

For example, get a MAC address, VLAN and ports from such 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 regular expression that describes the substrings needed:

```
In [2]: match = re.search('((\w{4}\.){2}\w{4}).+vlan (\d+).+port (\S+).+port (\S+)
↪', log)
```

The 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 the brackets 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+)` - the first interface
- `.+port (\S+)` - the second interface

The groups() method returns this result:

```
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 the brackets in the expression `(\w{4}\.){2}`.

In that case, we need to disable the group capturing. This is done by adding `?:` after the group bracket opens.

Now the expression looks like this:

```
In [4]: match = re.search('((?:\w{4}\.){2}\w{4}).+vlan (\d+).+port (\S+).+port␣
↪(\S+)', log)
```

Accordingly, the groups() method result:

```
In [5]: match.groups()
Out[5]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Repeating the captured result

When working with groups, it is possible to use the result that has fallen into the group further in the same expression.

For example, in the output of 'sh ip bgp' the last column describes the 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 the only one group here.

Additionally, the regular expression is preceded by the letter **r**. It is a so-called raw string.

It is more convenient to use it, because otherwise you will have to screen the backslash in order for the link to the group works correctly:

```
match = re.search('(\d+) \\1', line)
```

> **Warning:** When using regular expressions it is best to always use raw string.

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)
   ...:     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
```

Similarly, 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
```

(continues on next page)

(continued from previous page)

```
*  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 the 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 the iterator
- `compile()` - compiles regular expression. You can then apply all of the listed functions to this object
- `fullmatch()` - the entire line must conform to the regular expression described

In addition to functions that search matches, the module has the following functions:

- `re.sub` - for replacement in strings
- `re.split` - to split the 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 deals with 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 the object. Therefore, it is not necessary to rely on what is displayed in the match part as the displayed line is cut by a fixed number of characters.

group()

The group() method 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 the 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 the template and there are more than one match, the 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 brackets describes four letters or numbers, dot and then there is a plus. Accordingly, the first and the second part of the MAC address matched to expression in parentheses. But only the last expression is remembered and returned.

If named groups are used in the expression, the 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()

The group() method returns a tuple with strings in which the elements are those substrings that fall into the 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')
```

The group method has an optional parameter - default. It works when anything that comes into the 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 the space, nothing will get into the group. But the match will be because it is stated in regular expression 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()

The groupdict() method returns a dictionary in which the keys are group names and the 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 regular expression.

If the 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 transfer 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()

The span() method 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 the template
- returns the Match object if a substring is found
- returns None if no substring was found

The search() function is suitable when you need to find only one match in a string, for example when a regular expression describes the entire string or part of a string.

Consider an example of using the search() function to parse a log file.

The log.txt file 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
```

The MAC address can jump between several ports. In this case it is very important to know from which ports the MAC comes.

Try to figure out which ports and which VLAN was the problem. Check regular expression 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)
   ...:
```

The regular expression 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 the 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 obtain 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):

```
1 import re
2 from pprint import pprint
3
4
5 def parse_cdp(filename):
6     result = {}
7
```

(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 regular expression 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)

```
4
5  def parse_cdp(filename):
6      regex = ('Device ID: (?P<device>\S+)'
7               '|IP address: (?P<ip>\S+)'
8               '|Platform: (?P<platform>\S+ \S+),'
9               '|Cisco IOS Software, (?P<ios>.+), RELEASE')
10
11     result = {}
12
13     with open(filename) as f:
14         for line in f:
15             match = re.search(regex, line)
16             if match:
17                 if match.lastgroup == 'device':
18                     device = match.group(match.lastgroup)
19                     result[device] = {}
20                 elif device:
21                     result[device][match.lastgroup] = match.group(
22                         match.lastgroup)
23
24     return result
25
26
27 pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Explanations for the second option:

- in regular expression, all line variants are described via | sign (or)
- without checking a line the match is searched
- if a match is found, the lastgroup() method is checked
- lastgroup() method returns name of the last named group in regular expression for which a match has been found
- if a match was found for the *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',
```

(continues on next page)

(continued from previous page)

```
       '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'}}
```

Match function

Function `match()`:

- is used to search at the beginning of string that corresponds to the template
- returns Match object if substring is found
- returns None if no substring 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 the word *Host* 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+) '
```

(continues on next page)

(continued from previous page)

```
   ...:                 r'is flapping between port '
   ...:                 r'(\S+) and port (\S+)', log)
   ...:
```

The 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')
```

The 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 {} в VLAN {}'.format(', '.join(ports), vlan))
```

The result is:

```
$ python parse_log_match.py
Loop between ports Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Finditer function

Function finditer():

- is used to search for all disjoint matches in template
- returns an iterator with Match objects
- finditer() returns iterator even if no match is found

The finditer() function 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
   ...: '''
```

Regular expression 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 >
```

(continues on next page)

(continued from previous page)

```
<_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 >
<_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:

```
.. code:: python
```

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 generator:

```
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 the 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()
```

(continues on next page)

(continued from previous page)

```
with open('log.txt') as f:
    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 {} в 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 the 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)
```

(continues on next page)

(continued from previous page)

```
                result[device] = {}
            elif device:
                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

...
```

(continues on next page)

(continued from previous page)

```
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

For example, if you want to take the first IP address you can supplement a regular expression 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 disjoint matches in template
- returns:
  - list of strings that are described by the regular expression if there are no groups in regular expression
  - list of strings that match with the regular expression in the group if there is only one group in regular expression
  - list of tuples containing strings that matches with the 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
```

(continues on next page)

(continued from previous page)

```
 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 regular expression without groups. In this case findall() returns a list of strings that matches with regular expression.

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 regular expression, 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 the 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'),
```

(continues on next page)

(continued from previous page)

```
 ('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 desired 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 {} в 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 regular expression 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 regular expression and its use. In addition, using re.compile function creates a RegexObject object that has several additional features that are not present in the MatchObject object.

To compile a regular expression, use re.compile:

```
In [52]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')
```

It returns the 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 the module globally, but now they have to be applied to the 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 the method of *regex* object. And pass the string as an argument.

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 regular expression 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()

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('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

This is a modified example of finditer() usage. Description of regular expression 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 the string from where to start looking for a match
- endpos - specifies from which index the search should be started

Their use is similar to the 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+')
```

(continues on next page)

(continued from previous page)

```
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 functions or creating a compiled regular expression you can specify additional flags that affect the behavior of regular expression.

The **re** module supports such flags (in brackets - a short variant of flag designation):

- 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 considered. Information about other flags is available in documentation.

re.DOTALL

Regular expressions 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: ''
```

(continues on next page)

(continued from previous page)

```
   ...: 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 the necessary data without splitting.

In this expression, the strings with the 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 line feed character:

```
In [4]: print(re.search(regex, sh_cdp))
None
```

You can change the default behavior by using the 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 line feed character is now included, combination `.+` captures everything between data.

Now try to use this regular expression 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
```

(continues on next page)

(continued from previous page)

```
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

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.
```

(continues on next page)

(continued from previous page)

```
Compiled Fri 19-Jun-09 18:40 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Duplex: full
Management address(es):
```

Search for all regular expression 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 the desired parts of the output. Without the `re.DOTALL` flag, such an expression would capture everything before line feed character, but with the flag it captures the longest possible piece of text because + is greedy. As a result, the regular expression describes a string from the first Device ID to the last place where `Cisco IOS Software.+Version` meets.

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+),'
```

(continues on next page)

(continued from previous page)

```
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

The split() function works similary to split() method in strings, but in re.split function you can use regular expressions which means dividing the 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',
 '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']
```

The split() function has a peculiarity of working with groups (expressions in parentheses). If you specify the same expression with parentheses, the resulting list will include the 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 memorization 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

The re.sub function works similary to replace() method in strings. But in re.sub you can use regular expressions 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'

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
```

The regular expression is divided into groups:

- `(\d+)` - the first group. VLAN number gets here
- `([a-f,0-9]+).([a-f,0-9]+).([a-f,0-9]+)` - the following three groups (2, 3, 4) describe MAC address
- `(\S+)` - the fifth group. Describes an interface.

In a second regular expression these groups are used. To refer to a group a backslash and a group number are used. To avoid backslash screening, the *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.

Additional material

Regular expressions in Python:

- Regular expressions in Python from simple to complex. Details, examples, pictures, exercises (in Russian)
- 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

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 15.1

Create a get_ip_from_cfg() function that expects as argument the name of file with device configuration.

Function should process configuration and return IP addresses and masks, that are configured on interfaces, as a list of tuples:

- first element of tuple - IP address
- second element of tuple - 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 function with config_r1.txt file.

Please note that in this case you can skip checking the correctness of IP address, address ranges, etc., since command output is processed, not user input.

Task 15.1a

Copy get_ip_from_cfg() function from task 15.1 and redo it to return dictionary:

- key: interface name
- value: tuple with two strings:
  - IP address
  - mask

Add to dictionary only those interfaces where IP addresses are configured.

For 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 function with config_r1.txt file.

Please note that in this case you can skip checking the correctness of IP address, address ranges, etc., since command output is processed, not user input.

Task 15.1b

Check get_ip_from_cfg() function from 15.1a task based on config_r2.txt.

Note that there are two IP addresses assigned to interface e0/1:

```
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 dictionary that returns get_ip_from_cfg() function, only one IP (second) corresponds to interface Ethernet0/1.

Copy get_ip_from_cfg() function from 15.1a task and redo it so that it returns a list of tuples for each interface. If only one address is assigned to interface, one tuple will be listed. If multiple IP addresses are configured on interface, several tuples will be listed.

Check function with config_r2.txt configuration file and ensure that interface Ethernet0/1 corresponds to list of two tuples.

Please note that in this case you can skip checking the correctness of IP address, address ranges, etc., since command output is processed, not user input.

Task 15.2

Create parse_sh_ip_int_br() function that expects as an argument the name of file in which show ip int br output is found.

Function should process the output of show ip int br command and return such fields:

- Interface
- IP-Address
- Status
- Protocol

Информация должна возвращаться в виде списка кортежей:

```
[("FastEthernet0/0", "10.0.1.1", "up", "up"),
 ("FastEthernet0/1", "10.0.2.1", "up", "up"),
 ("FastEthernet0/2", "unassigned", "down", "down")]
```

Information should be returned in the form of a list of tuples:

To get this result, use regular expressions.

Check function with sh_ip_int_br.txt file.

Task 15.2a

Create convert_to_dict() function that expects two arguments:

- list of field names
- list of tuples with values

Function returns the result as a list of dictionaries, where keys are taken from first list and values are substituted from second list.

For example, if functions pass as arguments *headers* list and list

```
[("R1", "12.4(24)T1", "Cisco 3825"),
 ("R2", "15.2(2)T1", "Cisco 2911")]
```

Function should return such list with dictionaries:

```
[{"hostname": "R1", "ios": "12.4(24)T1", "platform": "Cisco 3825"},
 {"hostname": "R2", "ios": "15.2(2)T1", "platform": "Cisco 2911"}]
```

Function should not be tied to specific data or amount of headers/data in tuples.

Check function with:

- first argument - *headers* list
- second argument - *data* list

Restriction: All tasks must be performed using only covered topics.

```
headers = ["hostname", "ios", "platform"]

data = [
    ("R1", "12.4(24)T1", "Cisco 3825"),
    ("R2", "15.2(2)T1", "Cisco 2911"),
    ("SW1", "12.2(55)SE9", "Cisco WS-C2960-8TC-L"),
]
```

Task 15.3

Create convert_ios_nat_to_asa() function that converts NAT rules from cisco IOS syntax to cisco ASA.

Function expects such arguments:

- name of file containing NAT rules for Cisco IOS
- name of file to which to write resulting NAT rules for ASA

Function does not return anything.

Check function with cisco_nat_config.txt file.

Example of NAT rules for cisco IOS

```
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 corresponding NAT rules for 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 ASA rules file:

- no empty lines between rules
- “object network” lines should not be preceded by spaces
- there should be one space in front of the rest of lines

All ASA rules will have the same interfaces (inside,outside).

Task 15.4

Create get_ints_without_description() function that expects as argument the name of file in which device configuration is found.

Function should process configuration and return a list of interface names that do not have a description (*description* command)..

Example of interface with 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 function with config_r1.txt file.

Task 15.5

Create generate_description_from_cdp() function that expects as an argument the name of file containing show cdp neighbors command output.

Function should process the output of show cdp neighbors command and generate a description for interfaces based on command output.

For example, if R1 has this 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 interface Eth 0/0, you should generate this description `description Connected to SW1 port Eth 0/1`.

Function should return a dictionary where keys are interface names and values are command that defines interface description:

```
"Eth 0/0": "description Connected to SW1 port Eth 0/1"
```

Check function with sh_cdp_n_sw1.txt file.

5

IV. Data writing and transferring

This part of the book deals with data writing and transferring. 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 considered - writing information to a plain text file.

This section deals with 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 scripts. 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 the objects of language itself to be written into files and read through the Pickle module, but this aspect is not considered in this book.

Databases are also discussed in this part. Although you can write data to CSV or JSON based on a structure, it is not always convenient to query information from files in this format. This is particularly the case for more complex requests with more than one criterion.

For tasks of this kind, databases are excellent. Section 18 deals with SQLite as well as the basics of SQL language.

16. Unicode

The 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 the text, computer takes the 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 the text is an immutable sequence of codes (codepoints) Unicode.

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 12.1 (May 2019) describes 137 994 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, the letter "s" corresponds to the 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 the 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
```

The ord() function returns the value of Unicode code for the character:

```
In [6]: ord('ö')
Out[6]: 246
```

The chr() function returns the 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 the addition of letter “b” before the 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.

The 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 the string-byte and byte-string transformations must use the same encoding.

encode, decode

encode method 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 the *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 very simple rule, one that can avoid at least part of the problem. It's 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 transfer

Examples of converting between bytes and strings

Consider a few examples of working with bytes and converting bytes to string.

subprocess

The subprocess module 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)
   ...:
```

(continues on next page)

(continued from previous page)

```
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'
```

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
```

The subprocess module supports another conversion option - *encoding* parameter. If you specify it when you call the 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
```

(continues on next page)

(continued from previous page)

```
rtt min/avg/max/mdev = 53.368/54.534/55.564/0.941 ms
```

telnetlib

Depending on the module, conversion between strings and bytes can be performed automatically or may be required explicitly.

For example, the telnetlib module must transfer 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 the penultimate line uses decode.

pexpect

The pexpect module 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'
```

(continues on next page)

(continued from previous page)

```
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

So far, the following construction has been used to handle files:

```
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 the 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 the 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='')
   ...:
```

(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
!
```

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 the 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 cannot convert to Cyrillic bytes:

```
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')
```

(continues on next page)

(continued from previous page)

```
In [36]: hi_bytes.decode('ascii')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-36-aa0ada5e44e9> in <module>()
----> 1 hi_bytes.decode('ascii')

UnicodeDecodeError: 'ascii' codec can't decode byte 0xd0 in position 0: ordinal␣
↪not in range(128)
```

Another variant 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 me 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 the mode to ignore, as in encode, the 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'
```

Additional material

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 the objects of the 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 the following section covers databases.

For which YAML, JSON, CSV formats can be useful:

- you may have data about IP address and similar information to process in tables
  - o 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 because it has a nice syntax
  - for example, it can be settings for different objects (IP addresses, VLANs, etc.)
  - at least knowing the 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 the format name the separator can be not only a comma.

Although formats with a different separator may have their own name such as TSV (tab separated values), CSV is generally understood by all separators.

Example of a CSV file (sw_data.csv):

```
hostname,vendor,model,location
sw1,Cisco,3750,London
sw2,Cisco,3850,Liverpool
```

(continues on next page)

(continued from previous page)

```
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 the column names and remaining list contains the corresponding values.

Note that csv.reader returns the 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']]
```

(continues on next page)

(continued from previous page)

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 obtain dictionaries in which keys are column names and values are column values.

For this purpose, the 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
OrderedDict([('hostname', 'sw1'), ('vendor', 'Cisco'), ('model', '3750'), (
↪'location', 'London')])
sw1 3750
OrderedDict([('hostname', 'sw2'), ('vendor', 'Cisco'), ('model', '3850'), (
↪'location', 'Liverpool')])
sw2 3850
OrderedDict([('hostname', 'sw3'), ('vendor', 'Cisco'), ('model', '3650'), (
↪'location', 'Liverpool')])
sw3 3650
OrderedDict([('hostname', 'sw4'), ('vendor', 'Cisco'), ('model', '3650'), (
↪'location', 'London')])
sw4 3650
```

Dictreader does not create standard Python dictionaries but ordered dictionaries. Thus, the order of elements corresponds to order of the columns in the CSV file.

Note: Prior to Python 3.6 regular dictionaries were returned, not ordered dictionaries.

Otherwise, it is possible to work with ordered dictionaries using the same methods as in regular dictionaries.

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 the example above, strings from the 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 the quotation marks 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 quotation marks indicate where the value begins and ends.

The csv module allows you to control this. For all strings to be written in a CSV file with quotation marks you should change the 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'],
        ['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 the model number is given as a string in original list, it is quoted here as well.

Besides writerow() method, the 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'],
```

(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)
    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'
}]
```

(continues on next page)

(continued from previous page)

```
24
25 with open('csv_write_dictwriter.csv', 'w') as f:
26     writer = csv.DictWriter(
27         f, fieldnames=list(data[0].keys()), quoting=csv.QUOTE_NONNUMERIC)
28     writer.writeheader()
29     for d in data:
30         writer.writerow(d)
```

Delimiter

Sometimes other values are used as the separator. In this case, it should be possible to tell the 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):

```
1 import csv
2
3 with open('sw_data2.csv') as f:
4     reader = csv.reader(f, delimiter=';')
5     for row in reader:
6         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']}
```

(continues on next page)

(continued from previous page)

```
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):

```
1  import json
2
3  with open('sw_templates.json') as f:
4      file_content = f.read()
5      templates = json.loads(file_content)
6
7  print(templates)
8
9  for section, commands in templates.items():
10     print(section)
11     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:
    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}

```

(continues on next page)

(continued from previous page)

```
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 the person, this format is not very convenient to perceive.

Fortunately, the 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"
```

(continues on next page)

(continued from previous page)

```
]
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)
    ...:
    ...:

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 scripts in software. 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. The 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
  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 the csv and json modules.

Reading from YAML

Try 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
  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,
```

(continues on next page)

(continued from previous page)

```
  '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):

```
1  import yaml
2
3  trunk_template = [
4      'switchport trunk encapsulation dot1q', 'switchport mode trunk',
5      'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
6  ]
7
8  access_template = [
9      'switchport mode access', 'switchport access vlan',
10     'switchport nonegotiate', 'spanning-tree portfast',
11     'spanning-tree bpduguard enable'
12 ]
13
14 to_yaml = {'trunk': trunk_template, 'access': access_template}
15
16 with open('sw_templates.yaml', 'w') as f:
17     yaml.dump(to_yaml, f, default_flow_style=False)
18
19 with open('sw_templates.yaml') as f:
20     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
```

Additional material

In this section only basic read and write operations were considered 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

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 17.1

Create write_dhcp_snooping_to_csv() function that processes the output of show dhcp snooping binding command from different files and writes processed data to csv file.

Function arguments:

- filenames - list with file names with show dhcp snooping binding outputs
- output - file name in csv format to which the result will be written

Function does not return anything.

For example, if a list with one sw3_dhcp_snooping.txt file 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 have such 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
```

Check function with content of sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt files. The first column in a csv file should be retrieved from filename, the rest from contents of files.

Task 17.2

This task requires:

- take contents of several files with sh version command output
- parse command output with regular expressions and get device information
- write the received information to a CSV file

To complete task you need to create two functions.

Function parse_sh_version():

- expects as an argument the output of sh version command as one string (not file name)
- handles output using regular expressions
- returns a tuple of three elements:
  - ios - in format "12.4(5)T"
  - image - in format "flash:c2800-advipservicesk9-mz.124-5.T.bin"
  - uptime - in format "5 days, 3 hours, 3 minutes"

Function write_inventory_to_csv() should have two parameters:

- data_filenames - expects as argument list of file names with sh version command output
- csv_filename - expects as argument the name of file (for example, routers_inventory.csv) into which information will be written in CSV format

Function write_inventory_to_csv() writes content to a CSV file and returns nothing

Function write_inventory_to_csv() 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
- with parse_sh_version() function, information like ios, image, uptime should be obtained from each output
- hostname should be received from file name
- then all information should be written in CSV file

File routers_inventory.csv should have such columns: `hostname, ios, image, uptime`

In script, using glob module, a list of files whose name begins on *sh_vers* is created. You can uncomment string *print(sh_version_files)* to see the contents of list.

In addition, a list of headers was created that should be written in CSV.

```
import glob

sh_version_files = glob.glob("sh_vers*")
#print(sh_version_files)
```

(continues on next page)

(continued from previous page)

```
headers = ["hostname", "ios", "image", "uptime"]
```

Task 17.3

Create parse_sh_cdp_neighbors() function that handles the output of show cdp neighbors command.

Function expects as an argument the output of command as a string (not file name). Function should return a dictionary that describes connections between devices.

For example, if such output is given 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 such dictionary:

```
{"R4": {"Fa 0/1": {"R5": "Fa 0/1"},
        "Fa 0/2": {"R6": "Fa 0/0"}}}
```

Interfaces should be written with space. That is Fa 0/0, not Fa0/0.

Check function with contents of sh_cdp_n_sw1.txt file.

Task 17.3a

Create generate_topology_from_cdp() function that handles the output of show cdp neighbor command from multiple files and writes the resulting topology in one dictionary.

Function generate_topology_from_cdp() should be created with parameters:

- list_of_files - list of files from which to read the output of sh cdp neighbor command
- save_to_filename - name of file in YAML format that stores topology.
  - default value - None. By default, topology is not saved to file
  - topology is saved only if file name is specified for save_to_filename argument

Function should return a dictionary that describes connections between devices, regardless of whether topology is saved to file.

The structure of dictionary should be:

```
{"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 should be written with space. That is Fa 0/0, not Fa0/0.

Check generate_topology_from_cdp() function with 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 save_to_filename option and write the resulting dictionary to topology.yaml file.

Task 17.3b

Create transform_topology() function that converts topology into a format suitable for draw_topology() function.

Function expects as argument the name of file in YAML format in which topology is stored.

Function should read data from YAML file, convert it accordingly, so that the function returns a dictionary of this type:

```
{("R4", "Fa 0/1"): ("R5", "Fa 0/1"),
 ("R4", "Fa 0/2"): ("R6", "Fa 0/0")}
```

Function transform_topology() should not only change the format of topology representation but also remove duplicate connections (these are best seen in diagram generated by draw_topology).

Check function with topology.yaml file (should be created in previous task 17.2a). Based on resulting dictionary, you should generate a topology image using draw_topology() function. Do not copy draw_topology() function code.

Result should look the same as scheme in task_17_3b_topology.svg file

At the same time:

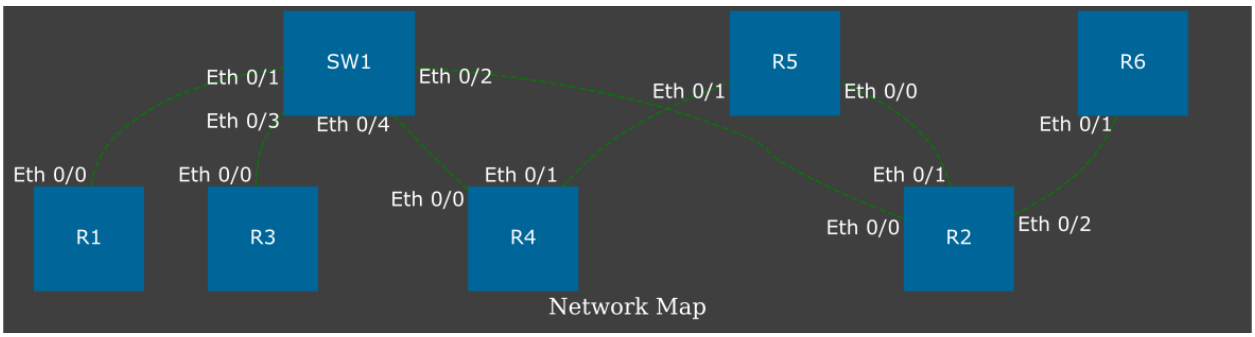
- Interfaces should be written with space Fa 0/0
- The arrangement of devices on diagram may be different

- Connections should follow the diagram
- There should be no duplicate links on diagram

Note: To complete this task, graphviz should be installed: apt-get install graphviz

And python module for working with graphviz: pip install graphviz

---



Task 17.4

Create write_last_log_to_csv() function.

Function arguments:

- source_log - name of csv file from which data is read (example mail_log.csv)
- output - name of csv file to which the result will be written

Function does not return anything.

Function write_last_log_to_csv() handles mail_log.csv file. File mail_log.csv contains logs of user name changes. At the same time, user cannot change email, only name.

Function write_last_log_to_csv() should select only the most recent entries for each user from mail_log.csv file and write them to another csv file. The *output* file should have column headers as a first line, similar to source_log file.

Some users have only one record and then only it should be written to final file. Some users have multiple entries with different names. For example, user with c3po@gmail.com has changed his name 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 entries, only one should be written to final file - the latest:

```
C-3PO,c3po@gmail.com,16/12/2019 17:24
```

It is convenient to use datetime objects from *datetime* module to compare dates. To simplify work with dates, convert_str_to_datetime() function was created - it converts a string with a date in format 11/10/2019 14:05 into an datetime object. The resulting datetime objects can be compared. The second convert_datetime_to_str() function does the reverse operation - converts datetime object into a string.

Functions convert_str_to_datetime() and convert_datetime_to_str() are not necessary to use.

```
import datetime


def convert_str_to_datetime(datetime_str):
    """
    Converts a string with a date in format 11/10/2019 14:05 into an datetime␣
↪object.
    """
    return datetime.datetime.strptime(datetime_str, "%d/%m/%Y %H:%M")


def convert_datetime_to_str(datetime_obj):
    """
    Converts datetime object into a string with a date in format 11/10/2019 14:05.
    """
    return datetime.datetime.strftime(datetime_obj, "%d/%m/%Y %H:%M")
```

6

V. Working with network equipment

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 equipment

This section discusses how to connect to equipment via:

- SSH
- Telnet

Python has several modules that allow you to connect to equipment 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 equipment 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 equipment

This section deals with all four 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 SSH.

The parameters used in the section:

- user: cisco
- password: cisco
- password for enable mode: cisco
- SSH version 2
- 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 the connection it is necessary to decide how the password will be transmitted:

- Request password at start of the script and read user input. Disadvantage is that you can see which characters the 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 desirable that 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 install:

```
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 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 the 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, the connection is established. Now you must specify which line to expect. In this case, wait for the password request:

```
In [6]: ssh.expect('[Pp]assword')
Out[6]: 0
```

Note how the line that pexpect expects is described: `[Pp]assword`. This is a regular expression that describes a *password* or *Password* string. That is, the expect() method can be used to pass a regular expression 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. The index appears here because you can transfer a list of strings. For example, you can transfer 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 the password using *sendline* command:

```
In [9]: ssh.sendline('cisco')
Out[9]: 6
```

Command *sendline* sends a string, automatically adds a line feed 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 options 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 pointed till which moment it should 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
```

The 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 template can be used:

- regular expression
- EOF - this template allows you to react to the EOF exception

- TIMEOUT - timeout exception (default timeout = 30 seconds)
- compiled re

Another very useful feature of pexpect.expect is that you can pass not one 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 the 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 the 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 the 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 #, the value 1 will be returned and in this case a message is displayed, that the 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 the 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'
```

(continues on next page)

(continued from previous page)

```
                'Lo0                                up             up       \n'
                'Lo9                                up             up       \n'
                'Lo19                               up             up       \n'
                '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)
```

(continues on next page)

(continued from previous page)

```
        ssh.sendline(command)
        output = ""

        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 line feed). Therefore, send() method is used here, not sendline - sendline automatically adds a line feed.

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 the telnet client implementation.

Note: It is also possible to connect via telnet using pexpect. Plus 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 the transfer of a byte string, rather than normal one.

The 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 the specified string.

Method write

Method write() is used for data transmission. Byte string has to be passed to it:

```
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>'
```

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
```

(continues on next page)

(continued from previous page)

```
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'>')
```

(continues on next page)

(continued from previous page)

```
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 transfer 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 the 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'[>#]'])

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.

File 2_telnetlib.py:

```
import telnetlib
import time
from pprint import pprint


def to_bytes(line):
    return f"{line}\n".encode("utf-8")


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)
```

(continues on next page)

(continued from previous page)

```
        pprint(result, width=120)
```

Since bytes need to be passed to write() method and line feed should be added each time, a small function to_bytes() is created that does the conversion to bytes and adds a line feed.

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'
                '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'
```

(continues on next page)

(continued from previous page)

```
                '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'
                 '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"#"])
```

(continues on next page)

(continued from previous page)

```
        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 = ""

        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. We will consider only client functionality.

Since Paramiko is not part of standard Python module library, it needs to be installed:

```
pip install paramiko
```

The connection is established in this way: first, client is created and client configuration is set, then connection is initiated and an interactive session is received:

```
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 the 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 the 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 brackets, 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#'
```

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,
```

(continues on next page)

(continued from previous page)

```
    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)
        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'
           '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,
```

(continues on next page)

(continued from previous page)

```
    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(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"]
```

(continues on next page)

(continued from previous page)

```
    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.

Netmiko first needs to install:

```
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', # predefined device type
                'ip': '192.168.1.1', # device IP address
                'username': 'user', # username
                'password': 'userpass', # user password
                'secret': 'enablepass', # enable password
                'port': 20022, # port SSH, by default 22
                 }
```

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')
```

The method works as follows:

- sends command to device and gets the output until the string with prompt or until the 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` - till which string read 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 of use:

```
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: Above 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 sufficient in the dictionary that defines connection parameters specify device type 'cisco_ios_telnet':

```
device = {
    "device_type": "cisco_ios_telnet",
    "ip": "192.168.100.1",
```

(continues on next page)

(continued from previous page)

```
    "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:
        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",
        "ip": "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,
)


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'
                 '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


```

(continues on next page)

(continued from previous page)

```
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:
                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"))
```

Additional matterial

Documentation:

- pexpect
- telnetlib
- paramiko Client
- paramiko Channel
- netmiko
- threading
- multiprocessing
- queue
- time

- datetime
- getpass

Articles:

- Netmiko Library
- Automate SSH connections with netmiko
- Network Automation Using Python: BGP Configuration

Tasks

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 18.1

Create send_show_command() function.

Function connects via SSH (using netmiko) to one device and performs specified command.

Function parameters:

- device - dictionary with device connection parameters
- command - command to execute

Function returns a string with command output.

Script should send *command* command to all devices from device.yaml file using send_show_command() function.

```
command = "sh ip int br"
```

Task 18.1a

Copy send_show_command() function from task 18.1 and redo it to process the exception that is generated when authentication on device fails.

If error occurs, exception message should be displayed on standard output stream.

To verify this, change your password on device or in devices.yaml

Task 18.1b

Copy send_show_command() function from task 18.1a and redo it in such a way that exception is generated not only when authentication on device fails, but also when device’s IP address is not available.

If error occurs, exception message should be displayed on standard output stream.

To verify this, change your password on device or in devices.yaml

Task 18.2

Create send_config_commands() function

Function connects via SSH (using netmiko) to device and performs a list of commands in configuration mode based on passed arguments.

Function parameters:

- device - dictionary with device connection parameters
- config_commands - list of commands to execute

Function returns a string with command output.

```
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
R1(config)#logging buffered 20010
R1(config)#no logging console
R1(config)#end
R1#
```

Script should send *command* command to all devices from device.yaml file using send_config_commands() function.

```
commands = [
    'logging 10.255.255.1', 'logging buffered 20010', 'no logging console'
```

(continues on next page)

(continued from previous page)

```
]
```

Task 18.2a

Copy send_config_commands() function from task 18.2 and add *verbose* parameter that controls whether information about to which device connection is established will be displayed in output.

Note: verbose - parameter of send_config_commands() function, not parameter of ConnectHandler!

By default, the result should be displayed.

Example of function execution:

```
In [13]: result = send_config_commands(r1, commands)
Connection to 192.168.100.1...

In [14]: result = send_config_commands(r1, commands, verbose=False)

In [15]:
```

Script should send commands list to all devices from devices.yaml file using the send_config_commands() function.

Task 18.2b

Copy send_config_commands() function from task 18.2a and add error check.

When executing each command, script should check the result for such errors:

- Invalid input detected, Incomplete command, Ambiguous command

If error occurs during execution of any of commands, function should output a message to standard output stream with information about: which error occurred, which command caused it and on which device. For example: “logging” command was executed with error “Incomplete command.” on device 192.168.100.1

Errors should always be displayed regardless of *verbose* parameter value. However, *verbose* still has to control whether the message will be displayed: Connecting to 192.168.100.1...

Function send_config_commands() should now return a tuple with two dictionaries:

- first dictionary with commands output that executed without error
- second dictionary with commands output that executed with errors

Both dictionaries in format:

- key - command
- value - output with execution of commands

Function can be checked on one device.

Example of send_config_commands() function execution:

```
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'
                            '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'
```

(continues on next page)

(continued from previous page)

```
            '% 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 send_config_commands() function from 18.2b task and redo it in the following way: If you have error when executing a command, ask user if you need to execute other commands.

Response options [y]/n:

- y - to execute other commands. This is the default, so pressing any combination is perceived as “y”
- n or no - do not execute other commands

Function send_config_commands() should still return a tuple with two dictionaries:

- first dictionary with commands output that executed without error
- second dictionary with commands output that executed with errors

Both dictionaries in format:

- key - command
- value - output with execution of commands

Function can be checked on one device.

Example of function execution:

```
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'
             '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 send_commands() function (netmiko is used to connect via SSH).

Function parameters:

- device - dictionary with device connection parameters
- show - one show command (string)
- config - list of commands to execute in configuration mode

Depending on which argument is passed, the function calls different functions within. When calling send_commands(), only one argumnet will be passed - show or config.

Then follows the combination of argument and corresponding fucntion:

- show - send_show_command() function from task 18.1
- config - send_config_commands() function from task 18.2

Function returns string with execution results of command or commands.

Check function with:

- *commands* - list of commands
- *command* - command

Example of function execution:

```
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 have to poll many devices, the connections will take quite a long time to connect in turn. Of course, this will be faster than manual connection but we'd like to get response as soon as possible.

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.

Module concurrent.futures is used for parallel connection to devices in this section.

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 of use:

```
from datetime import datetime
import time

start_time = datetime.now()

#Code is running here
time.sleep(5)

print(datetime.now() - start_time)
```

Result of execution:

```
$ 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 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 rather 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 variant of concurrent connections to device: asynchronous programming. This option is not discussed 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. Here the optimal number can be considered as 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 “will fit” on each other. This peculiarity appears very often, so do not trust the variant when “everything works” on basic examples, follow the rules of working with threads.

Before dealing with rules we have to deal with the term “thread safety”. Thread safety is a concept that describes work with multithreaded programs. The 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 the work 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 considered 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, the 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 the 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 considered 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 consider the fundamentals of logging module. It will be used instead of print() function which is not inherently thread-safe.

Module logging

Module logging - a module from the standard Python 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 variant, 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 should write `logging.info("test")`.

Example of script with logging settings: (logging_basics.py file)

```
from datetime import datetime
import logging
import netmiko
import yaml


# эта строка указывает, что лог-сообщения paramiko будут выводиться
# только если они уровня WARNING и выше
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
*12:26:18.137 UTC Wed Jun 5 2019
MainThread root INFO: ===> 12:26:18.413913 Connection: 192.168.100.2
```

(continues on next page)

(continued from previous page)

```
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

The concurrent.futures module 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 more understandable.

Concurrent.futures module allows to solve the problem of starting multiple threads/processes and getting data from them. For this purpose, the 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 the 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. Shutdown is responsible for the completion of threads/processes, when 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 called object. However, only functions will be considered further.

The shutdown() method indicates that the 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 the script itself will not exit until all the 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, the 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 the built-in map function: applying the func() function to one or more iterable objects. Each call to a function is then started in a separate thread/process. Method map() returns the iterator with function results for each element of the 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, the 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):

```
1  from datetime import datetime
2  import time
3  from itertools import repeat
4  from concurrent.futures import ThreadPoolExecutor
5  import logging
6
7  import netmiko
8  import yaml
9
10
11 logging.getLogger('paramiko').setLevel(logging.WARNING)
12
```

(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['ip']
    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['ip'], output)
```

Since function should be passed to map() method, the 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['ip']
    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))
```

(continues on next page)

(continued from previous page)

```
        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 the 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['ip'], output)
```

- `with ThreadPoolExecutor(max_workers=3) as executor:` - ThreadPoolExecutor class is initiated in *with* block with the 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 the 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 the 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 the 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 the 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:

```
1  from concurrent.futures import ThreadPoolExecutor
2  from pprint import pprint
3  from datetime import datetime
4  import time
5  from itertools import repeat
6  import logging
7
8  import yaml
9  from netmiko import ConnectHandler, NetMikoAuthenticationException
10
11
12 logging.getLogger('paramiko').setLevel(logging.WARNING)
13
14 logging.basicConfig(
15     format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
16     level=logging.INFO)
17
18
19 def send_show(device_dict, command):
20     start_msg = '===> {} Connection: {}'
21     received_msg = '<=== {} Received:   {}'
```

(continues on next page)

(continued from previous page)

```
    ip = device_dict['ip']
    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['ip']] = 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'))

```

The example is generally similar to the previous one but NetMikoAuthenticationException was introduced in the send_show() function, and the code that started send_show() function in the 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 the 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, the code can be executed further. And once all functions in threads are running, you can start requesting Future if the results are ready. Or take advantage of special function as_completed(), which requests the result itself and the 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 resquested for status (completed or not), and results or exceptions can be obtained from the work. Future does not need to create 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['ip']
    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()
```

(continues on next page)

(continued from previous page)

```
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    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)
    # то же самое в виде 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 only the 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 the 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>
<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>
```

(continues on next page)

(continued from previous page)

```
<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['ip']
    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}


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['ip']))
        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>
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 the 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

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)

```
start_msg = '===> {} Connection: {}'
received_msg = '<=== {} Received: {}'


def send_show(device_dict, command):
    ip = device_dict['ip']
    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
```

(continues on next page)

(continued from previous page)

```
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)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
```

(continues on next page)

(continued from previous page)

```
    received_msg = '<=== {} Received:   {}'
    ip = device_dict['ip']
    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['ip']] = 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',
 '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['ip']
    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:
        future_ssh = [
            executor.submit(send_show, device, command) for device in devices
        ]
        for f in as_completed(future_ssh):
            try:
```

(continues on next page)

(continued from previous page)

```
                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'}
```

Additional material

GIL

- Can't we get rid of the Global Interpreter Lock?
- GIL (in Russian)
- Understanding the Python GIL
- Python threads and the GIL

concurrent.futures

Python documentation:

- concurrent.futures — Launching parallel tasks
- PEP 3148
- PyMOTW. concurrent.futures — Manage Pools of Concurrent Tasks

Articles:

- 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

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 19.1

Create ping_ip_addresses() function that checks if IP addresses are pingable. Checking IP addresses should be performed in parallel in different threads.

Function parameters:

- ip_list - list of IP addresses
- limit - maximum number of parallel threads (default 3)

Function should return a tuple with two lists:

- list of available IP addresses
- list of unreachable IP addresses

You can create any additional functions to complete task.

To check availability of IP address, use ping.

Note: concurrent.futures hint: 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 with concurrent.futures (ping_ip_addresses() function should do this).

Task 19.2

Create send_show_command_to_devices() function that sends the same show command to different devices in parallel threads and then writes output of commands to a file. Output from devices in file can be in any order.

Function parameters:

- devices - list of dictionaries with connection parameters to devices
- command - command
- filename - name of text file into which the output of all commands will be written

- limit - maximum number of parallel threads (default 3)

Function does not return anything.

Output of commands should be written to a plain text file in this format (you should write host name and command itself before output of command):

```
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 task.

Check function with devices from device.yaml file

Task 19.3

Create send_command_to_devices() function that sends different show commands to different devices in parallel threads and then writes the output of commands to a file. Output from devices in file can be in any order.

Function parameters:

- devices - list of dictionaries with devices connection parameters
- commands_dict - dictionary that specifies which device to send which command. Example dictionary - *commands*
- filename - name of file into which the outputs of all commands will be written
- limit - maximum number of parallel threads (default 3)

Function does not return anything.

Output of commands should be written to a plain text file in this format (you should write host name and command itself before output of command):

```
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
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
```

You can create any additional functions to complete task.

Check function with devices from device.yaml file and commands dictionary

```
# This dictionary is only needed to check code operation, you can change IP␣
↪addresses in it
# test takes addresses from device.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 send_command_to_devices() function that sends a list of specified show commands to different devices in parallel threads and then writes the output of commands to a file. Output from devices in file can be in any order.

Function parameters:

- devices - list of dictionaries with devices connection parameters
- commands_dict - dictionary that specifies which device to send which command. Example dictionary - *commands*
- filename - name of file into which the outputs of all commands will be written
- limit - maximum number of parallel threads (default 3)

Function does not return anything.

Output of commands should be written to a plain text file in this format (you should write host name and command itself before output of command):

```
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 in file can be in any order.

To complete task you can create any additional functions and use functions created in previous tasks.

Check function with devices from device.yaml file and *commands* dictionary

```
# This dictionary is only needed to check code operation, you can change IP␣
↪addresses in it
# test takes addresses from device.yaml file
commands = {
    "192.168.100.3": ["sh ip int br", "sh ip route | ex -"],
    "192.168.100.1": ["sh ip int br", "sh int desc"],
    "192.168.100.2": ["sh int desc"],
}
```

Task 19.4

Create send_commands_to_devices() function that sends show or config command to different devices in parallel threads and then writes output to a file.

Function parameters:

- devices - list of dictionaries with devices connection parameters
- show - show command to send (default None)
- config - configuration mode commands to send (default None)
- filename - name of file into which outputs of all commands will be written
- limit - maximum number of parallel threads (default 3)

Function does not return anything.

Output of commands should be written to a file in this format (you should write host name and command itself before output of command):

```
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
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
```

Example of function call:

```
In [5]: send_commands_to_devices(devices, show='sh clock', filename='result.txt')

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
```

(continues on next page)

(continued from previous page)

```
*04:56:40.354 UTC Sat Mar 23 2019

In [11]: send_commands_to_devices(devices, config='logging 10.5.5.5', filename=
↪'result.txt')

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]: send_commands_to_devices(devices,
                                  config=['router ospf 55', 'network 0.0.0.0 255.
↪255.255.255 area 0'],
                                  filename='result.txt')

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.

7

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.

Although OOP is the basis of how everything works in Python, it is not necessary to use an object-oriented approach when writing code.

The point here is that in Python you don't have to create classes to do something.

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

Consider a practical example of OOP use.

In section “18. Working with databases” the first thing to do to work with database - connect to it:

```
In [1]: import sqlite3

In [2]: conn = sqlite3.connect('dhcp_snooping.db')
```

conn variable - an object that represents a real database connection. Using type() function you can find out which class instance the conn object belongs to:

```
In [3]: type(conn)
Out[3]: sqlite3.Connection
```

conn has its own methods and variables that depend on the state of current object. For example, conn.in_transaction instance variable is available in each instance of sqlite3.Connection class and returns True or False depending on whether all changes are commited:

```
In [15]: conn.in_transaction
Out[15]: False
```

Method execute() executes SQL command:

```
In [19]: query = 'insert into dhcp (mac, ip, vlan, interface) values (?, ?, ?, ?)'

In [5]: conn.execute(query, ('0000.1111.7777', '10.255.1.1', '10', 'Gi0/7'))
Out[5]: <sqlite3.Cursor at 0xb57328a0>
```

conn object saves the state: now instance variable conn.in_transaction returns True:

```
In [6]: conn.in_transaction
Out[6]: True
```

After calling commit() method, it is again False:

```
In [7]: conn.commit()

In [8]: conn.in_transaction
Out[8]: False
```

This example illustrates important aspects of OOP: data integration, data handling and state preservation.

So far in code writing, data and actions on data have been separated. Most often, actions are described as functions and data are transmitted as arguments to these functions. When creating a class, data and actions are combined. Of course, these data and actions are connected. That is, class methods become those actions that are specific to this type of object, not some arbitrary action.

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: By example with a string, it is clear that 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: invoke lower() method on **s** object. Invoking methods and variables is the same, but to call a method you have to add brackets 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 derive 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()

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('Hostname: {}\nModel: {}'.format(sw_obj.hostname, 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):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, 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 invoke method from instance itself. Another unclear thing - why we wrote *self* then?

The point is that 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 variant, so it is almost always used.

Note: When a class instance method is called the instance reference is passed by the first argument. 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)
```

(continues on next page)

(continued from previous page)

```
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 the method. Parameter **self** is a reference to a particular instance of the 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('Hostname: {}\nModel: {}'.format(sw_object.hostname, sw_
↪object.model))
    ...:
```

It will work the same way:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'

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 the class. First, the option of creating a usual variable within method:

```
In [5]: class Switch:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = ['{}{}'.format(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('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
   ...:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = ['{}{}'.format(intf_type, number) for number in␣
↪range(1, number_of_intf+1)]
   ...:         self.interfaces = interfaces
   ...:
```

Now, after generate_interfaces method is called the *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('Hostname: {}\nModel: {}'.format(self.hostname, 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('Hostname: {}\nModel: {}'.format(self.hostname, 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
```

(continues on next page)

(continued from previous page)

```
    ...:         self.model = model
    ...:
    ...:     def info(self):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, 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 use `__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.

Visibility area

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 the example below in `__init__` method, *hostname* and *model* variables are assigned to an instance and then used in info() due to the instance being passed as a first argument:

```
In [32]: class Switch:
    ...:     def __init__(self, hostname, model):
    ...:         self.hostname = hostname
    ...:         self.model = model
```

(continues on next page)

(continued from previous page)

```
    ...:
    ...:     def info(self):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, self.
↪model))
    ...:
```

Class variables

In addition to instance variables, there are also class variables. They are created when variables are specified within class itself, not method:

```
In [27]: class A:
    ...:     var_a = 5
    ...:
    ...:     def method(self):
    ...:         pass
    ...:
```

Now not only class but every instance of the class will have var_a variable:

```
In [40]: A.var_a
Out[40]: 5

In [30]: a1 = A()

In [31]: a1.var_a
Out[31]: 5

In [32]: a2 = A()

In [33]: a2.var_a
Out[33]: 5
```

An important point when using class variables is that within method they should still be called through name of the class (or **self**, but through name of the class better because then it is clear that it is a class variable). First, variant without class name:

```
In [37]: class A:
    ...:     var_a = 5
    ...:
    ...:     def method(self):
    ...:         print(var_a)
```

(continues on next page)

(continued from previous page)

```
    ...:

In [38]: a1 = A()

In [39]: a1.method()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-39-921b8753dbee> in <module>()
----> 1 a1.method()

<ipython-input-37-ef925c4e39d3> in method(self)
      3
      4     def method(self):
----> 5         print(var_a)
      6

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

And correct variant:

```
In [47]: class A:
    ...:     var_a = 5
    ...:
    ...:     def method(self):
    ...:         print(A.var_a)
    ...:

In [48]: a1 = A()

In [49]: a1.method()
5
```

Tasks

Warning: Starting from section "9. Functions" there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 22.1

Create Topology class that represents network topology.

When creating an instance of class, dictionary that describes topology is passed as an argument. Dictionary may contain duplicate connections.

Duplicate refers to situation where dictionary contains such couples:

```
('R1', 'Eth0/0'): ('SW1', 'Eth0/1') и ('SW1', 'Eth0/1'): ('R1', 'Eth0/0')
```

Each instance should have *topology* variable that contains topology dictionary, but without duplicates.

Example of class instance creation:

```
In [2]: top = Topology(topology_example)
```

After that, *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'),
                    ('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'),
```

(continues on next page)

(continued from previous page)

```
                    ('SW1', 'Eth0/2'): ('R2', 'Eth0/0'),
                    ('SW1', 'Eth0/3'): ('R3', 'Eth0/0')}
```

Task 22.1a

Copy Topology class from task 22.1 and change it.

If in task 22.1 duplicates removal was performed in __init__() method, it is necessary to transfer function of duplicates removal to _normalize() method.

The __init__ method has to look like this:

```
class Topology:
    def __init__(self, topology_dict):
        self.topology = self._normalize(topology_dict)
```

Task 22.1b

Change Topology class from task 22.1a or 22.1.

Add delete_link() method that removes specified connection. Method should also remove mirror connection if it exists (example below).

If there is no such connection, message “There is no such connection” is displayed.

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')}
```

Link removal:

```
In [9]: t.delete_link(('R3', 'Eth0/1'), ('R4', 'Eth0/0'))

In [10]: t.topology
Out[10]:
```

(continues on next page)

(continued from previous page)

```
{('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')}
```

Removal of mirror connection: dictionary has an entry ('R3', 'Eth0/2'): ('R5', 'Eth0/0'), but call of delete_link() with key and value in reverse order should remove 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, such message is displayed:

```
In [13]: t.delete_link(('R5', 'Eth0/0'), ('R3', 'Eth0/2'))
There is no such connection
```

Task 22.1c

Change Topology class from task 22.1b.

Add delete_node() method that removes all connections with specified device. If there is no such device, message "There is no such device" is displayed.

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'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Device removal:

```
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, such message is displayed:

```
In [5]: t.delete_node('SW1')
There is no such device
```

Task 22.1d

Change Topology class from task 22.1c

Add add_link() method that adds specified connection if it is not already in topology. If connection exists, display message “Such connection already exists”. If one of sides is in topology, display message “Connection with one of ports exists”.

Example of creating a topology and adding connections

```
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
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'),
```

(continues on next page)

(continued from previous page)

```
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [11]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/0'))
Such connection already exists

In [12]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/5'))
Connection with one of ports exists
```

Task 22.2

Create CiscoTelnet class that connects via Telnet to Cisco equipment.

When creating class instance, Telnet connection should be created as well as switching to enable mode. Class should use telnetlib module to connect via Telnet.

CiscoTelnet class, besides __init__(), should have at least two methods:

- _write_line() - takes as argument a string and sends to equipment a string converted to bytes and adds a line feed at the end. Method _write_line() should be used within class.
- send_show_command() - takes show command as argument and returns output received from device

Parameter of __init__() method:

- ip - IP address
- username - username
- password - password
- secret - enable password

Example of creating class instance:

```
In [2]: from task_22_2 import CiscoTelnet

In [3]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}
   ...:

In [4]: r1 = CiscoTelnet(**r1_params)
```

(continues on next page)

(continued from previous page)

```
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                ␣
↪190.16.200.1    YES NVRAM  up                    up      \r\nEthernet0/3        ␣
↪        192.168.130.1   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\nLoopback0                  10.1.1.1        YES NVRAM  up   ␣
↪                up      \r\nLoopback55                 5.5.5.5         YES␣
↪manual up                    up      \r\nR1#'
```

Note: Tip: Method _write_line() is needed to shorten line `self.telnet.write(line.encode("ascii") + b"\n")` to such line: `self._write_line(line)`. It should not do anything else.

Task 22.2a

Copy CiscoTelnet class from task 22.2 and change send_show_command() method by adding three parameters:

- parse - controls whether the usual command output or list of dictionaries received after processing with Textfsm will be returned. If parse=True, list of dictionaries should be returned and if parse=False, usual output should be returned. Default value is True.
- templates - path to template directory. Default value - “templates”
- index - name of file where mapping between commands and templates is stored. Default value - “index”

Example of class instance creation:

```
In [1]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [2]: from task_22_2a import CiscoTelnet

In [3]: r1 = CiscoTelnet(**r1_params)
```

Use of send_show_command() method:

```
In [4]: r1.send_show_command('sh ip int br', parse=False)
Out[4]: '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                ␣
↪190.16.200.1    YES NVRAM  up                    up      \r\nEthernet0/3        ␣
↪        192.168.130.1   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\nLoopback0                  10.1.1.1        YES NVRAM  up    ␣
↪                up      \r\nLoopback55                 5.5.5.5         YES␣
↪manual up                    up      \r\nR1#'

In [5]: r1.send_show_command('sh ip int br', parse=True)
Out[5]:
[{'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': '190.16.200.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/3',
  'address': '192.168.130.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/3.100',
  'address': '10.100.0.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/3.200',
  'address': '10.200.0.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/3.300',
  'address': '10.30.0.1',
```

(continues on next page)

(continued from previous page)

```
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Loopback0',
  'address': '10.1.1.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Loopback55',
  'address': '5.5.5.5',
  'status': 'up',
  'protocol': 'up'}]
```

Task 22.2b

Copy CiscoTelnet class from task 22.2a and add send_config_commands() method.

Method send_config_commands() should be able to send one configuration mode command or list of commands. Method should return output similar to send_config_set() method of netmiko (example of output below).

Example of class instance creation:

```
In [1]: from task_22_2b import CiscoTelnet

In [2]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [3]: r1 = CiscoTelnet(**r1_params)
```

Use of 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 CiscoTelnet class from task 22.2b and change send_config_commands() method by adding error check.

Method send_config_commands() should have an additional parameter *strict*:

- strict=True means that if error is detected, it is necessary to generate ValueError exception
- strict=False means that if error is detected, all you have to do is to display error message

Method should return output similar to send_config_set() method of netmiko (example of output below). Exception and error text in example below.

Example of class instance creation:

```
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', 'i']
In [5]: correct_commands = ['logging buffered 20010', 'ip http server']
In [6]: commands = commands_with_errors+correct_commands
```

Use of send_config_commands() method:

```
In [7]: print(r1.send_config_commands(commands, strict=False))
When executing command  "logging 0255.255.1" on device 192.168.100.1 error␣
↪occurred -> Invalid input detected at '^' marker.
When executing command "logging" on device 192.168.100.1 error occurred ->␣
↪Incomplete command.
When executing command "i" on device 192.168.100.1 error occurred -> Ambiguous␣
↪command:  "i"
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.
```

(continues on next page)

(continued from previous page)

```
R1(config)#i
% Ambiguous command:  "i"
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 command "logging 0255.255.1" on device 192.168.100.1␣
↪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, the expression `a + b` where a and b are numbers that is converted to such a call `a.__add__(b)`. That is, the 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 embedded 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 the 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):

```
.. code:: python
```

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 the 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:
            self._ssh.send('terminal length 0\n')
```

(continues on next page)

(continued from previous page)

```
        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 the 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
    ...:

In [17]: dir(CiscoSwitch)
```

(continues on next page)

(continued from previous page)

```
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:

```
In [1]: 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 display 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.

```
In [6]: 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)
Out[9]: 'IPAddress: 10.1.1.1'
```

(continues on next page)

(continued from previous page)

```
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 display and it should also return a string, but it would return a string by copying which you can get an instance of a class:

```
In [14]: 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__:

```
In [5]: 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 must 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. Exception and error message take from the analogous error of 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:

```
In [11]: 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 exception, 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 (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 iterated object becomes an iterator by calling `iter(name)` where *name* - name of iterable object. If __iter__ method is not present, Python iterates elements using __getitem__.

```
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 the iterator. If this method is not available, iter() function checks availability of __getitem__ method that can get elements by index. If __getitem__ method exists, the 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 returns 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 returned. In order for iterator to return elements again, it has to be re-created. Similar actions 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 the iterator and then __next__ method is called until Stopiteration exception occurs.

An example of my_for() function that works with any iterable object and imitates built-in function **for**:

```
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:

```
In [12]: 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 generates Stoeratipiton 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
```

(continues on next page)

(continued from previous page)

```
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 object

Very often it is sufficient for class to be an iterable object 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 object than an iterator.

In order for Network class to create iterable objects, the 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)`:

```
In [17]: 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')
```

(continues on next page)

(continued from previous page)

```
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']

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']
```

(continues on next page)

(continued from previous page)

```
<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
---> 14         return self.addresses[index]
     15

TypeError: list indices must be integers or slices, not str
```

You will find implementation of remaining methods of sequence protocol in tasks to this section:

- __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):
```

(continues on next page)

(continued from previous page)

```
        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()
        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()
        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
```

(continues on next page)

(continued from previous page)

```
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

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 23.1

In this task you need to create an IPaddress class.

When creating class instance, IP address and mask are passed as an argument and the correctness of address and mask should be checked:

Address is considered correct if it:

- consists of 4 numbers separated by a point
- each number in range 0 to 255

Mask is considered correct if it is between 8 and 32 inclusive

If mask or address didn’t pass verification, you should generate ValueError exception with appropriate text (output below).

Also, when creating a class, two instance variables should be created: *ip* and *mask* which contain address and mask, respectively.

Example of class instance creation:

```
In [1]: ip = IPAddress('10.1.1.1/24')

Атрибуты ip и mask
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
```

Address correctness check (traceback omitted)

Address correctness check (traceback omitted)

```
In [6]: ip1 = IPAddress('10.1.1.1/240')
------------------------------------------------------------------------
...
ValueError: Incorrect mask
```

Task 23.1a

Copy and change IPaddress class from task 23.1.

Add two string views for IPaddress class instances. What line views should look like is defined in the following output:

Instance creation

```
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

Add to CiscoTelnet class from task 22.2x support for work in context manager. When leaving context manager block, connection should be closed.

Example:

```
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#

In [17]: with CiscoTelnet(**r1_params) as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:     raise ValueError('Error occurred')
    ...:
sh clock
*19:17:38.828 UTC Sat Apr 6 2019
R1#
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-f3141be7c129> in <module>
      1 with CiscoTelnet(**r1_params) as r1:
      2     print(r1.send_show_command('sh clock'))
----> 3     raise ValueError('Error occurred')
      4

ValueError: Возникла ошибка
```

Task 23.3

Copy and change Topology class from task 22.1x.

Add method that allows you to perform addition of two instances of Topology class. As a result of addition, new instance of Topology class should be returned.

Creation of two topologies:

```
In [1]: t1 = Topology(topology_example)
```

(continues on next page)

(continued from previous page)

```
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')}

In [4]: t2 = Topology(topology_example2)

In [5]: t2.topology
Out[5]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Topology summation:

```
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')}
```

Check that original topologies have not 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')}
```

(continues on next page)

(continued from previous page)

```
In [10]: t2.topology
Out[10]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Task 23.3a

In this task, make sure that Topology class instances are iterable objects. The base of Topology class can be taken from any task 22.1x or task 23.3.

After class instance creation, instance should work as an iterable object. After each iteration, tuple that describes a single connection should return.

Example of class run:

```
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'))
```

Check class run.

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 variant of `ConnectSSH.send_show_command` explicitly specifies the name of parent class - this is the most understandable variant 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 of connection:

```
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

Warning: Starting from section "9. Functions" there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 24.1

Create CiscoSSH class that inherits BaseSSH class from base_connect_class.py.

Create __init__() method in CiscoSSH class in such a way that once connected by SSH, enable mode is activated.

To do this, __init__() method should first invoke __init__() method of ConnectSSH 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

Add CiscoSSH class from task 24.1.

Before connecting via SSH, you should check whether dictionary with parameters has: username, password, secret. If not, request them from user and then establish connection. If parameters exist, establish connection immediately.

```
In [1]: from task_24_1a import CiscoSSH

In [2]: device_params = {
   ...:         'device_type': 'cisco_ios',
   ...:         'ip': '192.168.100.1',
   ...: }

In [3]: r1 = CiscoSSH(**device_params)
Enter user name: cisco
Enter password:
Enter password for enable mode:

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 MyNetmiko class that inherits CiscoIosBase class from netmiko.

Rewrite __init__() method in MyNetmiko class in such a way that once connected via SSH, enable mode is activated.

To do this, __init__() method should first call __init__() method of CiscoIosBase class and then switch to enable mode.

Check that send_command() and send_config_set() methods are available in MyNetmiko class

```
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)

CiscoIosSSH class import:

```
from netmiko.cisco.cisco_ios import CiscoIosSSH


device_params = {
    "device_type": "cisco_ios",
    "ip": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}
```

Task 24.2a

Complete MyNetmikoclass from task 24.2.

Add _check_error_in_command() method that checks for such errors:

- Invalid input detected
- Incomplete command
- Ambiguous command

Method expects as an argument the command and output of command. If no error was found in output, method does not return anything. If error is found in output, method generates an ErrorInCommand exception with message about what error was detected, on which device and on which command.

Rewrite send_command netmiko() method by adding error check.

```
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      '
```

(continues on next page)

(continued from previous page)

```
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 command "sh ip br" on device 192.168.100.1 error␣
↪occurred "Invalid input detected at '^' marker."
```

ErrorInCommand exception:

```
class ErrorInCommand(Exception):
    """
    Exception is generated if error occurs while executing command on device.
    """
```

Task 24.2b

Copy MyNetmiko class from task 24.2a.

Complete send_config_set netmiko() method functionality and add error check using _check_error_in_command() method.

Method send_config_set() should send commands one at a time and check each for errors. If no errors are detected while executing commands, send_config_set() method returns output of 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 command "lo" on device 192.168.100.1 error␣
↪occurred "Incomplete command."
```

Задание 24.2c

Check that send_command() method of MyNetmiko class from task 24.2b accepts additional arguments (as in netmiko), except command.

If error occurs, redo method so that it accepts any arguments that support netmiko.

```
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      '

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 MyNetmiko class from task 24.2c or 24.2b.

Add ignore_errors parameter to send_config_set() method. If true value is passed, no error check should be performed and method should run in the same way as send_config_set() method in netmiko. If value is false, errors should be checked.

Errors should be ignored by default.

```
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 command "lo" on device 192.168.100.1 error␣
↪occurred "Incomplete command."
```

8

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 the DBMS it is possible to make information slices according to different parameters.

Database (DB) - the 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 the 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 the 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 the 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 the library (and therefore within program that uses it).

SQLite CLI

SQLite package also includes a command line utility for working with SQLite. The 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 the 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 the 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 of implementation:

```
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 deals with the SQL syntax.

If you are familiar with basic SQL syntax you can skip this section and move to section *Sqlite3 module*

CREATE

CREATE operator allows you to create tables.

First connect to the 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 the 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 the 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 the query. With the help of this operator it is possible to specify certain conditions under which the data are selected. If condition is met the corresponding value is returned from the 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 the 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 WHERE operator you can display only switches of 3850 model:

```
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 the 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 example, 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 operator 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 command:

```
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 the 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 the 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 the 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 the 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 derive entries from *switch* table by sorting them by the 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 the 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 the 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 transfer 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 (?, ?, ?, ?)"
```

The 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 transfer 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 |
```

(continues on next page)

(continued from previous page)

```
| 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
Time: 0.039s
sw_inventory.db>
```

Method executemany

Method executemany() allows one SQL command to be executed for parameter sequence (or for iterator).

Using executemany() method you can add a similar data list to *switch* table by a single command.

For example, you should add data from the *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          |
```

(continues on next page)

(continued from previous page)

```
+----------------+----------+------------+-------------------+
| 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 |
+----------------+----------+------------+-------------------+
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
```

(continues on next page)

(continued from previous page)

```
   ...:
('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'),
```

(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')]
```

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'),
```

(continues on next page)

(continued from previous page)

```
 ('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 the 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

The 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, the 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):

```
# -*- coding: utf-8 -*-
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_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')
```

Processing of exceptions

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')"

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 intercept *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):

```
# -*- coding: utf-8 -*-
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')
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 (create_sw_inventory_ver2_functions):

```
# -*- coding: utf-8 -*-
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'),
```

(continues on next page)

(continued from previous page)

```
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


def create_connection(db_name):
    '''
    Функция создает соединение с БД db_name
    и возвращает его
    '''
    connection = sqlite3.connect(db_name)
    return connection


def write_data_to_db(connection, query, data):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить
     * data - данные, которые надо передать в виде списка кортежей

    Функция пытается записать все данные из списка data.
    Если данные удалось записать успешно, изменения сохраняются в БД
    и функция возвращает True.
    Если в процессе записи возникла ошибка, транзакция откатывается
    и функция возвращает False.
    '''
    try:
        with connection:
            connection.executemany(query, data)
    except sqlite3.IntegrityError as e:
        print('Error occured: ', e)
        return False
    else:
        print('Запись данных прошла успешно')
        return True


def get_all_from_db(connection, query):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить

    Функция возвращает данные полученные из БД.
```

(continues on next page)

(continued from previous page)

```
    '''
    result = [row for row in connection.execute(query)]
    return result


if __name__ == '__main__':
    con = create_connection('sw_inventory3.db')

    print('Создание таблицы...')
    schema = '''create table switch
                (mac text primary key, hostname text, model text, location text)''
↪'
    con.execute(schema)

    query_insert = 'INSERT into switch values (?, ?, ?, ?)'
    query_get_all = 'SELECT * from switch'

    print('Запись данных в БД:')
    pprint(data)
    write_data_to_db(con, query_insert, data)
    print('\nПроверка содержимого БД')
    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:

```
# -*- coding: utf-8 -*-
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC-адрес sw7 совпадает с MAC-адресом коммутатора sw3 в списке data
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')]

con = dbf.create_connection('sw_inventory3.db')

query_insert = "INSERT into switch values (?, ?, ?, ?)"
query_get_all = "SELECT * from switch"

print("\nПроверка текущего содержимого БД")
pprint(dbf.get_all_from_db(con, query_get_all))

print('-' * 60)
print("Попытка записать данные с повторяющимся MAC-адресом:")
pprint(data2)
dbf.write_data_to_db(con, query_insert, data2)
print("\nПроверка содержимого БД")
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

In *data2* list the sw7 switch has the same MAC address as the 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')]
------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
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 the 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 the data in turn and if there is an error, only changes for specific data are rolled back:

```
# -*- coding: utf-8 -*-
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC-адрес sw7 совпадает с MAC-адресом коммутатора sw3 в списке data
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):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить
     * data - данные, которые надо передать в виде списка кортежей

```

(continues on next page)

(continued from previous page)

```
    Функция пытается записать поочереди кортежи из списка data.
    Если кортеж удалось записать успешно, изменения сохраняются в БД.
    Если в процессе записи кортежа возникла ошибка, транзакция откатывается.

    Флаг verbose контролирует то, будут ли выведены сообщения об удачной
    или неудачной записи кортежа.
    '''
    for row in data:
        try:
            with connection:
                connection.execute(query, row)
        except sqlite3.IntegrityError as e:
            if verbose:
                print("При записи данных '{}' возникла ошибка".format(
                    ', '.join(row), e))
        else:
            if verbose:
                print("Запись данных '{}' прошла успешно".format(
                    ', '.join(row)))


con = dbf.create_connection('sw_inventory3.db')

query_insert = 'INSERT into switch values (?, ?, ?, ?)'
query_get_all = 'SELECT * from switch'

print('\nПроверка текущего содержимого БД')
pprint(dbf.get_all_from_db(con, query_get_all))

print('-' * 60)
print('Попытка записать данные с повторяющимся MAC-адресом:')
pprint(data2)
write_rows_to_db(con, query_insert, data2, verbose=True)
print('\nПроверка содержимого БД')
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
```

(continues on next page)

(continued from previous page)

```
[('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
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 variant you can only see all devices connected to the 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 the 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.

The 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 the database and create a table (create_sqlite_ver1.py file):

```
1  import sqlite3
2
3  conn = sqlite3.connect('dhcp_snooping.db')
4
5  print('Creating schema...')
6  with open('dhcp_snooping_schema.sql', 'r') as f:
7      schema = f.read()
8      conn.executescript(schema)
9  print("Done")
10
11 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 the table has been created with sqlite3 utility which allows you to execute queries directly in command line.

The 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):

```
1  import sqlite3
2  import re
3
4  regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')
5
6  result = []
7
8  with open('dhcp_snooping.txt') as data:
9      for line in data:
10         match = regex.search(line)
11         if match:
12             result.append(match.groups())
```

(continues on next page)

(continued from previous page)

```
13
14 conn = sqlite3.connect('dhcp_snooping.db')
15
16 print('Creating schema...')
17 with open('dhcp_snooping_schema.sql', 'r') as f:
18     schema = f.read()
19     conn.executescript(schema)
20 print('Done')
21
22 print('Inserting DHCP Snooping data')
23
24 for row in result:
25     try:
26         with conn:
27             query = '''insert into dhcp (mac, ip, vlan, interface)
28                        values (?, ?, ?, ?)'''
29             conn.execute(query, row)
30     except sqlite3.IntegrityError as e:
31         print('Error occured: ', e)
32
33 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 the 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 Regular expressions
  - 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 the elements in the received list of tuples
- This script uses another version of database entry
  - *query* string describes the query. But instead of values, question marks are given. This query type allows dynamicly substite field values.
  - then execute() method is passed the query string and the *row* tuple where the 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
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:

```
1  import os
2  import sqlite3
3  import re
4
5  data_filename = 'dhcp_snooping.txt'
6  db_filename = 'dhcp_snooping.db'
7  schema_filename = 'dhcp_snooping_schema.sql'
8
9  regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')
10
11 result = []
```

(continues on next page)

(continued from previous page)

```
12
13 with open('dhcp_snooping.txt') as data:
14     for line in data:
15         match = regex.search(line)
16         if match:
17             result.append(match.groups())
18
19 db_exists = os.path.exists(db_filename)
20
21 conn = sqlite3.connect(db_filename)
22
23 if not db_exists:
24     print('Creating schema...')
25     with open(schema_filename, 'r') as f:
26         schema = f.read()
27     conn.executescript(schema)
28     print('Done')
29 else:
30     print('Database exists, assume dhcp table does, too.')
31
32 print('Inserting DHCP Snooping data')
33
34 for row in result:
35     try:
36         with conn:
37             query = '''insert into dhcp (mac, ip, vlan, interface)
38                        values (?, ?, ?, ?)'''
39             conn.execute(query, row)
40     except sqlite3.IntegrityError as e:
41         print('Error occured: ', e)
42
43 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
```

(continues on next page)

(continued from previous page)

```
$ 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
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 deals with sending queries to database and displaying results. It should:

- expect parameters from user:
  - parameter name
  - parameter value
- provide normal output on request

File get_data_ver1.py:

```
# -*- coding: utf-8 -*-
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

key, value = sys.argv[1:]
```

(continues on next page)

(continued from previous page)

```
keys = ['mac', 'ip', 'vlan', 'interface']
keys.remove(key)

conn = sqlite3.connect(db_filename)

#Позволяет далее обращаться к данным в колонках, по имени колонки
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:
    for k in keys:
        print('{:12}: {}'.format(k, row[k]))
    print('-' * 40)
```

Comments to the script:

- key, value are read from the arguments that passed to script
  - selected key is removed from the *keys* list. Thus, only parameters that you want to display are left in the list
- connecting to the DB
  - `conn.row_factory = sqlite3.Row` - allows further access data in column based on column names
- Select rows from database where the 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 the row formatting and the value by SQL tool.
  - Pay attention to `(value,)` - the tuple with one element is passed
- The resulting information is displayed to standard output stream:
  - iterate over the results obtained and display only those fields that are in the *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
----------------------------------------
mac         : 00:07:BC:3F:A6:50
ip          : 10.1.10.6
interface   : FastEthernet0/3
----------------------------------------
```

The second version of the script to obtain data with minor improvements:

- Instead of rows formatting, a dictionary that describes the queries corresponding to each key is used.
- Checking the key that was selected
- Method keys() is used to obtain all columns that match the query

File get_data_ver2.py:

```
# -*- coding: utf-8 -*-
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 = ?'
```

(continues on next page)

(continued from previous page)

```
}

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:
        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 the 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.

Additional material

Documentation:

- SQLite Tutorial - SQLite detailed description
- Module documentation sqlite3
- sqlite3 на сайте PyMOTW

Articles:

- A thorough guide to SQLite database operations in Python

Tasks

Warning: Starting from section “9. Functions” there are automatic tests for checking tasks. They help to check whether everything fits the task and also give feedback on what does not fit the task. As a rule, after first period of adaptation to tests, it becomes easier to do tasks with tests.

How to work with tests and basics of pytest.

Task 25.1

There are no tests for tasks of section 25!

Two scripts need to be created:

1. create_db.py
2. add_data.py

Code in scripts should be split into functions. You should decide what functions to create and how to divide code. Part of code could be global.

1. create_db.py - this script should have functionality to create database:
   - DB file existence verification should be performed
   - if there is no file, according to description of database scheme in the dhcp_snooping_schema.sql file, database should be created
   - database file name - dhcp_snooping.db

Database should have two tables (scheme is described in dhcp_snooping_schema.sql):

- switches - it contains switches data
- dhcp - it contains information derived from the output of *sh ip dhcp snooping binding* command

Example of script execution when there is no dhcp_snooping.db file:

```
$ python create_db.py
Building a database ...
```

After file creation:

```
$ python create_db.py
Database exists
```

2. add_data.py - this script adds data to database. Script should add data from output of *sh ip dhcp snooping binding* and information about switches

Accordingly, add_data.py file should have two parts:

- switch information is added to *switches* table
  - switches data, located in switches.yml file
- information based on output of *sh ip dhcp snooping binding* is added to *dhcp* table
  - output from three switches: files sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt
  - since *dhcp* table has changed and now has a *switch* field, it also needs to be filled in. Switch name is defined by the name of data file

Example of script run when a database has not yet been created:

```
$ python add_data.py
Database does not exist. Before adding data, you should create it
```

Example of first time script run after creating a database:

```
$ python add_data.py
Add data to switches table...
Adding data to dhcp table...
```

Example of script execution after data has been added to table (order in which data is added may be arbitrary, but messages should be displayed in the same way as output below):

```
$ python add_data.py
Add data to switches table...
When adding data: ('sw1', 'London, 21 New Globe Walk') Error occurred: UNIQUE␣
↪constraint failed: switches.hostname
When adding data: ('sw2', 'London, 21 New Globe Walk') Error occurred: UNIQUE␣
↪constraint failed: switches.hostname
When adding data: ('sw3', 'London, 21 New Globe Walk') Error occurred: UNIQUE␣
↪constraint failed: switches.hostname
Adding data to dhcp table..
When adding data: ('00:09:BB:3D:D6:58', '10.1.10.2', '10', 'FastEthernet0/1', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:04:A3:3E:5B:69', '10.1.5.2', '5', 'FastEthernet0/10', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:05:B3:7E:9B:60', '10.1.5.4', '5', 'FastEthernet0/9', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:07:BC:3F:A6:50', '10.1.10.6', '10', 'FastEthernet0/3', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7',
↪ 'sw1') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:E9:BC:3F:A6:50', '100.1.1.6', '3', 'FastEthernet0/20', 'sw3
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
```

(continues on next page)

(continued from previous page)

```
When adding data: ('00:E9:22:11:A6:50', '100.1.1.7', '3', 'FastEthernet0/21', 'sw3
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:A9:BB:3D:D6:58', '10.1.10.20', '10', 'FastEthernet0/7',
↪'sw2') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:B4:A3:3E:5B:69', '10.1.5.20', '5', 'FastEthernet0/5', 'sw2
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:C5:B3:7E:9B:60', '10.1.5.40', '5', 'FastEthernet0/9', 'sw2
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:A9:BC:3F:A6:50', '10.1.10.60', '20', 'FastEthernet0/2',
↪'sw2') Error occurred: UNIQUE constraint failed: dhcp.mac
```

Both scripts are called without argument.

Task 25.2

There are no tests for tasks of section 25!

In this task you need to create get_data.py script.

Code in script should be split into functions. You should decide what functions to create and how to divide code. Part of code could be global.

Script can be passed arguments and depending on arguments, different information should be displayed. If script is called:

- without arguments, output the entire contents of *dhcp* table
- with two arguments, output information from *dhcp* table which corresponds to field and value
- with any other number of arguments, output message that script supports only two or zero arguments

Database file can be copied from task 25.1.

Examples of output for different amount and value of arguments:

```
$ python get_data.py
dhcp table has such 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
```

(continues on next page)

(continued from previous page)

```
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
-----------------  ---------------  --  ----------------  ---

$ python get_data.py vlan 10

Information on devices with such 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 on devices with such 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 tasks of section 25!

In past tasks information was added to empty database. In this task, the situation when database already has information is considered.

Copy add_data.py script from task 25.1 and try to run it again on existing database. The result should be:

```
$ python add_data.py
Add data to switches table...
When adding data: ('sw1', 'London, 21 New Globe Walk') Error occurred: UNIQUE␣
↪constraint failed: switches.hostname
```

(continues on next page)

(continued from previous page)

```
When adding data: ('sw2', 'London, 21 New Globe Walk') Error occurred: UNIQUE␣
↪constraint failed: switches.hostname
When adding data: ('sw3', 'London, 21 New Globe Walk') Error occurred: UNIQUE␣
↪constraint failed: switches.hostname
Adding data to dhcp table..
When adding data: ('00:09:BB:3D:D6:58', '10.1.10.2', '10', 'FastEthernet0/1', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:04:A3:3E:5B:69', '10.1.5.2', '5', 'FastEthernet0/10', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:05:B3:7E:9B:60', '10.1.5.4', '5', 'FastEthernet0/9', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:07:BC:3F:A6:50', '10.1.10.6', '10', 'FastEthernet0/3', 'sw1
↪') Error occurred: UNIQUE constraint failed: dhcp.mac
When adding data: ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7',
↪ 'sw1') Error occurred: UNIQUE constraint failed: dhcp.mac
... (output ommited)
```

When creating a database schema, it was explicitly stated that MAC address field should be unique. Therefore, when an entry with the same MAC address is added an exception (error) occurs. In task 25.1, exception is processed and message is displayed on standard output stream.

In this task it is considered that information is periodically read from switches and written into files. After that, information from files should be transmitted to database. However, there may be changes in new data: MAC is missing, MAC has moved to another port/vlan, new MAC has appeared, etc.

In this task in *dhcp* table it is necessary to create a new *active* field that will indicate whether the entry is relevant. New database schema is in dhcp_snooping_schema.sql file.

Field *active* should take such values:

- 0 - means False. Used to mark the entry as inactive
- 1 - True. Used to indicate that the entry is active

Each time the information from DHCP snooping output files is re-added, all existing entries (for this switch) should be marked as inactive (active = 0). You can then update information and mark new entries as active (active = 1).

Thus, old entries will remain in database for MAC addresses that are not currently active and updated information for active addresses will appear.

For example, there are such entries in *dhcp* table:

```
mac                ip          vlan        interface          switch      active
-----------------  ----------  ----------  -----------------  ----------  ---------
↪-
00:09:BB:3D:D6:58  10.1.10.2   10          FastEthernet0/1    sw1         1
```

(continues on next page)

(continued from previous page)

```
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 have to add this information from 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 data, table should look like:

```
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
```

New information should overwrite previous information:

- MAC 00:04:A3:3E:5B:69 moved to another port and got another interface and got a different address
- MAC 00:07:BC:3F:A6:50 moved to another port

If there is no MAC address in new file, it should be left in database with active = 0: MAC-адреса 00:09:BC:3F:A6:50 50 is not in new information (computer is turned off).

Change add_data.py script to meet new conditions and fill in *active* field.

Code in script should be split into functions. You should decide what functions to create and how to divide code. Part of code could be global.

> To check the correctness of SQL query you can execute it in command line using sqlite3 utility.

To check task and work of new field, first add information from sw*_dhcp_snooping.txt files to database and then add information from new_data/sw*_dhcp_snooping.txt files

Data should look like this (the order of lines can be any)

```
----------------  ---------------  --  ----------------  ---  -
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
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 tasks of section 25!

Copy get_data file from 25.2 task. Add *active* column to script that we added to 25.3 task.

Now, when information is requested, active entries should be displayed first and then inactive entries. If there are no inactive entries, do not display title “Inactive entries”.

Examples of resulting script execution:

```
$ python get_data.py
dhcp table has such 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
```

(continues on next page)

(continued from previous page)

```
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:

-----------------  --------------  -  ---------------  ---  -
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 on devices with such 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 on devices with such 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 tasks of section 25!

After completing tasks 25.1 - 25.5 information about inactive entries is left in database. And if some MAC address has not appeared in new entries, the entry with it may remain in database forever.

Although it may be useful to see where MAC address was last, it is not very useful to keep this information permanently.

For example, if an entry in database is more than a month old, it can be deleted.

In order to make such a condition you need to enter a new field in which to write the most recent time of adding an entry.

New field is called *last_active* and should contain a string in format: YYYY-MM-DD HH:MM:SS.

This task requires:

- Amend *dhcp* table accordingly and add a new field.
  - table can be changed from cli sqlite but dhcp_snooping_schema.sql file also needs to be changed
- change add_data.py script to add time to each entry

You can get string with time and date in specified format using the datetime() function in SQL query. Syntax of the use:

```
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 value that is written to database you should specify datetime('now').

After this command such entry will appear in database:

```
mac                ip              vlan   interface        switch   active  ␣
↪last_active
-----------------  --------------  -----  ---------------  -------  -------  ----
↪---------------
00:09:BC:3F:A6:50  192.168.100.100  1      FastEthernet0/7  sw1      0       ␣
↪2019-03-08 11:26:56
```

Task 25.5a

There are no tests for tasks of section 25!

After completing task 25.5, *dhcp* table has a new field *last_active*.

Update add_data.py script to remove all entries that were active more than 7 days ago.

To get such entries you can simply manually update *last_active* field in some entries and set time 7 or more days.

Task file describes an example of working with datetime module objects. Shows how to get a date 7 days ago. With this date you will have to compare *last_active* time.

Note that you can compare lines with date that are written in database.

```
from datetime import timedelta, datetime

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 tasks of section 25!

There is a parse_dhcp_snooping.py file in this task. File parse_dhcp_snooping.py should not be changed.

File creates several functions and describes command line arguments that file accepts.

There is support for all actions that in previous tasks were executed in create_db.py, add_data.py and get_data.py.

File parse_dhcp_snooping.py has a line: import parse_dhcp_snooping_functions as pds

And the goal of this task is to create all necessary functions in parse_dhcp_snooping_functions.py file based on information in parse_dhcp_snooping.py.

From parse_dhcp_snooping.py file it is necessary to define:

- which functions should be in parse_dhcp_snooping_functions file.
- which parameters to create in these functions

It is necessary to create appropriate functions and transfer to them functionality described in previous tasks.

All necessary information is present in functions create(), add(), get() in parse_dhcp_snooping.py file.

To make it easier to start, try to create necessary functions in parse_dhcp_snooping_functions.py and simply display function arguments using print().

Then you can create functions that request information from database (database can be copied from previous tasks).

You can create any auxiliary functions in parse_dhcp_snooping_functions.py file, not only those that are called from parse_dhcp_snooping.py.

Check all operations:

- creation of the DB
- addition of information on switches
- add information based on the output of *sh ip dhcp snooping binding* from files
- fetching information from DB (by parameter and all information)

To make it easier to understand what a script call will look like, the following are a few examples. Examples show a variant where database has *active* and *last_active* fields, but you can also use variant 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 enties search
  -v VALUE              parameter value
  -a                    show all DB 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 be added

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  database name
  -s            if flag set, add switches data. Otherwise add DHCP records


$ python parse_dhcp_snooping.py add -h
```

(continues on next page)

(continued from previous page)

```
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                  filename [filename ...]

positional arguments:
  filename      file(s) to be added

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  database name
  -s            if flag set, add switches data. Otherwise add DHCP records


$ 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 enties search
  -v VALUE              parameter value
  -a                    show all DB content


$ python parse_dhcp_snooping.py create_db
Buidlding databse dhcp_snooping.db with schema dhcp_snooping_schema.sql
Buidlding databse...


$ python parse_dhcp_snooping.py add sw[1-3]_dhcp_snooping.txt
Reading inforamtion from files
sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt

Adding DHCP records data to dhcp_snooping.db


$ python parse_dhcp_snooping.py add -s switches.yml
Adding switches data

$ python parse_dhcp_snooping.py get
dhcp table has such entries:
```

(continues on next page)

(continued from previous page)

```
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 DB: dhcp_snooping.db
Information on devices with such parameters: vlan 10

Active entries:

-----------------  ----------  --  ---------------  ---  -  -------------------
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
```

(continues on next page)

(continued from previous page)

```
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')
```

9

VIII. Additional information

This section collects information that is not included in main sections of the book, but which can still be useful.

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 display of numbers.

For example, you can specify how many digits to display 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()
```

(continues on next page)

(continued from previous page)

```
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 feature of 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.

> 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
    ...:

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 form of display, 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)
```

(continues on next page)

(continued from previous page)

```
    ...:

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() fucntion

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 brackets, 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
```

(continues on next page)

(continued from previous page)

```
    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]
       ^
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']
```

(continues on next page)

(continued from previous page)

```
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

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 line feed 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
```

10

Continuing education

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 <https://natenka.github.io/pyneng-resources/>'__ where other materials can be found.

Scripting for workflow automation

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

Python without binding to network equipment

Books

Basic level:

- [Think Python](https://greenteapress.com/wp/think-python-2e/) - good book on Python basics. There are tasks in the book.
- [Python Crash Course: A Hands-On, Project-Based Introduction to Programming](https://www.amazon.com/Python-Crash-Course-Hands-Project-Based-ebook/dp/B018UXJ9RI/) - 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](https://automatetheboringstuff.com/). [In Russian](https://www.ozon.ru/context/detail/id/137673590/) - 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](https://www.amazon.com/Python-Tricks-Buffet-Awesome-Features-ebook/dp/B0785Q7GSY) - excellent for 2-3 books on Python. Book describes various aspects of Python and how to use it correctly. The book is fairly new (late 2017) and reviews Python 3.
- [Effective Python: 59 Specific Ways to Write Better Python (Effective Software Development Series)](https://www.amazon.com/Effective-Python-Specific-Software-Development-ebook-dp-B00TKGY0GU/dp/B00TKGY0GU/) - book of useful advice on how best to write code. At the end of 2019 [the second edition of book is planned] (https://www.amazon.com/Effective-Python-Specific-Software-Development/dp/0134853989/).
- [Dive Into Python 3](http://diveintopython3.problemsolving.io/) - briefly considered fundamentals of Python and then more advanced topics: closure, generators, tests and so on. Book written in 2009 but considered by Python 3 and 99% of topics remained unchanged.
- [Problem Solving with Algorithms and Data Structures using Python](https://runestone.academy/runestone/static/pythonds/index.html) - excellent book on data structures and

algorithms. Many examples and homework. [In Russian] (http://aliev.me/runestone/)

- [Fluent Python](https://www.amazon.com/gp/product/1491946008/) - 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](https://www.amazon.com/gp/product/1449340377/) - great recipe book. A huge number of scenarios are considered with solutions and explanations.

Cources

- 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

Resources with tasks

- 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
- Tiny-Python-3.6-Notebook - excellent Python 3 cheat sheet