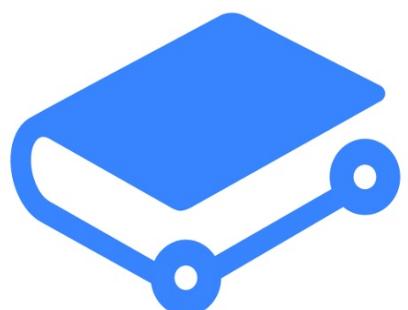


Python 3

для сетевых инженеров

Наташа Самойленко

Опубликовано
с помощью **GitBook**



Содержание

Введение	1.1
О книге	1.1.1
Изменения в книге	1.1.2
Ресурсы для обучения по книге	1.1.3
Как учиться по этой книге	1.1.4
Пример плана обучения	1.1.5
FAQ	1.1.6
Благодарности	1.1.7
I. Основы Python	1.2
1. Подготовка к работе	1.2.1
ОС и редактор	1.2.1.1
Система управления пакетами pip	1.2.1.2
Виртуальные окружения	1.2.1.3
Интерпретатор Python	1.2.1.4
Дополнительные материалы	1.2.1.5
Задания	1.2.1.6
2. Использование Git и GitHub	1.2.2
Основы Git	1.2.2.1
Отображение статуса репозитория в приглашении	1.2.2.1.1
Работа с Git	1.2.2.1.2
Дополнительные возможности	1.2.2.1.3
Аутентификация на GitHub	1.2.2.2
Работа со своим репозиторием	1.2.2.3
Работа с репозиторием заданий и примеров	1.2.2.4
Дополнительные материалы	1.2.2.5
Задания	1.2.2.6
3. Начало работы с Python	1.2.3
Синтаксис Python	1.2.3.1
Интерпретатор Python. iPython	1.2.3.2
Магия iPython	1.2.3.2.1

Переменные	1.2.3.3
Задания	1.2.3.4
4. Типы данных в Python	1.2.4
Числа	1.2.4.1
Строки (Strings)	1.2.4.2
Полезные методы для работы со строками	1.2.4.2.1
Форматирование строк	1.2.4.2.2
Объединение литералов строк	1.2.4.2.3
Список (List)	1.2.4.3
Полезные методы для работы со списками	1.2.4.3.1
Варианты создания списка	1.2.4.3.2
Словарь (Dictionary)	1.2.4.4
Полезные методы для работы со словарями	1.2.4.4.1
Варианты создания словаря	1.2.4.4.2
Кортеж (Tuple)	1.2.4.5
Множество (Set)	1.2.4.6
Полезные методы для работы с множествами	1.2.4.6.1
Операции с множествами	1.2.4.6.2
Варианты создания множества	1.2.4.6.3
Преобразование типов	1.2.4.7
Проверка типов	1.2.4.8
Дополнительные материалы	1.2.4.9
Задания	1.2.4.10
5. Создание базовых скриптов	1.2.5
Передача аргументов скрипту	1.2.5.1
Ввод информации пользователем	1.2.5.2
Задания	1.2.5.3
6. Контроль хода программы	1.2.6
if/elif/else	1.2.6.1
for	1.2.6.2
Вложенные for	1.2.6.2.1
Совмещение for и if	1.2.6.2.2
while	1.2.6.3
break, continue, pass	1.2.6.4

for/else, while/else	1.2.6.5
Работа с исключениями try/except/else/finally	1.2.6.6
Дополнительные материалы	1.2.6.7
Задания	1.2.6.8
7. Работа с файлами	1.2.7
Открытие файлов	1.2.7.1
Чтение файлов	1.2.7.2
Запись файлов	1.2.7.3
Закрытие файлов	1.2.7.4
Конструкция with	1.2.7.5
Дополнительные материалы	1.2.7.6
Задания	1.2.7.7
8. Примеры использования основ	1.2.8
Распаковка переменных	1.2.8.1
List, dict, set comprehensions	1.2.8.2
Работа со словарями	1.2.8.3
Дополнительные материалы	1.2.8.4
II. Повторное использование кода	1.3
9. Функции	1.3.1
Создание функций	1.3.1.1
Пространства имен. Области видимости	1.3.1.2
Параметры и аргументы функций	1.3.1.3
Типы параметров	1.3.1.3.1
Типы аргументов	1.3.1.3.2
Аргументы переменной длины	1.3.1.3.3
Распаковка аргументов	1.3.1.3.4
Пример использования	1.3.1.3.5
Дополнительные материалы	1.3.1.4
Задания	1.3.1.5
10. Полезные функции	1.3.2
Функция print	1.3.2.1
Функция range	1.3.2.2
Функция sorted	1.3.2.3

Функция enumerate	1.3.2.4
Функция zip	1.3.2.5
Функции any и all	1.3.2.6
11. Модули	1.3.3
Импорт модуля	1.3.3.1
Создание своих модулей	1.3.3.2
if name == "main"	1.3.3.3
Задания	1.3.3.4
12. Полезные модули	1.3.4
Модуль subprocess	1.3.4.1
Модуль os	1.3.4.2
Модуль ipaddress	1.3.4.3
Модуль argparse	1.3.4.4
Модуль tabulate	1.3.4.5
Модуль pprint	1.3.4.6
Дополнительные материалы	1.3.4.7
Задания	1.3.4.8
13. Итераторы, итерируемые объекты и генераторы	1.3.5
Итерируемый объект	1.3.5.1
Итератор	1.3.5.2
Generator	1.3.5.3
Дополнительные материалы	1.3.5.4
III. Регулярные выражения	1.4
14. Синтаксис регулярных выражений	1.4.1
Наборы символов	1.4.1.1
Символы повторения	1.4.1.2
Специальные символы	1.4.1.3
Жадность символов повторения	1.4.1.4
Группировка выражений	1.4.1.5
Пример использования именованных групп	1.4.1.5.1
Группа без захвата	1.4.1.6
Повторение захваченного результата	1.4.1.7
15. Модуль re	1.4.2
Объект Match	1.4.2.1

re.search	1.4.2.2
re.match	1.4.2.3
re.finditer	1.4.2.4
re.findall	1.4.2.5
re.compile	1.4.2.6
Флаги	1.4.2.7
re.split	1.4.2.8
re.sub	1.4.2.9
Дополнительные материалы	1.4.2.10
Задания	1.4.2.11
IV. Запись и передача данных	1.5
16. Unicode	1.5.1
Стандарт Unicode	1.5.1.1
Unicode в Python 3	1.5.1.2
Конвертация между байтами и строками	1.5.1.3
Примеры конвертации	1.5.1.4
Ошибки при конвертации	1.5.1.5
Дополнительные материалы	1.5.1.6
17. Работа с файлами в формате CSV, JSON, YAML	1.5.2
CSV	1.5.2.1
JSON	1.5.2.2
YAML	1.5.2.3
Дополнительные материалы	1.5.2.4
Задания	1.5.2.5
18. Работа с базами данных	1.5.3
SQL	1.5.3.1
SQLite	1.5.3.2
Основы SQL (в sqlite3 CLI)	1.5.3.3
CREATE	1.5.3.3.1
DROP	1.5.3.3.2
INSERT	1.5.3.3.3
SELECT	1.5.3.3.4
WHERE	1.5.3.3.5

ALTER	1.5.3.3.6
UPDATE	1.5.3.3.7
REPLACE	1.5.3.3.8
DELETE	1.5.3.3.9
ORDER BY	1.5.3.3.10
AND, OR, NOT, IN	1.5.3.3.11
Модуль sqlite3	1.5.3.4
Выполнение команд SQL	1.5.3.4.1
Получение результатов запроса	1.5.3.4.2
Cursor как итератор	1.5.3.4.3
Использование модуля sqlite3 без явного создания курсора	1.5.3.4.4
Обработка исключений	1.5.3.4.5
Connection как менеджер контекста	1.5.3.4.6
Пример использования SQLite	1.5.3.4.7
Дополнительные материалы	1.5.3.5
Задания	1.5.3.6
V. Работа с сетевым оборудованием	1.6
19. Подключение к оборудованию	1.6.1
Ввод пароля	1.6.1.1
Pexpect	1.6.1.2
Пример использования pexpect	1.6.1.2.1
Telnetlib	1.6.1.3
Пример использования telnetlib	1.6.1.3.1
Paramiko	1.6.1.4
Netmiko	1.6.1.5
Возможности netmiko	1.6.1.5.1
Дополнительные материалы	1.6.1.6
Задания	1.6.1.7
20. Одновременное подключение к нескольким устройствам	1.6.2
Измерение времени выполнения скрипта	1.6.2.1
Процессы и потоки в CPython	1.6.2.2
Модуль concurrent.futures	1.6.2.3
Метод map	1.6.2.3.1
Метод submit	1.6.2.3.2

Дополнительные материалы	1.6.2.4
Задания	1.6.2.5
21. Шаблоны конфигураций с Jinja2	1.6.3
Пример использования Jinja2	1.6.3.1
Программный интерфейс Jinja2	1.6.3.2
Синтаксис шаблонов Jinja2	1.6.3.3
Контроль символов whitespace	1.6.3.3.1
Переменные	1.6.3.3.2
for	1.6.3.3.3
if/elif/else	1.6.3.3.4
Фильтры	1.6.3.3.5
Тесты	1.6.3.3.6
Присваивание (set)	1.6.3.3.7
Include	1.6.3.3.8
Наследование шаблонов	1.6.3.4
Дополнительные материалы	1.6.3.5
Задания	1.6.3.6
22. Обработка вывода команд TextFSM	1.6.4
Синтаксис шаблонов TextFSM	1.6.4.1
Примеры использования TextFSM	1.6.4.2
CLI Table	1.6.4.3
Дополнительные материалы	1.6.4.4
Задания	1.6.4.5
VI. Ansible	1.7
23. Основы Ansible	1.7.1
Инвентарный файл	1.7.1.1
Ad-Hoc команды	1.7.1.2
Конфигурационный файл	1.7.1.3
Модули	1.7.1.4
Основы playbook	1.7.1.5
Переменные	1.7.1.5.1
Результат выполнения модуля	1.7.1.5.2
24. Сетевые модули	1.7.2

ios_command	1.7.2.1
ios_facts	1.7.2.2
ios_config	1.7.2.3
lines (commands)	1.7.2.3.1
parents	1.7.2.3.2
Отображение обновлений	1.7.2.3.3
save_when	1.7.2.3.4
backup	1.7.2.3.5
defaults	1.7.2.3.6
after	1.7.2.3.7
before	1.7.2.3.8
match	1.7.2.3.9
replace	1.7.2.3.10
src	1.7.2.3.11
ntc_ansible	1.7.2.4
Подробнее об Ansible	1.7.2.5
Дополнительные материалы	1.7.2.6
Задания	1.7.2.7
Дополнительная информация	1.8
Соглашение об именах	1.8.1
Подчеркивание в именах	1.8.1.1
Полезные функции	1.8.2
Функция lambda	1.8.2.1
Функция map	1.8.2.2
Функция filter	1.8.2.3
Основы threading и multiprocessing	1.8.3
Модуль threading	1.8.3.1
Модуль multiprocessing	1.8.3.2
Дополнительные материалы	1.8.3.3
Отличия Python 2.7 и Python 3	1.8.4
Продолжение обучения	1.9
Отзывы	1.10

Введение

Обсуждение

Для обсуждения книги, заданий, а также связанных вопросов используется [Slack](#). Обсуждения на [GitBook](#) закрыты. Все вопросы, предложения и замечания по книге также пишите в [Slack](#).

Зачем Вам учиться программировать?

Знание программирования для сетевого инженера сравнимо со знанием английского. Если Вы знаете английский хотя бы на уровне, который позволяет читать техническую документацию, Вы сразу же расширяете свои возможности:

- доступно в несколько раз больше литературы, форумов и блогов;
- практически для любого вопроса или проблемы достаточно быстро находится решение, если Вы ввели запрос в Google.

Знание программирования в этом очень похоже. Если Вы знаете, например, Python хотя бы на базовом уровне, Вы уже открываете массу новых возможностей для себя. Аналогия с английским подходит ещё и потому, что можно работать сетевым инженером и быть хорошим специалистом без знания английского. Английский просто даёт возможности, но он не является обязательным требованием.

О книге

Если "в двух словах", то это такой CCNA по Python. С одной стороны, книга достаточно базовая, чтобы её мог одолеть любой желающий, а с другой стороны, в книге рассматриваются все основные темы, которые позволяют 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.

Для кого эта книга

Для сетевых инженеров с опытом программирования и без. Все примеры и домашние задания будут построены с уклоном на сетевое оборудование. Эта книга будет полезна сетевым инженерам, которые хотят автоматизировать задачи, с которыми сталкиваются каждый день и хотели заняться программированием, но не знали, с какой стороны подойти.

Ещё не решили, нужно ли читать книгу? Почтайте [отзывы](#). Скажите "Спасибо!" на saythanks.io.



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).

О книге

Книга разделена на шесть частей.

[Первая часть](#) книги посвящена основам языка, которые являются фундаментом для дальнейшего его изучения. В [главах 1 – 3](#), посвящённых подготовке к работе с Python, рассматриваются выбор ОС и установка её пакетов, выбор редактора, использование Git и GitHub. Кроме того, в них рассматриваются система установки пакетов Python и способ изоляции разных версий и пакетов Python, а также готовые виртуальные машины для выполнения заданий. В [главах 4 – 7](#) изложены основы Python. Вы узнаете, какие базовые типы данных поддерживает Python, как с ними работать, какие возможности и ограничения есть у них. Создание базовых скриптов, получение информации от пользователя и передача аргументов созданному скрипту, рассматриваются в [главе 5](#). [Глава 6](#) описывает механизмы контроля хода программы: условия (`if, else`), циклы (`for, while`) и работу с исключениями. В [главе 7](#) завершается знакомство с основами Python описанием принципов работы с файлами, и, в завершение первой части, приводятся примеры использования пройденных тем для решения задач в [главе 8](#).

[Вторая часть](#) описывает различные техники повторного использования кода: функции, модули, итераторы и генераторы. В [главе 9](#) описано, как создать функцию, какие типы параметров и аргументов она поддерживает. Разговор о встроенных функциях и о том, как их использовать, ведётся в [главе 10](#). [Глава 11](#) знакомит с модулями, их созданием, а также с повторным использованием кода из других скриптов. В [главе 12](#) рассказывается про различные полезные модули Python, такие как `subprocess`, `ipaddress`, `argparse` и другие.

[Третья часть](#) знакомит Вас с регулярными выражениями. В ней рассматривается и синтаксис регулярных выражений, и как с ними работать в Python. После этой главы Вы с лёгкостью сможете получать нужную информацию из вывода команд.

[Четвёртая часть](#) содержит необходимую информацию про запись и передачу данных средствами Python. В [главе 16](#) рассматривается стандарт Unicode и его использование в Python, а [глава 17](#) знакомит с форматами CSV, YAML и JSON. Формат CSV позволит работать с табличными данными – это могут быть данные из таблицы, базы данных или системы мониторинга. YAML удобно использовать для записи параметров в структурированном формате – как вручную, так и автоматически. Более того, YAML используется как язык описания сценариев в Ansible. Формат JSON подойдет для сохранения полученной информации, и, кроме того, он часто используется в интернете

как формат передачи данных разных API. В [главе 18](#) рассматривается работа с базами данных на примере SQLite, рассматриваются и основы языка SQL, и как работать с базами данных из Python.

[Пятая часть](#) рассказывает про работу с сетевым оборудованием через Python. В [главе 19](#) рассказ посвящён подключению к сетевому оборудованию через Telnet и SSH, рассматривается использование нескольких полезных модулей – каждый из них использует несколько отличный от другого модуля подход. [Глава 20](#) объясняет, как подключаться к оборудованию параллельно, используя потоки и процессы, а в [главе 21](#) рассматривается язык шаблонов Jinja2. Он позволит создавать шаблоны конфигурации с нуля, и таким образом, вместо замены параметров в текстовом файле, Вы легко сможете генерировать нужные команды с помощью Python. В [главе 22](#) разговор посвящён модулю TextFSM, задача которого является обратной модулю Jinja2. Это библиотека от Google, которая позволяет обрабатывать вывод команд show (и любых других аналогичных), и получать из него значения в виде переменных, то есть проводить парсинг вывода. Для обработки команды создается отдельный шаблон, который описан с помощью регулярных выражений.

[Шестая часть](#) посвящена основам работы с Ansible, а также модулям для работы с оборудованием Cisco. Этой информации будет достаточно, чтобы начать использовать Ansible, но, так как это лишь небольшая часть его возможностей, то остальное вынесено в отдельную [книгу](#).

В приложениях собраны те темы, которые не вошли в другие разделы, но которые всё равно очень полезны.

И, наконец, в последней главе приводятся [рекомендации](#) по дальнейшему обучению.

Книга всегда будет оставаться бесплатной, поэтому Вам не нужно переживать, что она будет удалена.

Требуемые версии ОС и Python

Все примеры и выводы терминала в книге показываются на Debian Linux. В книге используется Python 3.6, но для большинства примеров подойдет и Python 3.4, и 3.5. Только в некоторых примерах требуется версия 3.6 или выше чем 3.5. Это всегда явно указано и, как правило, касается дополнительных возможностей.

Примеры

Все примеры, которые используются в книге, располагаются в [репозитории](#). Примеры, которые рассматриваются в разделах книги, являются обучающими. Это значит, что они не обязательно показывают лучший вариант решения задачи, так как они

основаны только на той информации, которая рассматривалась в предыдущих главах книги. Кроме того, довольно часто примеры, которые давались в разделах, развиваются в заданиях. То есть, в заданиях Вам нужно будет сделать лучшую, более универсальную, и, в целом, более правильную версию кода. Если есть возможность, лучше набирать код, который используется в книге, самостоятельно, или, как минимум, скачать примеры и попробовать что-то в них изменить – так информация будет лучше запоминаться. Если такой возможности нет, например, когда Вы читаете книгу в дороге, лучше повторить примеры самостоятельно позже. В любом случае, обязательно нужно делать задания вручную.

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#), том же, где располагаются примеры кода. Если в заданиях раздела есть задания с буквами (например, 5.2а), то нужно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают идею в соответствующем задании без буквы. Если получается, лучше делать задания по порядку. В книге специально не приведены ответы на задания, так как, к сожалению, когда есть ответы, очень часто вместо того, чтобы попытаться решить сложное задание самостоятельно, подглядывают в них. Конечно, иногда возникает ситуация, когда никак не получается решить задание – попробуйте отложить его, задать вопрос в [Slack](#) и сделать какое-либо другое.

На [Stack Overflow](#) есть ответы практически на любые вопросы. Так что, если Google отправил Вас на него, это, с большой вероятностью значит, что ответ найден. Запросы, конечно же, лучше писать на английском – по Python очень много материалов и, как правило, подсказку найти легко

Ответы могли бы показать, как ещё можно выполнить задание или же как лучше это сделать. Но на этот счёт не следует переживать, так как, скорее всего, в следующих разделах встретится пример, в котором будет показано, как писать такой код.

Тесты

Для части тем книги созданы тесты:

- [Типы данных. Часть 1](#)
- [Типы данных. Часть 2](#)
- [Контроль хода программы. Часть 1](#)
- [Контроль хода программы. Часть 2](#)
- [Функции и модули. Часть 1](#)
- [Функции и модули. Часть 2](#)

- [Регулярные выражения. Часть 1](#)
- [Регулярные выражения. Часть 2](#)
- [Базы данных](#)

Эти тесты можно использовать как для проверки знаний, так и в роли заданий. Очень полезно пройти тест после прочтения соответствующей темы. Он позволит Вам вспомнить материал темы, а также увидеть на практике разные аспекты работы с Python. Постарайтесь сначала ответить самостоятельно, а затем подсмотреть ответы в IPython по тем вопросам, в которых Вы сомневаетесь.

Презентации

Для всех тем книги есть презентации в [репозитории](#). По ним удобно быстро просматривать информацию и повторять. Если Вы знаете основы Python, то стоит их пролистать.

| Скачать все презентации в формате PDF можно в специальном [репозитории](#)

Форматы файлов книги

Книгу можно читать в нескольких форматах:

- [онлайн](#);
- [PDF/Mobi/ePub](#).

Они автоматически обновляются, поэтому всегда содержат одинаковую информацию.

| Пожалуйста, не выкладывайте скачанные версии книги. Вместо этого просто давайте ссылку на книгу

Обновление книги

В сентябре 2017 года книга была переведена на Python 3.6. Она ещё может дополняться, а также будут исправляться ошибки и опечатки. Поэтому, если Вы не будете читать книгу в ближайшее время, то лучше сохраните ссылку на онлайн-версию книги, а не PDF/Mobi/ePub, а когда решите читать, скачайте свежую версию.

| Подробнее об обновлениях можно почитать в [Changelog](#) книги

GitBook отображает, когда были сделаны последние изменения, поэтому легко можно определить, были ли изменения за последнее время.

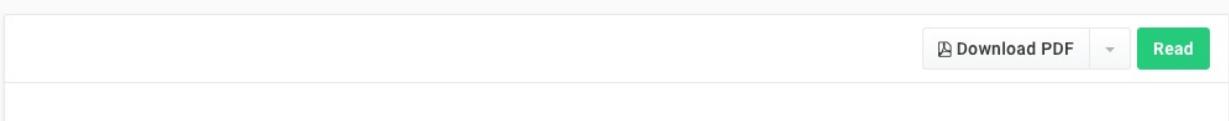
 **GitBook** WE ARE HIRING! Pricing Explore About Blog Sign In Sign Up

natenka > PyNEng

Updated 2 hours ago

ABOUT 1 DISCUSSION

★ Star 1  3

Изменения в книге

- 28.11.2017 – в задании 19.2б добавлены примеры команд с ошибками;
- 15.11.2017 – примеры в части Ansible проверены на версии 2.4.1;
- 05.11.2017 – задания 20.2, 20.2а переписаны, чтобы в них предполагалось использование concurrent.futures, задания 20.3, 20.3а удалены;
- 05.11.2017 – глава [Основы threading и multiprocessing](#) перенесена в часть [Дополнительная информация](#). В этих главах рассматриваются только основы модулей threading и multiprocessing, при этом, задача запуска функции в потоках и процессах намного проще решается в модуле concurrent.futures. К тому же, при его использовании, не надо переписывать существующий код. На случай, если задача будет более сложная и функционала concurrent.futures не хватит, оставлены основы модулей threading и multiprocessing. Разумеется, этих основ недостаточно, чтобы решать более сложные задачи, но это неплохой старт;
- 21.10.2017 – раздел [List, dict, set comprehensions](#) перенесён в главу 8;
- 15.10.2017 – реорганизация книги. Книга разделена на главы:
 - [Глава I. Основы Python](#)
 - [Глава II. Повторное использование кода](#)
 - [Глава III. Регулярные выражения](#)
 - [Глава IV. Запись и передача данных](#)
 - [Глава V. Работа с сетевым оборудованием](#)
 - [Глава VI. Ansible](#)

Изменена нумерация глав и некоторые разделы разбиты на несколько. Названия разделов и нумерация заданий изменены соответственно в [репозитории](#). Изменения по разделам:

- добавлена глава [Примеры использования основ](#). В ней показаны примеры на основе пройденных тем, а также находятся разделы [Распаковка переменных](#) и [List, dict, set comprehensions](#);
- раздел о Git и GitHub преобразован в [Главу 2](#);
- глава о функциях разделена на две части: [Функции](#) и [Полезные встроенные функции](#);
- глава о модулях разделена на две части: [Модули](#) и [Полезные модули](#).
- глава о Unicode перенесена в часть [Запись и передача данных](#).
- 14.10.2017 – в главу [Работа с файлами в формате CSV](#) добавлена информация о DictWriter;
- 27.09.2017 – раздел про форматирование строк разделён на две части. Ранее

примеры со старым и новым вариантом форматирования строк были перемешаны, теперь идёт сначала новый вариант форматирования строк, затем старый;

- 09.09.2017 – у книги появился замечательный редактор Слава Скороход. Все правки редактора внесены, теперь ошибок и опечаток намного меньше;
- 01.09.2017 – версия книги для Python 3 стала основной. Все изменения описаны в [статье](#) на сайте курса по книге. Содержимое книги обновлено до Python версии 3.6, все примеры, задания и содержимое книги протестированы. [Версия книги для Python 2.7](#) по-прежнему доступна.

Ресурсы для обучения по книге

Здесь собраны ссылки на все ресурсы, которые пригодятся в процессе обучения:

- [примерный план обучения](#);
- [варианты виртуальной машины для курса](#);
- [репозиторий](#) с примерами и заданиями;
- [тесты](#);
- [команда PyNEng](#) в Slack;
- [презентации](#) пригодятся для повторения материала.

Почти в каждой части книги есть глава Дополнительные материалы, в которой находятся полезные материалы и ссылки по теме, а также ссылки на документацию. Кроме того, я сделала [подборку](#) ресурсов по Python для сетевых инженеров, где можно найти много полезных статей, книг, видео курсов и подкастов.

Как учиться по этой книге

Есть два основных варианта обучения по данной книге.

Первый вариант

Этот вариант подходит в следующих случаях:

- Вы не уверены, что хотите читать всю книгу;
- Вам нужно просто автоматизировать подключение к устройствам и выполнение команд;
- Вы не уверены, что хотите изучать Python.

В этом случае начните с [Ansible](#). Для установки этой системы контроля конфигурациями и её базового использования, не требуется знать Python. Достаточно установить Ansible, взять пример использования и попробовать выполнить команды на оборудовании, например команды `show`. После этого можно использовать Ansible для отправки команд конфигурации, которые не влияют на передачу трафика, таких как:

- подписи интерфейсов;
- настройка общих параметров, например, `alias`;
- генерация конфигураций по шаблону для первичной настройки оборудования.

Если задачу не получается решить в Ansible, или Вы просто желаете изучить Python, то переходите ко второму варианту.

Второй вариант

Если Вы уже решили изучать Python и примерно представляете, что он может дать, то можно просто взять [Пример плана обучения](#) и идти по нему. Многие вещи, особенно в начале книги будут базовыми, но не пропускайте их, если Вы их не знаете. Можно с уверенностью сказать, что такие темы, как типы данных или управление ходом программы, будут основой всего, что мы будем изучать дальше.

Обязательно практикуйтесь! Пробуйте воспроизвести все примеры в главах (набирайте их вручную или повторяйте их по памяти). Пробуйте менять параметры в примерах, чтобы увидеть, как поведёт себя программа. Обязательно выполняйте все задания в конце каждой главы.

План занятий можно менять под себя, но не стоит делать большие перерывы между темами и не стоит заниматься слишком много: если делать большие перерывы, то всё, что было изучено раньше, начнёт забываться, а если торопиться, то, скорее всего, всё

сведется к чтению текста без выполнения заданий. В таком случае, будет очень мало пользы. Если следовать расписанию, пусть и с небольшими отклонениями, то к концу наберётся два месяца практики. За это время все базовые понятия хорошо усваются, и главное, после завершения курса, не бросать практиковаться и учиться.

По мере чтения книги, скорее всего, будут возникать идеи, что сделать по работе – это отлично! Пытайтесь реализовать это обязательно, записывайте идеи, чтобы не забыть, если не можете сразу сделать это. Примеры использования, которые придумываете Вы, будут особенно сильно помогать расти, так как они естественным образом вырастают из Ваших задач – будет добавляться функционал, будут улучшения, которые нужны именно Вам. Если же Вы чувствуете, что читаете книгу, но пока никаких идей нет, не огорчайтесь – в книге достаточно упражнений, на которых Вы сможете потренироваться писать код, и которые, весьма вероятно, натолкнут Вас на идеи.

После прочтения книги

К сожалению, новые знания очень быстро забываются без применения и без повторения. Не делайте слишком большой перерыв после курса. Если оставить новые знания без применения хотя бы на 2-4 недели, то большая часть из них выветрится.

Если Вам удалось изобрести себе задач по ходу курса – отлично, реализуйте их, напишите список и делайте задачи постепенно. Это прекрасный способ изучать язык дальше и повторять пройденное. Идеи сами будут двигать Вас дальше, Вы будете изучать новые темы и новые возможности более естественно, одновременно с развитием Ваших программ. Создайте репозиторий на GitHub и выкладывайте туда свои скрипты. Дорабатывайте их, поделитесь с коллегами. Отличный способ запомнить лучше определённую тему - рассказать её другому.

Если Вы хотите и дальше развиваться в этой области, продолжайте учиться и читать. В части [Продолжение обучения](#) я подготовила различные ссылки на видео, книги, курсы, блоги и задачки по Python. Выберите тот ресурс, который Вам нравится, и изучайте Python дальше, и обязательно практикуйтесь. Только читать книгу, статью или смотреть видео недостаточно, обязательно пишите код.

Пример плана обучения

Здесь приведён пример плана, по которому можно учиться, читая эту книгу. План рассчитан на то, что Вы изучаете Python с нуля, в нём всё разбито по неделям. Можно идти быстрее или медленнее, в зависимости от уровня знаний и наличия времени. По возможности, попробуйте придерживаться темпов, указанных в таблице, возможно, они Вам подойдут. Можно читать темы в выходные, а на рабочей неделе выполнять задания по этой теме.

Неделя	Темы	Задания
0	Подготовка к работе	Подготовка к работе
1	Начало работы с Python	-
2	Типы данных в Python	Типы данных в Python
3	Создание базовых скриптов	Создание базовых скриптов
4	Контроль хода программы	Контроль хода программы
5	Работа с файлами	Работа с файлами
6	Функции	Функции
7	Модули	Модули
8	Регулярные выражения	Регулярные выражения
9	Unicode	-
10	Сериализация данных	Сериализация данных
11	Работа с базами данных	Работа с базами данных
12	Подключение к оборудованию	Подключение к оборудованию
13	Шаблоны конфигураций с Jinja2	Шаблоны конфигураций с Jinja2
14	Обработка вывода команд с TextFSM	Обработка вывода команд с TextFSM
15	Ansible	Ansible

Часто задаваемые вопросы (FAQ)

Здесь собраны вопросы, которые наиболее часто возникают при чтении книги.

Чем это отличается от обычного вводного курса по Python?

Основных отличий три:

- основы даются достаточно коротко;
- подразумевается определённая предметная область знаний (сетевое оборудование);
- все примеры, по возможности, ориентированы на сетевое оборудование.

Я сетевик. Для чего мне нужна эта книга?

В первую очередь – для автоматизации рутинных задач. Автоматизация даёт несколько преимуществ:

- высокоровневое мышление – проще подняться над всем, когда Вы свободны от рутинной работы. У Вас появится время и возможность думать об улучшениях;
- доверие – Вы не будете бояться делать изменения, которые, как правило, сопряжены с риском, так как сеть это основа работы всех приложений и цена ошибки высока;
- консистентная конфигурация – Вы сможете автоматизированно создавать файлы настроек сетевого оборудования, от пользователей и подписей интерфейсов до функционала безопасности, и будете меньше переживать о том, забыли ли Вы нечто.

Конечно, не будет такого, что после прочтения книги, Вы "всё автоматизируете и наступит счастье", но это шаг в данном направлении. Я ни в коем случае не агитирую за то, чтобы автоматизация выполнялась кучей самописных скриптов. Если есть софт, который решает нужные Вам задачи, это отлично, используйте его. Но если его нет, или если Вы просто ещё о таком не думали, попробуйте начать с простого – Ansible, например, позволит выполнять многие задачи практически "из коробки".

Зачем тогда учить Python? Дело в том, что тот же Ansible не решит все вопросы. И, возможно, Вам понадобится добавить какой-то функционал самостоятельно. Кроме непосредственной настройки оборудования, есть ежедневные рутинные задачи, которые можно автоматизировать с помощью Python. Скажем так, если Вы не хотите

разбираясь с Python, но хотите автоматизировать процесс настройки и работы с оборудованием, обратите своё внимание на Ansible. Даже "из коробки" он будет очень полезен. Если же Вы потом войдете во вкус и захотите добавить своё, чего нет в Ansible, возвращайтесь :-)

И ещё, этот курс не только о том, как использовать Python для подключения к оборудованию и его настройке. Он и о том, как решать задачи, которые не касаются подключения к оборудованию, например, изменить что-то в нескольких файлах конфигурации, или обработать log-файл – Python поможет Вам решать в том числе и подобные задачи.

Почему книга именно для сетевых инженеров?

Есть несколько причин:

- сетевые инженеры уже обладают опытом работы в ИТ, и часть концепций им знакома, и, скорее всего, какие-то основы программирования большинству уже будут знакомы. Это означает, что будет гораздо проще разобраться с Python;
- работа в командной строке и написание скриптов вряд ли испугает их;
- у сетевых инженеров есть знакомая им предметная область, на которую можно опираться при составлении примеров и заданий.

Если рассказывать на абстрактных примерах "о котиках и зайчиках", это одно. Но когда в примерах есть возможность использовать идеи из предметной области, всё становится проще, рождаются конкретные идеи, как улучшить какую-либо программу, скрипт. А когда человек пытается её улучшить, он начинает разбираться с новым - это очень сильно помогает продвигаться вперёд.

Почему именно Python?

Причины следующие:

- в контексте работы с сетевым оборудованием, сейчас часто используется именно Python;
- на некотором оборудовании Python встроен или есть API, который поддерживает Python;
- Python достаточно прост для изучения (конечно, это относительно, и более простым может казаться другой язык, но, скорее, это будет из-за имеющегося опыта работы с языком, а не потому, что Python сложный);
- с Python Вы вряд ли быстро дойдете до границ возможностей языка;
- Python может использоваться не только для написания скриптов, но и для разработки приложений. Разумеется, это не является задачей этой книги, но, по

крайней мере, Вы потратите время на язык, который позволит Вам легко шагнуть дальше, чем написание простых скриптов;

- из программ, связанных с сетями, на Python написан, например, [GNS3](#).

И еще один момент – в контексте книги, Python нужно рассматривать не как единственно правильный вариант, и не как "правильный" язык. Нет, Python это просто инструмент, как отвертка например, и мы учимся им пользоваться для конкретных задач. То есть, никакой идеологической подоплеки здесь нет, никакого "только Python" и никакого поклонения тем более. Странно поклоняться отвертке :-) Всё просто - есть хороший и удобный инструмент, который подойдет к разным задачам. Он не лучший во всём и далеко не единственный язык в принципе. Начните с него, и потом Вы сможете самостоятельно выбрать нечто другое, если захотите – эти знания всё равно не пропадут.

Нужный мне модуль не поддерживает Python 3

Есть несколько вариантов решения:

- попробуйте найти альтернативный модуль, который поддерживает Python 3 (не обязательно последней версии языка);
- попробуйте найти community-версию этого модуля для Python 3. Возможно, официальной версии нет, но сообщество могло перевести его самостоятельно на версию 3, особенно если этот модуль популярен;
- используйте Python 2.7, ничего страшного не произойдет. Если Вы не собираетесь писать огромное приложение, а просто используете Python для автоматизации своих задач, Python 2.7 совершенно точно подойдет.

Я не знаю, нужно ли мне это.

Я, конечно же, считаю, что нужно :-) Иначе я бы не писала эту книгу. Совсем не факт, что Вам захочется погружаться во всё это, поэтому для начала попробуйте разобраться с [Ansible](#). Возможно, Вам хватит надолго его возможностей. Начните с простых команд show, попробуйте подключиться сначала к тестовому оборудованию (виртуальным машинам), затем попробуйте выполнить команду show на реальной сети, на 2-3 устройствах, потом на большем количестве. Если Вам этого будет достаточно, можно остановиться на этом. Следующим шагом я бы попробовала использование Ansible для генерации шаблонов конфигурации.

Зачем сетевому инженеру программирование?

На мой взгляд, для сетевого инженера умение программировать очень важно, и не потому, что сейчас все об этом говорят, или пугают SDN, потерей работы или чем-то подобным, а потому, что сетевой инженер постоянно сталкивается с:

- рутинными задачами;
- проблемами и решениями, которые надо протестировать;
- большим объемом однотипных и повторяющихся задач;
- большим количеством оборудования;

На текущий момент большое количество оборудования по-прежнему предлагает нам только интерфейс командной строки и неструктурированный вывод команд. Управляющий софт часто ограничен вендором, дорого стоит и имеет урезанные возможности – в итоге мы вручную снова и снова делаем одно и то же. Даже такие банальные вещи, как отправить одну и ту же команду `show` на 20 устройств, не всегда просто сделать. Допустим, ваш SSH-клиент поддерживает эту возможность. А если Вам теперь надо проанализировать вывод? Мы ограничены теми средствами, которые нам дали, а знание программирования, даже самое базовое, позволяет нам расширить наши средства и даже создавать новые. Я не считаю, что всем надо торопиться учиться программировать, но для инженера это очень важный навык. Именно для инженера, а не для всех на свете.

Сейчас явно наблюдается тенденция, которую можно описать фразой "все учимся программировать", и это, в целом, хорошо. Но программирование это не что-то элементарное, это сложно, в это нужно вкладывать много времени, особенно если Вы никогда не имели отношения к техническому миру. Может сложиться впечатление, что достаточно пройти "вот эти вот курсы" и через 3 месяца Вы крутой программист с высокой зарплатой. Нет, этот книга не об этом :-) Мы не говорим в ней о программировании как профессии и не ставим такую цель, мы говорим о программировании как инструменте, таком как, например, знание CLI Linux. Дело не в том, что инженеры какие-то особенные, просто, как правило:

- они уже имеют техническое образование;
- многие работают, так или иначе, с командной строкой;
- они сталкивались, как минимум, с одним языком программирования;
- у них "инженерный склад ума".

Это не значит, что всем остальным "не дано". Просто инженерам это будет проще.

Книга будет когда-то платной?

Нет, эта книга всегда будет бесплатной. Я читаю платно [онлайн курс "Python для сетевых инженеров"](#), но это не будет влиять на эту книгу - она всегда будет бесплатной.

Благодарности

Спасибо всем, кто проявил интерес к первому анонсу курса – ваш интерес подтвердил, что это будет кому-то нужно. Павел Пасынок, спасибо тебе за то, что согласился на курс. С вами было интересно работать, и это добавило мне мотивации завершить курс, и я особенно рада, что знания, которые Вы получили на курсе, нашли практическое применение. Алексей Кириллов, самое большое спасибо тебе :-) Я всегда могла обсудить с тобой любой вопрос по курсу. Ты помогал мне поддерживать мотивацию и не уходить в дебри. Общение с тобой вдохновляло меня продолжать, особенно в сложные моменты. Спасибо тебе за вдохновение, положительные эмоции и поддержку! Спасибо всем, кто писал комментарии к книге – благодаря вам в книге не только стало меньше опечаток и ошибок, но и содержание книги стало лучше. Спасибо всем слушателям онлайн-курса – благодаря вашим вопросам книга стала намного лучше. Слава Скороход, спасибо тебе огромное, что вызвался быть редактором – количество ошибок теперь стремится к нулю :-)

I. Основы Python

Первая часть книги посвящена основам Python. В ней рассматриваются:

- типы данных Python;
- как создавать базовые скрипты;
- контроль хода программы;
- работа с файлами.

Кроме того, в [главе 8](#) показаны примеры использования пройденных тем для решения задач.

Подготовка к работе

Для того, чтобы начать работать с Python, надо определиться с некоторыми вещами:

- какая операционная система будет использоваться;
- какой редактор будет использоваться;
- какая версия Python будет использоваться.

В книге используется Debian Linux (в других ОС вывод может незначительно отличаться) и Python 3.6.

Виртуальная машина

Для выполнения заданий в книге лучше всего сделать отдельную виртуальную машину. Имеются следующие варианты:

- взять подготовленную виртуалку для книги;
- использовать один из облачных сервисов;
- подготовить виртуалку самостоятельно.

Подготовленные виртуальные машины

Для книги подготовлены виртуальные машины, в которых установлены:

- Python 3.6 в виртуальном окружении;
- IPython;
- почти все модули, которые потребуются для выполнения заданий.

Есть два варианта подготовленных виртуальных машин (по ссылкам находятся инструкции для каждого варианта):

- [Vagrant](#) – логин и пароль vagrant/vagrant;
- [VMware](#) – логин и пароль python/python.

Вы можете выбрать одну из них или установить все самостоятельно, но будет лучше, если Вы выделите отдельную виртуальную машину.

Облачный сервис

Ещё один вариант – использовать один из следующих сервисов:

- [Cloud9](#) – выделяет виртуалку (контейнер), в котором можно полноценно работать, ставить пакеты и так далее. Кроме того, в этом контейнере есть графический

редактор, в котором можно делать задания. Базовая виртуалка бесплатна, но для регистрации понадобится ввести номер карточки;

- [PythonAnywhere](#) - выделяет отдельную виртуалку, но в бесплатном варианте Вы можете работать только из командной строки, то есть, нет графического текстового редактора;
- [repl.it](#) – этот сервис предоставляет онлайн-интерпретатор Python, а также графический редактор. [Пример использования](#).

Самостоятельная подготовка виртуальной машины

Если Вы используете Linux, Unix или Mac OS, то, скорее всего, Python уже установлен. Нужно только проверить, что установлена версия 3.6 (которая используется в книге), и если версия другая, надо установить Python 3.6. Подойдут и версии Python 3.4-3.5, но лучше использовать 3.6. Установка Python 3.6, если его нет в ОС, выполняется самостоятельно.

Процедура установки Python 3.6 на Debian:

```
$ sudo apt-get install build-essential ca-certificates curl gcc libbz2-dev libffi-dev  
libncurses5-dev libncursesw5-dev libreadline-dev libssl-dev libsdlite3-dev llvm make p  
ython3-dev tk-dev wget xz-utils zlib1g-dev  
$ wget https://www.python.org/ftp/python/3.6.3/Python-3.6.3.tgz  
$ tar xvf Python-3.6.3.tgz  
$ cd Python-3.6.3  
$ ./configure --enable-optimizations --enable-loadable-sqlite-extensions  
$ make -j8  
$ sudo make altinstall
```

Чтобы в виртуальном окружении по умолчанию использовался Python 3.6, создайте это окружение следующим образом (подробнее в разделе по виртуальным окружениям):

```
$ mkvirtualenv --python=/usr/local/bin/python3.6 rupeng-ruz
```

Если Вы используете Windows, то, скорее всего, Python нужно будет установить. Один из самых простых вариантов для Windows – установить окружение [Anaconda](#). В окружении есть IDE Spyder ([Integrated development environment](#)), который можно использовать вместо редактора. Windows не рекомендована в качестве ОС для обучения, например потому, что на ней нельзя установить Ansible.

Выбор редактора

Ещё один важный момент – выбор редактора. В следующем разделе приведены примеры редакторов для разных ОС. Вместо редактора можно использовать IDE. IDE это хорошая вещь, но не стоит переходить на IDE из-за таких вещей как:

- подсветка кода;
- подсказки синтаксиса;
- автоматические отступы (важно для Python).

Всё это есть в любом хорошем редакторе, но для этого может потребоваться установить дополнительные модули. В начале работы может получиться так, что IDE будет только отвлекать Вас обилием возможностей. Список IDE для Python можно можно посмотреть [здесь](#). Например, можно выбрать [PyCharm](#) или Spyder для Windows.

ОС и редактор

Можно выбрать любую ОС и любой редактор, но желательно использовать Python версии 3.6, так как здесь будет использоваться именно эта версия. Также для курса желательно не использовать Windows, так как, например, Ansible можно установить только на Linux, однако более половины заданий книги можно без проблем выполнить на Windows.

| В [документации](#) Python описано, как установить Python на Windows.

Популярные редакторы для разных ОС (vi, vim и emacs не указаны):

- Linux:
 - gedit;
 - Geany;
 - nano;
 - Sublime Text.
- Mac OS:
 - Geany;
 - TextMate;
 - TextWrangler.
- Windows:
 - Notepad++;
 - Geany.

Для начала работы можно взять первый из списка для соответствующей операционной системы. Далее выводы команд, интерпретатора и скриптов приводятся для Linux. В других ОС вывод может незначительно отличаться.

Система управления пакетами pip

Для установки пакетов Python, будет использоваться pip. Это система управления пакетами, которая используется для установки пакетов из Python Package Index (PyPI). Скорее всего, если у Вас уже установлен Python, то установлен и pip.

Проверка версии pip:

```
$ pip --version  
pip 9.0.1 from /home/vagrant/venv/py3_convert/lib/python3.6/site-packages (python 3.6)
```

Если команда выдала ошибку, значит, pip не установлен. Установка pip описана в [документации](#)

Установка модулей

Для установки модулей используется команда pip install:

```
$ pip install tabulate
```

Удаление пакета выполняется таким образом:

```
$ pip uninstall tabulate
```

Кроме того, иногда необходимо обновить пакет:

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

pip или pip3

В зависимости от того, как установлен и настроен Python в системе, может потребоваться использовать pip3, вместо pip. Чтобы проверить, какой вариант используется, надо выполнить команду "pip --version".

Вариант, когда pip соответствует Python 2.7:

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

Вариант, когда pip3 соответствует Python 3.4:

```
$ pip3 --version  
pip 1.5.6 from /usr/lib/python3/dist-packages (python 3.4)
```

Если в системе используется pip3, то каждый раз, когда в книге устанавливается модуль Python, нужно будет заменить pip на pip3.

Также можно использовать альтернативный вариант вызова pip:

```
$ python3.6 -m pip install tabulate
```

Таким образом, всегда понятно для какой именно версии Python устанавливается пакет.

Виртуальные окружения

Виртуальные окружения:

- позволяют изолировать различные проекты друг от друга;
- пакеты, которые нужны разным проектам, находятся в разных местах – если, например, в одном проекте требуется пакет версии 1.0, а в другом проекте требуется тот же пакет, но версии 3.1, то они не будут мешать друг другу;
- пакеты, которые установлены в виртуальных окружениях, не перебивают глобальные пакеты.

virtualenvwrapper

Виртуальные окружения создаются с помощью virtualenvwrapper.

Установка virtualenvwrapper с помощью pip:

```
$ sudo pip3.6 install virtualenvwrapper
```

После установки, в файле .bashrc, находящимся в домашней папке текущего пользователя, нужно добавить несколько строк:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3.6
export WORKON_HOME=~/venv
./usr/local/bin/virtualenvwrapper.sh
```

Если Вы используете командный интерпретатор, отличный от bash, посмотрите, поддерживается ли он в [документации](#) virtualenvwrapper. Переменная окружения VIRTUALENVWRAPPER_PYTHON указывает на бинарный файл командной строки Python, WORKON_HOME – на расположение виртуальных окружений. Третья строка указывает, где находится скрипт, установленный с пакетом virtualenvwrapper. Для того, чтобы скрипт virtualenvwrapper.sh выполнился и можно было работать с виртуальными окружениями, надо перезапустить bash.

Перезапуск командного интерпретатора:

```
$ exec bash
```

Такой вариант может быть не всегда правильным. Подробнее на [Stack Overflow](#).

Работа с виртуальными окружениями

Создание нового виртуального окружения, в котором Python 3.6 используется по умолчанию:

```
$ mkvirtualenv --python=/usr/local/bin/python3.6 pyneng
New python executable in PyNEng/bin/python
Installing distribute.....done.
Installing pip.....done.
(pyneng)$
```

В скобках перед стандартным приглашением отображается имя виртуального окружения. Это означает, что Вы находитесь в нём. В virtualenvwrapper по Tab работает автодополнение имени виртуального окружения. Это особенно удобно в тех случаях, когда виртуальных окружений много. Теперь в том каталоге, который был указан в переменной окружения WORKON_HOME, создан каталог pyneng:

```
(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
```

Выход из виртуального окружения:

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

Для перехода в созданное виртуальное окружение надо выполнить команду workon:

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

Если необходимо перейти из одного виртуального окружения в другое, то необязательно делать deactivate, можно перейти сразу через workon:

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

Если виртуальное окружение нужно удалить, то надо использовать команду `rmvirtualenv`:

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

Посмотреть, какие пакеты установлены в виртуальном окружении можно командой `lssitepackages`:

```
(pyneng)$ lssitepackages
ANSI.py
ANSI.pyc
decorator-4.0.4-py2.7.egg-info
decorator.py
decorator.pyc
distribute-0.6.24-py2.7.egg
easy-install.pth
fdpexpect.py
fdpexpect.pyc
FSM.py
FSM.pyc
IPython
ipython-4.0.0-py2.7.egg-info
ipython_genutils
ipython_genutils-0.1.0-py2.7.egg-info
path.py
path.py-8.1.1-py2.7.egg-info
path.pyc
repxpect
repxpect-3.3-py2.7.egg-info
pickleshare-0.5-py2.7.egg-info
pickleshare.py
pickleshare.pyc
pip-1.1-py2.7.egg
pxssh.py
pxssh.pyc
requests
requests-2.7.0-py2.7.egg-info
screen.py
screen.pyc
setuptools.pth
simplegeneric-0.8.1-py2.7.egg-info
simplegeneric.py
simplegeneric.pyc
test_path.py
test_path.pyc
traitlets
traitlets-4.0.0-py2.7.egg-info
```

Встроенный модуль `venv`

Начиная с версии 3.5, в Python рекомендуется использовать модуль `venv` для создания виртуальных окружений:

```
$ python3.6 -m venv new/pyneng
```

Вместо `python3.6` может использоваться `python` или `python3`, в зависимости от того, как установлен Python 3.6. Эта команда создаёт указанный каталог и все необходимые каталоги внутри него, если они не были созданы.

Команда создаёт следующую структуру каталогов:

```
$ 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
```

Для перехода в виртуальное окружение надо выполнить команду:

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

Для выхода из виртуального окружения используется команда deactivate:

```
$ deactivate
```

Подробнее о модуле `venv` в [документации](#).

Установка пакетов

Например, установим в виртуальном окружении пакет `simplejson`.

```
(pyneng)$ pip install simplejson
...
Successfully installed simplejson
Cleaning up...
```

Если перейти в IPython (рассматривается в [главе 3](#)) и импортировать `simplejson`, то он доступен и никаких ошибок нет:

```
(pyneng)$ ipython

In [1]: import simplejson

In [2]: simplejson
simplejson

In [2]: simplejson.
simplejson.Decimal      simplejson.decoder
simplejson.JSONDecodeError simplejson.dump
simplejson.JSONDecoder    simplejson.dumps
simplejson.JSONEncoder     simplejson.encoder
simplejson.JSONEncoderForHTML simplejson.load
simplejson.OrderedDict    simplejson.loads
simplejson.absolute_import  simplejson.scanner
simplejson.compat           simplejson.simple_first
```

Но если выйти из виртуального окружения и попытаться сделать то же самое, то такого модуля нет:

```
(pyneng)$ deactivate

$ ipython

In [1]: import simplejson
-----
ImportError                               Traceback (most recent call last)
<ipython-input-1-ac998a77e3e2> in <module>()
----> 1 import simplejson

ImportError: No module named simplejson
```

Интерпретатор Python

Перед началом работы надо проверить, что при вызове интерпретатора Python вывод будет таким:

```
$ python
Python 3.6.0 (default, May 31 2017, 07:04:38)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Вывод показывает, что установлен Python 3.6. Приглашение ">>>", это стандартное приглашение интерпретатора Python. Вызов интерпретатора выполняется командой `python`, а чтобы выйти, нужно набрать `quit()`, либо нажать `Ctrl+D`.

Дополнительные материалы

Документация:

- [Python Setup and Usage](#)
- [pip](#)
- [venv](#)
- [virtualenvwrapper](#)

Редакторы и IDE:

- [Python Editors](#)
- [Integrated Development Environments](#)
- [VIM and Python - a Match Made in Heaven](#)

Задания

Задание 1.1

Единственное задание в этом разделе: подготовка к работе.

Для этого нужно:

- определиться с ОС, которую Вы будете использовать:
 - так как все примеры в книге ориентированы на Linux (Debian), желательно использовать его;
 - желательно использовать новую виртуальную машину, чтобы было спокойнее экспериментировать.
- установить Python 3.6:
 - проверить, что Python и pip установлены.
- создать виртуальное окружение, в котором Вы будете работать;
- определиться с редактором;
- начиная с [главы 12](#), мы будем подключаться к оборудованию, поэтому Вам нужно подготовить виртуальное или реальное оборудование.

Использование Git и GitHub

В книге достаточно много заданий и нужно где-то их хранить. Один из вариантов – использование для этого Git и GitHub. Конечно, можно использовать для этого и другие средства, но, используя GitHub, можно постепенно разобраться с ним и затем использовать его для других задач. Задания и примеры из книги находятся в отдельном [репозитории](#) на GitHub. Конечно, их можно просто скачать как zip-архив, но лучше работать с репозиторием с помощью Git, тогда можно будет посмотреть внесённые изменения и легко обновить репозиторий. Если изучать Git с нуля и, особенно, если это первая система контроля версий, с которой Вы работаете, информации может быть очень много, поэтому в этой главе всё нацелено на практическую сторону вопроса, и рассказывается:

- как начать использовать Git и GitHub;
- как выполнить базовые настройки;
- как посмотреть информацию и/или изменения.

Теории в этом подразделе будет мало, но будут даны ссылки на полезные ресурсы. Попробуйте сначала провести все базовые настройки для выполнения заданий, а потом продолжайте читать книгу. И в конце, когда базовая работа с Git и GitHub будет уже привычным делом, почитайте о них подробнее. Для чего может пригодиться Git:

- для хранения конфигураций и всех изменений в них;
- для хранения документации и всех её версий;
- для хранения схем и всех их версий;
- для хранения кода и его версий.

GitHub позволяет централизованно хранить все перечисленные выше вещи, но следует учитывать, что эти ресурсы будут доступны и другим. У GitHub есть и приватные репозитории (платные), но даже в них, пожалуй, не стоит выкладывать такую информацию, как пароли. Конечно, основное использование GitHub это размещение кода различных проектов. Кроме этого, GitHub ещё и:

- хостинг для вашего сайта ([GitHub Pages](#));
- хостинг для онлайн-презентаций и инструмент для их создания ([GitPitch](#));
- вместе с [GitBook](#), это ещё и платформа для публикации книг, документации или подобного тому.

Основы Git

Git это распределённая система контроля версий (Version Control System, VCS), которая широко используется и выпущена под лицензией GNU GPL v2. Она может:

- отслеживать изменения в файлах;
- хранить несколько версий одного файла;
- отменять внесённые изменения;
- регистрировать, кто и когда сделал изменения.

Git хранит изменения как снимок (snapshot) всего репозитория. Этот снимок выполняется после каждого коммита (commit).

Установка Git:

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

Первичная настройка Git

Для начала работы с Git, необходимо указать имя и e-mail пользователя, которые будут использоваться для синхронизации локального репозитория с репозиторием на GitHub:

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

Посмотреть настройки Git можно таким образом:

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

Инициализация репозитория

Инициализация репозитория выполняется с помощью команды `git init`:

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

После выполнения этой команды, в текущем каталоге создаётся папка `.git`, в которой содержатся служебные файлы, необходимые для Git.

Отображение статуса репозитория в приглашении

Это дополнительный функционал, который не требуется для работы с Git, но очень помогает в этом. При работе с Git очень удобно, когда можно сразу определить, находитесь Вы в обычном каталоге или в репозитории Git. Кроме того, было бы хорошо понимать статус текущего репозитория. Для этого нужно установить специальную [утилиту](#), которая будет показывать статус репозитория. Для установки утилиты, надо скопировать её репозиторий в домашний каталог пользователя, под которым Вы работаете:

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

А затем добавить в конец файла .bashrc такие строки:

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

Для того, чтобы изменения применились, перезапустить bash:

```
exec bash
```

В моей конфигурации приглашение командной строки разнесено на несколько строк, поэтому у Вас оно будет отличаться. Главное, обратите внимание на то, что появляется дополнительная информация при переходе в репозиторий.

Теперь, если Вы находитесь в обычном каталоге, приглашение выглядит так:

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

Если же перейти в репозиторий Git:

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


Работа с Git

Для управления Git, используются различные команды, смысл которых поясняется далее.

git status

При работе с Git, важно понимать текущий статус репозитория. Для этого в Git есть команда git status:

```
[~/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 сообщает, что мы находимся в ветке master (эта ветка создаётся сама и используется по умолчанию), и что ему нечего добавлять в коммит. Кроме этого, Git предлагает создать или скопировать файлы и после этого воспользоваться командой git add, чтобы Git начал за ними следить.

Создание файла README и добавление в него строки "test":

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

После этого, приглашение выглядит таким образом:

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

В приглашении показано, что есть два файла, за которыми Git ещё не следит:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1...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)
```

Два файла получилось из-за того, что у меня настроены undo-файлы для Vim. Это специальные файлы, благодаря которым можно отменять изменения не только в текущей сессии файла, но и прошлые. Обратите внимание, что Git сообщает, что есть файлы, за которыми он не следит и подсказывает, какой командой это сделать.

Файл .gitignore

Undo-файл .README.un~ – служебный файл, который не нужно добавлять в репозиторий. В Git есть возможность указать, какие файлы или каталоги нужно игнорировать. Для этого нужно создать соответствующие шаблоны в файле .gitignore в каталоге репозитория.

Для того, чтобы Git игнорировал undo-файлы Vim, можно добавить, например, такую строку в файл .gitignore:

```
* .un~
```

Это значит, что Git должен игнорировать все файлы, которые заканчиваются на ".un~".

После этого, git status показывает:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1...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)
```

Обратите внимание, что теперь в выводе нет файла .README.un~. Как только в репозиторий был добавлен файл .gitignore, файлы, которые указаны в нём, стали игнорироваться.

git add

Для того, чтобы Git начал следить за файлами, используется команда git add.

Можно указать что надо следить за конкретным файлом:

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

Или за всеми файлами:

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

Вывод git status:

```
[~/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
```

Теперь файлы находятся в секции под названием "Changes to be committed".

git commit

После того, как все нужные файлы были добавлены в staging, можно закоммитить изменения. Staging это совокупность файлов, которые будут добавлены в следующий коммит. У команды git commit есть только один обязательный параметр – флаг "-m". Он позволяет указать сообщение для этого коммита.

```
[~/tools/first_repo]  
vagrant@jessie-i386: [master L1 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
```

После этого git status отображает:

```
[~/tools/first_repo]  
vagrant@jessie-i386: [master L1 ✓]  
13:47 $ git status  
On branch master  
nothing to commit, working directory clean
```

Фраза "nothing to commit, working directory clean" обозначает, что нет изменений, которые нужно добавить в Git или закоммитить.

Дополнительные возможности

git diff

Команда git diff позволяет просмотреть разницу между различными состояниями.

Например, в репозитории внесены изменения в файл README и .gitignore. Команда git status показывает, что оба файла изменены:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master 11+ 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")
```

Команда git diff показывает, какие изменения были внесены с последнего коммита:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master 11+ 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
```

Если добавить изменения в файлах и ещё раз выполнить команду git diff, она ничего не покажет:

```
[~/tools/first_repo]  
vagrant@jessie-i386: [master 1|+ 2]  
13:54 $ git add .  
  
[~/tools/first_repo]  
vagrant@jessie-i386: [master 1| 2]  
13:57 $ git diff
```

Чтобы показать отличия между staging и последним коммитом, надо добавить параметр --staged:

```
[~/tools/first_repo]  
vagrant@jessie-i386: [master 1| 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
```

Закоммитим изменения:

```
[~/tools/first_repo]  
vagrant@jessie-i386: [master 1| 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

Команда git log показывает, когда были выполнены последние изменения:

```
[~/tools/first_repo]
vagrant@jessie-i386: [master L1✓]
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
```

По умолчанию команда показывает все коммиты, начиная с самого свежего.

С помощью дополнительных параметров можно не только посмотреть информацию о коммитах, но и какие изменения были внесены.

Флаг -r позволяет отобразить отличия, которые были внесены каждым коммитом:

```
[~/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
```

Более короткий вариант вывода можно вывести с флагом `--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 ef8473307e0a119496ef154e0bcff703b1f8a71
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(+)
```

Аутентификация на GitHub

Для того, чтобы начать работать с GitHub, надо на нём [зарегистрироваться](#).

Для безопасной работы с GitHub лучше использовать аутентификацию по ключам SSH.

Эта же [инструкция на GitHub](#)

Генерация нового SSH ключа (используйте email, который привязан к GitHub):

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

На всех вопросах достаточно нажать enter (более безопасно использовать ключ с passphrase, но можно и без, если нажать enter, при вопросе).

Запуск ssh-agent:

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

Добавить ключ в ssh-agent:

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

Добавление SSH ключа на GitHub

Для добавления ключа надо его скопировать. Например, таким образом можно отобразить ключ для копирования:

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

После копирования надо перейти на GitHub.

Находясь на любой странице GitHub, в правом верхнем углу нажмите на картинку вашего профиля и в выпадающем списке выберите Settings. В настройках в левой панели надо выбрать поле "SSH and GPG keys".

После надо нажать "New SSH key" и в поле "Title" написать название ключа (например, "Home debian"), а в поле "Key" вставить содержимое, которое было скопировано из файла `~/.ssh/id_rsa.pub`

Если GitHub запросит пароль - введите пароль своего аккаунта GitHub.

Чтобы проверить, что всё прошло успешно, попробуйте выполнить команду локально:

```
$ ssh -T git@github.com
```

Вывод будет таким:

```
Hi username! You've successfully authenticated, but GitHub does not provide shell access.
```

Теперь Вы готовы работать с Git и GitHub.

Работа с Git и GitHub

Создание репозитория на GitHub

- Залогиниться на [GitHub](#)
- В правом верхнем углу нажать плюс и выбрать "New repository", чтобы создать новый репозиторий
- В открывшемся окне надо придумать название репозиторию
- По желанию можно нажать галочку "Initialize this repository with a README". Это создаст файл README.md, в котором будет находиться только название репозитория

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner Repository name

 natenka /

Great repository names are short and memorable. Need inspiration? How about [crispy-barnacle](#).

Description (optional)

 Public
Anyone can see this repository. You choose who can commit.

 Private
You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾ | Add a license: None ▾ 

Create repository

Копирование репозитория с GitHub

Для локальной работы с созданным репозиторием его нужно скопировать. Для этого используется команда git clone:

```
$ 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.
```

В этой команде нужно изменить:

- имя пользователя pyneng на имя своего пользователя
- имя репозитория "online-2-natasha-samoylenko" на свой репозиторий

В итоге, в текущем каталоге, в котором была выполнена команда git clone, появится каталог с именем, равным имени репозитория. В моем случае - online-2-natasha-samoylenko.

В этом каталоге теперь находится содержимое репозитория на GitHub.

Работа с репозиторием

Предыдущая команда не просто скопировала репозиторий локально, но и настроила соответствующим образом Git:

- был создан каталог .git
- скачаны все данные репозитория
- скачаны все изменения, которые были в репозитории
- репозиторий на GitHub настроен как remote для этого репозитория

Теперь локально готов полноценный репозиторий Git, в котором Вы можете работать.

Обычно последовательность работы будет такой:

- перед началом работы синхронизировать локальное содержимое с GitHub командой git pull (синхронизация из GitHub в локальный репозиторий)
- редактируете какие-то файлы репозитория
- добавляете их в staging командой git add
- делаете commit с помощью git commit
- когда Вы готовы закачать локальные изменения на GitHub, делаете git push

При работе с заданиями с работы и дома, надо не забывать первый и последний шаг:

- первый шаг - обновляет локальный репозиторий
- последний шаг - загружает изменения на GitHub

Синхронизация из GitHub в локальный репозиторий

Все команды выполняются внутри каталога репозитория (в примере выше - online-2-natasha-samoylenko)

Команда git pull:

```
$ git pull
```

Если содержимое локального репозитория одинаково с удаленным репозиторием на GitHub, вывод будет таким:

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

Если что-то было изменено, вывод будет примерно таким:

```
$ 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(+)
```

Добавление новых файлов или изменений в существующих файлах

Если необходимо добавить конкретный файл (в данном случае - README.md):

```
$ git add README.md
```

Добавление всех новых файлов или изменений в существующих:

```
$ git add .
```

Commit

При выполнении commit обязательно надо указать сообщение. Будет лучше, если сообщение будет со смыслом, а не просто "update" или подобное:

```
$ git commit -m "Сделал задания 4.1-4.3"
```

Push на GitHub

Для загрузки всех локальных изменений на GitHub используется git push:

```
$ 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
```

Перед выполнением git push можно выполнить команду `$ git log -p origin/master..` - она покажет, какие изменения Вы собираетесь добавлять в свой репозиторий на GitHub.

Работа с репозиторием заданий и примеров

Все примеры и задания книги выложены в отдельном [репозитории](#).

Копирование репозитория с GitHub

Примеры и задания периодически обновляются. Поэтому будет удобней скопировать локально этот репозиторий и обновлять его, когда были внесены какие-то изменения.

Для копирования репозитория с GitHub выполните команду 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.
```

Обновление локальной копии репозитория

При необходимости обновить локальную версию репозитория, чтобы синхронизировать её с версией на GitHub, надо выполнить git pull внутри созданного каталога pyneng-examples-exercises.

Если обновлений не было, вывод будет таким:

```
$ cd pyneng-examples-exercises/
$ git pull
Already up-to-date.
```

Если обновления были, вывод будет примерно таким:

```
$ 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(-)
```

Обратите внимание на информацию о том, что изменился только файл README.md.

Просмотр изменений

Если Вы хотите посмотреть, какие именно изменения были внесены, можно воспользоваться командой 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 @@
 * применить ACL к интерфейсу

ACL должен быть таким:
+
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
-
+
Проверьте работу playbook на маршрутизаторе R1.
```

В этой команде флаг `-p` указывает, что надо отобразить diff изменений, а не только сообщение commit, а `-1` указывает, что надо показать только один commit (самый свежий).

Посмотреть, какие изменения будут синхронизированы

Прошлый вариант опирается на количество коммитов. Но это не всегда удобно. До выполнения команды `git pull` можно посмотреть, какие изменения были выполнены с момента последней синхронизации. Для этого используется такая команда:

```
$ git log -p ..origin/master
commit 4c1821030d20b3682b67caf362fd777d098d9126
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 @@
+
+Тут находятся PDF версии руководств по настройке инструментов, которые используются на курсе.
```

В данном случае изменения были только в одном файле.

Эта команда будет очень полезна для того, чтобы посмотреть, например, какие изменения были внесены в формулировку заданий и каких именно заданий. Так будет легче сориентироваться, касается ли это заданий, которые Вы уже сделали, и надо ли что-то изменить.

Если изменения были в тех заданиях, которые Вы ещё не делали, этот вывод подскажет, какие файлы нужно скопировать с репозитория курса в ваш личный репозиторий (а может и весь раздел, если Вы ещё не делали задания из этого раздела).

`..origin/master` в этой команде означает показать все коммиты, которые есть в `origin/master` (в данном случае, это GitHub), но которых нет в вашей локальной копии репозитория.

Дополнительные материалы

Документация:

- [Informative git prompt for bash and fish](#)
- [Authenticating to GitHub](#)
- [Connecting to GitHub with SSH](#)

Git/GitHub:

- [GitHowTo](#) - интерактивный howto на русском
- [git/github guide. a minimal tutorial](#) - минимально необходимые знания для работы с Git и GitHub
- [Pro Git book. Эта же книга на русском](#)
- [Основы Git \(Hexlet\)](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 2.1

Клонировать [репозиторий с заданиями и примерами](#).

Попробовать обновить его (git pull). Должно отобразиться сообщение "Already up-to-date.".

В этом репозитории Вы не можете вносить изменения, поэтому он будет использоваться только для получения заданий и примеров. Для работы с ними, необходимо создать собственный репозиторий и скопировать туда задания.

Задание 2.2

В этом задании необходимо:

- создать свой репозиторий для выполнения заданий на GitHub
- клонировать ее на свою виртуалку

В репозитории можно придумать любую структуру каталогов.

Так как задания периодически обновляются, лучше выбрать такой подход к работе:

- обновляете [репозиторий с заданиями и примерами \(rulpeng-examples-exercises\)](#)
- копируете задания одного раздела из репозитория rulpeng-examples-exercises в свой репозиторий
- выполняете задания

- при переходе к следующему разделу, повторяете все с начала

Начало работы с Python

В этом разделе рассматриваются:

- синтаксис Python
- работа в интерактивном режиме
- переменные в Python

Синтаксис Python

Первое, что, как правило, бросается в глаза, если говорить о синтаксисе Python - это то, что отступы имеют значение:

- они определяют, какие выражения попадают в блок кода
- когда блок кода заканчивается

Пример кода Python:

```
a = 10
b = 5

if a > b:
    print("A больше B")
    print(a - b)
else:
    print("B больше или равно A")
    print(b - a)

print("The End")

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

Обратите внимание, что тут код показан для демонстрации синтаксиса. В следующих разделах рассматриваются [типы данных](#) Python и [создание скриптов](#).

Несмотря на то, что еще не рассматривалась конструкция if/else, всё должно быть понятно.

Python понимает, какие строки относятся к if на основе отступов. Выполнение части `if a > b` заканчивается, когда встречается строка с тем же отступом, что и сама строка `if a > b`.

Аналогично с блоком else.

Вторая особенность Python: после некоторых выражений должно идти двоеточие (например, после if a > b или после else).

Несколько правил и рекомендаций:

- В качестве отступов могут использоваться Tab или пробелы.

- лучше использовать пробелы, а точнее, настроить редактор так, чтобы Tab был равен 4 пробелам (тогда при использовании Tab будут ставиться 4 пробела)
- Количество пробелов должно быть одинаковым в одном блоке.
 - лучше, чтобы количество пробелов было одинаковым во всем коде
 - популярный вариант - использовать 2-4 пробела (в курсе используются 4 пробела)

Для того, чтобы проблем с отступами не было, надо сразу настроить редактор таким образом, чтобы отступы делались автоматически.

Еще одна особенность приведенного примера кода: пустые строки. Таким образом код форматируется, чтобы его было проще читать.

Остальные особенности синтаксиса будут показаны в процессе знакомства с структурами данных в Python.

Комментарии

При написании кода часто нужно оставить комментарий, например, чтобы описать особенности работы кода.

Комментарии в Python могут быть односторонними:

```
#Очень важный комментарий
a = 10
b = 5 #Очень нужный комментарий
```

Односторонние комментарии начинаются со знака #.

Обратите внимание, что комментарий может быть и в строке, где находится код, и сам по себе.

При необходимости написать несколько строк с комментариями, чтобы не ставить перед каждой решетку, можно сделать многострочный комментарий:

```
"""Очень важный
и длинный комментарий
"""
a = 10
b = 5
```

Многострочный комментарий может использовать три двойные или три одинарные кавычки.

Комментарии могут использоваться как для того, чтобы комментировать, что происходит в коде, так и для того, чтобы закомментировать временно какое-то выражение.

Интерпретатор Python. iPython

Интерпретатор позволяет получать моментальный отклик на выполненные действия.

Можно сказать, что интерпретатор работает как командная строка сетевых устройств: каждая команда будет выполняться сразу после нажатия enter (по крайней мере, похоже на cisco).

Но для интерпретатора Python есть исключение: более сложные объекты (например, циклы, функции) выполняются только после нажатия enter два раза.

В предыдущем разделе для проверки установки Python вызвался стандартный интерпретатор.

Но, кроме него, в Python есть усовершенствованный интерпретатор iPython ([документация iPython](#)).

iPython позволяет намного больше, чем стандартный интерпретатор, который вызывается по команде python.

Несколько примеров (возможности ipython намного шире):

- автодополнение команд по Tab или подсказка, если вариантов команд несколько
- более структурированный и понятный вывод команд
- автоматические отступы в циклах и других объектах
- история выполнения команд
 - по ней можно передвигаться
 - или посмотреть "волшебной" командой %history

Установить iPython можно с помощью pip (установка в виртуальном окружении):

```
pip install ipython
```

После этого зайди в ipython можно таким образом:

```
$ ipython
Python 3.6.3 (default, Oct  9 2017, 11:46:27)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Для выхода напишите quit

Далее как интерпретатор будет использоваться iPython.

Для знакомства с интерпретатором можно попробовать его использовать как калькулятор:

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

```
Out[1]: 3
```

```
In [2]: 22*45
```

```
Out[2]: 990
```

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

```
Out[3]: 8
```

В iPython ввод и вывод подписаны:

- In - это то, что написал пользователь
- Out - это вывод команды (если он есть)
- Числа после In и Out - это нумерация выполненных команд в текущей сессии iPython

Пример вывода строки:

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

```
Hello!
```

Когда в интерпретаторе создается, например, цикл, то внутри цикла приглашение меняется на Для выполнения цикла и выхода из этого подрежима необходимо дважды нажать Enter:

```
In [5]: for i in range(5):
```

```
...:     print(i)
```

```
...:
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

help

В ipython есть возможность посмотреть help по какому-то объекту, функции или методу:

```
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

    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.
```

Второй вариант:

```
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

Функция `print` позволяет вывести информацию на стандартный поток вывода.

Если необходимо вывести строку, то ее нужно обязательно заключить в кавычки (двойные или одинарные). Если же нужно вывести, например, результат вычисления или просто число, то кавычки не нужны:

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

```
Hello!
```

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

```
25
```

Если нужно вывести несколько значений, можно перечислить их через запятую:

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

```
5 10 15 20
```

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

```
one two three
```

Подробнее о [функции `print`](#)

По умолчанию в конце выражения будет перевод строки. Если необходимо, чтобы после вывода выражения не было перевода строки, надо указать дополнительный аргумент `end`.

По умолчанию он равен `\n`, поэтому к строке или строкам в `print` добавляется перевод строки.

Например, такое выражение выведет строки 'one' и 'two' в разных строках:

```
In [10]: print('one'), print('two')
```

```
one
```

```
two
```

```
Out[10]: (None, None)
```

Но если в первой функции `print` указать параметр `end` равным пустой строке, результат будет таким:

```
In [11]: print('one', end=''), print('two')
```

```
onetwo
```

```
Out[11]: (None, None)
```

dir

Команда `dir()` может использоваться для того, чтобы посмотреть, какие атрибуты и методы есть у объекта.

Например, для числа вывод будет таким (обратите внимание на различные методы, которые позволяют делать арифметические операции):

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

Для строки:

```
In [11]: dir('hello')
Out[11]:
['__add__',
 '__class__',
 '__contains__',
 ...
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

Если выполнить команду без передачи значения, то она показывает существующие методы, атрибуты и переменные, определенные в текущей сессии интерпретатора:

```
In [12]: dir()
Out[12]:
[ '__builtin__',
  '__builtins__',
  '__doc__',
  '__name__',
  '__dh',
  ...
  '__oh',
  '__sh',
  'exit',
  'get_ipython',
  'i',
  'quit']
```

Пример после создания переменной **a** и функции **test**:

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

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

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

Magic commands

В IPython есть специальные команды, которые упрощают работу с интерпретатором. Все они начинаются на %.

%history

Например, команда `%history` позволяет просмотреть историю текущей сессии:

```
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
```

Таким образом можно скопировать какой-то блок кода.

%cpaste

Еще одна очень полезная волшебная команда `%cpaste`

При вставке кода с отступами в IPython из-за автоматических отступов самого IPython начинает сдвигаться код:

```
In [1]: a = 10

In [2]: b = 5

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

In [4]: %hist
a = 10
b = 5
if a > b:
    print("A is bigger")
else:
    print("A is less or equal")
%hist

In [5]: if a > b:
...:     print("A is bigger")
...: else:
...:     print("A is less or equal")
...:
File "<ipython-input-8-4d18ff094f5c>", line 3
    else:
        ^
IndentationError: unindent does not match any outer indentation level
If you want to paste code into IPython, try the %paste and %cpaste magic functions.
```

Обратите внимание на последнюю строку. IPython подсказывает, какой командой воспользоваться, чтобы корректно вставить такой код.

Команды `%paste` и `%cpaste` работают немного по-разному.

%cpaste (после того, как все строки скопированы, надо завершить работу команды, набрав '--'):

```
In [9]: %cpaste
Pasting code; enter '---' alone on the line to stop or use Ctrl-D.
:if a > b:
:    print("A is bigger")
:else:
:    print("A is less or equal")
:---
A is bigger
```

%paste (требует установленного Tkinter):

```
In [10]: %paste
if a > b:
    print("A is bigger")
else:
    print("A is less or equal")

## -- End pasted text --
A is bigger
```

Подробнее об IPython можно почитать в [документация IPython](#).

Коротко информацию можно посмотреть в самом IPython командой %quickref:

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.
%%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 system 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.
```

Переменные

Переменные в Python:

- не требуют объявления типа переменной (Python - язык с динамической типизацией)
- являются ссылками на область памяти

Правила именования переменных:

- имя переменной может состоять только из букв, цифр и знака подчеркивания
- имя не может начинаться с цифры
- имя не может содержать специальных символов @, \$, %

Создавать переменные в Python очень просто:

```
In [1]: a = 3  
  
In [2]: b = 'Hello'  
  
In [3]: c, d = 9, 'Test'  
  
In [4]: print(a,b,c,d)  
3 Hello 9 Test
```

Обратите внимание, что в Python не нужно указывать, что a это число, а b это строка.

Переменные являются ссылками на область памяти. Это легко продемонстрировать с помощью функции `id()`, которая показывает идентификатор объекта:

```
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
```

В этом примере видно, что все три имени ссылаются на один и тот же идентификатор. То есть, это один и тот же объект, на который указывают три ссылки a, b и c.

С числами у Python есть ещё одна особенность, которая может немного сбить. Числа от -5 до 256 заранее созданы и хранятся в массиве (списке). Поэтому при создании числа из этого диапазона фактически создается ссылка на число в созданном массиве.

Эта особенность характерна именно для реализации CPython, которая рассматривается в курсе.

Это можно проверить таким образом:

```
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
```

Обратите внимание, что у `a`, `b` и числа `3` одинаковые идентификаторы. Все они просто являются ссылками на существующее число в списке.

Но если сделать то же самое с числом больше 256:

```
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
```

Идентификаторы у всех разные.

При этом, если сделать присваивание такого вида:

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

Идентификаторы будут у всех одинаковые:

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

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

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

Так как в таком варианте `a`, `b` и `c` просто ссылаются на один и тот же объект.

Имена переменных

Имена переменных не должны пересекаться с операторами и названиями модулей или других зарезервированных значений.

В Python есть рекомендации по именованию функций, классов и переменных:

- имена переменных обычно пишутся полностью большими или маленькими буквами
 - `DB_NAME`
 - `db_name`
- имена функций задаются маленькими буквами, с подчеркиваниями между словами
 - `get_names`
- имена классов задаются словами с заглавными буквами, без пробелов
 - `CiscoSwitch`

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 3.1

Выполните установку IPython в виртуальном окружении или глобально в системе, если виртуальные окружения не используются.

После установки, по команде ipython должен открываться интерпретатор IPython (вывод может незначительно отличаться):

```
$ ipython
Python 3.6.3 (default, Oct  9 2017, 11:46:27)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
```

In [1]:

Типы данных в Python

В Python есть несколько стандартных типов данных:

- Numbers (числа)
- Strings (строки)
- Lists (списки)
- Dictionaries (словари)
- Tuples (кортежи)
- Sets (множества)
- Boolean

Эти типы данных можно, в свою очередь, классифицировать по некоторым признакам:

- Изменяемые:
 - Списки
 - Словари
 - Множества
- Неизменяемые
 - Числа
 - Строки
 - Кортежи
- Упорядоченные:
 - Списки
 - Кортежи
 - Строки
- Неупорядоченные:
 - Словари
 - Множества

Числа

С числами можно выполнять различные математические операции.

```
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
```

Деление int и float:

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

```
Out[5]: 3.333333333333335
```

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

```
Out[6]: 3.333333333333335
```

С помощью функции round можно округлять числа до нужного количества знаков:

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

```
Out[9]: 3.33
```

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

```
Out[10]: 3.3333
```

Остаток от деления:

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

```
Out[11]: 1
```

Операторы сравнения

```
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
```

Функция `int()` позволяет выполнять конвертацию в тип `int`. Во втором аргументе можно указывать систему счисления:

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

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

```
Out[19]: 11
```

Если указать, что строку `a` надо воспринимать как двоичное число, то результат будет таким:

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

```
Out[20]: 3
```

Конвертация в `int` типа `float`:

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

```
Out[21]: 3
```

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

```
Out[22]: 3
```

Функция `bin` позволяет получить двоичное представление числа (обратите внимание, что результат - строка):

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

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

Аналогично, функция `hex()` позволяет получить шестнадцатеричное значение:

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

И, конечно же, можно делать несколько преобразований одновременно:

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

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

Для более сложных математических функций в Python есть модуль **math**:

```
In [28]: import math

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

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

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

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

Строки (Strings)

Строка в Python - это последовательность символов, заключенная в кавычки. Строки - это неизменяемый упорядоченный тип данных.

Примеры строк:

```
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 profi
le DMVPN\n'

In [13]: print(tunnel)

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

Строки можно суммировать. Тогда они объединяются в одну строку:

```
In [14]: intf = 'interface'
In [15]: tun = 'Tunnel0'

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

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

Строчку можно умножать на число. В этом случае, строка повторяется указанное количество раз:

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

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

То, что строки являются упорядоченным типом данных, позволяет обращаться к символам в строке по номеру, начиная с нуля:

```
In [20]: string1 = 'interface FastEthernet1/0'
In [21]: string1[0]
Out[21]: 'i'
```

Нумерация всех символов в строке идет с нуля. Но, если нужно обратиться к какому-то по счету символу, начиная с конца, то можно указывать отрицательные значения (на этот раз с единицами).

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

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

Кроме обращения к конкретному символу, можно делать срезы строки, указав диапазон номеров (срез выполняется по второе число, не включая его):

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

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

Если не указывается второе число, то срез будет до конца строки:

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

Срезать три последних символа строки:

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

Строка в обратном порядке:

```
In [28]: a = '0123456789'  
  
In [29]: a[::-]  
Out[29]: '0123456789'  
  
In [30]: a[::-1]  
Out[30]: '9876543210'
```

Записи `a[::-]` и `a[:]` дают одинаковый результат, но двойное двоеточие позволяет указывать, что надо брать не каждый элемент, а, например, каждый второй.

Например, таким образом можно получить все четные числа строки a:

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

Так можно получить нечетные:

```
In [32]: a[1::2]  
Out[32]: '13579'
```

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 (то есть, 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.

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
```

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
```

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`.

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
```

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'
```

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


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

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

По умолчанию, метод `strip()` убирает whitespace 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()`.

split()

Метод `split()` разбивает строку на части, используя как разделитель какой-то символ (или символы). По умолчанию, в качестве разделителя используются пробелы. Но в скобках можно указать любой разделитель.

В результате, строка будет разбита на части по указанному разделителю и представлена в виде частей, которые содержатся в списке:

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

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

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

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

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

В строке `string1` был символ пробела в начале и символ перевода строки в конце. В строке номер 54 с помощью метода `strip()` эти символы удаляются.

Метод `strip()` возвращает строку, которая обрабатывается методом `split()` и разделяет строку на части, используя пробел как разделитель. Итоговая строка присваивается переменной `commands`.

Используя тот же способ, что и со строками, к последнему объекту в списке `vlans` применяется метод `split()`. Но на этот раз внутри скобок указывается другой разделитель - запятая. В итоге, в списке `vlans` находятся номера VLAN.

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

```
In [58]: sh_ip_int_br = "FastEthernet0/0      15.0.15.1    YES manual up      up"  
  
In [59]: sh_ip_int_br.split()  
Out[59]: ['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 [60]: sh_ip_int_br.split(' ')  
Out[60]: ['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 два варианта форматирования строк:

- с оператором `%` (более старый вариант)
- методом `format()` (новый вариант)

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('{0}'.format('10.1.1.1'))
10.1.1.1

In [4]: print('{0}'.format(100))
100

In [5]: print('{0}'.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 оператором %

Пример использования оператора %:

```
In [2]: "interface FastEthernet0/%s" % '1'
Out[2]: 'interface FastEthernet0/1'
```

В старом синтаксисе форматирования строк используются такие обозначения:

- `%s` - строка или любой другой объект в котором есть строковое представление
- `%d` - integer
- `%f` - float

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("%15s %15s %15s" % (vlan, mac, intf))
        100    aabb.cc80.7000          Gi0/1
```

Выравнивание по левой стороне:

```
In [6]: print("%-15s %-15s %-15s" % (vlan, mac, intf))
100            aabb.cc80.7000  Gi0/1
```

С помощью форматирования строк можно также влиять на отображение чисел.

Например, можно указать, сколько цифр после запятой выводить:

```
In [8]: print("%.3f" % (10.0/3))  
3.333
```

У форматирования строк есть ещё много возможностей. Хорошие примеры и объяснения двух вариантов форматирования строк можно найти [тут](#).

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+ +'
        '(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 "{}" '\n....: 'возникла такая ошибка "{}".\n'....: 'Исключить эту команду из списка? [y/n]')\n\nIn [8]: message\nOut[8]: 'При выполнении команды "{}" возникла такая ошибка "{}".\nИсключить эту команду из списка? [y/n]'
```

Список (List)

Список - это изменяемый упорядоченный тип данных.

Список в Python - это последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки.

Примеры списков:

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

Так как список - это упорядоченный тип данных, то, как и в строках, в списках можно обращаться к элементу по номеру, делать срезы:

```
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]
```

Перевернуть список наоборот можно и с помощью метода `reverse()`:

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

In [11]: vlans.reverse()

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

Так как списки изменяемые, элементы списка можно менять:

```
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']
```

Можно создавать и список списков. И, как и в обычном списке, можно обращаться к элементам во вложенных списках:

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

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

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

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

Полезные методы для работы со списками

Список - это изменяемый тип данных, поэтому очень важно обращать внимание на то, что большинство методов для работы со списками меняют список на месте, при этом ничего не возвращая.

join()

Метод **join()** собирает список строк в одну строку с разделителем, который указан перед join:

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

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

Метод join на самом деле относится к строкам, но так как значение ему надо передавать как список, он рассматривается тут.

append()

Метод **append()** добавляет в конец списка указанный элемент:

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

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

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

Метод append меняет список на месте и ничего не возвращает.

extend()

Если нужно объединить два списка, то можно использовать два способа: метод **extend()** и операцию сложения.

У этих способов есть важное отличие - extend меняет список, к которому применен метод, а суммирование возвращает новый список, который состоит из двух.

Метод extend:

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

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

In [23]: vlans.extend(vlans2)

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

Суммирование списков:

```
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']
```

Обратите внимание на то, что при суммировании списков в ipython появилась строка Out. Это означает, что результат суммирования можно присвоить в переменную:

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

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

pop()

Метод **pop()** удаляет элемент, который соответствует указанному номеру. Но, что важно, при этом метод возвращает этот элемент:

```
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']
```

Без указания номера удаляется последний элемент списка.

remove()

Метод **remove()** удаляет указанный элемент.

`remove()` не возвращает удаленный элемент:

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

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

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

В методе `remove` надо указывать сам элемент, который надо удалить, а не его номер в списке. Если указать номер элемента, возникнет ошибка:

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

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

index()

Метод `index()` используется для того, чтобы проверить, под каким номером в списке хранится элемент:

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

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

insert()

Метод `insert()` позволяет вставить элемент на определенное место в списке:

```
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()

Метод `sort` сортирует список на месте:

Список (List)

```
In [40]: vlans = [1, 50, 10, 15]
In [41]: vlans.sort()
In [42]: vlans
Out[42]: [1, 10, 15, 50]
```

Варианты создания списка

Создание списка с помощью литерала:

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

Литерал - это выражение, которое создает объект.

Создание списка с помощью функции `list()`:

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

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

Генераторы списков:

```
In [4]: list2 = ['FastEthernet0/'+ str(i) for i in range(10)]

In [5]: list2
Out[6]:
['FastEthernet0/0',
 'FastEthernet0/1',
 'FastEthernet0/2',
 'FastEthernet0/3',
 'FastEthernet0/4',
 'FastEthernet0/5',
 'FastEthernet0/6',
 'FastEthernet0/7',
 'FastEthernet0/8',
 'FastEthernet0/9']
```

Генераторы списков требуют понимания работы цикла `for` и даже после этого, могут быть немного необычны.

После 6 раздела, можно вернуться к этой теме и почитать о них подробнее в разделе [Примеры использования основ](#).

Словарь (Dictionary)

Словари - это изменяемый неупорядоченный тип данных

В модуле [collections](#) доступны упорядоченные объекты, внешне идентичные словарям [OrderedDict](#).

Словарь (ассоциативный массив, хеш-таблица):

- данные в словаре - это пары `ключ: значение`
- доступ к значениям осуществляется по ключу, а не по номеру, как в списках
- словари не упорядочены, поэтому не стоит полагаться на порядок элементов словаря
- так как словари изменяемы, то элементы словаря можно менять, добавлять, удалять
- ключ должен быть объектом неизменяемого типа:
 - число
 - строка
 - кортеж
- значение может быть данными любого типа

Пример словаря:

```
london = {'name': 'London1', 'location': 'London Str', 'vendor': 'Cisco', 'model': '44  
51', 'ios': '15.4'}
```

Можно записывать и так:

```
london = {  
    'id': 1,  
    'name':'London',  
    'it_vlan':320,  
    'user_vlan':1010,  
    'mngmt_vlan':99,  
    'to_name': None,  
    'to_id': None,  
    'port':'G1/0/11'  
}
```

Для того, чтобы получить значение из словаря, надо обратиться по ключу, таким же образом, как это было в списках, только вместо номера будет использоваться ключ:

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

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

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

Аналогичным образом можно добавить новую пару ключ:значение:

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

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

В словаре в качестве значения можно использовать словарь:

```
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'
    }
}
```

Получить значения из вложенного словаря можно так:

```
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'
```

Полезные методы для работы со словарями

clear()

Метод **clear()** позволяет очистить словарь:

```
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()

Метод **copy()** позволяет создать полную копию словаря.

Если указать, что один словарь равен другому:

```
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'
```

В этом случае london2 это еще одно имя, которое ссылается на словарь. И при изменениях словаря london меняется и словарь london2, так как это ссылки на один и тот же объект.

Поэтому, если нужно сделать копию словаря, надо использовать метод **copy()**:

```
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()

Если при обращении к словарю указывается ключ, которого нет в словаре, возникает ошибка:

```
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'
```

Метод **get()** запрашивает ключ и, если его нет, вместо ошибки возвращает `None`.

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

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

Метод **get()** позволяет также указывать другое значение вместо `None`:

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

setdefault()

Метод **setdefault()** ищет ключ и, если его нет, вместо ошибки создает ключ со значением `None`.

```
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]: {'ios': None, 'location': 'London Str', 'name': 'London1', 'vendor': 'Cisco'}
```

Но, если ключ есть, `setdefault` возвращает значение, которое ему соответствует:

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

```
Out[25]: 'London1'
```

Второй аргумент позволяет указать, какое значение должно соответствовать ключу:

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

```
In [27]: print(model)
```

```
Cisco3580
```

```
In [28]: london
```

```
Out[28]:
```

```
{'ios': None,
```

```
'model': 'Cisco3580',
```

```
'location': 'London Str',
```

```
'name': 'London1',
```

```
'vendor': 'Cisco'}
```

Метод `setdefault` заменяет такую конструкцию:

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

keys(), values(), items()

Методы **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')])
```

Все три метода возвращают специальные объекты view, которые отображают ключи, значения и пары ключ-значение словаря соответственно.

Очень важная особенность view заключается в том, что они меняются вместе с изменением словаря. И фактически они лишь дают способ посмотреть на соответствующие объекты, но не создают их копию.

На примере метода 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'])
```

Сейчас переменной keys соответствует view dict_keys, в котором три ключа: name, location и vendor.

Но, если мы добавим в словарь еще одну пару ключ-значение, объект keys тоже поменяется:

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

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

Если нужно получить обычный список ключей, который не будет меняться с изменениями словаря, достаточно конвертировать view в список:

```
In [33]: list_keys = list(london.keys())
In [34]: list_keys
Out[34]: ['name', 'location', 'vendor', 'ip']
```

del

Удалить ключ и значение:

```
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

Метод update позволяет добавлять в словарь содержимое другого словаря:

```
In [38]: r1 = {'name': 'London1', 'location': 'London Str'}
In [39]: r1.update({'vendor': 'Cisco', 'ios':'15.2'})
In [40]: r1
Out[40]: {'ios': '15.2', 'location': 'London Str', 'name': 'London1', 'vendor': 'Cisco'}
```

Аналогичным образом можно обновить значения:

```
In [41]: r1.update({'name': 'london-r1', 'ios':'15.4'})
In [42]: r1
Out[42]:
{'ios': '15.4',
 'location': 'London Str',
 'name': 'london-r1',
 'vendor': 'Cisco'}
```

Варианты создания словаря

Литерал

Словарь можно создать с помощью литерала:

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

dict

Конструктор **dict** позволяет создавать словарь несколькими способами.

Если в роли ключей используются строки, можно использовать такой вариант создания словаря:

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

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

Второй вариант создания словаря с помощью **dict**:

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

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

dict.fromkeys

В ситуации, когда надо создать словарь с известными ключами, но, пока что, пустыми значениями (или одинаковыми значениями), очень удобен метод **fromkeys()**:

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

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

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

По умолчанию, метод `fromkeys` подставляет значение `None`. Но можно указывать и свой вариант значения:

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

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

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

Этот вариант создания словаря подходит не для всех случаев. Например, при использовании изменяемого типа данных в значении, будет создана ссылка на один и тот же объект:

```
In [11]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [12]: routers = dict.fromkeys(router_models, [])

In [13]: routers
Out[13]: {'ASR9002': [], 'ISR2811': [], 'ISR2911': [], 'ISR2921': []}

In [14]: routers['ASR9002'].append('london_r1')

In [15]: routers
Out[15]:
{'ASR9002': ['london_r1'],
 'ISR2811': ['london_r1'],
 'ISR2911': ['london_r1'],
 'ISR2921': ['london_r1']}
```

В данном случае каждый ключ ссылается на один и тот же список. Поэтому, при добавлении значения в один из списков обновляются и остальные.

Генератор словаря (dict comprehensions)

И последний метод создания словаря - **генераторы словарей**.

Сгенерируем словарь со списками в значении, как в предыдущем примере:

```
In [16]: router_models = ['ISR2811', 'ISR2911', 'ISR2921', 'ASR9002']

In [17]: routers = {key: [] for key in router_models}

In [18]: routers
Out[18]: {'ASR9002': [], 'ISR2811': [], 'ISR2911': [], 'ISR2921': []}

In [19]: routers['ASR9002'].append('london_r1')

In [20]: routers
Out[20]: {'ASR9002': ['london_r1'], 'ISR2811': [], 'ISR2911': [], 'ISR2921': []}
```

Кортеж (Tuple)

Кортеж - это неизменяемый упорядоченный тип данных.

Кортеж в Python - это последовательность элементов, которые разделены между собой запятой и заключены в скобки.

Грубо говоря, кортеж - это список, который нельзя изменить. То есть, в кортеже есть только права на чтение. Это может быть защитой от случайных изменений.

Создать пустой кортеж:

```
In [1]: tuple1 = tuple()  
  
In [2]: print(tuple1)  
()
```

Кортеж из одного элемента (обратите внимание на запятую):

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

Кортеж из списка:

```
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')
```

К объектам в кортеже можно обращаться, как и к объектам списка, по порядковому номеру:

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

Но так как кортеж неизменяем, присвоить новое значение нельзя:

```
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
```

Множество (Set)

Множество - это изменяемый неупорядоченный тип данных. В множестве всегда содержатся только уникальные элементы.

Множество в Python - это последовательность элементов, которые разделены между собой запятой и заключены в фигурные скобки.

С помощью множества можно легко убрать повторяющиеся элементы:

```
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}
```

Полезные методы для работы с множествами

add()

Метод `add()` добавляет элемент во множество:

```
In [1]: set1 = {10, 20, 30, 40}  
  
In [2]: set1.add(50)  
  
In [3]: set1  
Out[3]: {10, 20, 30, 40, 50}
```

discard()

Метод `discard()` позволяет удалять элементы, не выдавая ошибку, если элемента в множестве нет:

```
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()

Метод `clear()` очищает множество:

```
In [8]: set1 = {10, 20, 30, 40}  
  
In [9]: set1.clear()  
  
In [10]: set1  
Out[10]: set()
```


Операции с множествами

Множества полезны тем, что с ними можно делать различные операции и находить объединение множеств, пересечение и так далее.

Объединение множеств можно получить с помощью метода `union()` или оператора `|`:

```
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}
```

Пересечение множеств можно получить с помощью метода `intersection()` или оператора `&`:

```
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}
```

Варианты создания множества

Нельзя создать пустое множество с помощью литерала (так как в таком случае это будет не множество, а словарь):

```
In [1]: set1 = {}  
  
In [2]: type(set1)  
Out[2]: dict
```

Но пустое множество можно создать таким образом:

```
In [3]: set2 = set()  
  
In [4]: type(set2)  
Out[4]: set
```

Множество из строки:

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

Множество из списка:

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

Генератор множеств:

```
In [7]: set2 = {i + 100 for i in range(10)}  
  
In [8]: set2  
Out[8]: {100, 101, 102, 103, 104, 105, 106, 107, 108, 109}  
  
In [9]: print(set2)  
{100, 101, 102, 103, 104, 105, 106, 107, 108, 109}
```

Преобразование типов

В Python есть несколько полезных встроенных функций, которые позволяют преобразовать данные из одного типа в другой.

int()

`int()` - преобразует строку в int:

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

С помощью функции `int` можно преобразовать и число в двоичной записи в десятичную (двоичная запись должна быть в виде строки)

```
In [2]: int("11111111", 2)
Out[2]: 255
```

bin()

Преобразовать десятичное число в двоичный формат можно с помощью `bin()`:

```
In [3]: bin(10)
Out[3]: '0b1010'

In [4]: bin(255)
Out[4]: '0b11111111'
```

hex()

Аналогичная функция есть и для преобразования в шестнадцатеричный формат:

```
In [5]: hex(10)
Out[5]: '0xa'

In [6]: hex(255)
Out[6]: '0xff'
```

list()

Функция `list()` преобразует аргумент в список:

```
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()

Функция `set()` преобразует аргумент в множество:

```
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'}
```

Эта функция очень полезна, когда нужно получить уникальные элементы в последовательности.

tuple()

Функция `tuple()` преобразует аргумент в кортеж:

```
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')
```

Это может пригодиться в том случае, если нужно получить неизменяемый объект.

str()

Функция `str()` преобразует аргумент в строку:

```
In [16]: str(10)
Out[16]: '10'
```

Например, она пригодится в ситуации, когда есть список VLANов, который надо преобразовать в одну строку, где номера перечислены через запятую.

Если сделать `join` для списка чисел, возникнет ошибка:

```
In [17]: vlans = [10, 20, 30, 40]

In [18]: ','.join(vlans)
-----
TypeError          Traceback (most recent call last)
<ipython-input-39-d705aed3f1b3> in <module>()
----> 1 ','.join(vlans)

TypeError: sequence item 0: expected string, int found
```

Чтобы исправить это, нужно преобразовать числа в строки. Это удобно делать с помощью `list comprehensions`:

```
In [19]: ','.join([ str(vlan) for vlan in vlans ])
Out[19]: '10,20,30,40'
```

Проверка типов

При преобразовании типов данных могут возникнуть ошибки такого рода:

```
In [1]: int('a')
-----
ValueError      Traceback (most recent call last)
<ipython-input-42-b3c3f4515dd4> in <module>()
----> 1 int('a')

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

Ошибка абсолютно логичная. Мы пытаемся преобразовать в десятичный формат строку 'a'.

И если тут пример выглядит, возможно, глупым, тем не менее, когда нужно, например, пройтись по списку строк и преобразовать в числа те из них, которые содержат числа, можно получить такую ошибку.

Чтобы избежать её, было бы хорошо иметь возможность проверить, с чем мы работаем.

isdigit()

В Python такие методы есть. Например, чтобы проверить, состоит ли строка из одних цифр, можно использовать метод `isdigit()`:

```
In [2]: "a".isdigit()
Out[2]: False

In [3]: "a10".isdigit()
Out[3]: False

In [4]: "10".isdigit()
Out[4]: True
```

Пример использования метода:

```
In [5]: vlans = ['10', '20', '30', '40', '100-200']

In [6]: [ int(vlan) for vlan in vlans if vlan.isdigit() ]
Out[6]: [10, 20, 30, 40]
```

isalpha()

Метод `isalpha()` позволяет проверить, состоит ли строка из одних букв:

```
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()

Метод `isalnum()` позволяет проверить, состоит ли строка из букв и цифр:

```
In [11]: "a".isalnum()
Out[11]: True

In [12]: "a10".isalnum()
Out[12]: True
```

type()

Иногда, в зависимости от результата, библиотека или функция может выводить разные типы объектов. Например, если объект один, возвращается строка, если несколько, то возвращается кортеж.

Нам же надо построить ход программы по-разному, в зависимости от того, была ли возвращена строка или кортеж.

В этом может помочь функция `type()`:

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

In [14]: type("string") is str
Out[14]: True
```

Аналогично с кортежем (и другими типами данных):

```
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
```

Дополнительные материалы

Документация:

- [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](#)

Форматирование строк:

- Примеры использования форматирования строк
- [Документация по форматированию строк](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 4.1

Обработать строку NAT таким образом, чтобы в имени интерфейса вместо FastEthernet было GigabitEthernet.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
NAT = "ip nat inside source list ACL interface GigabitEthernet0/1 overload"
```

Задание 4.2

Преобразовать строку MAC из формата XXXX:XXXX:XXXX в формат XXXX.XXX.XXXX

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
MAC = 'AAAA:BBBB:CCCC'
```

Задание 4.3

Получить из строки CONFIG список VLANов вида: ['1', '3', '10', '20', '30', '100']

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
CONFIG = 'switchport trunk allowed vlan 1,3,10,20,30,100'
```

Задание 4.4

Из строк command1 и command2 получить список VLANов, которые есть и в команде command1 и в команде command2.

Для данного примера, результатом должен быть список: [1, 3, 100] Этот список содержит подсказку по типу итоговых данных.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
command1 = 'switchport trunk allowed vlan 1,3,10,20,30,100'  
command2 = 'switchport trunk allowed vlan 1,3,100,200,300'
```

Задание 4.5

Список VLANS это список VLANов, собранных со всех устройств сети, поэтому в списке есть повторяющиеся номера VLAN.

Из списка нужно получить уникальный список VLANов, отсортированный по возрастанию номеров.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.6

Обработать строку ospf_route и вывести информацию на стандартный поток вывода в виде:

Protocol:	OSPF
Prefix:	10.0.24.0/24
AD/Metric:	110/41
Next-Hop:	10.0.13.3
Last update:	3d18h
Outbound Interface:	FastEthernet0/0

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.7

Преобразовать MAC-адрес в двоичную строку (без двоеточий).

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
MAC = 'AAAA:BBBB:CCCC'
```

Задание 4.8

Преобразовать IP-адрес (переменная IP) в двоичный формат и вывести вывод столбцами на стандартный поток вывода, таким образом:

- первой строкой должны идти десятичные значения байтов
- второй строкой двоичные значения

Вывод должен быть упорядочен также, как в примере:

- столбцами
- ширина столбца 10 символов

Пример вывода:

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

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
IP = '192.168.3.1'
```

Создание базовых скриптов

Если говорить в целом, то скрипт - это обычный файл. В этом файле хранится последовательность команд, которые необходимо выполнить.

Начнем с базового скрипта. Выведем на стандартный поток вывода несколько строк.

Для этого надо создать файл access_template.py с таким содержимым:

```
access_template = ['switchport mode access',
                   'switchport access vlan {}',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print('\n'.join(access_template).format(5))
```

Сначала элементы списка объединяются в строку, которая разделена символом `\n`, а в строку подставляется номер VLAN, используя форматирование строк.

После этого надо сохранить файл и перейти в командную строку.

Так выглядит выполнение скрипта:

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

Ставить расширение .py у файла не обязательно.

Но, если Вы используете Windows, то это желательно делать, так как Windows использует расширение файла для определения того, как обрабатывать файл.

В курсе все скрипты, которые будут создаваться, используют расширение .py. Можно сказать, что это "хороший тон" - создавать скрипты Python с таким расширением.

Исполняемый файл

Для того, чтобы файл был исполняемым, и не нужно было каждый раз писать `python` перед вызовом файла, нужно:

- сделать файл исполняемым (для linux)
- в первой строке файла должна находиться строка `#!/usr/bin/env python` или `#!/usr/bin/env python3`, в зависимости от того, какая версия Python используется по умолчанию

Пример файла `access_template_exec.py`:

```
#!/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))
```

После этого:

```
chmod +x access_template_exec.py
```

Теперь можно вызывать файл таким образом:

```
$ ./access_template_exec.py
```

Передача аргументов скрипту (argv)

Очень часто скрипт решает какую-то общую задачу. Например, скрипт обрабатывает как-то файл конфигурации. Конечно, в таком случае, не хочется каждый раз руками в скрипте править название файла.

Гораздо лучше будет передавать имя файла как аргумент скрипта и затем использовать уже указанный файл.

В модуле sys есть очень простой и удобный способ для работы с аргументами - argv.

Посмотрим на пример (файл access_template_argv.py):

```
from sys import argv

interface, vlan = argv[1:]

access_template = ['switchport mode access',
                   'switchport access vlan {}',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

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

Проверяем работу скрипта:

```
$ 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
```

Аргументы, которые были переданы скрипту, подставляются как значения в шаблон.

Тут надо пояснить несколько моментов:

- argv - это список
- все аргументы находятся в списке в виде строк
- argv содержит не только аргументы, которые передали скрипту, но и название самого скрипта

В данном случае в списке argv находятся такие элементы:

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

Сначала идет имя самого скрипта, затем аргументы, в том же порядке.

Ещё один момент, который может быть не очень понятным:

```
interface, vlan = argv[1:]
```

Выражение `argv[1:]` должно быть знакомым. Это срез списка. То есть, в правой стороне остается список с двумя элементами: `['Gi0/7', '4']`.

Разберемся с двойным присваиванием.

В Python есть возможность за раз присвоить значения нескольким переменным.

Простой пример:

```
In [16]: a = 5
In [17]: b = 6
In [18]: c, d = 5, 6
In [19]: c
Out[19]: 5

In [20]: d
Out[20]: 6
```

Если вместо чисел список, как в случае с argv:

```
In [21]: arg = ['Gi0/7', '4']
In [22]: interface, vlan = arg

In [23]: interface
Out[23]: 'Gi0/7'

In [24]: vlan
Out[24]: '4'
```

Ввод информации пользователем

Иногда необходимо получить информацию от пользователя.

Попробуем сделать так, чтобы скрипт задавал вопросы пользователю и затем использовал этот ответ.

Ввод от пользователя может понадобиться, например, для того, чтобы ввести пароль.

Для получения информации от пользователя используется функция `input()` :

```
In [1]: print(input('Твой любимый протокол маршрутизации? '))
Твой любимый протокол маршрутизации? OSPF
OSPF
```

В данном случае информация просто тут же выводится пользователю, но, кроме этого, информация, которую ввел пользователь, может быть сохранена в какую-то переменную и может использоваться далее в скрипте.

```
In [2]: protocol = input('Твой любимый протокол маршрутизации? ')
Твой любимый протокол маршрутизации? OSPF

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

В скобках обычно пишется какой-то вопрос, который уточняет, какую информацию нужно ввести.

Текст в скобках, в принципе, писать не обязательно.

И можно сделать такой же вывод с помощью функции `print`:

```
In [4]: print('Твой любимый протокол маршрутизации? ')
Твой любимый протокол маршрутизации?

In [5]: protocol = input()
OSPF

In [6]: print(protocol)
OSPF
```

Но, как правило, нагляднее писать текст в самой функции `input()`.

Запрос информации из скрипта (файл `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))
```

В первых двух строках запрашивается информация у пользователя.

Еще появилась строка `print('\n' + '-' * 30)`.

Она используется просто для того, чтобы отделить запрос информации от вывода.

Выполняем скрипт:

```
$ 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
```

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 5.1

Запросить у пользователя ввод IP-сети в формате: 10.1.1.0/24

Затем вывести информацию о сети и маске в таком формате:

```
Network:  
10      1      1      0  
00001010  00000001  00000001  00000000  
  
Mask:  
/24  
255      255      255      0  
11111111  11111111  11111111  00000000
```

Проверить работу скрипта на разных комбинациях сеть/маска.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 5.1a

Всё, как в задании 5.1. Но, если пользователь ввел адрес хоста, а не адрес сети, то надо адрес хоста преобразовать в адрес сети и вывести адрес сети и маску, как в задании 5.1.

Пример адреса сети (все биты хостовой части равны нулю):

- 10.0.1.0/24
- 190.1.0.0/16

Пример адреса хоста:

- 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

Если пользователь ввел адрес 10.0.1.1/24, вывод должен быть таким:

```
Network:  
10      0      1      0  
00001010  00000000  00000001  00000000  
  
Mask:  
/24  
255      255      255      0  
11111111  11111111  11111111  00000000
```

Проверить работу скрипта на разных комбинациях сеть/маска.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 5.1b

Преобразовать скрипт из задания 5.1а таким образом, чтобы сеть/маска не запрашивались у пользователя, а передавались как аргумент скрипту.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 5.2

В задании создан словарь с информацией о разных устройствах.

Вам нужно запросить у пользователя ввод имени устройства (r1, r2 или sw1). И вывести информацию о соответствующем устройстве на стандартный поток вывода (информация будет в виде словаря).

Пример выполнения скрипта (ключи могут быть в другом порядке):

```
$ python task_5_2.py  
Enter device name: r1  
{'ios': '15.4', 'model': '4451', 'vendor': 'Cisco', 'location': '21 New Globe Walk',  
'ip': '10.255.0.1'}
```

Ограничение: нельзя изменять словарь london_co.

Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if.

```
london_co = {
    'r1' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    '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
    }
}
```

Задание 5.2а

Переделать скрипт из задания 5.2 таким образом, чтобы, кроме имени устройства, запрашивался также параметр устройства, который нужно отобразить.

Вывести информацию о соответствующем параметре, указанного устройства.

Пример выполнения скрипта:

```
$ python task_5_2a.py
Enter device name: r1
Enter parameter name: ios
15.4
```

Ограничение: нельзя изменять словарь london_co.

Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if.

```
london_co = {
    'r1' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    '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
    }
}
```

Задание 5.2b

Переделать скрипт из задания 5.2а таким образом, чтобы, при запросе параметра, отображался список возможных параметров.

Вывести информацию о соответствующем параметре, указанного устройства.

Пример выполнения скрипта:

```
$ python task_5_2b.py
Enter device name: r1
Enter parameter name (ios,model,vendor,location,ip): ip
10.255.0.1
```

Ограничение: нельзя изменять словарь london_co.

Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if.

```

london_co = {
    'r1' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    '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
    }
}

```

Задание 5.2с

Переделать скрипт из задания 5.2b таким образом, чтобы, при запросе параметра, которого нет в словаре устройства, отображалось сообщение 'Такого параметра нет'.

Попробуйте набрать неправильное имя параметра или несуществующий параметр, чтобы увидеть какой будет результат. А затем выполняйте задание.

Если выбран существующий параметр, вывести информацию о соответствующем параметре, указанного устройства.

Пример выполнения скрипта:

```

$ python task_5_2c.py
Enter device name: r1
Enter parameter name (ios,model,vendor,location,ip): io
Такого параметра нет

```

Ограничение: нельзя изменять словарь london_co.

Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if.

```
london_co = {
    'r1' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    '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
    }
}
```

Задание 5.2d

Переделать скрипт из задания 5.2c таким образом, чтобы, при запросе параметра, пользователь мог вводить название параметра в любом регистре.

Пример выполнения скрипта:

```
$ python task_5_2d.py
Enter device name: r1
Enter parameter name (ios,model,vendor,location,ip): IOS
15.4
```

Ограничение: нельзя изменять словарь london_co.

Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if.

```

london_co = {
    'r1' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.1'
    },
    'r2' : {
        'location': '21 New Globe Walk',
        'vendor': 'Cisco',
        'model': '4451',
        'ios': '15.4',
        'ip': '10.255.0.2'
    },
    '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
    }
}

```

Задание 5.3

Скрипт должен запрашивать у пользователя:

- информацию о режиме интерфейса (access/trunk),
 - пример текста запроса: 'Enter interface mode (access/trunk): '
- номере интерфейса (тип и номер, вида Gi0/3)
 - пример текста запроса: 'Enter interface type and number: '
- номер VLANa (для режима trunk будет вводиться список VLANов)
 - пример текста запроса: 'Enter vlan(s): '

В зависимости от выбранного режима, на стандартный поток вывода, должна возвращаться соответствующая конфигурация access или trunk (шаблоны команд находятся в списках access_template и trunk_template).

При этом, сначала должна идти строка interface и подставлен номер интерфейса, а затем соответствующий шаблон, в который подставлен номер VLANa (или список VLANов).

Ограничение: Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if и циклов for/while.

Ниже примеры выполнения скрипта, чтобы было проще понять задачу.

Пример выполнения скрипта, при выборе режима access:

```
$ python task_5_3.py
Enter interface mode (access/trunk): access
Enter interface type and number: Fa0/6
Enter vlan(s): 3

interface Fa0/6
switchport mode access
switchport access vlan 3
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

Пример выполнения скрипта, при выборе режима trunk:

```
$ python task_5_3.py
Enter interface mode (access/trunk): trunk
Enter interface type and number: Fa0/7
Enter vlan(s): 2,3,4,5

interface Fa0/7
switchport trunk encapsulation dot1q
switchport mode trunk
switchport trunk allowed vlan 2,3,4,5
```

Начальное содержимое скрипта:

```
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 {}']
```

Задание 5.3а

Дополнить скрипт из задания 5.3 таким образом, чтобы, в зависимости от выбранного режима, задавались разные вопросы в запросе о номере VLANа или списка VLANов:

- для access: 'Enter VLAN number:'
- для trunk: 'Enter allowed VLANs:'

Ограничение: Все задания надо выполнять используя только пройденные темы. То есть эту задачу можно решить без использования условия if и циклов for/while.

```
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 {}']
```

Задание 5.4

Найти индекс последнего вхождения элемента.

Например, для списка num_list, число 10 последний раз встречается с индексом 4; в списке word_list, слово 'ruby' последний раз встречается с индексом 6.

Сделать решение общим (то есть, не привязываться к конкретному элементу в конкретном списке) и проверить на двух списках, которые указаны и на разных элементах.

Для этого надо запросить у пользователя сначала ввод числа из списка num_list и затем вывести индекс его последнего появления в списке. А затем аналогично для списка word_list.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
num_list = [10, 2, 30, 100, 10, 50, 11, 30, 15, 7]
word_list = ['python', 'ruby', 'perl', 'ruby', 'perl', 'python', 'ruby', 'perl']
```

Контроль хода программы

В этом разделе рассматриваются возможности Python в управлении ходом программы.

Как минимум, стоит разобраться с конструкциями:

- `if/elif/else`
- циклом `for`
- циклом `while`

Остальные разделы можно прочесть позже.

Раздел про конструкции `for/else` и `while/else`, возможно, будет проще понять, если прочесть их после раздела об обработке исключений.

if/elif/else

Конструкция if/elif/else дает возможность выполнять различные действия в зависимости от условий.

В этой конструкции только if является обязательным, elif и else опциональны:

- Проверка if всегда идет первой.
- После оператора if должно быть какое-то условие: если это условие выполняется (возвращает True), то действия в блоке if выполняются.
- С помощью elif можно сделать несколько разветвлений, то есть, проверять входящие данные на разные условия.
 - блок elif это тот же if, но только следующая проверка. Грубо говоря, это "а если ..."
 - блоков elif может быть много
- Блок else выполняется в том случае, если ни одно из условий if или elif не было истинным.

Пример конструкции:

```
In [1]: a = 9

In [2]: if a == 10:
....:     print('a равно 10')
....: elif a < 10:
....:     print('a меньше 10')
....: else:
....:     print('a больше 10')
....:
a меньше 10
```

Условиями после if или elif могут быть, например, такие конструкции:

```
In [7]: 5 > 3
Out[7]: True

In [8]: 5 == 5
Out[8]: True

In [9]: 'vlan' in 'switchport trunk allowed vlan 10,20'
Out[9]: True

In [10]: 1 in [ 1, 2, 3 ]
Out[10]: True

In [11]: 0 in [ 1, 2, 3 ]
Out[11]: False
```

True и False

В Python:

- **True** (истина)
 - любое ненулевое число
 - любая непустая строка
 - любой непустой объект
- **False** (ложь)
 - 0
 - None
 - пустая строка
 - пустой объект

Остальные значения True или False, как правило, логически следуют из условия.

Например, так как пустой список это ложь, проверить, пустой ли список, можно таким образом:

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

In [13]: if list_to_test:
....:     print("В списке есть объекты")
....:
В списке есть объекты
```

Тот же результат можно было бы получить таким образом:

```
In [14]: if len(list_to_test) != 0:  
....:     print("В списке есть объекты")  
....:  
В списке есть объекты
```

Операторы сравнения

Операторы сравнения, которые могут использоваться в условиях:

```
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
```

Обратите внимание, что равенство проверяется двойным `==`.

Оператор `in`

Оператор `in` позволяет выполнять проверку на наличие элемента в последовательности (например, элемента в списке или подстроки в строке):

```
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
```

При использовании со словарями условие `in` выполняет проверку по ключам словаря:

```
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
```

Операторы `and`, `or`, `not`

В условиях могут также использоваться **логические операторы** `and` , `or` , `not` :

```
In [15]: r1 = {
....: 'IOS': '15.4',
....: 'IP': '10.255.0.1',
....: 'hostname': 'london_r1',
....: '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
```

Оператор and

В Python оператор `and` возвращает не булево значение, а значение одного из операторов.

Если оба операнда являются истиной, результатом выражения будет последнее значение:

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

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

Если один из операторов является ложью, результатом выражения будет первое ложное значение:

```
In [26]: '' and 'string1'
Out[26]: ''
```

```
In [27]: '' and [] and 'string1'
Out[27]: ''
```

Оператор or

Оператор `or`, как и оператор `and`, возвращает значение одного из операторов.

При оценке operandов возвращается первый истинный operand:

```
In [28]: '' or 'string1'  
Out[28]: 'string1'  
  
In [29]: '' or [] or 'string1'  
Out[29]: 'string1'  
  
In [30]: 'string1' or 'string2'  
Out[30]: 'string1'
```

Если все значения являются ложью, возвращается последнее значение:

```
In [31]: '' or [] or {}  
Out[31]: {}
```

Важная особенность работы оператора `or` - operandы, которые находятся после истинного, не вычисляются:

```
In [33]: '' or sorted([44, 1, 67])  
Out[33]: [1, 44, 67]  
  
In [34]: '' or 'string1' or sorted([44, 1, 67])  
Out[34]: 'string1'
```

Пример использования конструкции if/elif/else

Пример скрипта `check_password.py`, который проверяет длину пароля и есть ли в пароле имя пользователя:

```
# -*- coding: utf-8 -*-  
  
username = input('Введите имя пользователя: ')  
password = input('Введите пароль: ')  
  
if len(password) < 8:  
    print('Пароль слишком короткий')  
elif username in password:  
    print('Пароль содержит имя пользователя')  
else:  
    print('Пароль для пользователя {} установлен'.format(username))
```

Проверка скрипта:

```
$ python check_password.py
Введите имя пользователя: nata
Введите пароль: nata1234
Пароль содержит имя пользователя

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 123nata123
Пароль содержит имя пользователя

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 1234
Пароль слишком короткий

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 123456789
Пароль для пользователя nata установлен
```

Трехместное выражение (Ternary expressions)

Иногда удобнее использовать тернарный оператор, нежели развернутую форму:

```
s = [1, 2, 3, 4]
result = True if len(s) > 5 else False
```

for

Цикл for проходится по указанной последовательности и выполняет действия, которые указаны в блоке for.

Примеры последовательностей элементов, по которым может проходиться цикл for:

- строка
- список
- словарь
- функция `range()`
- любой [итерируемый объект](#)

Цикл for проходится по строке:

```
In [1]: for letter in 'Test string':  
...:     print(letter)  
...:  
T  
e  
s  
t  
  
s  
t  
r  
i  
n  
g
```

В цикле используется переменная с именем `letter`. Хотя имя может быть любое, удобно, когда имя подсказывает, через какие объекты проходит цикл.

Пример цикла for с функцией `range()`:

for

```
In [2]: for i in range(10):
...:     print('interface FastEthernet0/{}'.format(i))
...:
interface FastEthernet0/0
interface FastEthernet0/1
interface FastEthernet0/2
interface FastEthernet0/3
interface FastEthernet0/4
interface FastEthernet0/5
interface FastEthernet0/6
interface FastEthernet0/7
interface FastEthernet0/8
interface FastEthernet0/9
```

В этом цикле используется `range(10)`. Range генерирует числа в диапазоне от нуля до указанного числа (в данном примере - до 10), не включая его.

В этом примере цикл проходит по списку VLANов, поэтому переменную можно назвать `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
```

Когда цикл идет по словарю, то фактически он проходится по ключам:

for

```
In [5]: r1 = {  
    'IOS': '15.4',  
    'IP': '10.255.0.1',  
    'hostname': 'london_r1',  
    'location': '21 New Globe Walk',  
    'model': '4451',  
    'vendor': 'Cisco'}  
  
In [6]: for k in r1:  
....:     print(k)  
....:  
vendor  
IP  
hostname  
IOS  
location  
model
```

Если необходимо выводить пары ключ:значение в цикле:

```
In [7]: for key in r1:  
....:     print(key + ' => ' + r1[key])  
....:  
vendor => Cisco  
IP => 10.255.0.1  
hostname => london_r1  
IOS => 15.4  
location => 21 New Globe Walk  
model => 4451
```

В словаре есть специальный метод `items`, который позволяет проходиться в цикле сразу по паре ключ:значение:

```
In [8]: for key, value in r1.items():  
....:     print(key + ' => ' + value)  
....:  
vendor => Cisco  
IP => 10.255.0.1  
hostname => london_r1  
IOS => 15.4  
location => 21 New Globe Walk  
model => 4451
```

Метод `items` возвращает специальный объект `view`, который отображает пары ключ:значение:

```
In [9]: r1.items()
Out[9]: dict_items([('IOS', '15.4'), ('IP', '10.255.0.1'), ('hostname', 'london_r1'), ('location', '21 New Globe Walk'), ('model', '4451'), ('vendor', 'Cisco')])
```



Вложенные for

Циклы for можно вкладывать друг в друга.

В этом примере в списке commands хранятся команды, которые надо выполнить для каждого из интерфейсов в списке fast_int:

```
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
...
```

Первый цикл for проходится по интерфейсам в списке fast_int, а второй по командам в списке commands.

Совмещение for и if

Рассмотрим пример совмещения for и if.

Файл generate_access_port_config.py:

```
access_template = ['switchport mode access',
                   'switchport access vlan',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

fast_int = {'access': { '0/12':10,
                       '0/14':11,
                       '0/16':17,
                       '0/17':150} }

for intf, vlan in fast_int['access'].items():
    print('interface FastEthernet' + intf)
    for command in access_template:
        if command.endswith('access vlan'):
            print(' {} {}'.format(command, vlan))
        else:
            print(' {}'.format(command))
```

Комментарии к коду:

- В первом цикле for перебираются ключи и значения во вложенном словаре `fast_int['access']`
- Текущий ключ, на данный момент цикла, хранится в переменной `intf`
- Текущее значение, на данный момент цикла, хранится в переменной `vlan`
- Выводится строка `interface FastEthernet` с добавлением к ней номера интерфейса
- Во втором цикле for перебираются команды из списка `access_template`
- Так как к команде `switchport access vlan` надо добавить номер VLAN:
 - внутри второго цикла for проверяются команды
 - если команда заканчивается на `access vlan`
 - выводится команда, и к ней добавляется номер VLAN
 - во всех остальных случаях просто выводится команда

Результат выполнения скрипта:

```
$ python generate_access_port_config.py
interface FastEthernet0/12
switchport mode access
switchport access vlan 10
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet0/14
switchport mode access
switchport access vlan 11
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet0/16
switchport mode access
switchport access vlan 17
spanning-tree portfast
spanning-tree bpduguard enable
interface FastEthernet0/17
switchport mode access
switchport access vlan 150
spanning-tree portfast
spanning-tree bpduguard enable
```

while

Цикл while - это еще одна разновидность цикла в Python.

В цикле while, как и в выражении if, надо писать условие. Если условие истинно, выполняются действия внутри блока while. Но, в отличии от if, после выполнения while возвращается в начало цикла.

При использовании циклов while необходимо обращать внимание на то, будет ли достигнуто такое состояние, при котором условие цикла будет ложным.

Рассмотрим простой пример:

```
In [1]: a = 5

In [2]: while a > 0:
...:     print(a)
...:     a -= 1 # Эта запись равнозначна a = a - 1
...:

5
4
3
2
1
```

Сначала создается переменная a со значением 5.

Затем, в цикле while указано условие `a > 0`. То есть, пока значение a больше 0, будут выполняться действия в теле цикла. В данном случае, будет выводиться значение переменной a.

Кроме того, в теле цикла при каждом прохождении значение a становится на единицу меньше.

Запись `a -= 1` может быть немного необычной. Python позволяет использовать такой формат вместо `a = a - 1`.

Аналогичным образом можно писать: `a += 1`, `a *= 2`, `a /= 2`.

Так как значение a уменьшается, цикл не будет бесконечным, и в какой-то момент выражение `a > 0` станет ложным.

Следующий пример построен на основе примера про пароль из раздела о [конструкции if](#). В том примере приходилось заново запускать скрипт, если пароль не соответствовал требованиям.

С помощью цикла while можно сделать так, что скрипт сам будет запрашивать пароль заново, если он не соответствует требованиям.

Файл check_password_with_while.py:

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

username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

password_correct = False

while not password_correct:
    if len(password) < 8:
        print('Пароль слишком короткий\n')
        password = input('Введите пароль еще раз: ')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
        password = input('Введите пароль еще раз: ')
    else:
        print('Пароль для пользователя {} установлен'.format(username))
        password_correct = True
```

В этом случае цикл while полезен, так как он возвращает скрипт снова в начало проверок, позволяет снова набрать пароль, но при этом не требует перезапуска самого скрипта.

Теперь скрипт отрабатывает так:

```
$ python check_password_with_while.py
Введите имя пользователя: nata
Введите пароль: nata
Пароль слишком короткий

Введите пароль еще раз: natanata
Пароль содержит имя пользователя

Введите пароль еще раз: 123345345345
Пароль для пользователя nata установлен
```

break, continue, pass

В Python есть несколько операторов, которые позволяют менять поведение циклов по умолчанию.

Оператор break

Оператор **break** позволяет досрочно прервать цикл:

- **break** прерывает текущий цикл и продолжает выполнение следующих выражений
- если используется несколько вложенных циклов, **break** прерывает внутренний цикл и продолжает выполнять выражения, следующие за блоком
- **break** может использоваться в циклах **for** и **while**

Пример с циклом **for**:

```
In [1]: for num in range(10):
...:     if num < 7:
...:         print(num)
...:     else:
...:         break
...:

0
1
2
3
4
5
6
```

Пример с циклом **while**:

```
In [2]: i = 0
In [3]: while i < 10:
...:     if i == 5:
...:         break
...:     else:
...:         print(i)
...:     i += 1
...:

0
1
2
3
4
```

Использование break в примере с запросом пароля (файл `check_password_with_while_break.py`):

```
username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

while True:
    if len(password) < 8:
        print('Пароль слишком короткий\n')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
    else:
        print('Пароль для пользователя {} установлен'.format(username))
        # завершает цикл while
        break
password = input('Введите пароль еще раз: ')
```

Теперь можно не повторять строку `password = input('Введите пароль еще раз: ')` в каждом ответвлении, достаточно перенести ее в конец цикла.

И, как только будет введен правильный пароль, `break` выведет программу из цикла `while`.

Оператор `continue`

Оператор `continue` возвращает управление в начало цикла. То есть, `continue` позволяет "перепрыгнуть" оставшиеся выражения в цикле и перейти к следующей итерации.

Пример с циклом `for`:

```
In [4]: for num in range(5):
...:     if num == 3:
...:         continue
...:     else:
...:         print(num)
...:
0
1
2
4
```

Пример с циклом `while`:

```
In [5]: i = 0
In [6]: while i < 6:
....:     i += 1
....:     if i == 3:
....:         print("Пропускаем 3")
....:         continue
....:     print("Это никто не увидит")
....: else:
....:     print("Текущее значение: ", i)
....:
Текущее значение: 1
Текущее значение: 2
Пропускаем 3
Текущее значение: 4
Текущее значение: 5
Текущее значение: 6
```

Использование `continue` в примере с запросом пароля (файл `check_password_with_while_continue.py`):

```
username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

password_correct = False

while not password_correct:
    if len(password) < 8:
        print('Пароль слишком короткий\n')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
    else:
        print('Пароль для пользователя {} установлен'.format(username))
        password_correct = True
        continue
    password = input('Введите пароль еще раз: ')
```

Тут выход из цикла выполняется с помощью проверки флага `password_correct`. Когда был введен правильный пароль, флаг выставляется равным `True`, и с помощью `continue` выполняется переход в начало цикла, перескочив последнюю строку с запросом пароля.

Результат выполнения будет таким:

```
$ python check_password_with_while_continue.py
Введите имя пользователя: nata
Введите пароль: nata12
Пароль слишком короткий

Введите пароль еще раз: natalksdjf1sdjf
Пароль содержит имя пользователя

Введите пароль еще раз: asdfsujljhdfaskjdfh
Пароль для пользователя nata установлен
```

Оператор pass

Оператор `pass` ничего не делает. Фактически, это такая заглушка для объектов.

Например, `pass` может помочь в ситуации, когда нужно прописать структуру скрипта. Его можно ставить в циклах, функциях, классах. И это не будет влиять на исполнение кода.

Пример использования `pass`:

```
In [6]: for num in range(5):
....:     if num < 3:
....:         pass
....:     else:
....:         print(num)
....:
3
4
```

for/else, while/else

В циклах for и while опционально может использоваться блок else.

for/else

В цикле for:

- блок else выполняется в том случае, если цикл завершил итерацию списка
 - но else **не выполняется**, если в цикле был выполнен break

Пример цикла for с else (блок else выполняется после завершения цикла for):

```
In [1]: for num in range(5):
....:     print(num)
....: else:
....:     print("Числа закончились")
....:
0
1
2
3
4
Числа закончились
```

Пример цикла for с else и break в цикле (из-за break блок else не выполняется):

```
In [2]: for num in range(5):
....:     if num == 3:
....:         break
....:     else:
....:         print(num)
....: else:
....:     print("Числа закончились")
....:
0
1
2
```

Пример цикла for с else и continue в цикле (continue не влияет на блок else):

```
In [3]: for num in range(5):
....:     if num == 3:
....:         continue
....:     else:
....:         print(num)
....: else:
....:     print("Числа закончились")
....:

0
1
2
3
4
Числа закончились
```

while/else

В цикле while:

- блок else выполняется в том случае, если цикл завершил итерацию списка
 - но else **не выполняется**, если в цикле был выполнен break

Пример цикла while с else (блок else выполняется после завершения цикла while):

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

0
1
2
3
4
Конец
```

Пример цикла while с else и break в цикле (из-за break блок else не выполняется):

```
In [6]: i = 0

In [7]: while i < 5:
....:     if i == 3:
....:         break
....:     else:
....:         print(i)
....:     i += 1
....: else:
....:     print("Конец")
....:

0
1
2
```

Работа с исключениями try/except/else/finally

try/except

Если Вы повторяли примеры, которые использовались ранее, то наверняка были ситуации, когда высакивала ошибка. Скорее всего, это была ошибка синтаксиса, когда не хватало, например, двоеточия.

Как правило, Python довольно понятно реагирует на подобные ошибки, и их можно исправить.

Но, даже если код синтаксически написан правильно, все равно могут возникать ошибки. Эти ошибки называются **исключения (exceptions)**.

Примеры исключений:

```
In [1]: 2/0
-----
ZeroDivisionError: division by zero

In [2]: 'test' + 2
-----
TypeError: must be str, not int
```

В данном случае возникло два исключения: **ZeroDivisionError** и **TypeError**.

Чаще всего можно предсказать, какого рода исключение возникнут во время исполнения программы.

Например, если программа на вход ожидает два числа, а на выходе выдает их сумму, а пользователь ввел вместо одного из чисел строку, появится ошибка **TypeError**, как в примере выше.

Python позволяет работать с исключениями. Их можно перехватывать и выполнять определенные действия в том случае, если возникло исключение.

Когда в программе возникает исключение, она сразу завершает работу.

Для работы с исключениями используется конструкция `try/except`:

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

Конструкция try работает таким образом:

- сначала выполняются выражения, которые записаны в блоке try
- если при выполнении блока try не возникло никаких исключений, блок except пропускается, и выполняется дальнейший код
- если во время выполнения блока try в каком-то месте возникло исключение, оставшаяся часть блока try пропускается
 - если в блоке except указано исключение, которое возникло, выполняется код в блоке except
 - если исключение, которое возникло, не указано в блоке except, выполнение программы прерывается и выдается ошибка

Обратите внимание, что строка 'Cool!' в блоке try не выводится:

```
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
```

В конструкции try/except может быть много except, если нужны разные действия в зависимости от типа ошибки.

Например, скрипт divide.py делит два числа введенных пользователем:

```
# -*- coding: utf-8 -*-
try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    print("Результат: ", int(a)/int(b))
except ValueError:
    print("Пожалуйста, вводите только числа")
except ZeroDivisionError:
    print("На ноль делить нельзя")
```

Примеры выполнения скрипта:

```
$ python divide.py  
Введите первое число: 3  
Введите второе число: 1  
Результат: 3  
  
$ python divide.py  
Введите первое число: 5  
Введите второе число: 0  
На ноль делить нельзя  
  
$ python divide.py  
Введите первое число: qewr  
Введите второе число: 3  
Пожалуйста, вводите только числа
```

В данном случае исключение **ValueError** возникает, когда пользователь ввел строку вместо числа, во время перевода строки в число.

Исключение **ZeroDivisionError** возникает в случае, если второе число было равным 0.

Если нет необходимости выводить различные сообщения на ошибки **ValueError** и **ZeroDivisionError**, можно сделать так (файл `divide_ver2.py`):

```
# -*- coding: utf-8 -*-  
  
try:  
    a = input("Введите первое число: ")  
    b = input("Введите второе число: ")  
    print("Результат: ", int(a)/int(b))  
except (ValueError, ZeroDivisionError):  
    print("Что-то пошло не так...")
```

Проверка:

```
$ python divide_ver2.py  
Введите первое число: wer  
Введите второе число: 4  
Что-то пошло не так...  
  
$ python divide_ver2.py  
Введите первое число: 5  
Введите второе число: 0  
Что-то пошло не так...
```

В блоке except можно не указывать конкретное исключение или исключения. В таком случае будут перехватываться все исключения.

Это делать не рекомендуется!

try/except/else

В конструкции try/except есть опциональный блок else. Он выполняется в том случае, если не было исключений.

Например, если необходимо выполнять в дальнейшем какие-то операции с данными, которые ввел пользователь, можно записать их в блоке else (файл divide_ver3.py):

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

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
else:
    print("Результат в квадрате: ", result**2)
```

Пример выполнения:

```
$ python divide_ver3.py
Введите первое число: 10
Введите второе число: 2
Результат в квадрате: 25

$ python divide_ver3.py
Введите первое число: werq
Введите второе число: 3
Что-то пошло не так...
```

try/except/finally

Блок finally - это еще один опциональный блок в конструкции try. Он выполняется **всегда**, независимо от того, было ли исключение или нет.

Сюда ставятся действия, которые надо выполнить в любом случае. Например, это может быть закрытие файла.

Файл divide_ver4.py с блоком finally:

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

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
else:
    print("Результат в квадрате: ", result**2)
finally:
    print("Вот и сказочке конец, а кто слушал - молодец.")
```

Проверка:

```
$ python divide_ver4.py
Введите первое число: 10
Введите второе число: 2
Результат в квадрате: 25
Вот и сказочке конец, а кто слушал - молодец.

$ python divide_ver4.py
Введите первое число: qwerewr
Введите второе число: 3
Что-то пошло не так...
Вот и сказочке конец, а кто слушал - молодец.

$ python divide_ver4.py
Введите первое число: 4
Введите второе число: 0
Что-то пошло не так...
Вот и сказочке конец, а кто слушал - молодец.
```

Когда использовать исключения

Как правило, один и тот же код можно написать и с использованием исключений, и без них.

Например, это вариант кода:

```
while True:
    a = input("Введите число: ")
    b = input("Введите второе число: ")
    try:
        result = int(a)/int(b)
    except ValueError:
        print("Поддерживаются только числа")
    except ZeroDivisionError:
        print("На ноль делить нельзя")
    else:
        print(result)
        break
```

Можно переписать таким образом без try/except (файл try_except_divide.py):

```
while True:
    a = input("Введите число: ")
    b = input("Введите второе число: ")
    if a.isdigit() and b.isdigit():
        if int(b) == 0:
            print("На ноль делить нельзя")
        else:
            print(int(a)/int(b))
            break
    else:
        print("Поддерживаются только числа")
```

Но далеко не всегда аналогичный вариант без использования исключений будет простым и понятным.

Важно в каждой конкретной ситуации оценивать, какой вариант кода более понятный, компактный и универсальный - с исключениями или без.

Если Вы раньше использовали какой-то другой язык программирования, есть вероятность, что в нём использование исключений считалось плохим тоном. В Python этот не так. Чтобы немного больше разобраться с этим вопросом, посмотрите ссылки на дополнительные материалы в конце этого раздела.

Дополнительные материалы

Документация:

- [Compound statements \(if, while, for, try\)](#)
- [break, continue](#)
- [Errors and Exceptions](#)
- [Built-in Exceptions](#)

Статьи:

- [Write Cleaner Python: Use Exceptions](#)
- [Robust exception handling](#)
- [Python Exception Handling Techniques](#)

Stackoverflow:

- [Why does python use 'else' after for and while loops?](#)
- [Is it a good practice to use try-except-else in Python?](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 6.1

1. Запросить у пользователя ввод IP-адреса в формате 10.0.1.1.
2. Определить какому классу принадлежит IP-адрес.
3. В зависимости от класса адреса, вывести на стандартный поток вывода:
 - 'unicast' - если IP-адрес принадлежит классу A, B или C
 - 'multicast' - если IP-адрес принадлежит классу D
 - 'local broadcast' - если IP-адрес равен 255.255.255.255
 - 'unassigned' - если IP-адрес равен 0.0.0.0
 - 'unused' - во всех остальных случаях

Подсказка по классам (диапазон значений первого байта в десятичном формате):

- A: 1-127
- B: 128-191
- C: 192-223
- D: 224-239

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.1a

Сделать копию скрипта задания 6.1.

Дополнить скрипт:

- Добавить проверку введенного IP-адреса.
- Адрес считается корректно заданным, если он:
 - состоит из 4 чисел разделенных точкой,
 - каждое число в диапазоне от 0 до 255.

Если адрес задан неправильно, выводить сообщение:

- 'Incorrect IPv4 address'

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.1b

Сделать копию скрипта задания 6.1a.

Дополнить скрипт:

- Если адрес был введен неправильно, запросить адрес снова.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.2

Список mac содержит MAC-адреса в формате XXXX:XXXX:XXXX. Однако, в оборудовании cisco MAC-адреса используются в формате XXXX.XXXX.XXXX.

Создать скрипт, который преобразует MAC-адреса в формат cisco и добавляет их в новый список mac_cisco

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 6.3

В скрипте сделан генератор конфигурации для access-портов.

Сделать аналогичный генератор конфигурации для портов trunk.

В транках ситуация усложняется тем, что VLANов может быть много, и надо понимать, что с ними делать.

Поэтому в соответствии каждому порту стоит список и первый (нулевой) элемент списка указывает как воспринимать номера VLANов, которые идут дальше:

- add - значит VLANы надо будет добавить (команда switchport trunk allowed vlan add 10,20)
- del - значит VLANы надо удалить из списка разрешенных (команда switchport trunk allowed vlan remove 17)
- only - значит, что на интерфейсе должны остаться разрешенными только указанные VLANы (команда switchport trunk allowed vlan 11,30)

Задача для портов 0/1, 0/2, 0/4:

- сгенерировать конфигурацию на основе шаблона trunk_template
- с учетом ключевых слов add, del, only

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
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']

fast_int = {'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 fast_int['access'].items():
    print('interface FastEthernet' + intf)
    for command in access_template:
        if command.endswith('access vlan'):
            print(' {} {}'.format(command, vlan))
        else:
            print(' {}'.format(command))
```

Работа с файлами

В реальной жизни, для того чтобы полноценно использовать всё, что рассматривалось до этого раздела, надо разобраться как работать с файлами.

При работе с сетевым оборудованием (и не только), файлами могут быть:

- конфигурации (простые, не структурированные текстовые файлы)
 - работа с ними рассматривается в этом разделе
- шаблоны конфигураций
 - как правило, это какой-то специальный формат файлов.
 - в разделе [Шаблоны конфигураций с Jinja](#) рассматривается использование `Jinja2` для создания шаблонов конфигураций
- файлы с параметрами подключений
 - как правило, это структурированные файлы, в каком-то определенном формате: `YAML`, `JSON`, `CSV`
 - в разделе [Сериализация данных](#) рассматривается как работать с такими файлами
- другие скрипты Python
 - в разделе [Модули](#) рассматривается как работать с модулями (другими скриптами Python)

В этом разделе рассматривается работа с простыми текстовыми файлами. Например, конфигурационный файл Cisco.

В работе с файлами есть несколько аспектов:

- открытие/закрытие
- чтение
- запись

В этом разделе рассматривается только необходимый минимум для работы с файлами. Подробнее, в [документации Python](#).

Открытие файлов

Для начала работы с файлом, его надо открыть.

open()

Для открытия файлов, чаще всего, используется функция `open()` :

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

В функции `open()`:

- `'file_name.txt'` - имя файла
 - тут можно указывать не только имя, но и путь (абсолютный или относительный)
- `'r'` - режим открытия файла

Функция `open()` создает объект `file`, к которому потом можно применять различные методы, для работы с ним.

Режимы открытия файлов:

- `r` - открыть файл только для чтения (значение по умолчанию)
- `r+` - открыть файл для чтения и записи
- `w` - открыть файл для записи
 - если файл существует, то его содержимое удаляется
 - если файл не существует, то создается новый
- `w+` - открыть файл для чтения и записи
 - если файл существует, то его содержимое удаляется
 - если файл не существует, то создается новый
- `a` - открыть файл для дополнения записи. Данные добавляются в конец файла
- `a+` - открыть файл для чтения и записи. Данные добавляются в конец файла

| `r` - read; `a` - append; `w` - write

Чтение файлов

В Python есть несколько методов чтения файла:

- `read()` - считывает содержимое файла в строку
- `readline()` - считывает файл построчно
- `readlines()` - считывает строки файла и создает список из строк

Посмотрим как считывать содержимое файлов, на примере файла `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
!
```

read()

Метод `read()` - считывает весь файл в одну строку.

Пример использования метода `read()` :

```
In [1]: f = open('r1.txt')

In [2]: f.read()
Out[2]: '!\\nservice timestamps debug datetime msec localtime show-timezone year\\nseri
ce timestamps log datetime msec localtime show-timezone year\\nservice password-encrypt
ion\\nservice sequence-numbers\\n!\\nno ip domain lookup\\n!\\nip ssh version 2\\n!\\n'

In [3]: f.read()
Out[3]: ''
```

При повторном чтении файла в 3 строке, отображается пустая строка. Так происходит из-за того, что при вызове метода `read()`, считывается весь файл. И после того, как файл был считан, курсор остается в конце файла. Управлять положением курсора можно с помощью метода `seek()`.

readline()

Построчно файл можно считать с помощью метода `readline()` :

```
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'
```

Но, чаще всего, проще пройтись по объекту `file` в цикле, не используя методы `read...` :

```
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()

Еще один полезный метод - `readlines()` . Он считывает строки файла в список:

```
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']
```

Если нужно получить строки файла, но без перевода строки в конце, можно воспользоваться методом `split` и как разделитель, указать символ `\n`:

```
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',
 '!',
 '']
```

Обратите внимание, что последний элемент списка - пустая строка.

Если перед выполнением `split()`, воспользоваться методом `rstrip()`, список будет без пустой строки в конце:

```
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()

До сих пор, файл каждый раз приходилось открывать заново, чтобы снова его считать. Так происходит из-за того, что после методов чтения, курсор находится в конце файла. И повторное чтение возвращает пустую строку.

Чтобы ещё раз считать информацию из файла, нужно воспользоваться методом `seek`, который перемещает курсор в необходимое положение.

Пример открытия файла и считывания содержимого:

```
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
!
```

Если вызывать ещё раз метод `read`, возвращается пустая строка:

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

Но, с помощью метода `seek`, можно перейти в начало файла (0 означает начало файла):

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

После того, как, с помощью `seek`, курсор был переведен в начало файла, можно опять считывать содержимое:

```
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
!
```

Запись файлов

При записи, очень важно определиться с режимом открытия файла, чтобы случайно его не удалить:

- `w` - открыть файл для записи. Если файл существует, то его содержимое удаляется
- `a` - открыть файл для дополнения записи. Данные добавляются в конец файла

При этом, оба режима создают файл, если он не существует

Для записи в файл используются такие методы:

- `write()` - записать в файл одну строку
- `writelines()` - позволяет передавать в качестве аргумента список строк

`write()`

Метод `write` ожидает строку, для записи.

Для примера, возьмем список строк с конфигурацией:

```
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',  
...: '!']
```

Открытие файла `r2.txt` в режиме для записи:

```
In [2]: f = open('r2.txt', 'w')
```

Преобразуем список команд в одну большую строку с помощью `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\\nserve
ce timestamps log datetime msec localtime show-timezone year\\nservice password-encrypt
ion\\nservice sequence-numbers\\n!\\nno ip domain lookup\\n!\\nip ssh version 2\\n!'
```

Запись строки в файл:

```
In [5]: f.write(cfg_lines_as_string)
```

Аналогично можно добавить строку вручную:

```
In [6]: f.write('\\nhostname r2')
```

После завершения работы с файлом, его необходимо закрыть:

```
In [7]: f.close()
```

Так как ipython поддерживает команду cat, можно легко посмотреть содержимое файла:

```
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()

Метод `writelines()` ожидает список строк, как аргумент.

Запись списка строк `cfg_lines` в файл:

```
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 year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
no ip domain lookup
ip ssh version 2!
```

В результате, все строки из списка, записались в одну строку файла, так как в конце строк не было символа `\n`.

Добавить перевод строки можно по-разному.

Например, можно просто обработать список в цикле:

```
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',
```

Или использовать list comprehensions:

```
In [16]: cfg_lines3 = [ line + '\n' for line in cfg_lines ]  
  
In [17]: cfg_lines3  
Out[17]:  
['!\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']
```

Если любой, из получившихся списков записать заново в файл, то в нём уже будут переводы строк:

```
In [18]: f = open('r2.txt', 'w')  
  
In [19]: f.writelines(cfg_lines3)  
  
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  
!
```

Закрытие файлов

В реальной жизни, для закрытия файлов, чаще всего, используется конструкция `with`. Её намного удобней использовать, чем закрывать файл явно. Но, так как в жизни можно встретить и метод `close`, в этом разделе рассматривается как его использовать.

После завершения работы с файлом, его нужно закрыть.

В некоторых случаях, Python может самостоятельно закрыть файл.

Но лучше на это не рассчитывать и закрывать файл явно.

close()

Метод `close` встречался в разделе [запись файлов](#).

Там он был нужен для того, чтобы содержимое файла было записано на диск.

Для этого, в Python есть отдельный метод `flush()`.

Но, так как, в примере с записью файлов, не нужно было больше выполнять никаких операций, файл можно было закрыть.

Откроем файл `r1.txt`:

```
In [1]: f = open('r1.txt', 'r')
```

Теперь можно считать содержимое:

```
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
!
```

У объекта `file` есть специальный атрибут `closed`, который позволяет проверить закрыт файл или нет.

Если файл открыт, он возвращает `False`:

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

Теперь закрываем файл и снова проверяем `closed`:

```
In [4]: f.close()  
  
In [5]: f.closed  
Out[5]: True
```

Если попробовать прочитать файл, возникнет исключение:

```
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
```

Использование `try/finally` для работы с файлами

С помощью обработки исключений, можно:

- перехватывать исключения, которые возникают, при попытке прочитать несуществующий файл
- закрывать файл, после всех операций, в блоке `finally`

Если попытаться открыть для чтения файл, которого не существует, возникнет такое исключение:

```
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'
```

С помощью конструкции `try/except`, можно перехватить это исключение и вывести своё сообщение:

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

А с помощью части `finally`, можно закрыть файл, после всех операций:

```
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
```

Конструкция with

Конструкция with называется менеджером контекста.

В Python существует более удобный способ работы с файлами, чем те, которые использовались до сих пор - конструкция `with`:

```
In [1]: with open('r1.txt', 'r') as f:  
....:     for line in f:  
....:         print(line)  
....:  
!  
  
service timestamps debug datetime msec localtime show-timezone year  
  
service timestamps log datetime msec localtime show-timezone year  
  
service password-encryption  
  
service sequence-numbers  
  
!  
  
no ip domain lookup  
  
!  
  
ip ssh version 2  
  
!
```

Кроме того, конструкция `with` гарантирует закрытие файла автоматически.

Обратите внимание на то, какчитываются строки файла:

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

Когда с файлом нужно работать построчно, лучше использовать такой вариант.

В предыдущем выводе, между строками файла были лишние пустые строки, так как `print` добавляет ещё один перевод строки.

Чтобы избавиться от этого, можно использовать метод `rstrip`:

```
In [2]: with open('r1.txt', 'r') as f:
....:     for line in f:
....:         print(line.rstrip())
....:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!

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

И, конечно же, с конструкцией `with` можно использовать не только такой построчный вариант считывания, все методы, которые рассматривались до этого, также работают:

```
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
!
```

Открытие двух файлов

Иногда нужно работать одновременно с двумя файлами. Например, надо записать некоторые строки из одного файла, в другой.

В таком случае, в блоке `with` можно открывать два файла таким образом:

```
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
```

Это равнозначно таким двум блокам with:

```
In [7]: with open('r1.txt') as src:  
...:     with open('result.txt', 'w') as dest:  
...:         for line in src:  
...:             if line.startswith('service'):  
...:                 dest.write(line)  
...:
```

Дополнительные материалы

Документация:

- [Reading and Writing Files](#)
- [The with statement](#)

Статьи:

- [The Python "with" Statement by Example](#)

Stackoverflow:

- [What is the python “with” statement designed for?](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 7.1

Аналогично заданию 4.6 обработать строки из файла ospf.txt и вывести информацию по каждой в таком виде:

Protocol:	OSPF
Prefix:	10.0.24.0/24
AD/Metric:	110/41
Next-Hop:	10.0.13.3
Last update:	3d18h
Outbound Interface:	FastEthernet0/0

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.2

Создать скрипт, который будет обрабатывать конфигурационный файл config_sw1.txt:

- имя файла передается как аргумент скрипту

Скрипт должен возвращать на стандартный поток вывода команды из переданного конфигурационного файла, исключая строки, которые начинаются с '!'.
Между строками не должно быть дополнительного символа перевода строки.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.2a

Сделать копию скрипта задания 7.2.

Дополнить скрипт:

- Скрипт не должен выводить команды, в которых содержатся слова, которые указаны в списке ignore.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 7.2b

Дополнить скрипт из задания 7.2a:

- вместо вывода на стандартный поток вывода, скрипт должен записать полученные строки в файл config_sw1_cleared.txt

При этом, должны быть отфильтрованы строки, которые содержатся в списке ignore.

Строки, которые начинаются на '!' отфильтровывать не нужно.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 7.2c

Переделать скрипт из задания 7.2b:

- передавать как аргументы скрипту:
 - имя исходного файла конфигурации
 - имя итогового файла конфигурации

Внутри, скрипт должен отфильтровать те строки, в исходном файле конфигурации, в которых содержатся слова из списка ignore. И затем записать оставшиеся строки в итоговый файл.

Проверить работу скрипта на примере файла config_sw1.txt.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 7.3

Скрипт должен обрабатывать записи в файле CAM_table.txt таким образом чтобы:

- считывались только строки, в которых указаны MAC-адреса
- каждая строка, где есть MAC-адрес, должна обрабатываться таким образом, чтобы на стандартный поток вывода была выведена таблица вида:

```
100 01bb.c580.7000 Gi0/1
200 0a4b.c380.7010 Gi0/2
300 a2ab.c5a0.2000 Gi0/3
100 0a1b.1c80.7300 Gi0/4
500 02b1.3c80.7000 Gi0/5
200 1a4b.c580.5000 Gi0/6
300 0a1b.5c80.9010 Gi0/7
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.3a

Сделать копию скрипта задания 7.3

Дополнить скрипт:

- Отсортировать вывод по номеру VLAN

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.3b

Сделать копию скрипта задания 7.3a

Дополнить скрипт:

- Запросить у пользователя ввод номера VLAN.
- Выводить информацию только по указанному VLAN.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Примеры использования основ

В этом разделе собраны те темы, которые не вошли в предыдущие разделы, а также приведены примеры использования основ Python для решения задач.

Хотя большинство примеров будет ориентировано на работу с файлами, те же принципы обработки информации будут применимы и при работе с сетевым оборудованием. Только часть с чтением из файла будет заменена на получение вывода с оборудования.

Распаковка переменных

Распаковка переменных - это специальный синтаксис, который позволяет присваивать переменным элементы итерируемого объекта.

Достаточно часто этот функционал встречается под именем tuple unpacking. Но распаковка работает на любом итерируемом объекте, не только с кортежами

Пример распаковки переменных:

```
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'
```

Такой вариант намного удобней использовать, чем использование индексов:

```
In [5]: intf, ip, status, protocol = interface[0], interface[1], interface[2], interface[3]
```

При распаковке переменных каждый элемент списка попадает в соответствующую переменную. Но важно учитывать, что переменных слева должно быть ровно столько, сколько элементов в списке.

Если переменных больше или меньше, возникнет исключение:

```
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)
```

Замена ненужных элементов

Достаточно часто из всех элементов итерируемого объекта нужны только некоторые. Но выше был пример того, что синтаксис распаковки требует указать ровно столько переменных, сколько элементов в итерируемом объекте.

Если, например, из строки `line` надо получить только VLAN, MAC и интерфейс, надо все равно указать переменную для типа записи:

```
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'
```

Но, если тип записи не нужен в дальнейшем, можно заменить переменную `item_type` нижним подчеркиванием:

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

Таким образом явно указывается то, что этот элемент не нужен.

Нижнее подчеркивание можно использовать и несколько раз:

```
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'
```

Использование *

Распаковка переменных поддерживает специальный синтаксис, который позволяет распаковывать несколько элементов в один. Если поставить * перед именем переменной, в нее запишутся все элементы, кроме тех, что присвоены явно.

Например, так можно получить первый элемент в переменную first, а остальные в 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]
```

При этом переменная со звездочкой всегда будет содержать список:

```
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]
```

Если элемент всего один, распаковка все равно отработает:

```
In [25]: first, *rest = vlans
```

```
In [26]: first
```

```
Out[26]: 55
```

```
In [27]: rest
```

```
Out[27]: []
```

Такая переменная со звездочкой в выражении распаковки может быть только одна.

```
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
```

И, конечно же, такая переменная может находиться не только в конце выражения:

```
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
```

Таким образом можно указать, что нужен первый, второй и последний элемент:

```
In [34]: cdp = 'SW1           Eth 0/0           140           S I       WS-C3750-  Et
h 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'
```

Примеры распаковки

Распаковка итерируемых объектов

Эти примеры показывают, что распаковывать можно не только списки, кортежи и строки, но и любой другой итерируемый объект.

Распаковка range:

```
In [39]: first, *rest = range(1, 6)

In [40]: first
Out[40]: 1

In [41]: rest
Out[41]: [2, 3, 4, 5]
```

Распаковка 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)
```

Пример распаковки в цикле for

Пример цикла, который проходится по ключам:

```
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
```

Вместо этого можно проходиться по парам ключ-значение и сразу же распаковывать их в разные переменные:

```
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))
...:
```

Пример распаковки элементов списка в цикле:

```
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
```

Но еще лучше сделать так:

```
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 поддерживает специальные выражения, которые позволяют компактно создавать списки, словари и множества.

На английском эти выражения называются, соответственно:

- List comprehensions
- Dict comprehensions
- Set comprehensions

К сожалению, официальный перевод на русский звучит как [абстракция списков или списковое включение](#), что не особо помогает понять суть объекта.

В книге использовался перевод "генератор списка", что, к сожалению, тоже не самый удачный вариант, так как в Python есть отдельное понятие генератор и генераторные выражения. Но он лучше отображает суть выражения.

Эти выражения не только позволяют более компактно создавать соответствующие объекты, но и создают их быстрее. И хотя поначалу они требуют определенной привычки использования и понимания, они очень часто используются.

List comprehensions (генераторы списков)

Генератор списка - это выражение вида:

```
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']
```

В общем случае, это выражение, которое преобразует итерируемый объект в список. То есть, последовательность элементов преобразуется и добавляется в новый список.

Выражению выше аналогичен такой цикл:

```
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']
```

В list comprehensions можно использовать выражение if. Таким образом можно добавлять в список только некоторые объекты.

Например, такой цикл отбирает те элементы, которые являются числами, конвертирует их и добавляет в итоговый список only_digits:

```
In [6]: items = ['10', '20', 'a', '30', 'b', '40']

In [7]: only_digits = []

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

In [9]: print(only_digits)
[10, 20, 30, 40]
```

Аналогичный вариант в виде 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]
```

Конечно, далеко не все циклы можно переписать как генератор списка, но когда это можно сделать, и при этом выражение не усложняется, лучше использовать генераторы списка.

В Python генераторы списка могут также заменить функции filter и map и считаются более понятными вариантами решения.

С помощью генератора списка также удобно получать элементы из вложенных словарей:

```
In [13]: london_co = {
...:     'r1' : {
...:         'hostname': 'london_r1',
...:         'location': '21 New Globe Walk',
...:         'vendor': 'Cisco',
...:         'model': '4451',
...:         'IOS': '15.4',
...:         'IP': '10.255.0.1'
...:     },
...:     'r2' : {
...:         'hostname': 'london_r2',
...:         'location': '21 New Globe Walk',
...:         'vendor': 'Cisco',
...:         'model': '4451',
...:         'IOS': '15.4',
...:         'IP': '10.255.0.2'
...:     },
...:     'sw1' : {
...:         'hostname': 'london_sw1',
...:         'location': '21 New Globe Walk',
...:         'vendor': 'Cisco',
...:         'model': '3850',
...:         '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']
```

На самом деле, синтаксис генератора списка выглядит так:

```
[expression for item1 in iterable1 if condition1
    for item2 in iterable2 if condition2
    ...
    for itemN in iterableN if conditionN ]
```

Это значит, можно использовать несколько `for` в выражении.

Например, в списке `vlans` находятся несколько вложенных списков с VLAN'ами:

```
In [16]: vlans = [[10, 21, 35], [101, 115, 150], [111, 40, 50]]
```

Из этого списка надо сформировать один плоский список с номерами VLAN. Первый вариант, с помощью циклов `for`:

```
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]
```

Аналогичный вариант с генератором списков:

```
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]
```

Можно одновременно проходиться по двум последовательностям, используя 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 (генераторы словарей)

Генераторы словарей аналогичны генераторам списков, но они используются для создания словарей.

Например, такое выражение:

```
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}
```

Можно заменить генератором словаря:

```
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}
```

Еще один пример, в котором надо преобразовать существующий словарь и перевести все ключи в нижний регистр. Для начала, вариант решения без генератора словаря:

```
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'}
```

Аналогичный вариант с помощью генератора словаря:

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

Как и list comprehensions, dict comprehensions можно делать вложенными. Попробуем аналогичным образом преобразовать ключи во вложенных словарях:

```
In [39]: london_co = {
...:     'r1' : {
...:         'hostname': 'london_r1',
...:         'location': '21 New Globe Walk',
...:         'vendor': 'Cisco',
...:         'model': '4451',
...:         'IOS': '15.4',
...:         'IP': '10.255.0.1'
...:     },
...:     'r2' : {
...:         'hostname': 'london_r2',
...:         'location': '21 New Globe Walk',
...:         'vendor': 'Cisco',
...:         'model': '4451',
...:         'IOS': '15.4',
...:         'IP': '10.255.0.2'
...:     },
...:     'sw1' : {
...:         'hostname': 'london_sw1',
...:         'location': '21 New Globe Walk',
...:         'vendor': 'Cisco',
...:         'model': '3850',
...:         'IOS': '3.6.XE',
...:         'IP': '10.255.0.101'
...:     }
...: }
```

```
In [40]: lower_london_co = {}
```

```
In [41]: for device, params in london_co.items():
...:     lower_london_co[device] = {}
...:     for key, value in params.items():
...:         lower_london_co[device][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'}}
```

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

Set comprehensions (генераторы множеств)

Генераторы множеств в целом аналогичны генераторам списков.

Например, надо получить множество с уникальными номерами VLAN'ов:

```
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}
```

Аналогичное решение, без использования 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}
```

Работа со словарями

При обработке вывода команд или конфигурации часто надо будет записать итоговые данные в словарь.

Не всегда очевидно как обрабатывать вывод команд и каким образом в целом подходить к разбору вывода на части. В этом подразделе рассматриваются несколько примеров, с возрастающим уровнем сложности.

Разбор вывода столбцами

В этом примере будет разбираться вывод команды `sh ip int br`. Из вывода команды нам надо получить соответствия имя интерфейса - IP-адрес. То есть имя интерфейса - это ключ словаря, а IP-адрес - значение. При этом, соответствие надо делать только для тех интерфейсов, у которых назначен IP-адрес.

Пример вывода команды `sh ip int br` (файл `sh_ip_int_br.txt`):

```
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
```

Файл `working_with_dict_example_1.py`:

```
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)
```

Команда `sh ip int br` отображает вывод столбцами. Значит нужные поля находятся в одной строке. Скрипт обрабатывает вывод построчно и каждую строку разбивает с помощью метода `split`.

Полученный в итоге список содержит столбцы вывода. Так как из всего вывода нужны только интерфейсы на которых настроен IP-адрес, выполняется проверка первого символа второго столбца: если первый символ число, значит на интерфейсе назначен адрес и эту строку надо обрабатывать.

В строке `interface, address, *other = line` выполняется распаковка переменных. В переменную `interface` попадет имя интерфейса, в `address` попадет IP-адрес, а в `other` все остальные поля.

Так как для каждой строки есть пара ключ и значение, они присваиваются в словарь:
`result[interface] = address`.

Результатом выполнения скрипта будет такой словарь (тут он разбит на пары ключ-значение для удобства, в реальном выводе скрипта словарь будет отображаться в одну строку):

```
{'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'}
```

Получение ключа и значения из разных строк вывода

Очень часто вывод команд выглядит таким образом, что ключ и значение находятся в разных строках. И надо придумать каким образом обрабатывать вывод, чтобы получить нужное соответствие.

Например, из вывода команды `sh ip interface` надо получить соответствие имя интерфейса - MTU (файл `sh_ip_interface.txt`):

```

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
  ...

```

Имя интерфейса находится в строке вида `Ethernet0/0 is up, line protocol is up`, а MTU в строке вида `MTU is 1500 bytes`.

Например, попробуем запоминать каждый раз интерфейс и выводить его значение, когда встречается MTU, вместе со значением MTU:

```

In [2]: with open('sh_ip_interface.txt') as f:
    ...:     for line in f:
    ...:         if 'line protocol' in line:
    ...:             interface = line.split()[0]
    ...:         elif 'MTU is' in line:
    ...:             mtu = line.split()[-2]
    ...:             print('{:15}{}'.format(interface, mtu))
    ...:
Ethernet0/0    1500
Ethernet0/1    1500
Ethernet0/2    1500
Ethernet0/3    1500
Loopback0      1514

```

Вывод организован таким образом, что всегда сначала идет строка с интерфейсом, а затем через несколько строк - строка с MTU. Если запоминать имя интерфейса каждый раз, когда оно встречается, то на момент когда встретится строка с MTU, последний запомненный интерфейс - это тот к которому относится MTU.

Теперь, если необходимо создать словарь с соответствием интерфейс - МТУ, достаточно записать значения на момент, когда был найден МТУ.

Файл working_with_dict_example_2.py:

```
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)
```

Результатом выполнения скрипта будет такой словарь (тут он разбит на пары ключ-значение для удобства, в реальном выводе скрипта словарь будет отображаться в одну строку):

```
{'Ethernet0/0': '1500',
 'Ethernet0/1': '1500',
 'Ethernet0/2': '1500',
 'Ethernet0/3': '1500',
 'Loopback0': '1514'}
```

Этот прием будет достаточно часто полезен, так как вывод команд, в целом, организован очень похожим образом.

Вложенный словарь

Если из вывода команды надо получить несколько параметров, очень удобно использовать словарь с вложенным словарем.

Например, из вывода sh ip interface надо получить два параметра: IP-адрес и МТУ. Для начала, вывод информации:

```
In [2]: with open('sh_ip_interface.txt') as f:
    ...:     for line in f:
    ...:         if 'line protocol' in line:
    ...:             interface = line.split()[0]
    ...:         elif 'Internet address' in line:
    ...:             ip_address = line.split()[-1]
    ...:         elif 'MTU' in line:
    ...:             mtu = line.split()[-2]
    ...:         print('{:15}{:17}{}'.format(interface, ip_address, mtu))
    ...:
Ethernet0/0      192.168.100.1/24 1500
Ethernet0/1      192.168.200.1/24 1500
Ethernet0/2      19.1.1.1/24       1500
Ethernet0/3      192.168.230.1/24 1500
Loopback0        4.4.4.4/32       1514
```

Тут используется такой же прием, как в предыдущем примере, но добавляется еще одна вложенность словаря:

```
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)
```

Каждый раз, когда встречается интерфейс, в словаре `result` создается ключ с именем интерфейса, которому соответствует пустой словарь. Эта заготовка нужна для того, чтобы на момент когда встретится IP-адрес или MTU можно было записать параметр во вложенный словарь соответствующего интерфейса.

Результатом выполнения скрипта будет такой словарь (тут он разбит на пары ключ-значение для удобства, в реальном выводе скрипта словарь будет отображаться в одну строку):

```
{'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'}}
```

ВЫВОД С ПУСТЫМИ ЗНАЧЕНИЯМИ

Иногда, в выводе будут попадаться секции с пустыми значениями. Например, в случае с выводом `sh ip interface`, могут попадаться интерфейсы, которые выглядят так:

```
Ethernet0/1 is up, line protocol is up
  Internet protocol processing disabled
Ethernet0/2 is administratively down, line protocol is down
  Internet protocol processing disabled
Ethernet0/3 is administratively down, line protocol is down
  Internet protocol processing disabled
```

Соответственно тут нет MTU или IP-адреса.

И, если выполнить предыдущий скрипт для файла с такими интерфейсами, результат будет таким (вывод для файла `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'}}
```

Если необходимо добавлять интерфейсы в словарь только, когда на интерфейсе назначен IP-адрес, надо перенести создание ключа с именем интерфейса на момент, когда встречается строка с IP-адресом (файл `working_with_dict_example_4.py`):

```
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)
```

В этом случае, результатом будет такой словарь:

```
{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}
```

Дополнительные материалы

Документация:

- [PEP 3132 -- Extended Iterable Unpacking](#)

Статьи:

- [List, Dict And Set Comprehensions By Example](#) - хорошая статья. И в конце статьи есть несколько упражнений (с ответами)
- [Python List Comprehensions: Explained Visually](#) - отличное объяснение list comprehensions, плюс видео

Stackoverflow:

- [Ответ на stackoverflow со множеством вариантов распаковки](#)

Часть II. Повторное использование кода

При написании кода достаточно часто часть действий повторяется. Это может быть небольшой блок на 3-5 строк, а может быть и достаточно большая последовательность действий.

Копировать код плохая затея. Так как, если потом понадобится обновить одну из копий, надо будет обновлять и другие.

Вместо этого, надо создать специальный блок кода с именем - функцию. И каждый раз, когда код надо повторить, достаточно вызвать функцию. Функция позволяет не только назвать какой-то блок кода, но и сделать его более абстрактным за счет параметров. Параметры дают возможность передавать разные исходные данные для выполнения функции. И, соответственно, получать разный результат, в зависимости от входящих параметров.

Созданию функций посвящён [девятый раздел](#). Кроме того, в [десяттом разделе](#) рассматриваются полезные встроенные функции.

После разделения кода на функции, достаточно быстро наступает момент, когда необходимо использовать функцию в другом скрипте. Конечно же, копирование функции так же неудобно, как и копирование обычного кода. Для повторного использования кода из другого скрипта Python используются модули.

[Одиннадцатый раздел](#) посвящён созданию собственных модулей, а в [двенадцатом разделе](#) рассматриваются полезные модули из стандартной библиотеки Python.

[Последний раздел](#) этой части посвящён итерируемым объектам, итераторам и генераторам.

ФУНКЦИИ

Функция - это блок кода, выполняющий определенные действия:

- у функции есть имя, с помощью которого можно запускать этот блок кода сколько угодно раз
 - запуск кода функции называется вызовом функции
- при создании функции, как правило, определяются параметры функции.
 - параметры функции определяют, какие аргументы функция может принимать
- функциям можно передавать аргументы
 - соответственно, код функции будет выполняться с учетом указанных аргументов

Зачем нужны функции?

Как правило, задачи, которые решает код, очень похожи и часто имеют что-то общее.

Например, при работе с конфигурационными файлами каждый раз надо выполнять такие действия:

- открытие файла
- удаление (или пропуск) строк, которые начинаются на знак восклицания (для Cisco)
- удаление (или пропуск) пустых строк
- удаление символов перевода строки в конце строк
- преобразование полученного результата в список

Дальше действия могут отличаться в зависимости от того, что нужно делать.

Часто получается, что есть кусок кода, который повторяется. Конечно, его можно копировать из одного скрипта в другой. Но это очень неудобно, так как при внесении изменений в код нужно будет обновить его во всех файлах, в которые он скопирован.

Гораздо проще и правильней вынести этот код в функцию (это может быть и несколько функций).

И тогда будет просто производиться вызов этой функции - в этом файле или каком-то другом.

В этом разделе рассматривается ситуация, когда функция находится в том же файле.

А в разделе [Модули](#) будет рассматриваться, как повторно использовать объекты, которые находятся в других скриптах.

Создание функций

Создание функции:

- функции создаются с помощью зарезервированного слова **def**
- за def следуют имя функции и круглые скобки
- внутри скобок могут указываться параметры, которые функция принимает
- после круглых скобок идет двоеточие и с новой строки, с отступом, идет блок кода, который выполняет функция
- первой строкой, дополнительно, может быть комментарий, так называемая **docstring**
- в функциях может использоваться оператор **return**
 - он используется для прекращения работы функции и выхода из нее
 - чаще всего, оператор return возвращает какое-то значение

Код функций, которые используются в этом подразделе, можно скопировать из файла `create_func.py`

Пример функции:

```
In [1]: def open_file( filename ):
...:     """Documentation string"""
...:     with open(filename) as f:
...:         print(f.read())
...:
```

Когда функция создана, она ещё ничего не выполняет. Только при вызове функции действия, которые в ней перечислены, будут выполняться. Это чем-то похоже на ACL в сетевом оборудовании: при создании ACL в конфигурации, он ничего не делает до тех пор, пока не будет куда-то применен.

Вызов функции

При вызове функции нужно указать её имя и передать аргументы, если нужно.

Параметры - это переменные, которые используются при создании функции.

Аргументы - это фактические значения (данные), которые передаются функции при вызове.

Эта функция в качестве аргумента ожидает имя файла и затем выводит содержимое файла:

```
In [2]: open_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
!

In [3]: open_file('ospf.txt')
router ospf 1
router-id 10.0.0.3
auto-cost reference-bandwidth 10000
network 10.0.1.0 0.0.0.255 area 0
network 10.0.2.0 0.0.0.255 area 2
network 10.1.1.0 0.0.0.255 area 0
```

Первая строка в определении функции - это docstring, строка документации. Это комментарий, который используется как описание функции. Его можно отобразить так:

```
In [4]: open_file.__doc__
Out[4]: 'Documentation string'
```

Лучше не лениться писать краткие комментарии, которые описывают работу функции. Например, описать, что функция ожидает на вход, какого типа должны быть аргументы и что будет на выходе. Кроме того, лучше написать пару предложений о том, что делает функция. Это очень поможет, когда через месяц-два Вы будете пытаться понять, что делает функция, которую Вы же написали.

Оператор return

Оператор **return** используется для прекращения работы функции, выхода из неё, и, как правило, возврата какого-то значения. Функция может возвращать любой объект Python.

Функция `open_file` в примере выше просто выводит на стандартный поток вывода содержимое файла. Но, чаще всего, от функции нужно получить результат её работы.

В данном случае, если присвоить вывод функции переменной `result`, результат будет таким:

```
In [5]: result = open_file('ospf.txt')
router ospf 1
router-id 10.0.0.3
auto-cost reference-bandwidth 10000
network 10.0.1.0 0.0.0.255 area 0
network 10.0.2.0 0.0.0.255 area 2
network 10.1.1.0 0.0.0.255 area 0

In [6]: print(result)
None
```

Переменная `result` равна `None`. Так получилось из-за того, что функция ничего не возвращает. Она просто выводит сообщение на стандартный поток вывода.

Для того, чтобы функция возвращала значение, которое потом можно, например, присвоить переменной, используется оператор `return`:

```
In [7]: def open_file( filename ):
...:     """Documentation string"""
...:     with open(filename) as f:
...:         return f.read()
...:

In [8]: result = open_file('r1.txt')

In [9]: print(result)
!
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
!
```

Теперь в переменной `result` находится содержимое файла.

В реальной жизни практически всегда функция будет возвращать какое-то значение. Вместе с тем можно использовать выражение `print`, чтобы дополнительно выводить какие-то сообщения.

Ещё один важный аспект работы оператора `return`: выражения, которые идут после `return`, не выполняются.

То есть, в функции ниже, строка "Done" не будет выводиться, так как она стоит после `return`:

```
In [10]: def open_file( filename ):
...:     print("Reading file", filename)
...:     with open(filename) as f:
...:         return f.read()
...:     print("Done")
...:

In [11]: result = open_file('r1.txt')
Reading file r1.txt
```

Пространства имен. Области видимости

У переменных в Python есть область видимости. В зависимости от места в коде, где переменная была определена, определяется и область видимости, то есть, где переменная будет доступна.

При использовании имен переменных в программе, Python каждый раз ищет, создает или изменяет эти имена в соответствующем пространстве имен. Пространство имен, которое доступно в каждый момент, зависит от области, в которой находится код.

У Python есть правило LEGB, которым он пользуется при поиске переменных.

Например, если внутри функции выполняется обращение к имени переменной, Python ищет переменную в таком порядке по областям видимости (до первого совпадения):

- L (local) - в локальной (внутри функции)
- E (enclosing) - в локальной области объемлющих функций (это те функции, внутри которых находится наша функция)
- G (global) - в глобальной (в скрипте)
- B (built-in) - в встроенной (зарезервированные значения Python)

Соответственно, есть локальные и глобальные переменные:

- локальные переменные:
 - переменные, которые определены внутри функции
 - эти переменные становятся недоступными после выхода из функции
- глобальные переменные
 - переменные, которые определены вне функции
 - эти переменные 'глобальны' только в пределах модуля
 - например, чтобы они были доступны в другом модуле, их надо импортировать

Пример локальной и глобальной переменной result:

```
In [1]: result = 'test string'

In [2]: def open_file( filename ):
...:     with open(filename) as f:
...:         result = f.read()
...:     return result
...:

In [3]: open_file('r1.txt')
Out[3]: '!\\nservice timestamps debug datetime msec localtime show-timezone year\\nservi
ce timestamps log datetime msec localtime show-timezone year\\nservice password-encrypt
ion\\nservice sequence-numbers\\n!\\nno ip domain lookup\\n!\\nip ssh version 2\\n!\\n'

In [4]: result
Out[4]: 'test string'
```

Обратите внимание, что переменная `result` по-прежнему осталась равной 'test string', несмотря на то, что внутри функции ей присвоено содержимое файла.

Параметры и аргументы функций

Цель создания функции, как правило, заключается в том, чтобы вынести кусок кода, который выполняет определенную задачу, в отдельный объект.

Это позволяет использовать этот кусок кода многократно, не создавая его заново в программе.

Как правило, функция должна выполнять какие-то действия с входящими значениями и на выходе выдавать результат.

При работе с функциями важно различать:

- **параметры** - это переменные, которые используются при создании функции.
- **аргументы** - это фактические значения (данные), которые передаются функции при вызове.

Для того, чтобы функция могла принимать входящие значения, ее нужно создать с параметрами (файл func_params_args.py):

```
In [1]: def delete_exclamation_from_cfg(in_cfg, out_cfg):  
...:     with open(in_cfg) as in_file:  
...:         result = in_file.readlines()  
...:     with open(out_cfg, 'w') as out_file:  
...:         for line in result:  
...:             if not line.startswith('!'):   
...:                 out_file.write(line)  
...:
```

В данном случае, у функции delete_exclamation_from_cfg два параметра: `in_cfg` и `out_cfg`.

Функция открывает файл `in_cfg`, читает содержимое в список; затем открывает файл `out_cfg` и записывает в него только те строки, которые не начинаются на знак восклицания.

В данном случае функция ничего не возвращает.

Файл `r1.txt` будет использоваться как первый аргумент (`in_cfg`):

```
In [2]: cat 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
!
```

Пример использования функции `delete_exclamation_from_cfg`:

```
In [3]: delete_exclamation_from_cfg('r1.txt', 'result.txt')
```

Файл `result.txt` выглядит так:

```
In [4]: 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
no ip domain lookup
ip ssh version 2
```

При таком определении функции надо обязательно передать оба аргумента.

Если передать только один аргумент, возникнет ошибка:

```
In [5]: delete_exclamation_from_cfg('r1.txt')
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-12-66ae381f1c4f> in <module>()
      1 delete_exclamation_from_cfg('r1.txt')

TypeError: delete_exclamation_from_cfg() missing 1 required positional argument: 'out_
cfg'
```

Аналогично, возникнет ошибка, если передать три и больше аргументов.

Типы параметров функции

При создании функции можно указать, какие аргументы нужно передавать обязательно, а какие нет.

Соответственно, функция может быть создана с параметрами:

- **обязательными**
- **необязательными** (опциональными, параметрами со значением по умолчанию)

Обязательные параметры

Обязательные параметры - определяют, какие аргументы нужно передать функции обязательно. При этом, их нужно передать ровно столько, сколько указано параметров функции (нельзя указать большее или меньшее количество аргументов)

Функция с обязательными параметрами (файл func_params_types.py):

```
In [1]: def cfg_to_list(cfg_file, delete_exclamation):
....:     result = []
....:     with open( cfg_file ) as f:
....:         for line in f:
....:             if delete_exclamation and line.startswith('!'):
....:                 pass
....:             else:
....:                 result.append(line.rstrip())
....:     return result
```

Функция cfg_to_list ожидает два аргумента: cfg_file и delete_exclamation.

Внутри она открывает файл cfg_file для чтения, проходится по всем строкам и, если аргумент delete_exclamation истина и строка начинается с восклицательного знака, строка пропускается. Оператор `pass` означает, что ничего не выполняется.

Во всех остальных случаях в строке справа удаляются символы перевода строки, и строка добавляется в словарь result.

Пример вызова функции:

```
In [2]: cfg_to_list('r1.txt', True)
Out[2]:
['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']
```

Так как аргументу `delete_exclamation` передано значение `True`, в итоговом словаре нет строк с восклицательными знаками.

Вызов функции со значением `False` для аргумента `delete_exclamation`:

```
In [3]: cfg_to_list('r1.txt', False)
Out[3]:
['!',
 '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',
 '!']
```

Необязательные параметры (параметры со значением по умолчанию)

При создании функции можно указывать значение по умолчанию для параметра (файл `func_params_types.py`):

```
In [4]: def cfg_to_list(cfg_file, delete_exclamation=True):
....:     result = []
....:     with open( cfg_file ) as f:
....:         for line in f:
....:             if delete_exclamation and line.startswith('!'):
....:                 pass
....:             else:
....:                 result.append(line.rstrip())
....:     return result
....:
```

Так как теперь у параметра `delete_exclamation` значение по умолчанию равно `True`, соответствующий аргумент можно не указывать при вызове функции, если значение по умолчанию подходит:

```
In [5]: cfg_to_list('r1.txt')
Out[5]:
['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']
```

Но можно и указать, если нужно поменять значение по умолчанию:

```
In [6]: cfg_to_list('r1.txt', False)
Out[6]:
[ '!',
  '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',
  '!']
```

Типы аргументов функции

При вызове функции аргументы можно передавать двумя способами:

- как **позиционные** - передаются в том же порядке, в котором они определены при создании функции. То есть, порядок передачи аргументов определяет, какое значение получит каждый
- как **ключевые** - передаются с указанием имени аргумента и его значения. В таком случае, аргументы могут быть указаны в любом порядке, так как их имя указывается явно.

Позиционные и ключевые аргументы могут быть смешаны при вызове функции. То есть, можно использовать оба способа при передаче аргументов одной и той же функции. При этом, сначала должны идти позиционные аргументы, а только потом - ключевые.

Посмотрим на разные способы передачи аргументов на примере функции cfg_to_list (файл func_args_types.py):

```
In [1]: def cfg_to_list(cfg_file, delete_exclamation):
....:     result = []
....:     with open( cfg_file ) as f:
....:         for line in f:
....:             if delete_exclamation and line.startswith('!'):
....:                 pass
....:             else:
....:                 result.append(line.rstrip())
....:     return result
....:
```

Позиционные аргументы

Позиционные аргументы при вызове функции надо передать в правильном порядке (поэтому они и называются позиционные)

```
In [2]: cfg_to_list('r1.txt', False)
Out[2]:
['!',
 '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',
 '!']
```

Если при вызове функции поменять аргументы местами, скорее всего, возникнет ошибка, в зависимости от конкретной функции.

Ключевые аргументы

Ключевые аргументы:

- передаются с указанием имени аргумента
- за счет этого они могут передаваться в любом порядке

Если передать оба аргумента как ключевые, можно передавать их в любом порядке:

```
In [4]: cfg_to_list(delete_exclamation=False, cfg_file='r1.txt')
Out[4]:
['!',
 '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',
 '!']
```

Но, обратите внимание, что всегда сначала должны идти позиционные аргументы, а затем ключевые.

Если сделать наоборот, возникнет ошибка:

```
In [5]: cfg_to_list(delete_exclamation=False, 'r1.txt')
  File "<ipython-input-3-8f3a3aa16a22>", line 1
    cfg_to_list(delete_exclamation=False, 'r1.txt')
                           ^
SyntaxError: positional argument follows keyword argument
```

Но в такой комбинации можно:

```
In [6]: cfg_to_list('r1.txt', delete_exclamation=True)
Out[6]:
['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']
```

В реальной жизни зачастую намного понятней и удобней указывать флаги, такие как `delete_exclamation`, как ключевой аргумент. Если задать хорошее название параметра за счет указания его имени, сразу будет понятно, что именно делает этот аргумент.

Например, в функции `cfg_to_list` понятно, что аргумент `delete_exclamation` приводит к удалению восклицательных знаков.

Аргументы переменной длины

Иногда необходимо сделать так, чтобы функция принимала не фиксированное количество аргументов, а любое. Для такого случая в Python можно создавать функцию со специальным параметром, который принимает аргументы переменной длины. Такой параметр может быть как ключевым, так и позиционным.

Даже если Вы не будете использовать этот прием в своих скриптах, есть большая вероятность, что Вы встретите его в чужом коде.

Позиционные аргументы переменной длины

Параметр, который принимает позиционные аргументы переменной длины, создается добавлением перед именем параметра звездочки. Имя параметра может быть любым, но, по договоренности, чаще всего, используют имя `*args`

Пример функции:

```
In [1]: def sum_arg(a, *args):
....:     print(a, args)
....:     return a + sum(args)
....:
```

Функция `sum_arg` создана с двумя параметрами:

- параметр `a`
 - если передается как позиционный аргумент, должен идти первым
 - если передается как ключевой аргумент, то порядок не важен
- параметр `*args` - ожидает аргументы переменной длины
 - сюда попадут все остальные аргументы в виде кортежа
 - эти аргументы могут отсутствовать

Вызов функции с разным количеством аргументов:

```
In [2]: sum_arg(1, 10, 20, 30)
1 (10, 20, 30)
Out[2]: 61
```

```
In [3]: sum_arg(1, 10)
1 (10,)
Out[3]: 11
```

```
In [4]: sum_arg(1)
1 ()
Out[4]: 1
```

Можно создать и такую функцию:

```
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
```

Ключевые аргументы переменной длины

Параметр, который принимает ключевые аргументы переменной длины, создается добавлением перед именем параметра двух звездочек. Имя параметра может быть любым, но, по договоренности, чаще всего, используют имя `**kwargs` (от keyword arguments).

Пример функции:

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

Функция `sum_arg` создана с двумя параметрами:

- параметр `a`
 - если передается как позиционный аргумент, должен идти первым
 - если передается как ключевой аргумент, то порядок не важен

- параметр `**kwargs` - Ожидает ключевые аргументы переменной длины
 - сюда попадут все остальные ключевые аргументы в виде словаря
 - эти аргументы могут отсутствовать

Вызов функции с разным количеством ключевых аргументов:

```
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
```

Обратите внимание, что, хотя `a` можно указывать как позиционный аргумент, нельзя указывать позиционный аргумент после ключевого:

```
In [11]: sum_arg(10,b=10,c=20,d=30)
10 {'c': 20, 'b': 10, 'd': 30}
Out[11]: 70

In [12]: sum_arg(b=10,c=20,d=30,10)
File "<ipython-input-14-71c121dc2cf7>", line 1
    sum_arg(b=10,c=20,d=30,10)
          ^
SyntaxError: positional argument follows keyword argument
```

Распаковка аргументов

В Python выражения `*args` и `**kwargs` позволяют выполнять ещё одну задачу - **распаковку аргументов**.

До сих пор мы вызывали все функции вручную. И, соответственно, передавали все нужные аргументы.

Но в реальной жизни, как правило, данные необходимо передавать в функцию программно. И часто данные идут в виде какого-то объекта Python.

Распаковка позиционных аргументов

Например, при форматировании строк часто надо передать методу `format` несколько аргументов. И часто эти аргументы уже находятся в списке или кортеже. Чтобы их передать методу `format`, приходится использовать индексы таким образом:

```
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
```

Но, вместо этого, можно воспользоваться распаковкой аргументов и сделать так:

```
In [4]: items = [1, 2, 3]

In [5]: print('One: {}, Two: {}, Three: {}'.format(*items))
One: 1, Two: 2, Three: 3
```

Еще один пример - функция `config_interface` (файл `func_args_unpacking.py`):

```
def config_interface(intf_name, ip_address, cidr_mask):
    interface = 'interface {}'
    no_shut = 'no shutdown'
    ip_addr = 'ip address {} {}'
    result = []
    result.append(interface.format(intf_name))
    result.append(no_shut)

    mask_bits = int(cidr_mask.split('/')[-1])
    bin_mask = '1'*mask_bits + '0'*(32-mask_bits)
    dec_mask = [str(int(bin_mask[i:i+8], 2)) for i in range(0, 25, 8)]
    dec_mask_str = '.'.join(dec_mask)

    result.append(ip_addr.format(ip_address, dec_mask_str))
    return result
```

Функция ожидает как аргумент:

- `intf_name` - имя интерфейса
- `ip_address` - IP-адрес
- `cidr_mask` - маску в формате CIDR (допускается и формат /24, и просто 24)

На выходе она выдает список строк для настройки интерфейса.

Например:

```
In [1]: config_interface('Fa0/1', '10.0.1.1', '/25')
Out[1]: ['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.128']

In [2]: config_interface('Fa0/3', '10.0.0.1', '/18')
Out[2]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.192.0']

In [3]: config_interface('Fa0/3', '10.0.0.1', '/32')
Out[3]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.255.255']

In [4]: config_interface('Fa0/3', '10.0.0.1', '/30')
Out[4]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.255.252']

In [5]: config_interface('Fa0/3', '10.0.0.1', '30')
Out[5]: ['interface Fa0/3', 'no shutdown', 'ip address 10.0.0.1 255.255.255.252']
```

Допустим, теперь нужно вызвать функцию и передать ей информацию, которая была получена из другого источника, к примеру, из БД.

Например, список `interfaces_info`, в котором находятся параметры для настройки интерфейсов:

```
In [6]: interfaces_info = [['Fa0/1', '10.0.1.1', '/24'],
....:                         ['Fa0/2', '10.0.2.1', '/24'],
....:                         ['Fa0/3', '10.0.3.1', '/24'],
....:                         ['Fa0/4', '10.0.4.1', '/24'],
....:                         ['Lo0', '10.0.0.1', '/32']]
```

Если пройтись по списку в цикле и передавать вложенный список как аргумент функции, возникнет ошибка:

```
In [7]: for info in interfaces_info:
....:     print(config_interface(info))
....:
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-5-f7d6a9d80d48> in <module>()
      1 for info in interfaces_info:
----> 2     print(config_interface(info))
      3
TypeError: config_interface() missing 2 required positional arguments: 'ip_address' and 'cidr_mask'
```

Ошибка вполне логичная: функция ожидает три аргумента, а ей передан 1 аргумент - список.

В такой ситуации пригодится распаковка аргументов. Достаточно добавить `*` перед передачей списка как аргумента, и ошибки уже не будет:

```
In [8]: 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 сам 'распакует' список `info` и передаст в функцию элементы списка как аргументы.

Таким же образом можно распаковывать и кортеж.

Распаковка ключевых аргументов

Аналогичным образом можно распаковывать словарь, чтобы передать его как ключевые аргументы.

Функция config_to_list (файл func_args_unpacking.py):

```
def config_to_list(cfg_file, delete_excl=True,
                   delete_empty=True, strip_end=True):
    result = []
    with open(cfg_file) as f:
        for line in f:
            if strip_end:
                line = line.rstrip()
            if delete_empty and not line:
                pass
            elif delete_excl and line.startswith('!'):
                pass
            else:
                result.append(line)
    return result
```

Функция берет файл с конфигурацией, убирает часть строк и возвращает остальные строки как список.

Пример использования:

```
In [9]: config_to_list('r1.txt')
Out[9]:
['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']
```

Список словарей cfg , в которых указано имя файла и все аргументы:

```
In [10]: cfg = [dict(cfg_file='r1.txt', delete_excl=True, delete_empty=True, strip_end=True),
           ....:         dict(cfg_file='r2.txt', delete_excl=False, delete_empty=True, strip_end=True),
           ....:         dict(cfg_file='r3.txt', delete_excl=True, delete_empty=False, strip_end=True),
           ....:         dict(cfg_file='r4.txt', delete_excl=True, delete_empty=True, strip_end=False)]
```

Если передать словарь функции config_to_list , возникнет ошибка:

```
In [11]: for d in cfg:  
....:     print(config_to_list(d))  
....:  
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-4-8d1e8defad71> in <module>()  
      1 for d in cfg:  
----> 2     print(config_to_list(d))  
      3  
  
<ipython-input-1-6337ba2bfe7a> in config_to_list(cfg_file, delete_excl, delete_empty,  
strip_end)  
      2             delete_empty=True, strip_end=True):  
      3     result = []  
----> 4     with open( cfg_file ) as f:  
      5         for line in f:  
      6             if strip_end:  
  
TypeError: expected str, bytes or os.PathLike object, not dict
```

Ошибка такая, так как все параметры, кроме имени файла, опциональны. И на стадии открытия файла возникает ошибка, так как вместо файла передан словарь.

Если добавить `**` перед передачей словаря функции, функция нормально отработает:

```
In [12]: for d in cfg:  
....:     print(config_to_list(**d))  
....:  
['service timestamps debug datetime msec localtime show-timezone year', 'service times  
tamps log datetime msec localtime show-timezone year', 'service password-encryption',  
'service sequence-numbers', 'no ip domain lookup', 'ip ssh version 2']  
[ '!', 'service timestamps debug datetime msec localtime show-timezone year', 'service  
timestamps log datetime msec localtime show-timezone year', 'service password-encrypti  
on', 'service sequence-numbers', '!', 'no ip domain lookup', '!', 'ip ssh version 2',  
'!']  
['service timestamps debug datetime msec localtime show-timezone year', 'service times  
tamps log datetime msec localtime show-timezone year', 'service password-encryption',  
'service sequence-numbers', ' ', ' ', ' ', 'ip ssh version 2', '']  
['service timestamps debug datetime msec localtime show-timezone year\n', 'service tim  
estamps log datetime msec localtime show-timezone year\n', 'service password-encryptio  
n\n', 'service sequence-numbers\n', 'no ip domain lookup\n', 'ip ssh version 2\n']
```

Python распаковывает словарь и передает его в функцию как ключевые аргументы.

Пример использования ключевых аргументов переменной длины и распаковки аргументов

С помощью аргументов переменной длины и распаковки аргументов можно передавать аргументы между функциями. Посмотрим на примере.

Функция config_to_list (файл kwargs_example.py):

```
def config_to_list(cfg_file, delete_excl=True,
                   delete_empty=True, strip_end=True):
    result = []
    with open(cfg_file) as f:
        for line in f:
            if strip_end:
                line = line.rstrip()
            if delete_empty and not line:
                pass
            elif delete_excl and line.startswith('!'):
                pass
            else:
                result.append(line)
    return result
```

Функция берет файл с конфигурацией, убирает часть строк и возвращает остальные строки как список.

Вызов функции в ipython:

```
In [1]: config_to_list('r1.txt')
Out[1]:
['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']
```

По умолчанию из конфигурации убираются пустые строки, перевод строки в конце строк и строки, которые начинаются на знак восклицания.

Вызов функции со значением `delete_empty=False`:

```
In [2]: config_to_list('r1.txt', delete_empty=False)
Out[2]:
['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']
```

Теперь пустые строки появились в списке.

Сделаем 'оберточную' функцию `clear_cfg_and_write_to_file`, которая берет файл конфигурации с помощью функции `config_to_list`, удаляет лишние строки и затем записывает строки в указанный файл.

Но, при этом, мы не хотим терять возможность управлять тем, какие строки будут отброшены. То есть, необходимо, чтобы функция `clear_cfg_and_write_to_file` поддерживала те же параметры, что и функция `config_to_list`.

Конечно, можно просто продублировать все параметры функции и передать их в функцию `config_to_list`:

```
def clear_cfg_and_write_to_file(cfg, to_file, delete_excl=True,
                                delete_empty=True, strip_end=True):

    cfg_as_list = config_to_list(cfg, delete_excl=delete_excl,
                                 delete_empty=delete_empty, strip_end=strip_end)
    with open(to_file, 'w') as f:
        f.write('\n'.join(cfg_as_list))
```

Но, если воспользоваться возможностью Python принимать аргументы переменной длины, можно сделать функцию `clear_cfg_and_write_to_file` такой:

```
def clear_cfg_and_write_to_file(cfg, to_file, **kwargs):
    cfg_as_list = config_to_list(cfg, **kwargs)
    with open(to_file, 'w') as f:
        f.write('\n'.join(cfg_as_list))
```

В функции `clear_cfg_and_write_to_file` явно прописаны её аргументы, а всё остальное попадет в переменную `kwargs`. Затем переменная `kwargs` передается как аргумент в функцию `config_to_list`. Но, так как переменная `kwargs` - это словарь, её надо распаковать при передаче функции `config_to_list`.

Так функция `clear_cfg_and_write_to_file` выглядит проще и понятней. И, главное, в таком варианте в функцию `config_to_list` можно добавлять аргументы без необходимости дублировать их в функции `clear_cfg_and_write_to_file`.

В этом примере `**kwargs` используется и для того, чтобы указать, что функция `clear_cfg_and_write_to_file` может принимать аргументы переменной длины, и для того, чтобы 'распаковать' словарь `kwargs`, когда мы передаем его в функцию `config_to_list`.

Дополнительные материалы

Документация:

- [Defining Functions](#)
- [Built-in Functions](#)
- [Sorting HOW TO](#)
- [Functional Programming HOWTO](#)
- [Функция range](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2а), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2а, 5.2б, 5.3, 5.3а. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2а, 5.2б, 5.3а.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 9.1

Создать функцию, которая генерирует конфигурацию для access-портов.

Функция ожидает, как аргумент, словарь access-портов, вида:

```
{'FastEthernet0/12':10,  
 'FastEthernet0/14':11,  
 'FastEthernet0/16':17,  
 'FastEthernet0/17':150}
```

Функция должна возвращать список всех портов в режиме access с конфигурацией на основе шаблона `access_template`.

В конце строк в списке не должно быть символа перевода строки.

Пример итогового списка (перевод строки после каждого элемента сделан для удобства чтения):

```
[  
'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',  
...]
```

Проверить работу функции на примере словаря `access_dict`.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
def generate_access_config(access):  
    """  
    access - словарь access-портов,  
    для которых необходимо сгенерировать конфигурацию, вида:  
    { 'FastEthernet0/12':10,  
      'FastEthernet0/14':11,  
      'FastEthernet0/16':17}  
  
    Возвращает список всех портов в режиме access  
    с конфигурацией на основе шаблона  
    """  
  
    access_template = ['switchport mode access',  
                      'switchport access vlan',  
                      'switchport nonegotiate',  
                      'spanning-tree portfast',  
                      'spanning-tree bpduguard enable']  
  
    access_dict = { 'FastEthernet0/12':10,  
                  'FastEthernet0/14':11,  
                  'FastEthernet0/16':17,  
                  'FastEthernet0/17':150 }
```

Задание 9.1а

Сделать копию скрипта задания 9.1.

Дополнить скрипт:

- ввести дополнительный параметр, который контролирует будет ли настроен `port-`

security

- имя параметра 'psecurity'
- по умолчанию значение False

Проверить работу функции на примере словаря access_dict, с генерацией конфигурации port-security и без.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
def generate_access_config(access):
    """
    access - словарь access-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
        { 'FastEthernet0/12':10,
          'FastEthernet0/14':11,
          'FastEthernet0/16':17 }

    psecurity - контролирует нужна ли настройка Port Security. По умолчанию значение False
    else
        - если значение True, то настройка выполняется с добавлением шаблона port_security
    security
        - если значение False, то настройка не выполняется

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """

    access_template = ['switchport mode access',
                      'switchport access vlan',
                      'switchport nonegotiate',
                      'spanning-tree portfast',
                      'spanning-tree bpduguard enable']

    port_security = ['switchport port-security maximum 2',
                     'switchport port-security violation restrict',
                     'switchport port-security']

    access_dict = { 'FastEthernet0/12':10,
                   'FastEthernet0/14':11,
                   'FastEthernet0/16':17,
                   'FastEthernet0/17':150 }
```

Задание 9.1b

Сделать копию скрипта задания 9.1а.

Изменить скрипт таким образом, чтобы функция возвращала не список команд, а словарь:

- ключи: имена интерфейсов, вида 'FastEthernet0/12'

- значения: список команд, который надо выполнить на этом интерфейсе:

```
[ 'switchport mode access',
  'switchport access vlan 10',
  'switchport nonegotiate',
  'spanning-tree portfast',
  'spanning-tree bpduguard enable']
```

Проверить работу функции на примере словаря `access_dict`, с генерацией конфигурации `port-security` и без.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
def generate_access_config(access):
    """
    access - словарь access-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
        { 'FastEthernet0/12':10,
          'FastEthernet0/14':11,
          'FastEthernet0/16':17 }

    psecurity - контролирует нужна ли настройка Port Security. По умолчанию значение False
    если значение True, то настройка выполняется с добавлением шаблона port_security
    - если значение False, то настройка не выполняется

    Функция возвращает словарь:
    - ключи: имена интерфейсов, вида 'FastEthernet0/1'
    - значения: список команд, который надо выполнить на этом интерфейсе
    """

    access_template = [ 'switchport mode access',
                        'switchport access vlan',
                        'switchport nonegotiate',
                        'spanning-tree portfast',
                        'spanning-tree bpduguard enable']

    port_security = [ 'switchport port-security maximum 2',
                      'switchport port-security violation restrict',
                      'switchport port-security']

    access_dict = { 'FastEthernet0/12':10,
                   'FastEthernet0/14':11,
                   'FastEthernet0/16':17,
                   'FastEthernet0/17':150 }
```

Задание 9.2

Создать функцию, которая генерирует конфигурацию для trunk-портов.

Параметр `trunk` - это словарь trunk-портов.

Словарь `trunk` имеет такой формат (тестовый словарь `trunk_dict` уже создан):

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

Функция должна возвращать список команд с конфигурацией на основе указанных портов и шаблона `trunk_template`.

В конце строк в списке не должно быть символа перевода строки.

Проверить работу функции на примере словаря `trunk_dict`.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
def generate_trunk_config(trunk):  
    ...  
    trunk - словарь trunk-портов для которых необходимо сгенерировать конфигурацию.  
  
    Возвращает список всех команд, которые были сгенерированы на основе шаблона  
    ...  
    trunk_template = ['switchport trunk encapsulation dot1q',  
                      'switchport mode trunk',  
                      'switchport trunk native vlan 999',  
                      'switchport trunk allowed vlan']  
  
    trunk_dict = { 'FastEthernet0/1':[10,20,30],  
                  'FastEthernet0/2':[11,30],  
                  'FastEthernet0/4':[17] }
```

Задание 9.2а

Сделать копию скрипта задания 9.2

Изменить скрипт таким образом, чтобы функция возвращала не список команд, а словарь:

- ключи: имена интерфейсов, вида 'FastEthernet0/1'
- значения: список команд, который надо выполнить на этом интерфейсе

Проверить работу функции на примере словаря `trunk_dict`.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```

def generate_trunk_config(trunk):
    """
    trunk - словарь trunk-портов,
    для которых необходимо сгенерировать конфигурацию, вида:
    { 'FastEthernet0/1':[10,20],
      'FastEthernet0/2':[11,30],
      'FastEthernet0/4':[17] }

    Возвращает словарь:
    - ключи: имена интерфейсов, вида 'FastEthernet0/1'
    - значения: список команд, который надо выполнить на этом интерфейсе
    """

    trunk_template = ['switchport trunk encapsulation dot1q',
                      'switchport mode trunk',
                      'switchport trunk native vlan 999',
                      'switchport trunk allowed vlan']

    trunk_dict = { 'FastEthernet0/1':[10,20,30],
                  'FastEthernet0/2':[11,30],
                  'FastEthernet0/4':[17] }

```

Задание 9.3

Создать функцию `get_int_vlan_map`, которая обрабатывает конфигурационный файл коммутатора и возвращает два объекта:

- словарь портов в режиме `access`, где ключи номера портов, а значения `access VLAN`:

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

- словарь портов в режиме `trunk`, где ключи номера портов, а значения список разрешенных `VLAN`:

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

Функция ожидает в качестве аргумента имя конфигурационного файла.

Проверить работу функции на примере файла `config_sw1.txt`

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 9.3а

Сделать копию скрипта задания 9.3.

Дополнить скрипт:

- добавить поддержку конфигурации, когда настройка access-порта выглядит так:

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

То есть, порт находится в VLAN 1

В таком случае, в словарь портов должна добавляться информация, что порт в VLAN 1

Пример словаря:

```
{'FastEthernet0/12':10,
'FastEthernet0/14':11,
'FastEthernet0/20':1 }
```

Функция ожидает в качестве аргумента имя конфигурационного файла.

Проверить работу функции на примере файла config_sw2.txt

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 9.4

Создать функцию, которая обрабатывает конфигурационный файл коммутатора и возвращает словарь:

- Все команды верхнего уровня (глобального режима конфигурации), будут ключами.
- Если у команды верхнего уровня есть подкоманды, они должны быть в значении у соответствующего ключа, в виде списка (пробелы вначале можно оставлять).
- Если у команды верхнего уровня нет подкоманд, то значение будет пустым списком

Функция ожидает в качестве аргумента имя конфигурационного файла.

Проверить работу функции на примере файла config_sw1.txt

При обработке конфигурационного файла, надо игнорировать строки, которые начинаются с '!', а также строки в которых содержатся слова из списка ignore.

Для проверки надо ли игнорировать строку, использовать функцию ignore_command.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

def ignore_command(command, ignore):
    ...
    Функция проверяет содержится ли в команде слово из списка ignore.

    command - строка. Команда, которую надо проверить
    ignore - список. Список слов

    Возвращает True, если в команде содержится слово из списка ignore, False - если не
    т
    ...
    return any(word in command for word in ignore)
```

Задание 9.4a

Задача такая же, как и задании 9.4. Проверить работу функции надо на примере файла config_r1.txt

Обратите внимание на конфигурационный файл. В нём есть разделы с большей вложенностью, например, разделы:

- interface Ethernet0/3.100
- router bgp 100

Надо чтобы функция config_to_dict обрабатывала следующий уровень вложенности. При этом, не привязываясь к конкретным разделам. Она должна быть универсальной, и сработать, если это будут другие разделы.

Если уровня вложенности два:

- то команды верхнего уровня будут ключами словаря,
- а команды подуровней - списками

Если уровня вложенности три:

- самый вложенный уровень должен быть списком,
- а остальные - словарями.

На примере interface Ethernet0/3.100

```
{'interface Ethernet0/3.100':{
    'encapsulation dot1Q 100':[],
    'xconnect 10.2.2.2 12100 encapsulation mpls':
        ['backup peer 10.4.4.4 14100',
         'backup delay 1 1']}}
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

```
def check_ignore(command, ignore):
```

```
    ...
```

Функция проверяет содержится ли в команде слово из списка ignore.

command - строка. Команду, которую надо проверить

ignore - список. Список слов

Возвращает True, если в команде содержится слово из списка ignore, False - если не

```
T
```

```
...
```

```
return any(word in command for word in ignore)
```

Полезные встроенные функции

В этом разделе рассматриваются такие функции:

- print
- range
- sorted
- enumerate
- zip
- all, any

Функция print

Функция print уже не раз использовалась в книге, но до сих пор не встречался ее полный синтаксис:

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

Функция print выводит все элементы, разделяя их значением sep, и завершает вывод значением end.

Все элементы, которые передаются как аргументы, конвертируются в строки:

```
In [4]: def f(a):
...:     return a
...:

In [5]: print(1, 2, f, range(10))
1 2 <function f at 0xb4de926c> range(0, 10)
```

Для функций f и range результат равнозначен применению str():

```
In [6]: str(f)
Out[6]: '<function f at 0xb4de926c>'

In [7]: str(range(10))
Out[7]: 'range(0, 10)'
```

sep

Параметр sep контролирует то, какой разделитель будет использоваться между элементами.

По умолчанию используется пробел:

```
In [8]: print(1, 2, 3)
1 2 3
```

Но можно изменить значение sep на любую другую строку:

```
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
```

Обратите внимание на то, что все аргументы, которые управляют поведением функции print, надо передавать как ключевые, а не позиционные.

В некоторых ситуациях функция print может заменить метод join:

```
In [12]: items = [1,2,3,4,5]

In [13]: print(*items, sep=', ')
1, 2, 3, 4, 5
```

end

Параметр end контролирует то, какое значение выводится после вывода всех элементов.

По умолчанию используется перевод строки:

```
In [19]: print(1,2,3)
1 2 3
```

Можно изменить значение end на любую другую строку:

```
In [20]: print(1,2,3, end='\n'+'-'*10)
1 2 3
-----
```

file

Параметр `file` контролирует то, куда выводятся значения функции `print`. По умолчанию все выводится на стандартный поток вывода - `sys.stdout`.

Но Python позволяет передавать `file` как аргумент любой объект с методом `write(string)`. За счет этого с помощью `print` можно записывать строки в файл:

```
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

По умолчанию при записи в файл или выводе на стандартный поток вывода вывод буферизируется. Функция `print` позволяет отключать буферизацию.

Это можно контролировать и в файле

Пример скрипта, который выводит число от 0 до 10 каждую секунду (файл `print_nums.py`):

```
import time

for num in range(10):
    print(num)
    time.sleep(1)
```

Попробуйте запустить скрипт и убедитесь, что числа выводятся раз в секунду.

Теперь, аналогичный скрипт, но числа будут выводиться в одной строке (файл `print_nums_oneline.py`):

```
import time

for num in range(10):
    print(num, end=' ')
    time.sleep(1)
```

Попробуйте запустить функцию. Числа не выводятся по одному в секунду, а выводятся все через 10 секунд.

Это связано с тем, что при выводе на стандартный поток вывода flush выполняется после перевода строки.

Чтобы скрипт отрабатывал как нужно, необходимо установить flush равным True (файл print_nums_oneline_fixed.py):

```
import time

for num in range(10):
    print(num, end=' ', flush=True)
    time.sleep(1)
```

Функция range

Функция range возвращает неизменяемую последовательность чисел в виде объекта range.

Синтаксис функции:

```
range(stop)
range(start, stop[, step])
```

Параметры функции:

- **start** - с какого числа начинается последовательность. По умолчанию - 0
- **stop** - до какого числа продолжается последовательность чисел. Указанное число не включается в диапазон
- **step** - с каким шагом растут числа. По умолчанию 1

Функция range хранит только информацию о значениях start, stop и step и вычисляет значения по мере необходимости. Это значит, что, независимо от размера диапазона, который описывает функция range, она всегда будет занимать фиксированный объем памяти.

Самый простой вариант range - передать только значение stop:

```
In [1]: range(5)
Out[1]: range(0, 5)

In [2]: list(range(5))
Out[2]: [0, 1, 2, 3, 4]
```

Если передаются два аргумента, то первый используется как start, а второй - как stop:

```
In [3]: list(range(1, 5))
Out[3]: [1, 2, 3, 4]
```

И, чтобы указать шаг последовательности, надо передать три аргумента:

```
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]
```

Функция range

С помощью `range` можно генерировать и убывающие последовательности чисел:

```
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]
```

Для получения убывающей последовательности надо использовать отрицательный шаг и соответственно указать `start` - большим числом, а `stop` - меньшим.

В убывающей последовательности шаг тоже может быть разным:

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

Функция поддерживает отрицательные значения `start` и `stop`:

```
In [9]: list(range(-10, 0, 1))
Out[9]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
```

```
In [10]: list(range(0, -10, -1))
Out[10]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Объект `range` поддерживает все операции, которые поддерживают последовательности в Python, кроме сложения и умножения.

Проверка, входит ли число в диапазон, который описывает `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
```

Начиная с версии Python 3.2, эта проверка выполняется за постоянное время ($O(1)$).

Можно получить конкретный элемент диапазона:

```
In [15]: nums = range(5)

In [16]: nums[0]
Out[16]: 0

In [17]: nums[-1]
Out[17]: 4
```

Range поддерживает срезы:

```
In [18]: nums = range(5)

In [19]: nums[1:]
Out[19]: range(1, 5)

In [20]: nums[:3]
Out[20]: range(0, 3)
```

Можно получить длину диапазона:

```
In [21]: nums = range(5)

In [22]: len(nums)
Out[22]: 5
```

А также минимальный и максимальный элемент:

```
In [23]: nums = range(5)

In [24]: min(nums)
Out[24]: 0

In [25]: max(nums)
Out[25]: 4
```

Кроме того, объект range поддерживает метод index:

```
In [26]: nums = range(1, 7)

In [27]: nums.index(3)
Out[27]: 2
```

Функция sorted

Функция `sorted()` возвращает новый отсортированный список, который получен из итерируемого объекта, который был передан как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.

Первый аспект, на который важно обратить внимание - `sorted` возвращает список.

Если сортировать список элементов, то возвращается новый список:

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']
In [2]: sorted(list_of_words)
Out[2]: ['', 'dict', 'list', 'one', 'two']
```

При сортировке кортежа также возвращается список:

```
In [3]: tuple_of_words = ('one', 'two', 'list', '', 'dict')
In [4]: sorted(tuple_of_words)
Out[4]: ['', 'dict', 'list', 'one', 'two']
```

Сортировка множества:

```
In [5]: set_of_words = {'one', 'two', 'list', '', 'dict'}
In [6]: sorted(set_of_words)
Out[6]: ['', 'dict', 'list', 'one', 'two']
```

Сортировка строки:

```
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']
```

Если передать `sorted` словарь, функция вернет отсортированный список ключей:

```
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

Флаг `reverse` позволяет управлять порядком сортировки. По умолчанию сортировка будет по возрастанию элементов.

Указав флаг `reverse`, можно поменять порядок:

```
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

С помощью параметра `key` можно указывать, как именно выполнять сортировку. Параметр `key` ожидает функцию, с помощью которой должно быть выполнено сравнение.

Например, таким образом можно отсортировать список строк по длине строки:

```
In [14]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [15]: sorted(list_of_words, key=len)
Out[15]: ['', 'one', 'two', 'list', 'dict']
```

Если нужно отсортировать ключи словаря, но при этом игнорировать регистр строк:

```
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']
```

Параметру `key` можно передавать любые функции, не только встроенные. Также тут удобно использовать анонимную функцию `lambda`.

С помощью параметра `key` можно сортировать объекты не по первому элементу, а по любому другому. Но для этого надо использовать или функцию `lambda`, или специальные функции из модуля `operator`.

Например, чтобы отсортировать список кортежей из двух элементов по второму элементу, надо использовать такой прием:

```
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

Иногда, при переборе объектов в цикле for, нужно не только получить сам объект, но и его порядковый номер. Это можно сделать, создав дополнительную переменную, которая будет расти на единицу с каждым прохождением цикла. Однако, гораздо удобнее это делать с помощью итератора `enumerate()`.

Базовый пример:

```
In [15]: list1 = ['str1', 'str2', 'str3']

In [16]: for position, string in enumerate(list1):
...:     print(position, string)
...:
0 str1
1 str2
2 str3
```

`enumerate()` умеет считать не только с нуля, но и с любого значение, которое ему указали после объекта:

```
In [17]: list1 = ['str1', 'str2', 'str3']

In [18]: for position, string in enumerate(list1, 100):
...:     print(position, string)
...:
100 str1
101 str2
102 str3
```

Иногда нужно проверить, что сгенерировал итератор, как правило, на стадии написания скрипта. Если необходимо увидеть содержимое, которое сгенерирует итератор, полностью, можно воспользоваться функцией list:

```
In [19]: list1 = ['str1', 'str2', 'str3']

In [20]: list(enumerate(list1, 100))
Out[20]: [(100, 'str1'), (101, 'str2'), (102, 'str3')]
```

Пример использования enumerate для ЕЕМ

В этом примере используется Cisco EEM. Если в двух словах, то EEM позволяет выполнять какие-то действия (action) в ответ на событие (event).

Выглядит applet EEM так:

```
event manager applet Fa0/1_no_shut
  event syslog pattern "Line protocol on Interface FastEthernet0/0, changed state to do
    wn"
    action 1 cli command "enable"
    action 2 cli command "conf t"
    action 3 cli command "interface fa0/1"
    action 4 cli command "no sh"
```

В EEM, в ситуации, когда действий выполнить нужно много, неудобно каждый раз набирать `action x cli command`. Плюс, чаще всего, уже есть готовый кусок конфигурации, который должен выполнить EEM.

С помощью простого скрипта Python можно сгенерировать команды EEM на основании существующего списка команд (файл enumerate_eem.py):

```
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()))
```

В данном примере командычитываются из файла, а затем к каждой строке добавляется приставка, которая нужна для EEM.

Файл с командами выглядит так (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
```

Вывод будет таким:

```
$ 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"
```

Функция zip

Функция zip():

- на вход функции передаются последовательности
- zip() возвращает итератор с кортежами, в котором n-ый кортеж состоит из n-ых элементов последовательностей, которые были переданы как аргументы
 - например, десятый кортеж будет содержать десятый элемент каждой из переданных последовательностей
- если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности
- порядок элементов соблюдается

| Так как zip - это итератор, для отображение его содержимого используется list()

Пример использования 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)]
```

Использование zip() со списками разной длины:

```
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)]
```

Использование zip для создания словаря:

Пример использования zip для создания словаря:

```
In [4]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
In [5]: d_values = ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255
.0.1']

In [6]: list(zip(d_keys,d_values))
Out[6]:
[('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('vendor', 'Cisco'),
 ('model', '4451'),
 ('IOS', '15.4'),
 ('IP', '10.255.0.1')]

In [7]: dict(zip(d_keys,d_values))
Out[7]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
In [8]: r1 = dict(zip(d_keys,d_values))

In [9]: r1
Out[9]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

В примере ниже есть отдельный список, в котором хранятся ключи, и словарь, в котором хранится в виде списка (чтобы сохранить порядок) информация о каждом устройстве.

Соберем их в словарь с ключами из списка и информацией из словаря data:

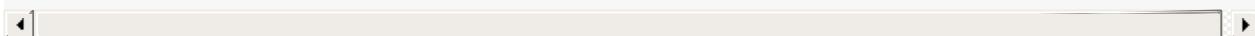
```
In [10]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

In [11]: data = {
....:     'r1': ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.1'],
....:     'r2': ['london_r2', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.2'],
....:     'sw1': ['london_sw1', '21 New Globe Walk', 'Cisco', '3850', '3.6.XE', '10.255.0.101']
....: }

In [12]: london_co = {}

In [13]: for k in data.keys():
....:     london_co[k] = dict(zip(d_keys, data[k]))
....:

In [14]: london_co
Out[14]:
{'r1': {'IOS': '15.4',
'IP': '10.255.0.1',
'hostname': 'london_r1',
'location': '21 New Globe Walk',
'model': '4451',
'vendor': 'Cisco'},
'r2': {'IOS': '15.4',
'IP': '10.255.0.2',
'hostname': 'london_r2',
'location': '21 New Globe Walk',
'model': '4451',
'vendor': 'Cisco'},
'sw1': {'IOS': '3.6.XE',
'IP': '10.255.0.101',
'hostname': 'london_sw1',
'location': '21 New Globe Walk',
'model': '3850',
'vendor': 'Cisco'}}
```



Функция all

Функция `all()` возвращает `True`, если все элементы истина (или объект пустой).

```
In [1]: all([False, True, True])
Out[1]: False

In [2]: all([True, True, True])
Out[2]: True

In [3]: all([])
Out[3]: True
```

Например, с помощью `all` можно проверить, все ли октеты в IP-адресе являются числами:

```
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
```

Функция any

Функция `any()` возвращает `True`, если хотя бы один элемент истина.

```
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
```

Например, с помощью `any`, можно заменить функцию `ignore_command`:

```
def ignore_command(command):
    """
    Функция проверяет содержится ли в команде слово из списка ignore.
    * command - строка. Команду, которую надо проверить
    * Возвращает True, если в команде содержится слово из списка ignore, False - если
нет
    """
    ignore = ['duplex', 'alias', 'Current configuration']

    ignore_command = False

    for word in ignore:
        if word in command:
            return True
    return ignore_command
```

На такой вариант:

```
def ignore_command(command):
    """
    Функция проверяет содержится ли в команде слово из списка ignore.
    command - строка. Команда, которую надо проверить
    Возвращает True, если в команде содержится слово из списка ignore, False - если не
т
    """
    ignore = ['duplex', 'alias', 'Current configuration']

    return any(word in command for word in ignore)
```

Модули

Модуль в Python - это обычный текстовый файл с кодом Python и расширением **.py**. Он позволяет логически упорядочить и сгруппировать код.

Разделение на модули может быть, например, по такой логике:

- разделение данных, форматирования и логики кода
- группировка функций и других объектов по функционалу

Модули хороши тем, что позволяют повторно использовать уже написанный код и не копировать его (например, не копировать когда-то написанную функцию).

Импорт модуля

В Python есть несколько способов импорта модуля:

- `import module`
- `import module as`
- `from module import object`
- `from module import *`

import module

Вариант `import module`:

```
In [1]: dir()
Out[1]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'quit']

In [2]: import os

In [3]: dir()
Out[3]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'os',
 'quit']
```

После импорта модуль `os` появился в выводе `dir()`. Это значит, что он теперь в текущем именном пространстве.

Чтобы вызвать какую-то функцию или метод из модуля `os`, надо указать `os.` и затем имя объекта:

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

Этот способ импорта хорош тем, что объекты модуля не попадают в именное пространство текущей программы. То есть, если создать функцию с именем `getlogin()`, она не будет конфликтовать с аналогичной функцией модуля `os`.

Если в имени файла содержится точка, стандартный способ импортирования не будет работать. Для таких случаев используется [другой способ](#).

import module as

Конструкция `import module as` позволяет импортировать модуль под другим именем (как правило, более коротким):

```
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--- 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'
```

from module import object

Вариант `from module import object` удобно использовать, когда из всего модуля нужны только одна-две функции:

```
In [1]: from os import getlogin, getcwd
```

Теперь эти функции доступны в текущем именном пространстве:

```
In [2]: dir()
Out[2]:
['In',
 'Out',
 ...
 'exit',
 'get_ipython',
 'getcwd',
 'getlogin',
 'quit']
```

Их можно вызывать без имени модуля:

```
In [3]: getlogin()
Out[3]: 'natasha'

In [4]: getcwd()
Out[4]: '/Users/natasha/Desktop/Py_net_eng/code_test'
```

from module import *

Вариант **from module import *** импортирует все имена модуля в текущее именное пространство:

```
In [1]: from os import *

In [2]: dir()
Out[2]:
['EX_CANTCREATE',
 'EX_CONFIG',
 ...
 'wait',
 'wait3',
 'wait4',
 'waitpid',
 'walk',
 'write']

In [3]: len(dir())
Out[3]: 218
```

В модуле `os` очень много объектов, поэтому вывод сокращен. В конце указана длина списка имен текущего именного пространства.

Такой вариант импорта лучше не использовать. При таком импорте по коду непонятно, что какая-то функция взята, например, из модуля `os`. Это заметно усложняет понимание кода.

Создание своих модулей

Так как модуль - это просто файл с расширение .py и кодом Python, мы можем легко создать несколько своих модулей.

Например, разделим скрипт из раздела [Совмещение for и if](#) на несколько частей: шаблоны портов, данные и формирование команд будут в разных файлах.

Файл `sw_int_templates.py`:

```
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']

l3int_template = ['no switchport', 'ip address']
```

Файл `sw_data.py`:

```
sw1_fast_int = {
    'access':{
        '0/12':'10',
        '0/14':'11',
        '0/16':'17'}}
```

Совмещаем всё вместе в файле `generate_sw_int_cfg.py`:

```
import sw_int_templates as sw_temp
from sw_data import sw1_fast_int

def generate_access_cfg(sw_dict):
    result = []
    for intf, vlan in sw_dict['access'].items():
        result.append('interface FastEthernet' + intf)
        for command in sw_temp.access_template:
            if command.endswith('access vlan'):
                result.append(' {} {}'.format(command, vlan))
            else:
                result.append(' {}'.format(command))
    return result

print('\n'.join(generate_access_cfg(sw1_fast_int)))
```

В первых двух строках импортируются объекты из других файлов:

- `import sw_int_templates` - импорт всего из файла
 - пример использования одного из шаблонов: `sw_int_templates.access_template`
- `from sw_data import sw1_fast_int` - из модуля `sw_data` импортируется только `sw1_fast_int`
 - при таком импорте можно напрямую обращаться к имени `sw1_fast_int`

Результат выполнения скрипта:

```
$ python generate_sw_int_cfg.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
```

```
if name == "main"
```

```
if __name__ == "__main__"
```

Достаточно часто скрипт может выполнятся и самостоятельно, и может быть импортирован как модуль другим скриптом.

Например, скрипт filter_functions.py содержит такой код:

```
from pprint import pprint

def filter_file_lines(filename, substring):
    result = []
    with open(filename) as f:
        for line in f:
            if substring in line:
                result.append(line)
    return result

pprint(filter_file_lines('config_r1.txt', 'ip address'))
```

В скрипте содержится одна функция, которая отбирает из файла только те строки, в которых содержится указанная подстрока.

Результат выполнения скрипта:

```
$ python filter_functions.py
[' ip address 10.1.1.1 255.255.255.255\n',
 ' ip address 10.0.13.1 255.255.255.0\n',
 ' no ip address\n',
 ' ip address 10.0.19.1 255.255.255.0\n',
 ' no ip address\n',
 ' no ip address\n']
```

Скрипт get_data.py импортирует функцию filter_file_lines из скрипта filter_functions.py и использует её для получения строк в которых содержится слово interface:

```
from filter_functions import filter_file_lines
from pprint import pprint

pprint(filter_file_lines('config_r1.txt', 'interface'))
```

Выполнение скрипта get_data.py выглядит таким образом:

```
if name == "main"
```

```
$ python get_data.py
[' ip address 10.1.1.1 255.255.255.255\n',
 ' ip address 10.0.13.1 255.255.255.0\n',
 ' no ip address\n',
 ' ip address 10.0.19.1 255.255.255.0\n',
 ' no ip address\n',
 ' no ip address\n']
['interface Loopback0\n',
 'interface Tunnel0\n',
 'interface Ethernet0/0\n',
 'interface Ethernet0/1\n',
 'interface Ethernet0/2\n',
 'interface Ethernet0/3\n',
 'interface Ethernet0/3.100\n',
 'interface Ethernet1/0\n',
 ' event neighbor-discovery interface regexp .*Ethernet.* cdp add\n',
 ' action 3.0 cli command "interface $_nd_local_intf_name"\n']
```

Полученный вывод содержит не только список со строками, в которых содержится слово `interface`, но и вывод из скрипта `filter_functions.py`.

Так происходит из-за того, что при импорте модуля, Python выполняет его.

Python выполняет весь модуль, независимо от того как именно импортируется модуль: `import module`, `from module import function` ИЛИ `from module import *`.

В Python есть специальный прием, который позволяет указать, что какой-то код должен выполняться, только когда файл запускается напрямую.

Файл `filter_functions.py`:

```
from pprint import pprint

def filter_file_lines(filename, substring):
    result = []
    with open(filename) as f:
        for line in f:
            if substring in line:
                result.append(line)
    return result

if __name__ == "__main__":
    pprint(filter_file_lines('config_r1.txt', 'ip address'))
```

Обратите внимание на запись:

```
if name == "main"
```

```
if __name__ == '__main__':
    pprint(filter_file_lines('config_r1.txt', 'ip address'))
```

Переменная `__name__` - это специальная переменная, которая будет равна `"__main__"`, если файл запускается как основная программа, и выставляется равной имени модуля, если модуль импортируется.

Таким образом, условие `if __name__ == '__main__'` проверяет, был ли файл запущен напрямую.

Теперь, при выполнении скрипта `get_data.py`, вывод такой:

```
$ python get_data.py
['interface Loopback0\n',
 'interface Tunnel0\n',
 'interface Ethernet0/0\n',
 'interface Ethernet0/1\n',
 'interface Ethernet0/2\n',
 'interface Ethernet0/3\n',
 'interface Ethernet0/3.100\n',
 'interface Ethernet1/0\n',
 'event neighbor-discovery interface regexp .*Ethernet.* cdp add\n',
 'action 3.0 cli command "interface $_nd_local_intf_name"\n']
```

Строки, которые находятся в блоке `if __name__ == '__main__'` не выполняются при импорте.

При выводе информации на стандартный поток вывода, проще всего заметить тот факт, что модуль выполняется при импорте, но гораздо больше проблем возникает когда, например, надо импортировать функцию из скрипта, который выполняет подключение к сотням устройств. В таком случае, во время импорта будет выполняться подключение, а только затем сможет выполниться скрипт, который импортировал другой модуль.

При создании функции, она не выполняется, поэтому в блок `if __name__ == '__main__'` выносится код, который вызывает функции.

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 11.1

Создать функцию `parse_cdp_neighbors`, которая обрабатывает вывод команды `show cdp neighbors`.

Функция ожидает, как аргумент, вывод команды одной строкой (а не имя файла).

Функция должна возвращать словарь, который описывает соединения между устройствами.

Например, если как аргумент был передан такой вывод:

```
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
```

Функция должна вернуть такой словарь:

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

Интерфейсы могут быть записаны с пробелом Fa 0/0 или без Fa0/0.

Проверить работу функции на содержимом файла `sw1_sh_cdp_neighbors.txt`

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 11.2

Для выполнения этого задания, должен быть установлен graphviz:

```
apt-get install graphviz
```

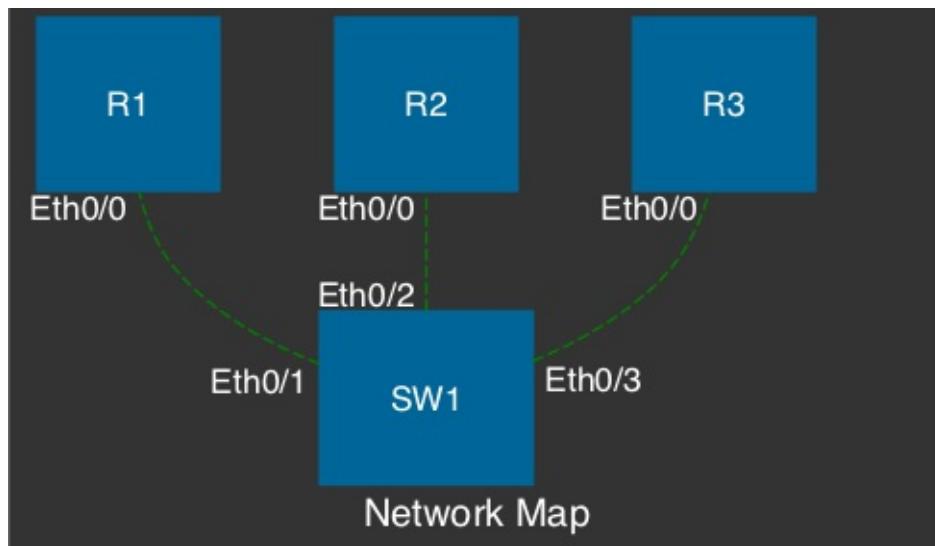
И модуль python для работы с graphviz:

```
pip install graphviz
```

С помощью функции `parse_cdp_neighbors` из задания 11.1 и функции `draw_topology` из файла `draw_network_graph.py`, сгенерировать топологию, которая соответствует выводу команды `sh cdp neighbor` в файле `sw1_sh_cdp_neighbors.txt`

Не копировать код функций `parse_cdp_neighbors` и `draw_topology`.

В итоге, должен быть сгенерировано изображение топологии. Результат должен выглядеть так же, как схема в файле `task_11_2_topology.svg`



При этом:

- Интерфейсы могут быть записаны с пробелом Fa 0/0 или без Fa0/0.
- Расположение устройств на схеме может быть другим
- Соединения должны соответствовать схеме

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 11.2а

Для выполнения этого задания, должен быть установлен graphviz:

```
apt-get install graphviz
```

И модуль python для работы с graphviz:

```
pip install graphviz
```

С помощью функции `parse_cdp_neighbors` из задания 11.1 и функции `draw_topology` из файла `draw_network_graph.py`, сгенерировать топологию, которая соответствует выводу команды `sh cdp neighbor` из файлов:

- `sh_cdp_n_sw1.txt`
- `sh_cdp_n_r1.txt`
- `sh_cdp_n_r2.txt`
- `sh_cdp_n_r3.txt`

Не копировать код функций `parse_cdp_neighbors` и `draw_topology`.

В итоге, должен быть сгенерировано изображение топологии. Результат должен выглядеть так же, как схема в файле `task_11_2a_topology.svg`



При этом:

- Интерфейсы могут быть записаны с пробелом Fa 0/0 или без Fa0/0.
- Расположение устройств на схеме может быть другим
- Соединения должны соответствовать схеме

Ограничение: Все задания надо выполнять используя только пройденные темы.

Полезные модули

В этом разделе описаны такие модули:

- subprocess
- os
- argparse
- ipaddress
- pprint
- tabulate

Модуль subprocess

Модуль subprocess позволяет создавать новые процессы.

При этом он может подключаться к [стандартным потокам ввода/вывода/ошибок](#) и получать код возврата.

С помощью subprocess можно, например, выполнять любые команды Linux из скрипта. И, в зависимости от ситуации, получать вывод или только проверять, что команда выполнилась без ошибок.

Синтаксис модуля subprocess изменился в Python 3.5. Если Вы используете Python версии 3.4 или ранее, используйте [синтаксис для Python 2.7](#)

Функция subprocess.run()

Функция `subprocess.run()` - основной способ работы с модулем subprocess.

Самый простой вариант использования функции - запуск её таким образом:

```
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
```

В переменной `result` теперь содержится специальный объект `CompletedProcess`. Из этого объекта можно получить информацию о выполнении процесса, например, о коде возврата:

```
In [3]: result
Out[3]: CompletedProcess(args='ls', returncode=0)

In [4]: result.returncode
Out[4]: 0
```

Код 0 означает, что программа выполнилась успешно.

Обратите внимание, что если необходимо вызвать команду с аргументами, её нужно передавать таким образом (как список):

```
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
```

При попытке выполнить команду с использованием wildcard выражений, например, использовать `*`, возникнет ошибка:

```
In [6]: result = subprocess.run(['ls', '-ls', '*md'])
ls: cannot access *md: No such file or directory
```

Чтобы вызывать команды, в которых используются wildcard выражения, нужно добавлять аргумент `shell` и вызывать команду таким образом:

```
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
```

Ещё одна особенность функции `run()` - она ожидает завершения выполнения команды. Если попробовать, например, запустить команду `ping`, то этот аспект будет заметен:

```
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
```

Получение результата выполнения команды

По умолчанию функция `run` возвращает результат выполнения команды на стандартный поток вывода.

Если нужно получить результат выполнения команды, надо добавить аргумент `stdout` и указать ему значение `subprocess.PIPE`:

```
In [9]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE)
```

Теперь можно получить результат выполнения команды таким образом:

```
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'
```

Обратите внимание на букву `b` перед строкой. Она означает, что модуль вернул вывод в виде байтовой строки.

Для перевода её в unicode есть два варианта:

- выполнить `decode` полученной строки
- указать аргумент `encoding`

Вариант с `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
```

Вариант с `encoding`:

```
In [12]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE, encoding='utf-8')

In [13]: print(result.stdout)
total 28
4 -rw-r--r-- 1 vagrant vagrant 56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant 277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:31 useful_modules
4 -rw-r--r-- 1 vagrant vagrant 49 Jun  7 19:35 version_control.md
```

Отключение вывода

Иногда достаточно получения кода возврата и нужно отключить вывод результата выполнения на стандартный поток вывода, и при этом сам результат не нужен.

Это можно сделать, передав функции run аргумент stdout со значением subprocess.DEVNULL:

```
In [14]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.DEVNULL)

In [15]: print(result.stdout)
None

In [16]: print(result.returncode)
0
```

Работа со стандартным потоком ошибок

Если команда была выполнена с ошибкой или не отработала корректно, вывод команды попадет на стандартный поток ошибок.

Получить этот вывод можно так же, как и стандартный поток вывода:

```
In [17]: result = subprocess.run(['ping', '-c', '3', '-n', 'a'], stderr=subprocess.PIPE,
encoding='utf-8')
```

Теперь в result.stdout пустая строка, а в result.stderr находится стандартный поток вывода:

```
In [18]: print(result.stdout)
None

In [19]: print(result.stderr)
ping: unknown host a

In [20]: print(result.returncode)
2
```

Примеры использования модуля

Пример использования модуля subprocess (файл subprocess_run_basic.py):

```
import subprocess

reply = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])

if reply.returncode == 0:
    print('Alive')
else:
    print('Unreachable')
```

Результат выполнения будет таким:

```
$ 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
```

То есть, результат выполнения команды выводится на стандартный поток вывода.

Функция `ping_ip` проверяет доступность IP-адреса и возвращает `True` и `stdout`, если адрес доступен, или `False` и `stderr`, если адрес недоступен (файл `subprocess_ping_function.py`):

```
import subprocess

def ping_ip(ip_address):
    """
    Ping IP address and return tuple:
    On success:
        * True
        * command output (stdout)
    On failure:
        * False
        * error output (stderr)
    """
    reply = subprocess.run(['ping', '-c', '3', '-n', ip_address],
                          stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE,
                          encoding='utf-8')
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stderr

print(ping_ip('8.8.8.8'))
print(ping_ip('a'))
```

Результат выполнения будет таким:

```
$ 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')
```

На основе этой функции, можно сделать функцию, которая будет проверять список IP-адресов и возвращать в результате выполнения два списка: доступные и недоступные адреса.

Это вынесено в задания к разделу

Если количество IP-адресов, которые нужно проверить, большое, можно использовать модуль `threading` или `multiprocessing`, чтобы ускорить проверку.

Модуль os

Модуль `os` позволяет работать с файловой системой, с окружением, управлять процессами.

Мы рассмотрим лишь несколько полезных возможностей. За более полным описанием возможностей модуля Вы можете обратиться к [документации](#) или [статье на сайте PyMOTW](#).

Модуль `os` позволяет создавать каталоги:

```
In [1]: import os  
  
In [2]: os.mkdir('test')  
  
In [3]: ls -ls  
total 0  
0 drwxr-xr-x 2 nata nata 68 Jan 23 18:58 test/
```

Кроме того, в модуле есть соответствующие проверки на существование. Например, если попробовать повторно создать каталог, возникнет ошибка:

```
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'
```

В таком случае пригодится проверка `os.path.exists`:

```
In [5]: os.path.exists('test')  
Out[5]: True  
  
In [6]: if not os.path.exists('test'):  
...:     os.mkdir('test')  
...:
```

Метод `listdir` позволяет посмотреть содержимое каталога:

```
In [7]: os.listdir('.')  
Out[7]: ['cover3.png', 'dir2', 'dir3', 'README.txt', 'test']
```

С помощью проверок `os.path.isdir` и `os.path.isfile` можно получить отдельно список файлов и список каталогов:

```
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']
```

Также в модуле есть отдельные методы для работы с путями:

```
In [12]: os.path.basename(file)  
Out[12]: 'README.md'  
  
In [13]: os.path.dirname(file)  
Out[13]: 'Programming/PyNEng/book/25_additional_info'  
  
In [14]: os.path.split(file)  
Out[14]: ('Programming/PyNEng/book/25_additional_info', 'README.md')
```

Модуль `ipaddress`

Модуль `ipaddress` может пригодиться для работы с IP-адресами.

С версии Python 3.3 модуль `ipaddress` входит в стандартную библиотеку Python.

`ipaddress.ip_address()`

Функция `ipaddress.ip_address()` позволяет создавать объект `IPv4Address` или `IPv6Address` соответственно.

IPv4 адрес:

```
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
```

У объекта есть несколько методов и атрибутов:

```
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
```

С помощью атрибутов `is_` можно проверить, к какому диапазону принадлежит адрес:

```
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
```

С полученными объектами можно выполнять различные операции:

```
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()

Функция `ipaddress.ip_network()` позволяет создать объект, который описывает сеть (IPv4 или IPv6).

Сеть IPv4:

```
In [20]: subnet1 = ipaddress.ip_network('80.0.1.0/28')
```

Как и у адреса, у сети есть различные атрибуты и методы:

```
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
```

Метод `hosts()` возвращает генератор, поэтому, чтобы посмотреть все хосты, надо применить функцию `list`:

```
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')]
```

Метод `subnets` позволяет разбивать на подсети. По умолчанию он разбивает сеть на две подсети:

```
In [27]: list(subnet1.subnets())
Out[27]: [IPv4Network('80.0.1.0/29'), IPv4Network(u'80.0.1.8/29')]
```

Но можно передать параметр `prefixlen_diff`, чтобы указать количество бит для подсетей:

```
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')]
```

Или с помощью параметра new_prefix просто указать, какая маска должна быть у подсетей:

```
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')]
```

По IP-адресам в сети можно проходиться в цикле:

```
In [31]: for ip in subnet1:
....:     print(ip)
....:
80.0.1.0
80.0.1.1
80.0.1.2
80.0.1.3
80.0.1.4
80.0.1.5
80.0.1.6
80.0.1.7
80.0.1.8
80.0.1.9
80.0.1.10
80.0.1.11
80.0.1.12
80.0.1.13
80.0.1.14
80.0.1.15
```

Или обращаться к конкретному адресу:

```
In [32]: subnet1[0]
Out[32]: IPv4Address('80.0.1.0')

In [33]: subnet1[5]
Out[33]: IPv4Address('80.0.1.5')
```

Таким образом можно проверять, находится ли IP-адрес в сети:

```
In [34]: ip1 = ipaddress.ip_address('80.0.1.3')

In [35]: ip1 in subnet1
Out[35]: True
```

`ipaddress.ip_interface()`

Функция `ipaddress.ip_interface()` позволяет создавать объект `IPv4Interface` или `IPv6Interface` соответственно.

Попробуем создать интерфейс:

```
In [36]: int1 = ipaddress.ip_interface('10.0.1.1/24')
```

Используя методы объекта `IPv4Interface`, можно получать адрес, маску или сеть интерфейса:

```
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')
```

Пример использования модуля

Так как в модуль встроены проверки корректности адресов, можно ими пользоваться, например, чтобы проверить, является ли адрес адресом сети или хоста:

```
In [40]: IP1 = '10.0.1.1/24'

In [41]: IP2 = '10.0.1.0/24'

In [42]: def check_if_ip_is_network(ip_address):
....:     try:
....:         ipaddress.ip_network(ip_address)
....:         return True
....:     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
```

Модуль argparse

argparse - это модуль для обработки аргументов командной строки.

Примеры того, что позволяет делать модуль:

- создавать аргументы и опции, с которыми может вызываться скрипт
- указывать типы аргументов, значения по умолчанию
- указывать, какие действия соответствуют аргументам
- выполнять вызов функции при указании аргумента
- отображать сообщения с подсказками по использованию скрипта

argparse не единственный модуль для обработки аргументов командной строки.

И даже не единственный такой модуль в стандартной библиотеке.

Мы будем рассматривать только argparse. Но, если Вы столкнетесь с необходимостью использовать подобные модули, обязательно посмотрите и на те модули, которые не входят в стандартную библиотеку Python.

Например, на [click](#).

[Очень хорошая статья](#), которая сравнивает разные модули обработки аргументов командной строки (рассматриваются argparse, click и docopt).

Пример скрипта ping_function.py:

```

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)

```

Создание парсера:

- `parser = argparse.ArgumentParser(description='Ping script')`

Добавление аргументов:

- `parser.add_argument('-a', action="store", dest="ip")`
 - аргумент, который передается после опции `-a`, сохранится в переменную `ip`
- `parser.add_argument('-c', action="store", dest="count", default=2, type=int)`
 - аргумент, который передается после опции `-c`, будет сохранен в переменную `count`, но, прежде, будет конвертирован в число. Если аргумент не было указан, по умолчанию будет значение 2

Строка `args = parser.parse_args()` указывается после того, как определены все аргументы.

После её выполнения в переменной `args` содержатся все аргументы, которые были переданы скрипту.

К ним можно обращаться, используя синтаксис `args.ip`.

Попробуем вызвать скрипт с разными аргументами.

Если переданы оба аргумента:

```
$ 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 - это объект, который возвращает метод `parse_args()`

Передаем только IP-адрес:

```
$ 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
```

Вызов скрипта без аргументов:

```
$ 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
```

Если бы функция была вызвана без аргументов, когда не используется argparse, возникла бы ошибка, что не все аргументы указаны.

Но, из-за argparse, фактически аргумент передается, только он равен `None`. Это видно в строке `Namespace(count=2, ip=None)`.

В таком скрипте, очевидно, IP-адрес необходимо указывать всегда. И в argparse можно указать, что аргумент является обязательным.

Надо изменить опцию `-a` : добавить в конце `required=True` :

```
parser.add_argument('-a', action="store", dest="ip", required=True)
```

Теперь, если вызвать скрипт без аргументов, вывод будет таким:

```
$ python ping_function.py
usage: ping_function.py [-h] -a IP [-c COUNT]
ping_function.py: error: the following arguments are required: -a
```

Теперь отображается понятное сообщение, что надо указать обязательный аргумент, и подсказка usage.

Также, благодаря argparse, доступен help:

```
$ python ping_function.py -h
usage: ping_function.py [-h] -a IP [-c COUNT]

Ping script

optional arguments:
  -h, --help  show this help message and exit
  -a IP
  -c COUNT
```

Обратите внимание, что в сообщении все опции находятся в секции optional arguments .

argparse сам определяет, что указаны опции, так как они начинаются с - и в имени только одна буква.

Зададим IP-адрес как позиционный аргумент.

Файл ping_function_ver2.py:

```

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_addresses),
                          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)

```

Теперь, вместо указания опции `-a`, можно просто передать IP-адрес.

Он будет автоматически сохранен в переменной `host`.

И автоматически считается обязательным.

То есть, теперь не нужно указывать `required=True` и `dest="ip"`.

Кроме того, в скрипте указаны сообщения, которые будут выводиться при вызове `help`.

Теперь вызов скрипта выглядит так:

```
$ 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
```

А сообщение help так:

```
$ 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
```

Вложенные парсеры

Рассмотрим один из способов организации более сложной иерархии аргументов.

Этот пример покажет больше возможностей argparse, но они этим не ограничиваются, поэтому, если Вы будете использовать argparse, обязательно посмотрите [документацию модуля](#) или [статью на РуМОТВ](#).

Файл parse_dhcp_snooping.py:

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

# Default values:
DFLT_DB_NAME = 'dhcp_snooping.db'
DFLT_DB_SCHEMA = 'dhcp_snooping_schema.sql'

def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))

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("Getting data from DB: {}".format(args.db_file))
        print("Request data for host(s) with {} {}".format((args.key, args.value)))
    elif args.key or args.value:
        print("Please give two or zero args\n")
        print(show_subparser_help('get'))
    else:
        print("Showing {} content...".format(args.db_file))

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers(title='subcommands',
                                    description='valid subcommands',
                                    help='description')

create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults( func=create )

add_parser = subparsers.add_parser('add', help='add data to db')
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
add_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db name')
add_parser.add_argument('-s', dest='sw_true', action='store_true',
                       help='add switch data if set, else add normal data')
add_parser.set_defaults( func=add )

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)
```

Теперь создается не только парсер, как в прошлом примере, но и вложенные парсеры. Вложенные парсеры будут отображаться как команды.

Но, фактически, они будут использоваться как обязательные аргументы.

С помощью вложенных парсеров создается иерархия аргументов и опций.

Аргументы, которые добавлены во вложенный парсер, будут доступны как аргументы этого парсера.

Например, в этой части создан вложенный парсер `create_db`, и к нему добавлена опция `-n`:

```
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')
```

Синтаксис создания вложенных парсеров и добавления к ним аргументов одинаков:

```
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)
```

Метод `add_argument` добавляет аргумент.

Тут синтаксис точно такой же, как и без использования вложенных парсеров.

В строке `create_parser.set_defaults(func=create)` указывается, что при вызове парсера `create_parser` будет вызвана функция `create`.

Функция `create` получает как аргумент все аргументы, которые были переданы.

И внутри функции можно обращаться к нужным:

```
def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))
```

Если вызвать `help` для этого скрипта, вывод будет таким:

```
$ 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
```

Обратите внимание, что каждый вложенный парсер, который создан в скрипте, отображается как команда в подсказке usage:

```
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...
```

У каждого вложенного парсера теперь есть свой 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
```

Кроме вложенных парсеров, в этом примере также есть несколько новых возможностей argparse.

metavar

В парсере create_parser используется новый аргумент - `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')
```

Аргумент `metavar` позволяет указывать имя аргумента для вывода в сообщении usage и 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
```

Посмотрите на разницу между опциями `-n` и `-s`:

- после опции `-n` в usage, и в help указывается имя, которое указано в параметре metavar
- после опции `-s` указывается имя переменной, в которую сохраняется значение

nargs

В парсере add_parser используется `nargs`:

```
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
```

`nargs` позволяет указать, что в этот аргумент должно попасть определенное количество элементов.

В этом случае все аргументы, которые были переданы скрипту после имени аргумента `filename`, попадут в список `nargs`.

Но должен быть передан хотя бы один аргумент.

Сообщение help в таком случае выглядит так:

```
$ 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
```

Если передать несколько файлов, они попадут в список.

А так как функция add просто выводит имена файлов, вывод получится таким:

```
$ 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` поддерживает такие значения:

- `N` - должно быть указанное количество аргументов. Аргументы будут в списке (даже если указан 1)
- `?` - 0 или 1 аргумент
- `*` - все аргументы попадут в список
- `+` - все аргументы попадут в список, но должен быть передан хотя бы один аргумент

choices

В парсере `get_parser` используется `choices`:

```
get_parser.add_argument('-k', dest="key",
                      choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                      help='host key (parameter) to search')
```

Для некоторых аргументов важно, чтобы значение было выбрано только из определенных вариантов.

Для таких случаев можно указывать `choices`.

Для этого парсера `help` выглядит так:

```
$ 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
```

А если выбрать неправильный вариант:

```
$ 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')
```

В данном примере важно указать варианты на выбор, так как затем на основании выбранного варианта генерируется SQL-запрос. И, благодаря `choices`, нет возможности указать какой-то параметр, кроме разрешенных.

Импорт парсера

В файле `parse_dhcp_snooping.py` последние две строки будут выполняться только в том случае, если скрипт был вызван как основной.

```
if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

А значит, если импортировать файл, эти строки не будут вызваны.

Попробуем импортировать парсер в другой файл (файл `call_pds.py`):

```
from parse_dhcp_snooping import parser

args = parser.parse_args()
args.func(args)
```

Вызов сообщения `help`:

```
$ 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
```

Вызов аргумента:

```
$ 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
```

Всё работает без проблем.

Передача аргументов вручную

Последняя особенность argparse - возможность передавать аргументы вручную.

Аргументы можно передать как список при вызове метода `parse_args()` (файл `call_pds2.py`):

```
from parse_dhcp_snooping import parser, get

args = parser.parse_args('add test.txt test2.txt'.split())
args.func(args)
```

Необходимо использовать метод `split()`, так как метод `parse_args` ожидает список аргументов.

Результат будет таким, как если бы скрипт был вызван с аргументами:

```
$ python call_pds2.py
Reading info from file(s)
test.txt, test2.txt

Adding data to db dhcp_snooping.db
```

Модуль tabulate

tabulate - это библиотека, которая позволяет красиво отображать табличные данные.

tabulate не входит в стандартную библиотеку Python, поэтому его нужно установить:

```
pip install tabulate
```

Модуль поддерживает такие типы табличных данных:

- список списков (в общем случае iterable of iterables)
- список словарей (или любой другой итерируемый объект со словарями). Ключи используются как имена столбцов
- словарь с итерируемыми объектами. Ключи используются как имена столбцов

Для генерации таблицы используется функция tabulate:

```
In [1]: from tabulate import tabulate

In [2]: sh_ip_int_br = [('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
...: ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
...: ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
...: ('Loopback0', '10.1.1.1', 'up', 'up'),
...: ('Loopback100', '100.0.0.1', 'up', 'up')]
...:

In [4]: print(tabulate(sh_ip_int_br))
-----
FastEthernet0/0  15.0.15.1    up    up
FastEthernet0/1  10.0.12.1    up    up
FastEthernet0/2  10.0.13.1    up    up
Loopback0        10.1.1.1    up    up
Loopback100     100.0.0.1   up    up
-----
```

headers

Параметр headers позволяет передавать дополнительный аргумент, в котором указаны имена столбцов:

```
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
```

Достаточно часто первый набор данных - это заголовки. Тогда достаточно указать `headers` равным "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
```

Если данные в виде списка словарей, надо указать `headers` равным "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
```

Стиль таблицы

tabulate поддерживает разные стили отображения таблицы.

Формат grid:

```
In [24]: print(tabulate(list_of_dict, headers='keys', tablefmt="grid"))
+-----+-----+-----+-----+
| Interface | IP      | Status | Protocol |
+-----+-----+-----+-----+
| FastEthernet0/0 | 15.0.15.1 | up     | up      |
+-----+-----+-----+-----+
| FastEthernet0/1 | 10.0.12.1 | up     | up      |
+-----+-----+-----+-----+
| FastEthernet0/2 | 10.0.13.1 | up     | up      |
+-----+-----+-----+-----+
| Loopback0      | 10.1.1.1  | up     | up      |
+-----+-----+-----+-----+
| Loopback100    | 100.0.0.1 | up     | up      |
+-----+-----+-----+-----+
```

Таблица в формате Markdown:

```
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      |
```

Таблица в формате HTML:

```
In [26]: print(tabulate(list_of_dict, headers='keys', tablefmt='html'))


| 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       |


```

Выравнивание столбцов

Можно указывать выравнивание для столбцов:

```
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 |
```

Обратите внимание, что тут не только столбцы отобразились с выравниванием по центру, но и соответственно изменился синтаксис Markdown.

Дополнительные материалы

- [Документация tabulate](#)

Статьи от автора tabulate:

- [Pretty printing tables in Python](#)
- [Tabulate 0.7.1 with LaTeX & MediaWiki tables](#)

Stackoverflow:

- [Printing Lists as Tabular Data](#). Обратите внимание на [ответ](#) - в нём указаны другие аналоги tabulate.

Модуль pprint

Модуль pprint позволяет красиво отображать объекты Python. При этом сохраняется структура объекта и отображение, которое выводит pprint, можно использовать для создания объекта.

Модуль pprint входит в стандартную библиотеку Python.

Самый простой вариант использования модуля - функция pprint.

Например, словарь с вложенными словарями отобразится так:

```
In [6]: london_co = {'r1': {'hostname': 'london_r1', 'location': '21 New Globe Wal
...: k', 'vendor': 'Cisco', 'model': '4451', 'IOS': '15.4', 'IP': '10.255.0.1'}
...: , 'r2': {'hostname': 'london_r2', 'location': '21 New Globe Walk', 'vendor
...: ': 'Cisco', 'model': '4451', 'IOS': '15.4', 'IP': '10.255.0.2'}, 'sw1': {''
...: hostname': 'london_sw1', 'location': '21 New Globe Walk', 'vendor': 'Cisco
...: ', 'model': '3850', 'IOS': '3.6.XE', 'IP': '10.255.0.101'}}
...:

In [7]: from pprint import pprint

In [8]: pprint(london_co)
{'r1': {'IOS': '15.4',
        'IP': '10.255.0.1',
        'hostname': 'london_r1',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'r2': {'IOS': '15.4',
        'IP': '10.255.0.2',
        'hostname': 'london_r2',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'sw1': {'IOS': '3.6.XE',
        'IP': '10.255.0.101',
        'hostname': 'london_sw1',
        'location': '21 New Globe Walk',
        'model': '3850',
        'vendor': 'Cisco'}}
```

Список списков:

```
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']]
```

Строка:

```
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 profi
le DMVPN\n'

In [19]: pprint(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')
```

Ограничение вложенности

У функции pprint есть дополнительный параметр depth, который позволяет ограничивать глубину отображения структуры данных.

Например, есть такой словарь:

```
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 igr-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']]}
```

Можно отобразить только ключи, указав глубину равной 1:

```
In [5]: pprint(result, depth=1)
{'interface Tunnel0': [...],
 'ip access-list standard LDP': [...],
 'router bgp 100': {...},
 'router ospf 1': [...]}
```

Скрытые уровни сложности заменяются

Если указать глубину равно 2, отобразится следующий уровень:

```
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 igrp-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 - это функция, которая отображает результат в виде строки. Ее удобно использовать, если необходимо записать структуру данных в какой-то файл, например, для логирования.

```
In [15]: from pprint import pformat

In [16]: formatted_result = pformat(result)

In [17]: print(formatted_result)
{'interface Tunnel0': [' ip unnumbered Loopback0',
                       '   tunnel mode mpls traffic-eng',
                       '   tunnel destination 10.2.2.2',
                       '   tunnel mpls traffic-eng priority 7 7',
                       '   tunnel mpls traffic-eng bandwidth 5000',
                       '   tunnel mpls traffic-eng path-option 10 dynamic',
                       '   no routing dynamic'],
 'ip access-list standard LDP': [' deny  10.0.0.0 0.0.255.255',
                                  ' permit 10.0.0.0 0.255.255.255'],
 'router bgp 100': {' address-family vpng4': [' neighbor 10.2.2.2 activate',
                                                '   neighbor 10.2.2.2 '
                                                'send-community both',
                                                '   exit-address-family'],
                     ' bgp bestpath igrp-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']}
```

Дополнительные материалы

Документация:

- [pprint — Data pretty printer](#)
- [PyMOTW. pprint — Pretty-Print Data Structures](#)

Дополнительные материалы

Стандартная библиотека модулей Python:

- [Индекс модулей](#)
- [Python 3 Module of the Week](#)

Документация:

- [Python tutorial. Modules](#)
- [os](#)
- [argparse](#)
- [subprocess](#)
- [ipaddress](#)

Видео:

- [David Beazley - Modules and Packages: Live and Let Die! - PyCon 2015](#)

argparse

- [Документация модуля](#)
- [Статья на PyMOTW](#)

tabulate

- [Документация tabulate](#)

Статьи от автора tabulate:

- [Pretty printing tables in Python](#)
- [Tabulate 0.7.1 with LaTeX & MediaWiki tables](#)

Stackoverflow:

- [Printing Lists as Tabular Data](#). Обратите внимание на [ответ](#) - в нем указаны другие аналоги tabulate.

pprint

- [pprint — Data pretty printer](#)
- [PyMOTW. pprint — Pretty-Print Data Structures](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 12.1

Создать функцию `check_ip_addresses`, которая проверяет доступность IP-адресов.

Функция ожидает как аргумент список IP-адресов. И возвращает два списка:

- список доступных IP-адресов
- список недоступных IP-адресов

Для проверки доступности IP-адреса, используйте `ping`. Адрес считается доступным, если на три ICMP-запроса пришли три ответа.

Задание 12.2

Функция `check_ip_addresses` из задания 12.1 принимает только список адресов, но было бы удобно иметь возможность указывать адреса с помощью диапазона, например, `192.168.100.1-10`.

Создать функцию `check_ip_availability`, которая проверяет доступность IP-адресов.

Функция ожидает как аргумент список IP-адресов.

IP-адреса могут быть в формате:

- `10.1.1.1`
- `10.1.1.1-10.1.1.10`
- `10.1.1.1-10`

Если адрес указан в виде диапазона, надо проверить доступность всех адресов диапазона включая последний.

Для упрощения задачи, можно считать, что в диапазоне всегда меняется только последний октет адреса.

Функция возвращает два списка:

- список доступных IP-адресов
- список недоступных IP-адресов

Для выполнения задачи можно воспользоваться функцией `check_ip_addresses` из задания 12.1.

Задание 12.3

Создать функцию `ip_table`, которая отображает таблицу доступных и недоступных IP-адресов.

Функция ожидает как аргументы два списка:

- список доступных IP-адресов
- список недоступных IP-адресов

Результат работы функции - вывод на стандартный поток вывода таблицы вида:

Reachable	Unreachable
10.1.1.1	10.1.1.7
10.1.1.2	10.1.1.8
	10.1.1.9

Функция не должна изменять списки, которые переданы ей как аргументы. То есть, до выполнения функции и после списки должны выглядеть одинаково.

Iterator, generator expression

В этом разделе рассматриваются:

- итерируемые объекты (iterable)
- итераторы (iterator)
- генераторные выражения (generator expression)

Итерируемый объект

Итерация - это общий термин, который описывает процедуру взятия элементов чего-то по очереди.

В более общем смысле, это последовательность инструкций, которая повторяется определенное количество раз или до выполнения указанного условия.

Итерируемый объект (`iterable`) - это объект, который способен возвращать элементы по одному. Кроме того, из объекта, из которого можно получить итератор.

Примеры итерируемых объектов:

- все последовательности: список, строка, кортеж
- словари
- файлы

В Python за получение итератора отвечает функция `iter()`:

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

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

Функция `iter()` отработает на любом объекте, у которого есть метод `__iter__` или метод `__getitem__`.

Метод `__iter__` возвращает итератор. Но если этого метода нет, функция `iter()` проверяет, нет ли метода `__getitem__` - метода, который позволяет получать элементы по индексу.

Если метод `__getitem__` есть, возвращается итератор, который проходится по элементам, используя индекс (начиная с 0).

На практике, использование метода `__getitem__` означает, что все последовательности элементов - это итерируемые объекты. Например, список, кортеж, строка. Хотя у этих типов данных есть и метод `__iter__`.

Итераторы

Итератор (iterator) - это объект, который возвращает свои элементы по одному за раз.

С точки зрения Python - это любой объект, у которого есть метод `__next__`. Этот метод возвращает следующий элемент, если он есть, или возвращает исключение `StopIteration`, когда элементы закончились.

Кроме того, итератор запоминает, на каком объекте он остановился в последнюю итерацию.

В Python у каждого итератора присутствует метод `__iter__` - то есть, любой итератор является итерируемым объектом. Этот метод просто возвращает сам итератор.

Пример создания итератора из списка:

```
In [3]: lista = [1, 2, 3]
In [4]: i = iter(lista)
```

Теперь можно использовать функцию `next()`, которая вызывает метод `__next__`, чтобы взять следующий элемент:

```
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:
```

После того, как элементы закончились, возвращается исключение `StopIteration`.

Для того, чтобы итератор снова начал возвращать элементы, его надо заново создать.

Аналогичные действия выполняются, когда цикл for проходится по списку:

```
In [9]: for item in lista:  
...:     print(item)  
...:  
1  
2  
3
```

Когда мы перебираем элементы списка, к списку сначала применяется функция `iter()`, чтобы создать итератор, а затем вызывается его метод `__next__` до тех пор, пока не возникнет исключение `StopIteration`.

Итераторы полезны тем, что они отдают элементы по одному. Например, при работе с файлом это полезно тем, что в памяти будет находиться не весь файл, а только одна строка файла.

Файл как итератор

Один из самых распространенных примеров итератора - файл.

Файл `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  
!
```

Если открыть файл обычной функцией `open`, мы получим объект, который представляет файл:

```
In [10]: f = open('r1.txt')
```

Этот объект является итератором, что можно проверить, вызвав метод `__next__`:

```
In [11]: f.__next__()  
Out[11]: '!\\n'  
  
In [12]: f.__next__()  
Out[12]: 'service timestamps debug datetime msec localtime show-timezone year\\n'
```

Аналогичным образом можно перебирать строки в цикле for:

```
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  
!
```

При работе с файлами, использование файла как итератора не просто позволяет перебирать файл построчно - в каждую итерацию загружена только одна строка. Это очень важно при работе с большими файлами на тысячи и сотни тысяч строк, например, с лог-файлами.

Поэтому при работе с файлами в Python чаще всего используется конструкция вида:

```
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 (генератор)

Генераторы - это специальный класс функций, который позволяет легко создавать свои итераторы. В отличии от обычных функций, генератор не просто возвращает значение и завершает работу, а возвращает итератор, который отдает элементы по одному.

Обычная функция завершает работу, если:

- встретилось выражение `return`
- закончился код функции (это срабатывает как выражение `return None`)
- возникло исключение

После выполнения функции управление возвращается, и программа выполняется дальше. Все аргументы, которые передавались в функцию, локальные переменные, все это теряется. Остается только результат, который вернула функция.

Функция может возвращать список элементов, несколько объектов или возвращать разные результаты в зависимости от аргументов, но она всегда возвращает какой-то один результат.

Генератор же генерирует значения. При этом значения возвращаются по запросу, и после возврата одного значения выполнение функции-генератора приостанавливается до запроса следующего значения. Между запросами генератор сохраняет свое состояние.

Python позволяет создавать генераторы двумя способами:

- генераторное выражение
- функция-генератор

Ниже пример генераторного выражения, а по функциям-генераторам - [отдельная заметка](#)

generator expression (генераторное выражение)

Генераторное выражение использует такой же синтаксис, как list comprehensions, но возвращает итератор, а не список.

Генераторное выражение выглядит точно так же, как list comprehensions, но используются круглые скобки:

```
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
```

Обратите внимание, что это не tuple comprehensions, а генераторное выражение.

Оно полезно в том случае, когда надо работать с большим итерируемым объектом или бесконечным итератором.

Дополнительные материалы

Документация Python:

- [Sequence types](#)
- [Iterator types](#)
- [Functional Programming HOWTO](#)

Статьи:

- [Iterables vs. Iterators vs. Generators](#)

Регулярные выражения

Регулярное выражение - это последовательность из обычных и специальных символов. Эта последовательность задает шаблон, который позже используется для поиска подстрок.

При работе с сетевым оборудованием регулярные выражения могут использоваться, например, для:

- получения информации из вывода команд show
- отбора части строк из вывода команд show, которые совпадают с шаблоном
- проверки, есть ли определенные настройки в конфигурации

Несколько примеров:

- обработав вывод команды show version, можно собрать информацию про версию ОС и uptime оборудования.
- получить из log-файла те строки, которые соответствуют шаблону.
- получить из конфигурации те интерфейсы, на которых нет описания (description)

Кроме того, в самом сетевом оборудовании регулярные выражения можно использовать для фильтрации вывода любых команд show.

В целом, использование регулярных выражений будет связано с получением части текста из большого вывода. Но это не единственное, в чем они могут пригодиться. Например, с помощью регулярных выражений можно выполнять замены в строках или разделение строки на части.

Эти области применения пересекаются с методами, которые применяются к строкам. И, если задача понятна и просто решается с помощью методов строк, лучше использовать их. Такой код будет проще понять и, кроме того, методы строк быстрее работают.

Но методы строк могут справиться не со всеми задачами или могут сильно усложнить решение задачи. В этом случае могут помочь регулярные выражения.

Синтаксис регулярных выражений

В Python для работы с регулярными выражениями используется модуль `re`. Соответственно, для начала работы с регулярными выражениями надо его импортировать.

В первой половине этого раздела для всех примеров будет использоваться функция `search`. А в следующих подразделах будут рассматриваться остальные функции модуля `re`.

Синтаксис функции `search` такой:

```
match = re.search(regex, string)
```

У функции `search` два обязательных параметра:

- `regex` - регулярное выражение
- `string` - строка, в которой ищется совпадение

Если совпадение было найдено, функция вернет специальный объект `Match`. Если же совпадения не было, функция вернет `None`.

При этом особенность функции `search` в том, что она ищет только первое совпадение. То есть, если в строке есть несколько подстрок, которые соответствуют регулярному выражению, `search` вернет только первое найденное совпадение.

Чтобы получить представление о регулярных выражениях, рассмотрим несколько примеров.

Самый простой пример регулярного выражения - подстрока:

```
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)
```

В этом примере:

- сначала импортируется модуль `re`
- затем идет пример строки `int_line`
- и в 3 строке функции `search` передается выражение, которое надо искать, и строка `int_line`, в которой ищется совпадение

В данном случае мы просто ищем, есть ли подстрока 'MTU' в строке int_line.

Если она есть, в переменной match будет находиться специальный объект Match:

```
In [4]: print(match)
<_sre.SRE_Match object; span=(2, 5), match='MTU'>
```

У объекта Match есть несколько методов, которые позволяют получать разную информацию о полученном совпадении. Например, метод group показывает, что в строке совпало с описанным выражением.

В данном случае это просто подстрока 'MTU':

```
In [5]: match.group()
Out[5]: 'MTU'
```

Если совпадения не было, в переменной match будет значение None:

```
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
```

Полностью возможности регулярных выражений проявляются при использовании специальных символов. Например, символ `\d` означает цифру, а `+` означает повторение предыдущего символа один или более раз. Если их совместить `\d+`, получится выражение, которое означает одну или более цифр.

Используя это выражение, можно получить часть строки, в которой описана пропускная способность:

```
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'
```

Особенно полезны регулярные выражения в получении определенных подстрок из строки. Например, необходимо получить VLAN, MAC и порты из вывода такого лог-сообщения:

```
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'
```

Это можно сделать с помощью такого регулярного выражения:

```
In [13]: re.search('Host (\S+) in vlan (\d+) is flapping between port (\S+) and port (  
\S+)', log2).groups()  
Out[13]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Метод `groups` возвращает только те части исходной строки, которые попали в круглые скобки. Таким образом, заключив часть выражения в скобки, можно указать, какие части строки надо запомнить.

Выражение `\d+` уже использовалось ранее - оно описывает одну или более цифр. А выражение `\S+` описывает все символы, кроме whitespace (пробел, таб и другие).

В следующих подразделах мы разберемся со специальными символами, которые используются в регулярных выражениях.

Если Вы знаете, что означают специальные символы в регулярных выражениях, можно пропустить следующий подраздел и сразу переключиться на подраздел о модуле `re`.

Наборы символов

В Python есть специальные обозначения для наборов символов:

- `\d` - любая цифра
- `\D` - любое нечисловое значение
- `\s` - whitespace (`\t\n\r\f\v`)
- `\S` - все, кроме whitespace
- `\w` - любая буква, цифра или нижнее подчеркивание
- `\W` - все, кроме букв, цифр или нижнего подчеркивания

Это не все наборы символов, которые поддерживает Python. Подробнее смотрите в [документации](#).

Наборы символов позволяют писать более короткие выражения без необходимости перечислять все нужные символы.

Например, получим время из строки лог-файла:

```
In [1]: log = '*Jul  7 06:15:18.695: %LINEPROTO-5-UPDOWN: Line protocol on Interface E
thernet0/3, changed state to down'

In [2]: re.search('\d\d:\d\d:\d\d', log).group()
Out[2]: '06:15:18'
```

Выражение `\d\d:\d\d:\d\d` описывает 3 пары чисел, разделенных двоеточиями.

Получение MAC-адреса из лог-сообщения:

```
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'
```

Выражение `\w\w\w\w\.\w\w\w\w\.\w\w\w\w` описывает 12 букв или цифр, которые разделены на три группы по четыре символа точками.

Группы символов очень удобны, но, пока что, приходится вручную указывать повторение символа. В следующем подразделе рассматриваются символы повторения, которые упростят описание выражений.

Символы повторения

- `regex+` - одно или более повторений предшествующего элемента
- `regex*` - ноль или более повторений предшествующего элемента
- `regex?` - ноль или одно повторение предшествующего элемента
- `regex{n}` - ровно n повторений предшествующего элемента
- `regex{n,m}` - от n до m повторений предшествующего элемента
- `regex{n,}` - n или более повторений предшествующего элемента

+

Плюс указывает, что предыдущее выражение может повторяться сколько угодно раз, но, как минимум, один раз.

Например, тут повторение относится к букве а:

```
In [1]: line = '100      aab1.a1a1.a5d3      FastEthernet0/1'

In [2]: re.search('a+', line).group()
Out[2]: 'aa'
```

А в этом выражении повторяется строка 'a1':

```
In [3]: line = '100      aab1.a1a1.a5d3      FastEthernet0/1'

In [4]: re.search('(a1)+', line).group()
Out[4]: 'a1a1'
```

В выражении `(a1)+` скобки используются для того, чтобы указать, что повторение относится к последовательности символов 'a1'.

IP-адрес можно описать выражением `\d+\.\d+\.\d+\.\d+`. Тут плюс используется, чтобы указать, что цифр может быть несколько. А также встречается выражение `\..`.

Оно необходимо из-за того, что точка является специальным символом (она обозначает любой символ). И чтобы указать, что нас интересует именно точка, надо ее экранировать - поместить перед точкой обратный слеш.

Используя это выражение, можно получить IP-адрес из строки `sh_ip_int_br`:

```
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'
```

Еще один пример выражения: `\d+\s+\S+` - оно описывает строку, в которой идут цифры, пробел (whitespace), не whitespace символы, то есть, все, кроме пробела, таба и других whitespace символов. С его помощью можно получить VLAN и MAC-адрес из строки:

```
In [7]: line = '1500      aab1.a1a1.a5d3      FastEthernet0/1'
In [8]: re.search('\d+\s+\S+', line).group()
Out[8]: '1500      aab1.a1a1.a5d3'
```

*

Звездочка указывает, что предыдущее выражение может повторяться 0 или более раз.

Например, если звездочка стоит после символа, она означает повторение этого символа.

Выражение `ba*` означает b, а затем ноль или более повторений a:

```
In [9]: line = '100      a011.baaa.a5d3      FastEthernet0/1'
In [10]: re.search('ba*', line).group()
Out[10]: 'baaa'
```

Если в строке line до подстроки baaa встретится b, то совпадением будет b:

```
In [11]: line = '100      ab11.baaa.a5d3      FastEthernet0/1'
In [12]: re.search('ba*', line).group()
Out[12]: 'b'
```

Допустим, необходимо написать регулярное выражение, которое описывает email'ы двух форматов: user@example.com и user.test@example.com. То есть, в левой части адреса может быть или одно слово, или два слова, разделенные точкой.

Первый вариант на примере адреса без точки:

```
In [13]: email1 = 'user1@gmail.com'
```

Этот адрес можно описать таким выражением `\w+@\w+\.\w+` :

```
In [14]: re.search('\w+@\w+\.\w+', email1).group()
Out[14]: 'user1@gmail.com'
```

Но такое выражение не подходит для email с точкой:

```
In [15]: email2 = 'user2.test@gmail.com'

In [16]: re.search('\w+@\w+\.\w+', email2).group()
Out[16]: 'test@gmail.com'
```

Регулярное выражение для адреса с точкой:

```
In [17]: re.search('\w+\.\w+@\w+\.\w+', email2).group()
Out[17]: 'user2.test@gmail.com'
```

Чтобы описать оба варианта адресов, надо указать, что точка в адресе опциональна:

```
'\w+\.*\w+@\w+\.\w+'
```

Такое регулярное выражение описывает оба варианта:

```
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'
```

?

В последнем примере регулярное выражение указывает, что точка опциональна. Но, в то же время, указывает и то, что точка может появиться много раз.

В этой ситуации логичней использовать знак вопроса. Он обозначает ноль или одно повторение предыдущего выражения или символа. Теперь регулярное выражение выглядит так `\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}

С помощью фигурных скобок можно указать, сколько раз должно повторяться предшествующее выражение.

Например, выражение `\w{4}\.\w{4}\.\w{4}` описывает 12 букв или цифр, которые разделены на три группы по четыре символа точками. Таким образом можно получить MAC-адрес:

```
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'
```

В фигурных скобках можно указывать и диапазон повторений. Например, попробуем получить все номера VLAN'ов из строки `mac_table`:

```
In [26]: mac_table = '''
...: sw1#sh mac address-table
...:          Mac Address Table
...: -----
...:
...:   Vlan      Mac Address          Type      Ports
...:   ----      -----              -----      -----
...:   100      a1b2.ac10.7000      DYNAMIC    Gi0/1
...:   200      a0d4.cb20.7000      DYNAMIC    Gi0/2
...:   300      acb4.cd30.7000      DYNAMIC    Gi0/3
...:   1100     a2bb.ec40.7000      DYNAMIC    Gi0/4
...:   500      aa4b.c550.7000      DYNAMIC    Gi0/5
...:   1200     a1bb.1c60.7000      DYNAMIC    Gi0/6
...:   1300     aa0b.cc70.7000      DYNAMIC    Gi0/7
...: '''

In [27]: mac_table
```

Так как `search` ищет только первое совпадение, в выражение `\d{1,4}` попадет номер VLAN:

```
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
```

Выражение `\d{1,4}` описывает от одной до четырех цифр.

Обратите внимание, что в выводе команды нет первого VLAN. Такой результат получился из-за того, что в имени коммутатора есть цифра и она совпала с выражением.

Чтобы исправить это, достаточно дополнить выражение и указать, что после цифр должен идти хотя бы один пробел:

```
In [28]: for line in mac_table.split('\n'):
...:     match = re.search('\d{1,4} +', line)
...:     if match:
...:         print('VLAN: ', match.group())
...:
VLAN: 100
VLAN: 200
VLAN: 300
VLAN: 1100
VLAN: 500
VLAN: 1200
VLAN: 1300
```

Специальные символы

- `.` - любой символ, кроме символа новой строки
- `^` - начало строки
- `$` - конец строки
- `[abc]` - любой символ в скобках
- `[^abc]` - любой символ, кроме тех, что в скобках
- `a|b` - элемент a или b
- `(regex)` - выражение рассматривается как один элемент. Кроме того, подстрока, которая совпала с выражением, запоминается

•

Точка обозначает любой символ.

Чаще всего, точка используется с символами повторения `+` и `*`, чтобы указать, что между определенными выражениями могут находиться любые символы.

Например, с помощью выражения `Interface.+Port ID.+` можно описать строку с интерфейсами в выводе `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'
```

В результат попала только одна строка, так как точка обозначает любой символ, кроме символа перевода строки. Кроме того, символы повторения `+` и `*` по умолчанию захватывают максимально длинную строку. Этот аспект рассматривается в подразделе "Жадность символов повторения".

•

Символ `^` означает начало строки. Выражению `^\d+` соответствует подстрока:

```
In [3]: line = "100      aa12.35fe.a5d3      FastEthernet0/1"

In [4]: re.search('^\d+', line).group()
Out[4]: '100'
```

Символы с начала строки и до решетки (включая решетку):

```
In [5]: prompt = 'SW1#show cdp neighbors detail'

In [6]: re.search('^.+#', prompt).group()
Out[6]: 'SW1#'
```

`$`

Символ `$` обозначает конец строки.

Выражение `\s+\$` описывает любые символы, кроме whitespace в конце строки:

```
In [7]: line = "100      aa12.35fe.a5d3      FastEthernet0/1"

In [8]: re.search('\s+$', line).group()
Out[8]: 'FastEthernet0/1'
```

`[]`

Символы, которые перечислены в квадратных скобках, означают, что любой из этих символов будет совпадением. Таким образом можно описывать разные регистры:

```
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'
```

С помощью квадратных скобок можно указать, какие символы могут встречаться на конкретной позиции. Например, выражение `^.+[>#]` описывает символы с начала строки и до решетки или знака больше (включая их). С помощью такого выражения можно получить имя устройства:

```
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#
```

В квадратных скобках можно указывать диапазоны символов. Например, таким образом можно указать, что нас интересует любая цифра от 0 до 9:

```
In [14]: line = "100      aa12.35fe.a5d3      FastEthernet0/1"

In [15]: re.search('[0-9]+', line).group()
Out[15]: '100'
```

Аналогичным образом можно указать буквы:

```
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'
```

В квадратных скобках можно указывать несколько диапазонов:

```
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'
```

Выражение `[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+` описывает три группы символов, разделенных точкой. Символами в каждой группе могут быть буквы a-f или цифры 0-9. Это выражение описывает MAC-адрес.

Еще одна особенность квадратных скобок - специальные символы внутри квадратных скобок теряют свое специальное значение и обозначают просто символ. Например, точка внутри квадратных скобок будет обозначать точку, а не любой символ.

Выражение `[a-f0-9]+[./][a-f0-9]+` описывает три группы символов:

1. буквы a-f или цифры от 0 до 9
2. точка или слеш
3. буквы a-f или цифры от 0 до 9

Для строки `line` совпадением будет такая подстрока:

```
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'
```

Если после открывающейся квадратной скобки указан символ `^`, совпадением будет любой символ, кроме указанных в скобках:

```
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      '
```

В данном случае выражение описывает все, кроме букв.

|

Вертикальная черта работает как 'или':

```
In [25]: line = "100      aa12.35fe.a5d3      FastEthernet0/1"
In [26]: re.search('Fast|0/1', line).group()
Out[26]: 'Fast'
```

Обратите внимание на то, как срабатывает `|` - `Fast` и `0/1` воспринимаются как целое выражение. То есть, в итоге выражение означает, что мы ищем `Fast` или `0/1`, а не то, что мы ищем `Fas`, затем `t` или `0` и `0/1`.

()

Скобки используются для группировки выражений. Как и в математических выражениях, с помощью скобок можно указать, к каким элементам применяется операция.

Например, выражение `[0-9]([a-f]|[0-9])[0-9]` описывает три символа: цифра, потом буква или цифра и цифра:

```
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'
```

Скобки позволяют указывать, какое выражение является одним целым. Это особенно полезно при использовании символов повторения:

```
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'
```

Скобки позволяют не только группировать выражения. Стока, которые совпала с выражением в скобках, запоминается. Ее можно получить отдельно с помощью специальных методов `groups` и `group(n)`. Это рассматривается в подразделе "Группировка выражений".

Жадность символов повторения

По умолчанию символы повторения в регулярных выражениях жадные (greedy). Это значит, что результирующая подстрока, которая соответствует шаблону, будет наиболее длинной.

Пример жадного поведения:

```
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>'
```

То есть, в данном случае выражение захватило максимально возможный кусок символов, заключенный в <>.

Если нужно отключить жадность, достаточно добавить знак вопроса после символов повторения:

```
In [5]: line = '<text line> some text>'

In [6]: match = re.search('<.*?>', line)

In [7]: match.group()
Out[7]: '<text line>'
```

Зачастую жадность наоборот полезна. Например, без отключения жадности последнего плюса, выражение `\d+\s+\S+` описывает такую строку:

```
In [8]: line = '1500      aab1.a1a1.a5d3      FastEthernet0/1'

In [9]: re.search('\d+\s+\S+', line).group()
Out[9]: '1500      aab1.a1a1.a5d3'
```

Символ `\s` обозначает все, кроме whitespace. Поэтому выражение `\s+` с жадным символом повторения описывает максимально длинную строку до первого whitespace символа. В данном случае - до первого пробела.

Но если отключить жадность, результат будет таким:

```
In [10]: re.search('\d+\s+\S+?', line).group()  
Out[10]: '1500      a'
```

Группировка выражений

Группировка выражений указывает, что последовательность символов надо рассматривать как одно целое. Но это не единственное преимущество группировки.

Кроме этого, с помощью групп можно получать только определенную часть строки, которая была описана выражением. Это очень полезно в ситуациях, когда надо описать строку достаточно подробно, чтобы отобрать нужные строки, но, в то же время, из самой строки надо получить только определенное значение.

Например, из log-файла надо отобрать строки, в которых встречается "%SW_MTM-4-MACFLAP_NOTIF", а затем из каждой такой строки получить MAC-адрес, VLAN и интерфейсы. В этом случае регулярное выражение просто должно описывать строку, а все части строки, которые надо получить в результате, просто заключаются в скобки.

В Python есть два варианта использования групп:

- Нумерованные группы
- Именованные группы

Нумерованные группы

Группа определяется помещением выражения в круглые скобки `()`.

Внутри выражения группы нумеруются слева направо, начиная с 1.

Затем к группам можно обращаться по номерам и получать текст, который соответствует выражению в группе.

Пример использования групп:

```
In [8]: line = "FastEthernet0/1      10.0.12.1      YES manual up
          up"
In [9]: match = re.search('(\S+)\s+(\w+)\s+.*', line)
```

В данном примере указаны две группы:

- первая группа - любые символы, кроме whitespaces
- вторая группа - любая буква или цифра (символ `\w`) или точка

Вторую группу можно было описать так же, как и первую. Другой вариант сделан просто для примера

Теперь можно обращаться к группам по номеру. Группа 0 - это строка, которая соответствует всему шаблону:

```
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'
```

При необходимости можно перечислить несколько номеров групп:

```
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')
```

Начиная с версии Python 3.6, к группам можно обращаться таким образом:

```
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'
```

Для вывода всех подстрок, которые соответствуют указанным группам, используется метод `groups`:

```
In [18]: match.groups()
Out[18]: ('FastEthernet0/1', '10.0.12.1')
```

Именованные группы

Когда выражение сложное, не очень удобно определять номер группы.

Плюс, при дополнении выражения, может получиться так, что порядок групп изменился, и придется изменить и код, который ссылается на группы.

Именованные группы позволяют задавать группе имя.

Синтаксис именованной группы (`(?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)
```

Теперь к этим группам можно обращаться по имени:

```
In [21]: match.group('intf')  
Out[21]: 'FastEthernet0/1'  
  
In [22]: match.group('address')  
Out[22]: '10.0.12.1'
```

Также очень полезно то, что с помощью метода `groupdict()`, можно получить словарь, где ключи - имена групп, а значения - подстроки, которые им соответствуют:

```
In [23]: match.groupdict()  
Out[23]: {'address': '10.0.12.1', 'intf': 'FastEthernet0/1'}
```

И, в таком случае, можно добавить группы в регулярное выражение и полагаться на их имя, а не на порядок:

```
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'}
```

Разбор вывода команды show ip dhcp snooping с помощью именованных групп

Рассмотрим еще один пример использования именованных групп.

В этом примере задача в том, чтобы получить из вывода команды show ip dhcp snooping binding поля: MAC-адрес, IP-адрес, VLAN и интерфейс.

В файле dhcp_snooping.txt находится вывод команды 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	FastEthernet0/1
0					
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					

Для начала попробуем разобрать одну строку:

```
In [1]: line = '00:09:BB:3D:D6:58 10.1.10.2 86250 dhcp-snooping 10 FastEthernet0/1'
```

В регулярном выражении именованные группы используются для тех частей вывода, которые нужно запомнить:

```
In [2]: match = re.search('(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P<port>\S+)', line)
```

Комментарии к регулярному выражению:

- (?P<mac>\S+) + - в группу с именем 'mac' попадают любые символы, кроме whitespace. Получается, что выражение описывает последовательность любых символов до пробела
- (?P<ip>\S+) + - тут аналогично, последовательность любых символов, кроме whitespace, до пробела. Имя группы 'ip'
- (\d+) + - числовая последовательность (одна или более цифр), а затем один или более пробелов
 - сюда попадет значение Lease
- \S+ + - последовательность любых символов, кроме whitespace
 - сюда попадает тип соответствия (в данном случае все они dhcp-snooping)

- `(?P<vlan>\d+)` + - именованная группа 'vlan'. Сюда попадают только числовые последовательности с одним или более символами
- `(?P<int>.\S+)` - именованная группа 'int'. Сюда попадают любые символы, кроме whitespace

В результате, метод groupdict вернет такой словарь:

```
In [3]: match.groupdict()
Out[3]:
{'int': 'FastEthernet0/1',
 'ip': '10.1.10.2',
 'mac': '00:09:BB:3D:D6:58',
 'vlan': '10'}
```

Так как регулярное выражение отработало как нужно, можно создавать скрипт.

В скрипте перебираются все строки файла dhcp_snooping.txt, и на стандартный поток вывода выводится информация об устройствах.

Файл 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('К коммутатору подключено {}'.format(len(result)))

for num, comp in enumerate(result, 1):
    print('Параметры устройства {}'.format(num))
    for key in comp:
        print('{:10}: {:10}'.format(key, comp[key]))
```

Результат выполнения:

```
$ python parse_dhcp_snooping.py
К коммутатору подключено 4 устройства
Параметры устройства 1:
    int:      FastEthernet0/1
    ip:       10.1.10.2
    mac:      00:09:BB:3D:D6:58
    vlan:     10
Параметры устройства 2:
    int:      FastEthernet0/10
    ip:       10.1.5.2
    mac:      00:04:A3:3E:5B:69
    vlan:     5
Параметры устройства 3:
    int:      FastEthernet0/9
    ip:       10.1.5.4
    mac:      00:05:B3:7E:9B:60
    vlan:     5
Параметры устройства 4:
    int:      FastEthernet0/3
    ip:       10.1.10.6
    mac:      00:09:BC:3F:A6:50
    vlan:     10
```

Группа без захвата

По умолчанию все, что попало в группу, запоминается. Это называется группа с захватом.

Но иногда скобки нужны для указания части выражения, которое повторяется. И, при этом, не нужно запоминать выражение.

Например, надо получить MAC-адрес, VLAN и порты из такого лог-сообщения:

```
In [1]: log = 'Jun 3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v  
lan 10 is flapping between port Gi0/5 and port Gi0/15'
```

Регулярное выражение, которое описывает нужные подстроки:

```
In [2]: match = re.search('(([0-9a-fA-F]{4}\.){2}[0-9a-fA-F]{4}).+vlan (\d+).+port (\$  
+).+port (\S+)', log)
```

Выражение состоит из таких частей:

- `(([0-9a-fA-F]{4}\.){2}[0-9a-fA-F]{4})` - сюда попадет MAC-адрес
 - `[0-9a-fA-F]{4}\.` - эта часть описывает 4 буквы или цифры и точку
 - `([0-9a-fA-F]{4}\.){2}` - тут скобки нужны, чтобы указать, что 4 буквы или цифры и точка повторяются два раза
 - `[0-9a-fA-F]{4}` - затем 4 буквы или цифры
- `.+vlan (\d+)` - в группу попадет номер VLAN
- `.+port (\S+)` - первый интерфейс
- `.+port (\S+)` - второй интерфейс

Метод `groups` вернет такой результат:

```
In [3]: match.groups()  
Out[3]: ('f03a.b216.7ad7', 'b216.', '10', 'Gi0/5', 'Gi0/15')
```

Второй элемент, по сути, лишний. Он попал в вывод из-за скобок в выражении `([0-9a-fA-F]{4}\.){2}`.

В этом случае нужно отключить захват в группе. Это делается добавлением `?:` после открывающейся скобки группы.

Теперь выражение выглядит так:

```
In [4]: match = re.search('((?:[0-9a-fA-F]{4}\.){2}[0-9a-fA-F]{4})+vlan (\d+).+port (\S+).+port (\S+)', log)
```

И, соответственно, группы:

```
In [5]: match.groups()
Out[5]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Повторение захваченного результата

При работе с группами можно использовать результат, который попал в группу, дальше в этом же выражении.

Например, в выводе `sh ip bgp` последний столбец описывает атрибут AS Path (через какие автономные системы прошел маршрут):

```
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
...: *>                  192.168.89.8
...: * 192.168.67.0/24  192.168.79.7      0
...: *>                  192.168.89.8
...: * 192.168.88.0/24  192.168.79.7
...: *>                  192.168.89.8      0
...: '''
```

Допустим, надо получить те префиксы, у которых в пути несколько раз повторяется один и тот же номер AS.

Это можно сделать с помощью ссылки на результат, который был захвачен группой. Например, такое выражение отображает все строки, в которых один и тот же номер повторяется хотя бы два раза:

```
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  700 700 700 i
* 192.168.88.0/24  192.168.79.7      0  700 700 700 i
*>                  192.168.89.8      0  800 800 i
```

В этом выражении обозначение `\1` подставляет результат, который попал в группу. Номер один указывает на конкретную группу. В данном случае это группа 1, она же единственная.

Кроме того, в этом выражении перед строкой регулярного выражения стоит буква `r`. Это так называемая raw строка.

Тут удобней использовать ее, так как иначе надо будет экранировать обратный слеш, чтобы ссылка на группу сработала корректно:

```
match = re.search('(\d+) \\\1', line)
```

При использовании регулярных выражений лучше всегда использовать raw строки.

Аналогичным образом можно описать строки, в которых один и тот же номер встречается три раза:

```
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      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      0 700 700 700 i
```

Аналогичным образом можно ссылаться на результат, который попал в именованную группу:

```
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      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      0 700 700 700 i
*>                  192.168.89.8          0      0 800 800 i
```

Модуль re

В Python для работы с регулярными выражениями используется модуль `re`.

Основные функции модуля `re`:

- `match()` - ищет последовательность в начале строки
- `search()` - ищет первое совпадение с шаблоном
- `.findall()` - ищет все совпадения с шаблоном. Выдает результирующие строки в виде списка
- `finditer()` - ищет все совпадения с шаблоном. Выдает итератор
- `compile()` - компилирует регулярное выражение. К этому объекту затем можно применять все перечисленные функции
- `fullmatch()` - вся строка должна соответствовать описанному регулярному выражению

Кроме функций для поиска совпадений, в модуле есть такие функции:

- `re.sub` - для замены в строках
- `re.split` - для разделения строки на части

Объект Match

В модуле `re` несколько функций возвращают объект `Match`, если было найдено совпадение:

- `search`
- `match`
- `finditer` возвращает итератор с объектами `Match`

В этом подразделе рассматриваются методы объекта `Match`.

Пример объекта `Match`:

```
In [1]: log = 'Jun 3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v  
lan 10 is flapping between port Gi0/5 and port Gi0/15'  
  
In [2]: match = re.search('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>
```

Вывод в 3 строке просто отображает информацию об объекте. Поэтому не стоит полагаться на то, что отображается в части `match`, так как отображаемая строка обрезается по фиксированному количеству знаков.

group()

Метод `group` возвращает подстроку, которая совпала с выражением или с выражением в группе.

Если метод вызывается без аргументов, отображается вся подстрока:

```
In [4]: match.group()  
Out[4]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'
```

На самом деле в этом случае метод `group` вызывается с группой 0:

```
In [13]: match.group(0)  
Out[13]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/1  
5'
```

Другие номера отображают только содержимое соответствующей группы:

```
In [14]: match.group(1)
Out[14]: 'f03a.b216.7ad7'

In [15]: match.group(2)
Out[15]: '10'

In [16]: match.group(3)
Out[16]: 'Gi0/5'

In [17]: match.group(4)
Out[17]: 'Gi0/15'
```

Если вызвать метод `group` с номером группы, который больше, чем количество существующих групп, возникнет ошибка:

```
In [18]: match.group(5)
-----
IndexError                                Traceback (most recent call last)
<ipython-input-18-9df93fa7b44b> in <module>()
      1 match.group(5)
----> 1 IndexError: no such group
```

Если вызвать метод с несколькими номерами групп, результатом будет кортеж со строками, которые соответствуют совпадениям:

```
In [19]: match.group(1, 2, 3)
Out[19]: ('f03a.b216.7ad7', '10', 'Gi0/5')
```

В группу может ничего не попасть, тогда ей будет соответствовать пустая строка:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v
lan 10 is flapping between port Gi0/5 and port Gi0/15'

In [34]: match = re.search('Host (\S+) in vlan (\D*)', log)

In [36]: match.group(2)
Out[36]: ''
```

Если группа описывает часть шаблона и совпадений было несколько, метод отобразит последнее совпадение:

```
In [1]: log = 'Jun 3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v  
lan 10 is flapping between port Gi0/5 and port Gi0/15'  
  
In [44]: match = re.search('Host (\w{4}\.){1}', log)  
  
In [45]: match.group(1)  
Out[46]: 'b216.'
```

Такой вывод получился из-за того, что выражение в скобках описывает 4 буквы или цифры, и после этого стоит плюс. Соответственно, сначала с выражением в скобках совпала первая часть MAC-адреса, потом вторая. Но запоминается и возвращается только последнее выражение.

Если в выражении использовались именованные группы, методу group можно передать имя группы и получить соответствующую подстроку:

```
In [1]: log = 'Jun 3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v  
lan 10 is flapping between port Gi0/5 and port Gi0/15'  
  
In [55]: match = re.search('Host (?P<mac>\S+)'  
...:             'in vlan (?P<vlan>\d+) .*'  
...:             'port (?P<int1>\S+)'  
...:             'and port (?P<int2>\S+)',  
...:             log)  
...:  
  
In [53]: match.group('mac')  
Out[53]: 'f03a.b216.7ad7'  
  
In [54]: match.group('int2')  
Out[54]: 'Gi0/15'
```

Но эти же группы доступны и по номеру:

```
In [58]: match.group(3)  
Out[58]: 'Gi0/5'  
  
In [59]: match.group(4)  
Out[59]: 'Gi0/15'
```

groups()

Метод groups() возвращает кортеж со строками, в котором элементы - это те подстроки, которые попали в соответствующие группы:

```
In [63]: 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 [64]: match = re.search('Host (\S+) '
...:                      'in vlan (\d+) .* '
...:                      'port (\S+) '
...:                      'and port (\S+)',
...:                      log)
...:

In [65]: match.groups()
Out[65]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

У метода `groups` есть опциональный параметр - `default`. Он срабатывает в ситуации, когда все, что попадает в группу, опционально.

Например, при такой строке, совпадение будет и в первой группе, и во второй:

```
In [76]: line = '100      aab1.a1a1.a5d3      FastEthernet0/1'

In [77]: match = re.search('(\d+) +(\w+)?', line)

In [78]: match.groups()
Out[78]: ('100', 'aab1')
```

Если же в строке нет ничего после пробела, в группу ничего не попадет. Но совпадение будет, так как в регулярном выражении описано, что группа опциональна:

```
In [80]: line = '100      '

In [81]: match = re.search('(\d+) +(\w+)?', line)

In [82]: match.groups()
Out[82]: ('100', None)
```

Соответственно, для второй группы значением будет `None`.

Но если передать методу `groups` аргумент, он будет возвращаться вместо `None`:

```
In [83]: line = '100      '
In [84]: match = re.search('(\d+) +(\w+)?', line)
In [85]: match.groups(0)
Out[85]: ('100', 0)
In [86]: match.groups('No match')
Out[86]: ('100', 'No match')
```

groupdict()

Метод groupdict возвращает словарь, в котором ключи - имена групп, а значения - соответствующие строки:

```
In [63]: 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 [88]: match = re.search('Host (?P<mac>\S+) '
...:                     'in vlan (?P<vlan>\d+) .* '
...:                     'port (?P<int1>\S+) '
...:                     'and port (?P<int2>\S+)',
...:                     log)
...:

In [89]: match.groupdict()
Out[89]: {'int1': 'Gi0/5', 'int2': 'Gi0/15', 'mac': 'f03a.b216.7ad7', 'vlan': '10'}
```

start(), end()

Методы start и end возвращают индексы начала и конца совпадения с регулярным выражением.

Если методы вызываются без аргументов, они возвращают индексы для всего совпадения:

```
In [101]: line = ' 10      aab1.a1a1.a5d3      FastEthernet0/1  '
In [102]: match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)
In [103]: match.start()
Out[103]: 2
In [104]: match.end()
Out[104]: 42
In [105]: line[match.start():match.end()]
Out[105]: '10      aab1.a1a1.a5d3      FastEthernet0/1'
```

Методам можно передавать номер или имя группы. Тогда они возвращают индексы для этой группы:

```
In [108]: match.start(2)
Out[108]: 9
In [109]: match.end(2)
Out[109]: 23
In [110]: line[match.start(2):match.end(2)]
Out[110]: 'aab1.a1a1.a5d3'
```

Аналогично для именованных групп:

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vIan 10 is flapping between port Gi0/5 and port Gi0/15'
In [88]: match = re.search('Host (?P<mac>\S+) '
...:                     'in vIan (?P<vlan>\d+) .* '
...:                     'port (?P<int1>\S+) '
...:                     'and port (?P<int2>\S+)',
...:                     log)
...
In [9]: match.start('mac')
Out[9]: 52
In [10]: match.end('mac')
Out[10]: 66
```

span()

Метод `span` возвращает кортеж с индексом начала и конца подстроки. Он работает аналогично методам `start`, `end`, но возвращает пару чисел.

Без аргументов метод span возвращает индексы для всего совпадения:

```
In [112]: line = ' 10      aab1.a1a1.a5d3      FastEthernet0/1  '
In [113]: match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)
In [114]: match.span()
Out[114]: (2, 42)
```

Но ему также можно передать номер группы:

```
In [115]: line = ' 10      aab1.a1a1.a5d3      FastEthernet0/1  '
In [116]: match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)
In [117]: match.span(2)
Out[117]: (9, 23)
```

Аналогично для именованных групп:

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vIan 10 is flapping between port Gi0/5 and port Gi0/15'
In [88]: match = re.search('Host (?P<mac>\S+) '
...:                 'in vIan (?P<vlan>\d+) .* '
...:                 'port (?P<int1>\S+) '
...:                 'and port (?P<int2>\S+)',
...:                 log)
...
In [14]: match.span('mac')
Out[14]: (52, 66)
In [15]: match.span('vlan')
Out[15]: (75, 77)
```

re.search()

Функция `search()`:

- используется для поиска подстроки, которая соответствует шаблону
- возвращает объект `Match`, если подстрока найдена
- возвращает `None`, если подстрока не найдена

Функция `search` подходит в том случае, когда надо найти только одно совпадение в строке, например, когда регулярное выражение описывает всю строку или часть строки.

Рассмотрим пример использования функции `search` в разборе лог-файла.

В файле `log.txt` находятся лог-сообщения с информацией о том, что один и тот же MAC слишком быстро переучивается то на одном, то на другом интерфейсе. Одна из причин таких сообщений - петля в сети.

Содержимое файла `log.txt`:

```
%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
```

При этом, MAC-адрес может прыгать между несколькими портами. В таком случае очень важно знать, с каких портов прилетает MAC. И, если это вызвано петлей, выключить все порты, кроме одного.

Попробуем вычислить, между какими портами и в каком VLAN образовалась проблема.

Проверка регулярного выражения с одной строкой из log-файла:

```
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('Host \S+ '
...:                     'in vlan (\d+) '
...:                     'is flapping between port '
...:                     '(\S+) and port (\S+)', log)
...:
```

Регулярное выражение для удобства чтения разбито на части. В нём есть три группы:

- `(\d+)` - описывает номер VLAN
- `(\S+) and port (\S+)` - в это выражение попадают номера портов

В итоге, в группы попали такие части строки:

```
In [4]: match.groups()
Out[4]: ('10', 'Gi0/16', 'Gi0/24')
```

В итоговом скрипте файл `log.txt` обрабатывается построчно, и из каждой строки собирается информация о портах. Так как порты могут дублироваться, сразу добавляем их в множество, чтобы получить подборку уникальных интерфейсов (файл `parse_log_search.py`):

```
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))
```

Результат выполнения скрипта такой:

```
$ python parse_log_search.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Обработка вывода show cdp neighbors detail

Попробуем получить параметры устройств из вывода sh cdp neighbors detail.

Пример вывода информации для одного соседа:

```
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 S
OFTWARE (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
```

Задача получить такие поля:

- имя соседа (Device ID: SW2)
- IP-адрес соседа (IP address: 10.1.1.2)
- платформу соседа (Platform: cisco WS-C2960-8TC-L)
- версию IOS (Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE SOFTWARE (fc1))

И, для удобства, надо получить данные в виде словаря. Пример итогового словаря для коммутатора SW2:

```
{'SW2': {'ip': '10.1.1.2',
          'platform': 'cisco WS-C2960-8TC-L',
          'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9'}}
```

Пример проверяется на файле sh_cdp_neighbors_sw1.txt.

Первый вариант решения (файл parse_sh_cdp_neighbors_detail_ver1.py):

```
import re
from pprint import pprint

def parse_cdp(filename):
    result = {}

    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'))
```

Тут нужные строки отбираются с помощью метода строк startswith. И в строке с помощью регулярного выражения получается требуемая часть строки.

В итоге все собирается в словарь.

Результат выглядит так:

```
$ 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'}}}
```

Все получилось как нужно. Но, с помощью регулярных выражений эту задачу можно решить более компактно.

Вторая версия решения (файл parse_sh_cdp_neighbors_detail_ver2.py):

```
import re
from pprint import pprint

def parse_cdp(filename):
    regex = ('Device ID: (?P<device>\S+)')
             '| IP address: (?P<ip>\S+)'
             '| Platform: (?P<platform>\S+ \S+), '
             '| Cisco IOS Software, (?P<ios>>.+), RELEASE')

    result = {}

    with open('sh_cdp_neighbors_sw1.txt') as f:
        for line in f:
            match = re.search(regex, line)
            if match:
                if match.lastgroup == 'device':
                    device = match.group(match.lastgroup)
                    result[device] = {}
                elif device:
                    result[device][match.lastgroup] = match.group(match.lastgroup)

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Пояснения ко второму варианту:

- в регулярном выражении описаны все варианты строк через знак или |
- без проверки строки ищется совпадение
- если совпадение найдено, проверяется метод lastgroup
 - метод lastgroup возвращает имя последней именованной группы в регулярном выражении, для которой было найдено совпадение
 - если было найдено совпадение для группы device, в переменную device записывается значение, которое попало в эту группу
 - иначе в словарь записывается соответствие 'имя группы': соответствующее значение

У этого решения ограничение в том, что подразумевается, что в каждой строке может быть только одно совпадение. И в регулярных выражениях, которые записаны через знак |, может быть только одна группа. Это можно исправить, расширив решение.

Результат будет таким же:

```
$ python parse_sh_cdp_neighbors_detail_ver2.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
         'ip': '10.1.1.1',
         'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
         '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'}}
```

re.match()

Функция `match()`:

- используется для поиска в начале строки подстроки, которая соответствует шаблону
- возвращает объект `Match`, если подстрока найдена
- возвращает `None`, если подстрока не найдена

Функция `match` отличается от `search` тем, что `match` всегда ищет совпадение в начале строки. Например, если повторить пример, который использовался для функции `search`, но уже с `match`:

```
In [2]: import re

In [3]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping be
tween port Gi0/16 and port Gi0/24'

In [4]: match = re.match('Host \S+ '
...:                     'in vlan (\d+) '
...:                     'is flapping between port '
...:                     '(\S+) and port (\S+)', log)
...:
```

Результатом будет `None`:

```
In [6]: print(match)
None
```

Так получилось из-за того, что `match` ищет слово `Host` в начале строки. Но это сообщение находится в середине.

В данном случае можно легко исправить выражение, чтобы функция `match` находила совпадение:

```
In [4]: match = re.match('\s+: Host \S+ '
...:                     'in vlan (\d+) '
...:                     'is flapping between port '
...:                     '(\S+) and port (\S+)', log)
...:
```

Перед словом `Host` добавлено выражение `\s+:`. Теперь совпадение будет найдено:

```
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')
```

Пример аналогичен тому, который использовался в функции search, с небольшими изменениями (файл parse_log_match.py):

```
import re

regex = ('\S+: 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.match(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))
```

Результат:

```
$ python parse_log_match.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

re.finditer()

Функция `finditer()`:

- используется для поиска всех непересекающихся совпадений в шаблоне
- возвращает итератор с объектами Match

Функция `finditer` отлично подходит для обработки тех команд, вывод которых отображается столбцами. Например, `sh ip int br`, `sh mac address-table` и др. В этом случае его можно применять ко всему выводу команды.

Пример вывода `sh ip int br`:

```
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
...: '''
```

Регулярное выражение для обработки вывода:

```
In [9]: result = re.finditer('(\S+) +'
...:                         '([\d.]+) +'
...:                         '\w+ +\w+ +'
...:                         '(up|down|administratively down) +'
...:                         '(up|down)',
...:                         sh_ip_int_br)
...:'
```

В переменной `result` находится итератор:

```
In [12]: result
Out[12]: <callable_iterator at 0xb583f46c>
```

В итераторе находятся объекты Match:

```
In [16]: groups = []

In [18]: for match in result:
...:     print(match)
...:     groups.append(match.groups())
...:
<_sre.SRE_Match object; span=(103, 171), match='FastEthernet0/0'      15.0.15.1
YES manual >
<_sre.SRE_Match object; span=(172, 240), match='FastEthernet0/1'      10.0.12.1
YES manual >
<_sre.SRE_Match object; span=(241, 309), match='FastEthernet0/2'      10.0.13.1
YES manual >
<_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 >
```

Теперь в списке `groups` находятся кортежи со строками, которые попали в группы:

```
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')]
```

Аналогичный результат можно получить с помощью генератора списков:

```
In [20]: regex = '(\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')]
```

Теперь разберем тот же лог-файл, который использовался в подразделах `search` и `match`.

В этом случае вывод можно не перебирать построчно, а передать все содержимое файла (файл `parse_log_finditer.py`):

```
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 m in re.finditer(regex, 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))
```

В реальной жизни log-файл может быть очень большим. В таком случае, его лучше обрабатывать построчно.

Вывод будет таким же:

```
$ python parse_log_finditer.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Обработка вывода show cdp neighbors detail

С помощью finditer можно обработать вывод sh cdp neighbors detail, так же, как и в подразделе re.search.

Скрипт почти полностью аналогичен варианту с re.search (файл parse_sh_cdp_neighbors_detail_finditer.py):

```

import re
from pprint import pprint

def parse_cdp(filename):
    regex = ('Device ID: (?P<device>\S+)')
            '|IP address: (?P<ip>\S+)'
            '|Platform: (?P<platform>\S+ \S+), '
            '|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open('sh_cdp_neighbors_sw1.txt') as f:
        match_iter = re.finditer(regex, f.read())
        for match in match_iter:
            if match.lastgroup == 'device':
                device = match.group(match.lastgroup)
                result[device] = {}
            elif device:
                result[device][match.lastgroup] = match.group(match.lastgroup)

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))

```

Теперь совпадения ищутся во всем файле, а не в каждой строке отдельно:

```

with open('sh_cdp_neighbors_sw1.txt') as f:
    match_iter = re.finditer(regex, f.read())

```

Затем перебираются совпадения:

```

with open('sh_cdp_neighbors_sw1.txt') as f:
    match_iter = re.finditer(regex, f.read())
    for match in match_iter:

```

Остальное аналогично.

Результат будет таким:

```
$ 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'}}
```

Хотя результат аналогичный, с finditer больше возможностей, так как можно указывать не только то, что должно находиться в нужной строке, но и в строках вокруг.

Например, можно точнее указать, какой именно IP-адрес надо взять:

```
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP

...
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

Например, если нужно взять первый IP-адрес, можно так дополнить регулярное выражение:

```
regex = ('Device ID: (?P<device>\S+)')
      '|Entry address.*\n +IP address: (?P<ip>\S+)'
      '|Platform: (?P<platform>\S+ \S+),''
      '|Cisco IOS Software, (?P<ios>>.+), RELEASE')
```

re.findall()

Функция `.findall()`:

- используется для поиска всех непересекающихся совпадений в шаблоне
- возвращает:
 - список строк, которые описаны регулярным выражением, если в регулярном выражении нет групп
 - список строк, которые совпали с регулярным выражением в группе, если в регулярном выражении одна группа
 - список кортежей, в которых находятся строки, которые совпали с выражением в группе, если групп несколько

Рассмотрим работу `findall` на примере вывода команды `sh mac address-table`:

```
In [2]: mac_address_table = open('CAM_table.txt').read()

In [3]: print(mac_address_table)
sw1#sh mac address-table
      Mac Address Table
-----
  VLAN      MAC ADDRESS      TYPE      PORTS
  --  -----  -----  -----
  100  a1b2.ac10.7000  DYNAMIC  Gi0/1
  200  a0d4.cb20.7000  DYNAMIC  Gi0/2
  300  acb4.cd30.7000  DYNAMIC  Gi0/3
  100  a2bb.ec40.7000  DYNAMIC  Gi0/4
  500  aa4b.c550.7000  DYNAMIC  Gi0/5
  200  a1bb.1c60.7000  DYNAMIC  Gi0/6
  300  aa0b.cc70.7000  DYNAMIC  Gi0/7
```

Первый пример - регулярное выражение без групп. В этом случае `findall` возвращает список строк, которые совпали с регулярным выражением.

Например, с помощью `findall` можно получить список строк с соответствиями `vlan` - `mac-interface` и избавиться от заголовка в выводе команды:

```
In [4]: re.findall('\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']
```

Обратите внимание, что `findall` возвращает список строк, а не объект `Match`.

Но как только в регулярном выражении появляется группа, `findall` ведет себя по-другому.

Если в выражении используется одна группа, `findall` возвращает список строк, которые совпали с выражением в группе:

```
In [5]: re.findall('(\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']
```

При этом `findall` ищет совпадение всей строки, но возвращает результат, похожий на метод `groups()` в объекте `Match`.

Если же групп несколько, `findall` вернет список кортежей:

```
In [6]: re.findall('(\d+) +(\S+) +\w+ +(\S+)', mac_address_table)
Out[6]:
[('100', 'a1b2.ac10.7000', 'Gi0/1'),
 ('200', 'a0d4.cb20.7000', 'Gi0/2'),
 ('300', 'acb4.cd30.7000', 'Gi0/3'),
 ('100', 'a2bb.ec40.7000', 'Gi0/4'),
 ('500', 'aa4b.c550.7000', 'Gi0/5'),
 ('200', 'a1bb.1c60.7000', 'Gi0/6'),
 ('300', 'aa0b.cc70.7000', 'Gi0/7')]
```

Если такие особенности работы функции `findall` мешают получить необходимый результат, то лучше использовать функцию `finditer`. Но иногда такое поведение подходит и удобно использовать.

Пример использования `findall` в разборе лог-файла (файл `parse_log.findall.py`):

```
import re

regex = ('Host \S+ '
         'in vlan (\d+) '
         'is flapping between port '
         '(\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('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат:

```
$ python parse_log.findall.py
Петля между портами Gi0/19, Gi0/16, Gi0/24 в VLAN 10
```

re.compile()

В Python есть возможность заранее скомпилировать регулярное выражение, а затем использовать его. Это особенно полезно в тех случаях, когда регулярное выражение много используется в скрипте.

Использование компилированного выражения может ускорить обработку, и, как правило, такой вариант удобней использовать, так как в программе разделяется создание регулярного выражения и его использование. Кроме того, при использовании функции `re.compile` создается объект `RegexObject`, у которого есть несколько дополнительных возможностей, которых нет в объекте `MatchObject`.

Для компиляции регулярного выражения используется функция `re.compile`:

```
In [52]: regex = re.compile('\d+ +\S+ +\w+ +\S+')
```

Она возвращает объект `RegexObject`:

```
In [53]: regex
Out[53]: re.compile(r'\d+ +\S+ +\w+ +\S+', re.UNICODE)
```

У объекта `RegexObject` доступны такие методы и атрибуты:

```
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']
```

Обратите внимание, что у объекта `Regex` доступны методы `search`, `match`, `finditer`, `findall`. Это те же функции, которые доступны в модуле глобально, но теперь их надо применять к объекту.

Пример использования метода `search`:

```
In [67]: line = ' 100      a1b2.ac10.7000      DYNAMIC      Gi0/1'
In [68]: match = regex.search(line)
```

Теперь search надо вызывать как метод объекта regex. И передать как аргумент строку.

Результатом будет объект Match:

```
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'
```

Пример компиляции регулярного выражения и его использования на примере разбора лог-файла (файл parse_log_compile.py):

```
import re

regex = re.compile('Host \S+
                   ^in vlan (\d+)
                   ^is flapping between port
                   ^(\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))
```

Это модифицированный пример с использованием finditer. Тут изменилось описание регулярного выражения:

```
regex = re.compile('Host \S+
                   ^in vlan (\d+)
                   ^is flapping between port
                   ^(\S+) and port (\S+)')
```

И вызов finditer теперь выполняется как метод объекта regex:

```
for m in regex.finditer(f.read()):
```

Параметры, которые доступны только при использовании `re.compile`

При использовании функции `re.compile` в методах `search`, `match`, `findall`, `finditer` и `fullmatch` появляются дополнительные параметры:

- `pos` - позволяет указывать индекс в строке, с которого надо начать искать совпадение
- `endpos` - указывает, до какого индекса надо выполнять поиск

Их использование аналогично выполнению среза строки.

Например, таким будет результат без указания параметров `pos`, `endpos`:

```
In [75]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [76]: line = ' 100      a1b2.ac10.7000      DYNAMIC      Gi0/1'

In [77]: match = regex.search(line)

In [78]: match.group()
Out[78]: '100      a1b2.ac10.7000      DYNAMIC      Gi0/1'
```

В этом случае указывается начальная позиция поиска:

```
In [79]: match = regex.search(line, 2)

In [80]: match.group()
Out[80]: '00      a1b2.ac10.7000      DYNAMIC      Gi0/1'
```

Указание начальной позиции аналогично срезу строки:

```
In [81]: match = regex.search(line[2:])

In [82]: match.group()
Out[82]: '00      a1b2.ac10.7000      DYNAMIC      Gi0/1'
```

И последний пример, с указанием двух индексов:

```
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'
```

И аналогичный срез строки:

```
In [94]: match = regex.search(line[2:40])

In [95]: match.group()
Out[95]: '00      a1b2.ac10.7000      DYNAMIC      Gi'
```

В методах `match`, `findall`, `finditer` и `fullmatch` параметры `pos` и `endpos` работают аналогично.

Флаги

При использовании функций или создании скомпилированного регулярного выражения можно указывать дополнительные флаги, которые влияют на поведение регулярного выражения.

Модуль `re` поддерживает такие флаги (в скобках короткий вариант обозначения флага):

- `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`

В этом подразделе для примера рассматривается флаг `re.DOTALL`. Информация об остальных флагах доступна в [документации](#).

`re.DOTALL`

С помощью регулярных выражений можно работать и с многострочной строкой.

Например, из строки `table` надо получить только строки с соответствиями VLAN-MAC-interface:

```
In [11]: table = '''
...: sw1#sh mac address-table
...:          Mac Address Table
...: -----
...:
...:   Vlan      Mac Address          Type      Ports
...:   --      -----
...:   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
...: '''
```

Конечно, в этом случае можно разделить строку на части и работать с каждой строкой отдельно.

Но можно получить часть с MAC-адресами и без разделения.

В этом примере нужно вырезать часть вывода, которая содержит соответствия.

В этом выражении описана строка с MAC-адресом:

```
In [12]: m = re.search(' *\d+ +[a-f0-9.]+ +\w+ +\$+', table)
```

В результат попадет первая строка с MAC-адресом:

```
In [13]: m.group()
Out[13]: ' 100      aabb.cc80.7000      DYNAMIC      Gi0/1'
```

Учитывая то, что по умолчанию регулярные выражения жадные, можно получить все соответствия таким образом:

```
In [14]: m = re.search('( *\d+ +[a-f0-9.]+ +\w+ +\$+\n)+', table)

In [15]: print(m.group())
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
```

Тут описана строка с MAC-адресом, перевод строки, и указано, что это выражение должно повторяться, как минимум, один раз.

Получается, что в данном случае надо получить все строки, начиная с первого соответствия VLAN-MAC-интерфейс.

Это можно описать таким образом:

```
In [16]: m = re.search(' *\d+ +[a-f0-9.]+ +\w+ +\$.*', table)

In [17]: print(m.group())
100      aabb.cc10.7000      DYNAMIC      Gi0/1
```

Пока что в результате только одна строка, так как по умолчанию точка не включает в себя перевод строки.

Но, если добавить специальный флаг, `re.DOTALL`, точка будет включать и перевод

строки, и в результат попадут все соответствия:

```
In [18]: m = re.search(' *\d+ *[a-f0-9.]+ *\w+ *\S+.*', table, re.DOTALL)

In [19]: print(m.group())
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
```

re.split

Функция split работает аналогично методу split в строках.

Но в функции re.split можно использовать регулярные выражения, а значит, разделять строку на части по более сложным условиям.

Например, строку ospf_route надо разбить на элементы по пробелам (как в методе str.split):

```
In [1]: ospf_route = '0      10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0'

In [2]: re.split(' +', ospf_route)
Out[2]:
['0',
 '10.0.24.0/24',
 '[110/41]',
 'via',
 '10.0.13.3',
 '3d18h',
 'FastEthernet0/0']
```

Аналогичным образом можно избавиться и от запятых:

```
In [3]: re.split('[ ,]+', ospf_route)
Out[3]:
['0',
 '10.0.24.0/24',
 '[110/41]',
 'via',
 '10.0.13.3',
 '3d18h',
 'FastEthernet0/0']
```

И, если нужно, от квадратных скобок:

```
In [4]: re.split('[ ,\[\]]+', ospf_route)
Out[4]: ['0', '10.0.24.0/24', '110/41', 'via', '10.0.13.3', '3d18h', 'FastEthernet0/0']
```

У функции split есть особенность работы с группами (выражения в круглых скобках).

Если указать то же выражение с помощью круглых скобок, в итоговый список попадут и разделители.

Например, в выражении как разделитель добавлено слово via:

re.split

```
In [5]: re.split('(via|[\ ,\[\]]+)', ospf_route)
Out[5]:
['0',
 '',
 '10.0.24.0/24',
 '[',
 '110/41',
 '',
 '10.0.13.3',
 '',
 '3d18h',
 '',
 'FastEthernet0/0']
```

Для отключения такого поведения надо сделать группу noncapture.

То есть, отключить запоминание элементов группы:

```
In [6]: re.split(?:via|[\ ,\[\]]+, ospf_route)
Out[6]: ['0', '10.0.24.0/24', '110/41', '10.0.13.3', '3d18h', 'FastEthernet0/0']
```

re.sub

Функция re.sub работает аналогично методу replace в строках.

Но в функции re.sub можно использовать регулярные выражения, а значит, делать замены по более сложным условиям.

Заменим запятые, квадратные скобки и слово via на пробел в строке ospf_route:

```
In [7]: ospf_route = '0      10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0'

In [8]: re.sub('(via|[,\\[\\]])', ' ', ospf_route)
Out[8]: '0      10.0.24.0/24 110/41    10.0.13.3 3d18h  FastEthernet0/0'
```

С помощью re.sub можно трансформировать строку.

Например, преобразовать строку mac_table таким образом:

```
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(' *(\d+) +'
...:                   '([a-f0-9]+)\. '
...:                   '([a-f0-9]+)\. '
...:                   '([a-f0-9]+) +\w+ +'
...:                   '(\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
```

Регулярное выражение раздelenо на группы:

- `(\d+)` - первая группа. Сюда попадет номер VLAN
- `([a-f, 0-9]+).([a-f, 0-9]+).([a-f, 0-9]+)` - три следующие группы (2, 3, 4) описывают MAC-адрес
- `(\s+)` - пятая группа. Описывает интерфейс.

Во втором регулярном выражении эти группы используются.

Для того, чтобы сослаться на группу, используется обратный слеш и номер группы.

Чтобы не пришлось экранировать обратный слеш, используется raw строка.

В итоге вместо номеров групп будут подставлены соответствующие подстроки.

Для примера, также изменен формат записи MAC-адреса.

Дополнительные материалы

Регулярные выражения в Python:

- [Регулярные выражения в Python](#) от простого к сложному. Подробности, примеры, картинки, упражнения
- [Regular Expression HOWTO](#)
- [Python 3 Module of the Week. Модуль re](#)

Сайты для проверки регулярных выражений:

- [для Python](#) - тут можно указывать и методы search, match, findall, и флаги. [Пример регулярного выражения](#). К сожалению, иногда не все выражения воспринимает.
- [Еще один сайт для Python](#) - не поддерживает методы, но хорошо работает и отработал те выражения, на которые ругнулся предыдущий сайт. Подходит для односторочного текста отлично. С многострочным надо учитывать, что в питоне будет другая ситуация. [Пример регулярного выражения](#)
- [regex101](#)

Общие руководства по использованию регулярных выражений:

- Множество примеров использования регулярных выражений от основ до более сложных тем
- Книга [Mastering Regular Expressions](#)

Помощь в изучении регулярных выражений:

- [Визуализация регулярного выражения](#)
- [Regex Crossword](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 15.1

Создать скрипт, который будет ожидать два аргумента:

1. имя файла, в котором находится вывод команды show
2. регулярное выражение

В результате выполнения скрипта, на стандартный поток вывода должны быть выведены те строки из файла с выводом команды show, в которых было найдено совпадение с регулярным выражением.

Проверить работу скрипта на примере вывода команды sh ip int br (файл sh_ip_int_br.txt). Например, попробуйте вывести информацию только по интерфейсу FastEthernet0/1.

Пример работы скрипта:

```
$ python task_9_1.py sh_ip_int_br.txt "Fas"
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

$ python task_9_1.py sh_ip_int_br.txt "manual"
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
Loopback0            10.1.1.1      YES manual up          up
Loopback100          100.0.0.1     YES manual up          up

$ python task_9_1.py sh_ip_int_br.txt "up +up"
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
Loopback0            10.1.1.1      YES manual up          up
Loopback100          100.0.0.1     YES manual up          up
```

В данном случае, скрипт будет работать как фильтр `include` в CLI Cisco. Вы можете попробовать использовать регулярные выражения для [фильтрации вывода команд `show`](#).

Задание 15.1a

Напишите регулярное выражение, которое отобразит строки с интерфейсами 0/1 и 0/3 из вывода `sh ip int br`.

Проверьте регулярное выражение, используя скрипт, который был создан в задании 15.1, и файл `sh_ip_int_br.txt`.

В файле задания нужно написать только регулярное выражение.

Задание 15.1b

Переделайте регулярное выражение из задания 15.1a таким образом, чтобы оно, по-прежнему, отображало строки с интерфейсами 0/1 и 0/3, но, при этом, в регулярном выражении было не более 7 символов (не считая кавычки вокруг регулярного выражения).

Проверьте регулярное выражение, используя скрипт, который был создан в задании 15.1, и файл `sh_ip_int_br.txt`.

В файле задания нужно написать только регулярное выражение.

Задание 15.1с

Проверить работу скрипта из задания 15.1 и регулярного выражения из задания 15.1a или 15.1b на выводе sh ip int br из файла sh_ip_int_br_switch.txt.

Если, в результате выполнения скрипта, были выведены не только строки с интерфейсами 0/1 и 0/3, исправить регулярное выражение. В результате, должны выводиться только строки с интерфейсами 0/1 и 0/3.

В файле задания нужно написать только регулярное выражение.

Задание 15.2

Создать функцию return_match, которая ожидает два аргумента:

- имя файла, в котором находится вывод команды show
- регулярное выражение

Функция должна обрабатывать вывод команды show построчно и возвращать список подстрок, которые совпали с регулярным выражением (не всю строку, где было найдено совпадение, а только ту подстроку, которая совпала с выражением).

Проверить работу функции на примере вывода команды sh ip int br (файл sh_ip_int_br.txt). Вывести список всех IP-адресов из вывода команды.

Соответственно, регулярное выражение должно описывать подстроку с IP-адресом (то есть, совпадением должен быть IP-адрес).

Обратите внимание, что в данном случае, мы можем не проверять корректность IP-адреса, диапазоны адресов и так далее, так как мы обрабатываем вывод команды, а не ввод пользователя.

Задание 15.3

Создать функцию parse_cfg, которая ожидает как аргумент имя файла, в котором находится конфигурация устройства.

Функция должна обрабатывать конфигурацию и возвращать IP-адреса и маски, которые настроены на интерфейсах, в виде списка кортежей:

- первый элемент кортежа - IP-адрес
- второй элемент кортежа - маска

Например (взяты произвольные адреса):

```
[('10.0.1.1', '255.255.255.0'), ('10.0.2.1', '255.255.255.0')]
```

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла config_r1.txt.

Обратите внимание, что в данном случае, мы можем не проверять корректность IP-адреса, диапазоны адресов и так далее, так как мы обрабатываем конфигурацию, а не ввод пользователя.

Задание 15.3a

Переделать функцию parse_cfg из задания 15.3 таким образом, чтобы она возвращала словарь:

- ключ: имя интерфейса
- значение: кортеж с двумя строками:
 - IP-адрес
 - маска

Например (взяты произвольные адреса):

```
{'FastEthernet0/1': ('10.0.1.1', '255.255.255.0'),
 'FastEthernet0/2': ('10.0.2.1', '255.255.255.0')}
```

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла config_r1.txt.

Задание 15.3b

Проверить работу функции parse_cfg из задания 15.3а на конфигурации config_r2.txt.

Обратите внимание, что на интерфейсе e0/1 назначены два IP-адреса:

```
interface Ethernet0/1
 ip address 10.255.2.2 255.255.255.0
 ip address 10.254.2.2 255.255.255.0 secondary
```

А в словаре, который возвращает функция parse_cfg, интерфейсу Ethernet0/1 соответствует только один из них (второй).

Переделайте функцию `parse_cfg` из задания 15.3а таким образом, чтобы она возвращала список кортежей для каждого интерфейса. Если на интерфейсе назначен только один адрес, в списке будет один кортеж. Если же на интерфейсе настроены несколько IP-адресов, то в списке будет несколько кортежей.

Проверьте функцию на конфигурации `config_r2.txt` и убедитесь, что интерфейсу `Ethernet0/1` соответствует список из двух кортежей.

Обратите внимание, что в данном случае, можно не проверять корректность IP-адреса, диапазоны адресов и так далее, так как обрабатывается вывод команды, а не ввод пользователя.

Задание 15.4

Создать функцию `parse_sh_ip_int_br`, которая ожидает как аргумент имя файла, в котором находится вывод команды `show`

Функция должна обрабатывать вывод команды `show ip int br` и возвращать такие поля:

- 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', 'up', 'up')]
```

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла `sh_ip_int_br_2.txt`.

Задание 15.4а

Создать функцию `convert_to_dict`, которая ожидает два аргумента:

- список с названиями полей
- список кортежей с результатами отработки функции `parse_sh_ip_int_br` из задания 15.4

Функция возвращает результат в виде списка словарей (порядок полей может быть другой):

```
[{"interface": "FastEthernet0/0", "status": "up", "protocol": "up", "address": "10.0.1.1"},  
 {"interface": "FastEthernet0/1", "status": "up", "protocol": "up", "address": "10.0.2.1"}]
```

Проверить работу функции на примере файла sh_ip_int_br_2.txt:

- первый аргумент - список headers
- второй аргумент - результат, который возвращает функции parse_show из прошлого задания.

Функцию parse_sh_ip_int_br не нужно копировать. Надо импортировать или саму функцию, и использовать то же регулярное выражение, что и в задании 9.4, или импортировать результат выполнения функции parse_show.

```
headers = ['interface', 'address', 'status', 'protocol']
```

Запись и передача данных

В этой части книги рассматриваются вопросы сохранения и передачи данных. Данными могут быть, например:

- вывод команд
- обработанный вывод команд в виде словаря, списка и подобного
- информация полученная из системы мониторинга

До сих пор рассматривался только самый простой вариант - запись информации в обычный текстовый файл.

В этой части рассматривается чтение и запись данных в форматах CSV, JSON и YAML:

- CSV - это табличный формат представления данных. Он может быть получен, например, при экспорте данных из таблицы или базе данных. Аналогичным образом данные могут быть записаны в этом формате для последующего импорта в таблицу.
- JSON - это формат, который очень часто используется в API. Кроме того, этот формат позволит сохранить такие структуры данных как словари или списки в структурированном формате и затем прочитать их из файла в формате JSON и получить те же структуры данных в Python.
- Формат YAML очень часто используется для описания сценариев. Например, он используется в Ansible. Кроме того, в этом формате удобно записывать вручную параметры, которые должны считываться скрипты.

Python позволяет записывать объекты самого языка в файлы и считывать их с помощью модуля Pickle, но этот аспект в книге не рассматривается.

Также в этой части рассматриваются базы данных. Хотя данные можно записать с соблюдением структуры и в CSV или JSON, запрашивать нужную информацию из файлов в этом формате не всегда удобно. Особенно, когда речь идет о более сложных запросах, в которых указаны несколько критериев.

Для задач такого рода отлично подходят базы данных. В разделе 18 рассматривается СУБД SQLite, а также основы языка SQL.

Unicode

Программы, которые мы пишем, не изолированы в себе. Они скачивают данные из Интернета, читают и записывают данные на диск, передают данные через сеть.

Поэтому очень важно понимать разницу между тем, как компьютер хранит и передает данные, и как эти данные воспринимает человек. Мы воспринимаем текст, а компьютер - байты.

В Python 3, соответственно, есть две концепции:

- текст - неизменяемая последовательность Unicode символов. Для хранения этих символов используется тип строки (`str`)
- данные - неизменяемая последовательность байтов. Для хранения используется тип `bytes`

Более корректно будет сказать, что текст - это неизменяемая последовательность кодов (codepoints) Unicode.

Стандарт Юникод

Юникод - это стандарт, который описывает представление и кодировку почти всех языков и других символов.

Несколько фактов о Юникод:

- стандарт версии 10.0 (июнь 2017) описывает 136 690 кодов
- каждый код - это номер, который соответствует определенному символу
- стандарт также определяет кодировки - способ представления кода символа в байтах

Каждому символу в Юникод соответствует определенный код. Это число, которое обычно записывается таким образом: `U+0073`, где 0073 - это шестнадцатеричные цифры.

Кроме кода, у каждого символа есть свое уникальное имя. Например, букве "s" соответствует код `U+0073` и имя "LATIN SMALL LETTER S".

Примеры кодов, имен и соответствующих символов:

- `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" - ☺

Кодировки

Кодировки позволяют записывать код символа в байтах.

Юникод поддерживает несколько кодировок:

- UTF-8
- UTF-16
- UTF-32

Одна из самых популярных кодировок на сегодняшний день - UTF-8. Эта кодировка использует переменное количество байт для записи символов Юникод.

Примеры символов Юникод и их представление в байтах в кодировке UTF-8:

- H - `48`
- i - `69`

- - 01 f6 c0
- - 01 f6 80
- ☀ - 26 03

Юникод в Python 3

В Python 3 есть:

- строки - неизменяемая последовательность Unicode символов. Для хранения этих символов используется тип строка (`str`)
- байты - неизменяемая последовательность байтов. Для хранения используется тип `bytes`

Строки

Примеры строк:

```
In [11]: hi = 'привет'

In [12]: hi
Out[12]: 'привет'

In [15]: type(hi)
Out[15]: str

In [13]: beautiful = 'schön'

In [14]: beautiful
Out[14]: 'schön'
```

Так как строки - это последовательность кодов Юникод, можно записать строку разными способами.

Символ Юникод можно записать, используя его имя:

```
In [1]: "\N{LATIN SMALL LETTER O WITH DIAERESIS}"
Out[1]: 'ö'
```

Или использовав такой формат:

```
In [4]: "\u00f6"
Out[4]: 'ö'
```

Строчку можно записать как последовательность кодов Юникод:

```
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
```

Функция `ord` возвращает значение кода Unicode для символа:

```
In [6]: ord('ö')
Out[6]: 246
```

Функция `chr` возвращает символ Юникод, который соответствует коду:

```
In [7]: chr(246)
Out[7]: 'ö'
```

Байты

Тип `bytes` - это неизменяемая последовательность байтов.

Байты обозначаются так же, как строки, но с добавлением буквы "b" перед строкой:

```
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
```

В Python байты, которые соответствуют символам ASCII, отображаются как эти символы, а не как соответствующие им байты. Это может немного путать, но всегда можно распознать тип `bytes` по букве `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'
```

Если попытаться написать не ASCII символ в байтовом литерале, возникнет ошибка:

```
In [44]: bytes3 = b'привет'
          File "<ipython-input-44-dc8b23504fa7>", line 1
            bytes3 = b'привет'
                      ^
SyntaxError: bytes can only contain ASCII literal characters.
```

Конвертация между байтами и строками

Избежать работы с байтами нельзя. Например, при работе с сетью или файловой системой, чаще всего, результат возвращается в байтах.

Соответственно, надо знать, как выполнять преобразование байтов в строку и наоборот. Для этого и нужна кодировка.

Кодировку можно представлять как ключ шифрования, который указывает:

- как "зашифровать" строку в байты (str -> bytes). Используется метод encode (похож на encrypt)
- как "расшифровать" байты в строку (bytes -> str). Используется метод decode (похож на decrypt)

Эта аналогия позволяет понять, что преобразования строка-байты и байты-строка должны использовать одинаковую кодировку.

encode, decode

Для преобразования строки в байты используется метод **encode**:

```
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')
```

Чтобы получить строку из байт, используется метод **decode**:

```
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

Метод encode есть также в классе str (как и другие методы работы со строками):

```
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'
```

А метод decode есть у класса bytes (как и другие методы):

```
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'
```

В этих методах кодировка может указываться как ключевой аргумент (примеры выше) или как позиционный:

```
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'
```

Как работать с Юникод и байтами

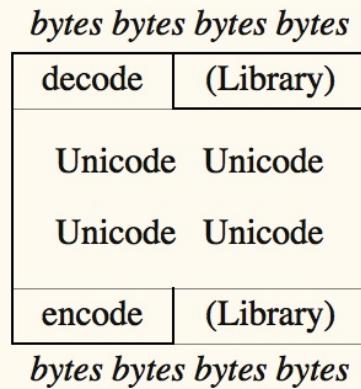
Есть очень простое правило, придерживаясь которого, можно избежать, как минимум, части проблем. Оно называется "Юникод сэндвич":

- байты, которые программа считывает, надо как можно раньше преобразовать в юникод (строку)
- внутри программы работать с юникод
- юникод надо преобразовать в байты как можно позже, перед передачей

Pro tip #1: Unicode sandwich

Bytes on the outside, unicode on the inside

Encode/decode at the edges



@nedbat

bit.ly/unipain

Примеры конвертации между байтами и строками

Рассмотрим несколько примеров работы с байтами и конвертации байт в строки.

subprocess

Модуль subprocess возвращает результат команды в виде байт:

```
In [1]: import subprocess

In [2]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'],
...:                         stdout=subprocess.PIPE)
...:

In [3]: result.stdout
Out[3]: b'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=59.4 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms\n64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=55.1 ms\n--- 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'
```

Если дальше необходимо работать с этим выводом, надо сразу конвертировать его в строку:

```
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
```

Модуль subprocess поддерживает еще один вариант преобразования - параметр encoding. Если указать его при вызове функции run, результат будет получен в виде строки:

```
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--- 8.8.8.8 ping statistics ---\n3 packets transmitted, 3 received, 0% packet loss, time 2003ms\nrtt min/avg/max/mdev = 53.368/54.534/55.564/0.941 ms\n'

In [8]: print(result.stdout)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=55.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.3 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 53.368/54.534/55.564/0.941 ms
```

telnetlib

В зависимости от модуля, преобразование между строками и байтами может выполняться автоматически, а может требоваться явно.

Например, в модуле telnetlib необходимо передавать байты в методах `read_until` и `write`:

```
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)
```

И возвращает метод байты, поэтому в предпоследней строке используется `decode`.

pexpect

Модуль pexpect как аргумент ожидает строку, а возвращает байты:

```
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_futu
res\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 3 07:59 iterator_generator\r\n'

In [12]: output.decode('utf-8')
Out[12]: 'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_futu
res\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 3 07:59 iterator_generator\r\n'
```

И также поддерживает вариант передачи кодировки через параметр encoding:

```
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_futu
res\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 3 07:59 iterator_generator\r\n'
```

Работа с файлами

До сих пор при работе с файлами использовалась такая конструкция:

```
with open(filename) as f:
    for line in f:
        print(line)
```

Но на самом деле, при чтении файла происходит конвертация байт в строки. И при этом использовалась кодировка по умолчанию:

```
In [1]: import locale

In [2]: locale.getpreferredencoding()
Out[2]: 'UTF-8'
```

Кодировка по умолчанию в файле:

```
In [2]: f = open('r1.txt')

In [3]: f
Out[3]: <_io.TextIOWrapper name='r1.txt' mode='r' encoding='UTF-8'>
```

При работе с файлами лучше явно указывать кодировку, так как в разных ОС она может отличаться:

```
In [4]: with open('r1.txt', encoding='utf-8') as f:
...:     for line in f:
...:         print(line, end='')

!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Выводы

Эти примеры показаны тут для того, чтобы показать, что разные модули могут по-разному подходить к вопросу конвертации между строками и байтами. И разные функции и методы этих модулей могут ожидать аргументы и возвращать значения разных типов. Однако все эти вещи написаны в документации.

Ошибки при конвертации

При конвертации между строками и байтами очень важно точно знать, какая кодировка используется, а также знать о возможностях разных кодировок.

Например, кодировка ASCII не может преобразовать в байты кириллицу:

```
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)
```

Аналогично, если строка "привет" преобразована в байты, и попробовать преобразовать ее в строку с помощью ascii, тоже получим ошибку:

```
In [34]: hi_unicode = 'привет'

In [35]: hi_bytes = hi_unicode.encode('utf-8')

In [36]: hi_bytes.decode('ascii')
-----
UnicodeDecodeError                                 Traceback (most recent call last)
<ipython-input-36-aa0ada5e44e9> in <module>()
----> 1 hi_bytes.decode('ascii')

UnicodeDecodeError: 'ascii' codec can't decode byte 0xd0 in position 0: ordinal not in
range(128)
```

Еще один вариант ошибки, когда используются разные кодировки для преобразований:

```
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
```

Но на самом деле, предыдущие ошибки - это хорошо. Они явно говорят, в чем проблема.

Хуже, когда получается так:

```
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\xbd\xd0\xbb\xd1\x82'

In [43]: hi_bytes.decode('utf-16')
Out[43]: '뿐腴론닐뗐苑'
```

Обработка ошибок

У методов encode и decode есть режимы обработки ошибок, которые указывают, как реагировать на ошибку преобразования.

Параметр errors в encode

По умолчанию encode использует режим 'strict' - при возникновении ошибок кодировки генерируется исключение UnicodeError. Примеры такого поведения были выше.

Вместо этого режима можно использовать replace, чтобы заменить символ знаком вопроса:

```
In [44]: de_hi_unicode = 'grüezi'

In [45]: de_hi_unicode.encode('ascii', 'replace')
Out[45]: b'gr?ezi'
```

Или `namereplace`, чтобы заменить символ именем:

```
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'
```

Кроме того, можно полностью игнорировать символы, которые нельзя закодировать:

```
In [48]: de_hi_unicode = 'grüezi'

In [49]: de_hi_unicode.encode('ascii', 'ignore')
Out[49]: b'grezi'
```

Параметр `errors` в `decode`

В методе `decode` по умолчанию тоже используется режим `strict` и генерируется исключение `UnicodeDecodeError`.

Если изменить режим на `ignore`, как и в `encode`, символы будут просто игнорироваться:

```
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'
```

Режим `replace` заменит символы:

```
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'
```


Дополнительные материалы

Документация Python:

- [What's New In Python 3: Text Vs. Data Instead Of Unicode Vs. 8-bit](#)
- [Unicode HOWTO](#)

Статьи:

- [Pragmatic Unicode](#) - статья, презентация и видео
- [Раздел "Strings" книги "Dive Into Python 3"](#) - очень хорошо написано о Unicode, кодировках и как все это работает в Python

Без привязки к 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 \(Wikipedia\)](#)

Работа с файлами в формате CSV, JSON, YAML

Сериализация данных - это сохранение данных в каком-то формате, чаще всего, структурированном.

Например, это могут быть:

- файлы в формате YAML или JSON
- файлы в формате CSV
- база данных

Кроме того, Python позволяет записывать объекты самого языка (этот аспект в курсе не рассматривается, но, если Вам интересно, посмотрите на модуль Pickle).

В этом разделе рассматриваются форматы CSV, JSON, YAML, а в следующем разделе - базы данных.

Для чего могут пригодится форматы YAML, JSON, CSV:

- у Вас могут быть данные об IP-адресах и подобной информации, которую нужно обработать, в таблицах
 - таблицу можно экспортить в формат CSV и обрабатывать её с помощью Python
- управляющий софт может возвращать данные в JSON. Соответственно, преобразовав эти данные в объект Python, с ними можно работать и делать что угодно
- YAML очень удобно использовать для описания параметров, так как у него довольно приятный синтаксис
 - например, это могут быть параметры настройки различных объектов (IP-адреса, VLAN и др.)
 - как минимум, знание формата YAML пригодится при использовании Ansible

Для каждого из этих форматов в Python есть модуль, который существенно упрощает работу с ними.

Работа с файлами в формате CSV

CSV (comma-separated value) - это формат представления табличных данных (например, это могут быть данные из таблицы или данные из БД).

В этом формате каждая строка файла - это строка таблицы. Но, несмотря на название формата, разделителем может быть не только запятая.

И, хотя у форматов с другим разделителем может быть и собственное название, например, TSV (tab separated values), тем не менее, под форматом CSV понимают, как правило, любые разделители.

Пример файла в формате CSV (sw_data.csv):

```
hostname, vendor, model, location
sw1, Cisco, 3750, London
sw2, Cisco, 3850, Liverpool
sw3, Cisco, 3650, Liverpool
sw4, Cisco, 3650, London
```

В стандартной библиотеке Python есть модуль csv, который позволяет работать с файлами в CSV формате.

Чтение

Пример чтения файла в формате CSV (файл csv_read.py):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Вывод будет таким:

```
$ python csv_read.py
['hostname', 'vendor', 'model', 'location']
['sw1', 'Cisco', '3750', 'London']
['sw2', 'Cisco', '3850', 'Liverpool']
['sw3', 'Cisco', '3650', 'Liverpool']
['sw4', 'Cisco', '3650', 'London']
```

В первом списке находятся названия столбцов, а в остальных соответствующие значения.

Обратите внимание, что сам csv.reader возвращает итератор:

```
In [1]: import csv

In [2]: with open('sw_data.csv') as f:
...:     reader = csv.reader(f)
...:     print(reader)
...:
<_csv.reader object at 0x10385b050>
```

При необходимости его можно превратить в список таким образом:

```
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']]
```

Чаще всего заголовки столбцов удобней получить отдельным объектом. Это можно сделать таким образом (файл csv_read_headers.py):

```
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)
```

Иногда в результате обработки гораздо удобней получить словари, в которых ключи - это названия столбцов, а значения - значения столбцов.

Для этого в модуле есть **DictReader** (файл csv_read_dict.py):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row)
        print(row['hostname'], row['model'])
```

Вывод будет таким:

```
$ 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 создает не стандартные словари Python, а упорядоченные словари. За счет этого порядок элементов соответствует порядку столбцов в CSV файле.

До Python 3.6 возвращались обычные словари, а не упорядоченные.

В остальном с упорядоченными словарями можно работать, используя те же методы, что и в обычных словарях.

Запись

Аналогичным образом с помощью модуля csv можно и записать файл в формате CSV (файл csv_write.py):

```

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())

```

В примере выше строки из списка сначала записываются в файл, а затем содержимое файла выводится на стандартный поток вывода.

Вывод будет таким:

```

$ 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"

```

Обратите внимание на интересную особенность: строки в последнем столбце взяты в кавычки, а остальные значения - нет.

Так получилось из-за того, что во всех строках последнего столбца есть запятая. И кавычки указывают на то, что именно является целой строкой. Когда запятая находятся в кавычках, модуль csv не воспринимает её как разделитель.

Иногда лучше, чтобы все строки были в кавычках. Конечно, в данном случае достаточно простой пример, но когда в строках больше значений, то кавычки позволяют указать, где начинается и заканчивается значение.

Модуль csv позволяет управлять этим. Для того, чтобы все строки записывались в файл csv с кавычками, надо изменить скрипт таким образом (файл csv_write_quoting.py):

```

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())

```

Теперь вывод будет таким:

```

$ 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"

```

Теперь все значения с кавычками. И, так как номер модели задан как строка в изначальном списке, тут он тоже в кавычках.

Кроме метода `writerow`, поддерживается метод `writerows`. Ему можно передать любой итерируемый объект.

Например, предыдущий пример можно записать таким образом (файл `csv_writerows.py`):

```

import csv

data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]

with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    writer.writerows(data)

with open('sw_data_new.csv') as f:
    print(f.read())

```

DictWriter

С помощью DictWriter можно записать словари в формат csv.

В целом DictWriter работает так же, как writer, но так как словари не упорядочены, надо указывать явно в каком порядке будут идти столбцы в файле. Для этого используется параметр fieldnames (файл csv_write_dict.py):

```
import csv

data = [{'hostname': 'sw1',
         'location': 'London',
         'model': '3750',
         'vendor': 'Cisco'},
        {'hostname': 'sw2',
         'location': 'Liverpool',
         'model': '3850',
         'vendor': 'Cisco'},
        {'hostname': 'sw3',
         'location': 'Liverpool',
         'model': '3650',
         'vendor': 'Cisco'},
        {'hostname': 'sw4',
         'location': 'London',
         'model': '3650',
         'vendor': 'Cisco']]

with open('csv_write_dictwriter.csv', 'w') as f:
    writer = csv.DictWriter(f, fieldnames=list(data[0].keys()),
                           quoting=csv.QUOTE_NONNUMERIC)
    writer.writeheader()
    for d in data:
        writer.writerow(d)
```

Указание разделителя

Иногда в качестве разделителя используются другие значения. В таком случае должна быть возможность подсказать модулю, какой именно разделитель использовать.

Например, если в файле используется разделитель ; (файл sw_data2.csv):

```
hostname;vendor;model;location
sw1;Cisco;3750;London
sw2;Cisco;3850;Liverpool
sw3;Cisco;3650;Liverpool
sw4;Cisco;3650;London
```

Достаточно просто указать, какой разделитель используется в reader (файл csv_read_delimiter.py):

```
import csv

with open('sw_data2.csv') as f:
    reader = csv.reader(f, delimiter=';')
    for row in reader:
        print(row)
```

Работа с файлами в формате JSON

JSON (JavaScript Object Notation) - это текстовый формат для хранения и обмена данными.

JSON по синтаксису очень похож на Python и достаточно удобен для восприятия.

Как и в случае с CSV, в Python есть модуль, который позволяет легко записывать и читать данные в формате JSON.

Чтение

Файл 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"  
    ]  
}
```

Для чтения в модуле json есть два метода:

- `json.load()` - метод считывает файл в формате JSON и возвращает объекты Python
- `json.loads()` - метод считывает строку в формате JSON и возвращает объекты Python

json.load()

Чтение файла в формате JSON в объект Python (файл json_read_load.py):

```

import json

with open('sw_templates.json') as f:
    templates = json.load(f)

for section, commands in templates.items():
    print(section)
    print('\n'.join(commands))

```

Вывод будет таким:

```

$ 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 99
              9', 'switchport trunk allowed vlan']}
access
switchport mode access
switchport access vlan
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
trunk
switchport trunk encapsulation dot1q
switchport mode trunk
switchport trunk native vlan 999
switchport trunk allowed vlan

```

json.loads()

Считывание строки в формате JSON в объект Python (файл json_read_loads.py):

```

import json

with open('sw_templates.json') as f:
    file_content = f.read()
    templates = json.loads(file_content)

print(templates)

for section, commands in templates.items():
    print(section)
    print('\n'.join(commands))

```

Результат будет аналогичен предыдущему выводу.

Запись

Запись файла в формате JSON также осуществляется достаточно легко.

Для записи информации в формате JSON в модуле json также два метода:

- `json.dump()` - метод записывает объект Python в файл в формате JSON
- `json.dumps()` - метод возвращает строку в формате JSON

json.dumps()

Преобразование объекта в строку в формате JSON (`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())
```

Метод `json.dumps()` подходит для ситуаций, когда надо вернуть строку в формате JSON. Например, чтобы передать ее API.

json.dump()

Запись объекта Python в файл в формате JSON (файл `json_write_dump.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:
    json.dump(to_json, f)

with open('sw_templates.json') as f:
    print(f.read())
```

Когда нужно записать информацию в формате JSON в файл, лучше использовать метод `dump`.

Дополнительные параметры методов записи

Методам `dump` и `dumps` можно передавать дополнительные параметры для управления форматом вывода.

По умолчанию эти методы записывают информацию в компактном представлении. Как правило, когда данные используются другими программами, визуальное представление данных не важно. Если же данные в файле нужно будет считывать человеку, такой формат не очень удобно воспринимать.

К счастью, модуль `json` позволяет управлять подобными вещами.

Передав дополнительные параметры методу `dump` (или методу `dumps`), можно получить более удобный для чтения вывод (файл `json_write_indent.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:
    json.dump(to_json, f, sort_keys=True, indent=2)

with open('sw_templates.json') as f:
    print(f.read())

```

Теперь содержимое файла `sw_templates.json` выглядит так:

```
{
  "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"
  ]
}
```

Изменение типа данных

Еще один важный аспект преобразования данных в формат JSON: данные не всегда будут того же типа, что исходные данные в Python.

Например, кортежи при записи в JSON превращаются в списки:

```
In [1]: import json

In [2]: trunk_template = ('switchport trunk encapsulation dot1q',
...:                      'switchport mode trunk',
...:                      'switchport trunk native vlan 999',
...:                      'switchport trunk allowed vlan')

In [3]: print(type(trunk_template))
<class 'tuple'>

In [4]: with open('trunk_template.json', 'w') as f:
...:     json.dump(trunk_template, f, sort_keys=True, indent=2)
...:

In [5]: cat trunk_template.json
[
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
]
In [6]: templates = json.load(open('trunk_template.json'))

In [7]: type(templates)
Out[7]: list

In [8]: print(templates)
['switchport trunk encapsulation dot1q', 'switchport mode trunk', 'switchport trunk na
tive vlan 999', 'switchport trunk allowed vlan']
```

Так происходит из-за того, что в JSON используются другие типы данных и не для всех типов данных Python есть соответствия.

Таблица конвертации данных Python в JSON:

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

Таблица конвертации JSON в данные Python:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Ограничение по типам данных

В формат JSON нельзя записать словарь, у которого ключи - кортежи:

```
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
```

Но с помощью дополнительного параметра можно игнорировать подобные ключи:

```
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"]}
```

Кроме того, в JSON ключами словаря могут быть только строки. Но, если в словаре Python использовались числа, ошибки не будет. Вместо этого выполнится конвертация чисел в строки:

```
In [28]: d = {1:100, 2:200}  
In [29]: json.dumps(d)  
Out[29]: '{"1": 100, "2": 200}'
```

Работа с файлами в формате YAML

YAML (YAML Ain't Markup Language) - еще один текстовый формат для записи данных.

YAML более приятен для восприятия человеком, чем JSON, поэтому его часто используют для описания сценариев в ПО. Например, в Ansible.

Синтаксис YAML

Как и Python, YAML использует отступы для указания структуры документа. Но в YAML можно использовать только пробелы и нельзя использовать знаки табуляции.

Еще одна схожесть с Python: комментарии начинаются с символа # и продолжаются до конца строки.

Список

Список может быть записан в одну строку:

```
[switchport mode access, switchport access vlan, switchport nonegotiate, spanning-tree  
portfast, spanning-tree bpduguard enable]
```

Или каждый элемент списка в своей строке:

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

Когда список записан таким блоком, каждая строка должна начинаться с - (минуса и пробела), и все строки в списке должны быть на одном уровне отступа.

Словарь

Словарь также может быть записан в одну строку:

```
{ vlan: 100, name: IT }
```

Или блоком:

```
vlan: 100
name: IT
```

Строки

Строки в YAML не обязательно брать в кавычки. Это удобно, но иногда всё же следует использовать кавычки. Например, когда в строке используется какой-то специальный символ (специальный для YAML).

Такую строку, например, нужно взять в кавычки, чтобы она была корректно воспринята YAML:

```
command: "sh interface | include Queueing strategy:"
```

Комбинация элементов

Словарь, в котором есть два ключа: `access` и `trunk`. Значения, которые соответствуют этим ключам - списки команд:

```
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
```

Список словарей:

```
- 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
```

Модуль PyYAML

Для работы с YAML в Python используется модуль PyYAML. Он не входит в стандартную библиотеку модулей, поэтому его нужно установить:

```
pip install pyyaml
```

Работа с ним аналогична модулям csv и json.

Чтение из YAML

Попробуем преобразовать данные из файла YAML в объекты Python.

Файл info.yaml:

```
- 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
```

Чтение из YAML (файл yaml_read.py):

```
import yaml
from pprint import pprint

with open('info.yaml') as f:
    templates = yaml.load(f)

pprint(templates)
```

Результат:

```
$ python yaml_read.py
[{'BS': 1550,
 'IT': 791,
 'id': 11,
 'name': 'Liverpool',
 'to_id': 1,
 'to_name': 'LONDON'},
 {'BS': 1510,
 'IT': 793,
 'id': 12,
 'name': 'Bristol',
 'to_id': 1,
 'to_name': 'LONDON'},
 {'BS': 1650,
 'IT': 892,
 'id': 14,
 'name': 'Coventry',
 'to_id': 2,
 'to_name': 'Manchester'}]
```

Формат YAML очень удобен для хранения различных параметров, особенно, если они заполняются вручную.

Запись в YAML

Запись объектов Python в YAML (файл `yaml_write.py`):

```
import yaml

trunk_template = ['switchport trunk encapsulation dot1q',
                  'switchport mode trunk',
                  'switchport trunk native vlan 999',
                  'switchport trunk allowed vlan']

access_template = ['switchport mode access',
                  'switchport access vlan',
                  'switchport nonegotiate',
                  'spanning-tree portfast',
                  'spanning-tree bpduguard enable']

to_yaml = {'trunk':trunk_template, 'access':access_template}

with open('sw_templates.yaml', 'w') as f:
    yaml.dump(to_yaml, f)

with open('sw_templates.yaml') as f:
    print(f.read())
```

Файл `sw_templates.yaml` выглядит таким образом:

```
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]
```

По умолчанию список записался в одну строку. Это можно изменить.

Для того, чтобы изменить формат записи, надо добавить параметр `default_flow_style=False` (файл `yaml_write_default_flow_style.py`):

```
import yaml

trunk_template = ['switchport trunk encapsulation dot1q',
                  'switchport mode trunk',
                  'switchport trunk native vlan 999',
                  'switchport trunk allowed vlan']

access_template = ['switchport mode access',
                   'switchport access vlan',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

to_yaml = {'trunk':trunk_template, 'access':access_template}

with open('sw_templates.yaml', 'w') as f:
    yaml.dump(to_yaml, f, default_flow_style=False)

with open('sw_templates.yaml') as f:
    print f.read()
```

Теперь содержимое файла `sw_templates.yaml` выглядит таким образом:

```
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
```


Дополнительные материалы

В этом разделе рассматривались только базовые операции чтения и записи, без дополнительных параметров. Подробнее можно почитать в документации модулей.

- [CSV](#)
- [JSON](#)
- [YAML](#)

Кроме того, на сайте [PyMOTW](#) очень хорошо расписываются все модули Python, которые входят в стандартную библиотеку (устанавливаются вместе с самим Python):

- [CSV](#)
- [JSON](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 17.1

В этом задании нужно:

- взять содержимое нескольких файлов с выводом команды `sh version`
- распарсить вывод команды с помощью регулярных выражений и получить информацию об устройстве
- записать полученную информацию в файл в CSV формате

Для выполнения задания нужно создать две функции.

Функция `parse_sh_version`:

- ожидает аргумент `output` в котором находится вывод команды `sh version` (не имя файла)
- обрабатывает вывод, с помощью регулярных выражений
- возвращает кортеж из трёх элементов:
 - `ios` - в формате "12.4(5)T"
 - `image` - в формате "flash:c2800-advpipservicesk9-mz.124-5.T.bin"
 - `uptime` - в формате "5 days, 3 hours, 3 minutes"

Функция `write_to_csv`:

- ожидает два аргумента:
 - имя файла, в который будет записана информация в формате CSV
 - данные в виде списка списков, где:

- первый список - заголовки столбцов,
- остальные списки - содержимое
- функция записывает содержимое в файл, в формате CSV и ничего не возвращает

Остальное содержимое скрипта может быть в скрипте, а может быть в ещё одной функции.

Скрипт должен:

- обработать информацию из каждого файла с выводом sh version:
 - sh_version_r1.txt, sh_version_r2.txt, sh_version_r3.txt
- с помощью функции parse_sh_version, из каждого вывода должна быть получена информация ios, image, uptime
- из имени файла нужно получить имя хоста
- после этого вся информация должна быть записана в файл routers_inventory.csv

В файле routers_inventory.csv должны быть такие столбцы:

- hostname, ios, image, uptime

В скрипте, с помощью модуля glob, создан список файлов, имя которых начинается на sh_ver. Вы можете раскомментировать строку print(sh_version_files), чтобы посмотреть содержимое списка.

Кроме того, создан список заголовков (headers), который должен быть записан в CSV.

```
import glob

sh_version_files = glob.glob('sh_ver*')
#print(sh_version_files)

headers = ['hostname', 'ios', 'image', 'uptime']
```

Задание 17.2

Создать функцию parse_sh_cdp_neighbors, которая обрабатывает вывод команды show cdp neighbors.

Функция ожидает, как аргумент, вывод команды одной строкой (не имя файла).

Функция должна возвращать словарь, который описывает соединения между устройствами.

Например, если как аргумент был передан такой вывод:

```
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

Функция должна вернуть такой словарь:

```
{'R4': {'Fa0/1': {'R5': 'Fa0/1',
                  'Fa0/2': {'R6': 'Fa0/0'}}}}
```

При этом интерфейсы могут быть записаны с пробелом Fa 0/0 или без Fa0/0.

Проверить работу функции на содержимом файла sh_cdp_n_sw1.txt

Задание 17.2а

С помощью функции parse_sh_cdp_neighbors из задания 17.2, обработать вывод команды sh cdp neighbor из файлов:

- 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

Объединить все словари, которые возвращает функция parse_sh_cdp_neighbors, в один словарь topology и записать его содержимое в файл topology.yaml.

Структура словаря topology должна быть такой:

```
{'R4': {'Fa0/1': {'R5': 'Fa0/1',
                  'Fa0/2': {'R6': 'Fa0/0'}},
         'R5': {'Fa0/1': {'R4': 'Fa0/1'}},
         'R6': {'Fa0/0': {'R4': 'Fa0/2'}}}}
```

При этом интерфейсы могут быть записаны с пробелом Fa 0/0 или без Fa0/0.

Не копировать код функции parse_sh_cdp_neighbors

Задание 17.2б

Переделать функциональность скрипта из задания 17.2а, в функцию `generate_topology_from_cdp`.

Функция `generate_topology_from_cdp` должна быть создана с параметрами:

- `list_of_files` - список файлов из которых надо считать вывод команды `sh cdp neighbor`
- `save_to_file` - этот параметр управляет тем, будет ли записан в файл, итоговый словарь
 - значение по умолчанию - `True`
- `topology_filename` - имя файла, в который сохранится топология.
 - по умолчанию, должно использоваться имя `topology.yaml`.
 - топология сохраняется только, если аргумент `save_to_file` указан равным `True`

Функция возвращает словарь, который описывает топологию. Словарь должен быть в том же формате, что и в задании 17.2а.

Проверить работу функции `generate_topology_from_cdp` на файлах:

- `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`

Записать полученный словарь в файл `topology.yaml`.

Не копировать код функции `parse_sh_cdp_neighbors`

Задание 17.2с

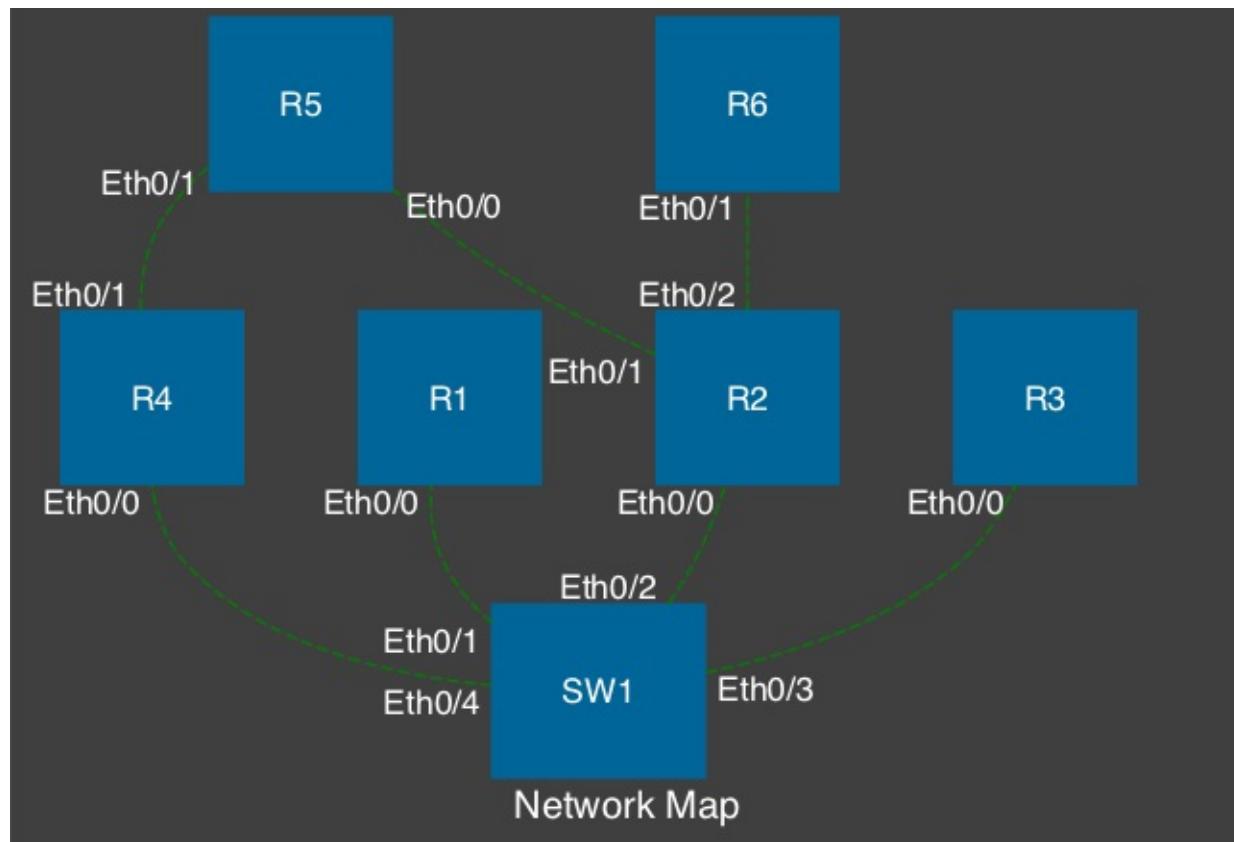
С помощью функции `draw_topology` из файла `draw_network_graph.py` сгенерировать топологию, которая соответствует описанию в файле `topology.yaml`

Обратите внимание на то, какой формат данных ожидает функция `draw_topology`. Описание топологии из файла `topology.yaml` нужно преобразовать соответствующим образом, чтобы использовать функцию `draw_topology`.

Для решения задания можно создать любые вспомогательные функции.

Не копировать код функции `draw_topology`.

В итоге, должно быть сгенерировано изображение топологии. Результат должен выглядеть так же, как схема в файле `task_17_2c_topology.svg`



При этом:

- Интерфейсы могут быть записаны с пробелом Fa 0/0 или без Fa0/0.
- Расположение устройств на схеме может быть другим
- Соединения должны соответствовать схеме

Для выполнения этого задания, должен быть установлен graphviz: `apt-get install graphviz`

И модуль python для работы с graphviz: `pip install graphviz`

Работа с базами данных

Использование баз данных - это еще один способ хранения информации.

Но базы данных полезны не только в хранении информации. Используя СУБД, можно делать срезы информации по различным параметрам.

База данных (БД) - это данные, которые хранятся в соответствии с определенной схемой. В этой схеме каким-то образом описаны соотношения между данными.

Язык БД (лингвистические средства) - используется для описания структуры БД, управления данными (добавление, изменение, удаление, получение), управления правами доступа к БД и ее объектам, управления транзакциями.

Система управления базами данных (СУБД) - это программные средства, которые дают возможность управлять БД. СУБД должны поддерживать соответствующий язык (языки) для управления БД.

SQL

SQL (structured query language) - используется для описания структуры БД, управления данными (добавление, изменение, удаление, получение), управления правами доступа к БД и ее объектам, управления транзакциями.

Язык SQL подразделяется на такие категории:

- DDL (Data Definition Language) - язык описания данных
- DML (Data Manipulation Language) - язык манипулирования данными
- DCL (Data Control Language) - язык определения доступа к данным
- TCL (Transaction Control Language) - язык управления транзакциями

В каждой категории есть свои операторы (перечислены не все операторы):

- DDL
 - CREATE - создание новой таблицы, СУБД, схемы
 - ALTER - изменение существующей таблицы, колонки
 - DROP - удаление существующих объектов из СУБД
- DML
 - SELECT - выбор данных
 - INSERT - добавление новых данных
 - UPDATE - обновление существующих данных
 - DELETE - удаление данных
- DCL
 - GRANT - предоставление пользователям разрешения на чтение/запись определенных объектов в СУБД
 - REVOKE - отзыв ранее предоставленных разрешений
- TCL
 - COMMIT Transaction - применение транзакции
 - ROLLBACK Transaction - откат всех изменений, сделанных в текущей транзакции

SQL и Python

Для работы с реляционной СУБД в Python можно использовать два подхода:

- работать с библиотекой, которая соответствует конкретной СУБД, и использовать для работы с БД язык SQL
 - Например, для работы с SQLite используется модуль sqlite3
- работать с [ORM](#), которая использует объектно-ориентированный подход для

работы с БД

- Например, SQLAlchemy

SQLite

[SQLite](#) — встраиваемая в процесс реализация SQL-машины.

Слово SQL-сервер здесь не используем, потому что как таковой сервер там не нужен — весь функционал, который встраивается в SQL-сервер, реализован внутри библиотеки (и, соответственно, внутри программы, которая её использует).

На практике SQLite часто используется как встроенная СУБД в приложениях.

SQLite CLI

В комплекте поставки SQLite идёт также утилита для работы с SQLite в командной строке. Утилита представлена в виде исполняемого файла sqlite3 (sqlite3.exe для Windows), и с ее помощью можно вручную выполнять команды SQL.

С помощью этой утилиты очень удобно проверять правильность команд SQL, а также в целом знакомиться с языком SQL.

Попробуем с помощью этой утилиты разобраться с базовыми командами SQL, которые понадобятся для работы с БД.

Для начала разберемся, как создавать БД.

Если Вы используете Linux или Mac OS, то, скорее всего, sqlite3 установлен. Если Вы используете Windows, то можно скачать sqlite3 [тут](#).

Для того, чтобы создать БД (или открыть уже созданную), надо просто вызвать sqlite3 таким образом:

```
$ sqlite3 testDB.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite>
```

Внутри sqlite3 можно выполнять команды SQL или так называемые метакоманды (или dot-команды).

Метакоманды

К метакомандам относятся несколько специальных команд для работы с SQLite. Они относятся только к утилите sqlite3, а не к SQL языку. В конце этих команд ; ставить не нужно.

Примеры метакоманд:

- `.help` - подсказка со списком всех метакоманд
- `.exit` ИЛИ `.quit` - выход из сессии sqlite3
- `.databases` - показывает присоединенные БД
- `.tables` - показывает доступные таблицы

Примеры выполнения:

```
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
```

Основы SQL (в sqlite3 CLI)

В этом разделе рассматривается синтаксис языка SQL.

Если Вы знакомы с базовым синтаксисом SQL, этот раздел можно пропустить и сразу перейти к разделу [Модуль sqlite3](#).

CREATE

Оператор create позволяет создавать таблицы.

Создадим таблицу switch, в которой хранится информация о коммутаторах:

```
sqlite> CREATE table switch (
...>     mac          text not NULL primary key,
...>     hostname     text,
...>     model        text,
...>     location     text
...> );
```

Аналогично можно было создать таблицу и таким образом:

```
sqlite> create table switch (mac text not NULL primary key, hostname text, model text,
location text);
```

В данном примере мы описали таблицу switch: определили, какие поля будут в таблице, и значения какого типа будут в них находиться.

Кроме того, поле mac является первичным ключом. Это автоматически значит, что:

- поле должно быть уникальным
- в нём не может находиться значение NULL (в SQLite это надо задавать явно)

В этом примере это вполне логично, так как MAC-адрес должен быть уникальным.

На данный момент записей в таблице нет, есть только ее определение. Просмотреть определение можно такой командой:

```
sqlite> .schema switch
CREATE TABLE switch (
mac          text not NULL primary key,
hostname     text,
model        text,
location     text
);
```

DROP

Оператор DROP удаляет таблицу вместе со схемой и всеми данными.

Удалить таблицу можно так:

```
sqlite> DROP table switch;
```

INSERT

Оператор `insert` используется для добавления данных в таблицу.

Есть несколько вариантов добавления записей, в зависимости от того, все ли поля будут заполнены, и будут ли они идти по порядку определения полей или нет.

Если Вы удалили таблицу, выполнив `drop`, надо ее заново создать:

```
create table switch (mac text not NULL primary key, hostname text, model text, location text);
```

Если указываются значения для всех полей, добавить запись можно таким образом (порядок полей должен соблюдаться):

```
sqlite> INSERT into switch values ('0010.A1AA.C1CC', 'sw1', 'Cisco 3750', 'London, Green Str');
```

Если нужно указать не все поля или указать их в произвольном порядке, используется такая запись:

```
sqlite> INSERT into switch (mac, model, location, hostname)
...> values ('0020.A2AA.C2CC', 'Cisco 3850', 'London, Green Str', 'sw2');
```

SELECT

Оператор select позволяет запрашивать информацию в таблице.

Например:

```
sqlite> SELECT * from switch;
0010.A1AA.C1CC|sw1|Cisco 3750|London, Green Str
0020.A2AA.C2CC|sw2|Cisco 3850|London, Green Str
```

`select *` означает, что нужно вывести все поля таблицы. Следом указывается, из какой таблицы запрашиваются данные: `from switch`.

В данном случае в отображении таблицы не хватает названия полей. Включить это можно с помощью команды `.headers ON`.

```
sqlite> .headers ON
sqlite> 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
```

Теперь отобразились заголовки, но, в целом, отображение не очень приятное. Хотелось бы, чтобы все выводилось в виде колонок. За форматирование вывода отвечает команда `.mode`.

Режим `.mode column` включает отображение в виде колонок:

```
sqlite> .mode column
sqlite> 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
```

При желании можно выставить и ширину колонок. Для этого используется команда `.width`. Например, попробуйте выставить `.width 20`.

Если нужно сделать так, чтобы эти параметры использовались по умолчанию, добавьте их в файл `.sqliterc` в домашнем каталоге пользователя, под которым Вы работаете.

Например, чтобы вывод заголовков столбцов и вывод столбцами использовались по умолчанию, файл `.sqliterc` должен выглядеть так:

```
.headers on  
.mode column
```

В следующих подразделах вывод команд показан с включенными `.headers on` и `.mode column`

WHERE

Оператор WHERE используется для уточнения запроса. С помощью этого оператора можно указывать определенные условия, по которым отбираются данные. Если условие выполнено, возвращается соответствующее значение из таблицы, если нет - не возвращается.

Сейчас в таблице switch всего две записи:

```
sqlite> 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
```

Чтобы в таблице было больше записей, надо создать еще несколько строк. В SQLite есть метакоманда .read, которая позволяет загружать команды SQL из файла.

Для добавления записей заготовлен файл add_rows_to_testdb.txt:

```
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')
;
```

Для загрузки команд из файла надо выполнить команду:

```
sqlite> .read add_rows_to_testdb.txt
```

Теперь таблица switch выглядит так:

```
sqlite> 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
```

С помощью оператора WHERE можно показать только те коммутаторы, модель которых 3850:

```
sqlite> 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
```

Оператор WHERE позволяет указывать не только конкретное значение поля. Если добавить к нему оператор LIKE, можно указывать шаблон поля.

LIKE с помощью символов _ и % указывает, на что должно быть похоже значение:

- _ - обозначает один символ или число
- % - обозначает ноль, один или много символов

Например, если поле model записано в разном формате, с помощью предыдущего запроса не получится вывести нужные коммутаторы.

Например, у коммутатора sw6 поле model записано в таком формате: C3750, а у коммутаторов sw1 и sw3 в таком: Cisco 3750.

В таком варианте запрос с оператором WHERE не покажет sw6:

```
sqlite> 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
```

Но если вместе с оператором WHERE использовать оператор LIKE :

```
sqlite> 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
```

ALTER

Оператор ALTER позволяет менять существующую таблицу: добавлять новые колонки или переименовывать таблицу.

Добавим в таблицу новые поля:

- mngmt_ip - IP-адрес коммутатора в менеджмент VLAN
- mngmt_vid - VLAN ID (номер VLAN) для менеджмент VLAN

Добавление записей с помощью команды ALTER:

```
sqlite> ALTER table switch ADD COLUMN mngmt_ip text;
sqlite> ALTER table switch ADD COLUMN mngmt_vid integer;
```

Теперь таблица выглядит так (новые поля установлены в значение NULL):

```
sqlite> SELECT * from switch;
mac           hostname    model      location      mngmt_ip      mngmt_vid
-----        -----
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
```

UPDATE

Оператор UPDATE используется для изменения существующей записи таблицы.

Обычно, UPDATE используется вместе с оператором WHERE, чтобы уточнить, какую именно запись необходимо изменить.

С помощью UPDATE можно заполнить новые столбцы в таблице.

Например, добавить IP-адрес для коммутатора sw1:

```
sqlite> UPDATE switch set mngmt_ip = '10.255.1.1' WHERE hostname = 'sw1';
```

Теперь таблица выглядит так:

```
sqlite> SELECT * from switch;
mac          hostname    model      location      mngmt_ip      mngmt_vid
-----        -----       -----      -----       -----
0010.A1AA.C1CC  sw1        Cisco 3750  London, Green Str  10.255.1.1
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
```

Аналогичным образом можно изменить и номер VLAN:

```
sqlite> UPDATE switch set mngmt_vid = 255 WHERE hostname = 'sw1';
sqlite> 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
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
```

И можно изменить несколько полей за раз:

```
sqlite> UPDATE switch set mngmt_ip = '10.255.1.2', mngmt_vid = 255 WHERE hostname = 'sw2';
sqlite> 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
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
```

Чтобы не заполнять поля mngmt_ip и mngmt_vid вручную, заполним остальное из файла 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';
```

После загрузки команд таблица выглядит так:

```
sqlite> .read update_fields_in_testdb.txt

sqlite> 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
```

Теперь предположим, что sw1 был заменен с модели 3750 на модель 3850. Соответственно, изменилось не только поле модель, но и поле MAC-адрес.

Внесение изменений:

```
sqlite> UPDATE switch set model = 'Cisco 3850', mac = '0010.D1DD.E1EE' WHERE hostname = 'sw1';
```

Результат будет таким:

```
sqlite> 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
```

REPLACE

Оператор REPLACE используется для добавления или замены данных в таблице.

Оператор REPLACE может поддерживаться не во всех СУБД.

Когда возникает нарушение условия уникальности поля, выражение с оператором REPLACE:

- удаляет существующую строку, которая вызвала нарушение
- добавляет новую строку

У выражения REPLACE есть два вида:

```
sqlite> INSERT OR REPLACE INTO switch
...> VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850', 'London, Green Str', '10.255.1.
3', 255);
```

Или более короткий вариант:

```
sqlite> REPLACE INTO switch
...> VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850', 'London, Green Str', '10.255.1.
3', 255);
```

Результатом любой из этих команд будет замена модели коммутатора sw3:

```
sqlite> 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
```

В данном случае MAC-адрес в новой записи совпадает с уже существующей, поэтому происходит замена.

Если были указаны не все поля, в новой записи будут только те поля, которые были указаны. Это связано с тем, что replace сначала удаляет существующую запись.

При добавлении записи, для которой не возникает нарушения уникальности поля, replace работает как обычный insert:

```
sqlite> REPLACE INTO switch
...> VALUES ('0080.A8AA.C8CC', 'sw8', 'Cisco 3850', 'London, Green Str', '10.255.1.
8', 255);

sqlite> 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
```

DELETE

Оператор delete используется для удаления записей.

Как правило, он используется вместе с оператором where.

Например, таблица switch выглядит так:

```
sqlite> 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
```

Удаление информации про коммутатор sw8 выполняется таким образом:

```
sqlite> DELETE from switch where hostname = 'sw8';
```

Теперь в таблице нет строки с коммутатором sw8:

```
sqlite> 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
```

ORDER BY

Оператор ORDER BY используется для сортировки вывода по определенному полю, по возрастанию или убыванию. Для этого он добавляется к оператору SELECT.

Если выполнить простой запрос SELECT, вывод будет таким:

```
sqlite> 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
```

С помощью оператора ORDER BY можно вывести записи в таблице switch, отсортировав их по имени коммутаторов:

```
sqlite> 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
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
```

По умолчанию сортировка выполняется по возрастанию, поэтому в запросе можно было не указывать параметр ASC:

```
sqlite> 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
```

Сортировка по IP-адресу по убыванию:

```
sqlite> 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
```

AND

Оператор AND позволяет группировать несколько условий:

```
sqlite> select * from switch where model = 'Cisco 3750' and ip LIKE '10.0.%';
mac           hostname   model      location      ip       vlan
-----        -----      -----      -----      -----
0010.A11A.C1CC  sw1       Cisco 3750  London, Green Str  10.0.255.1  255
0020.A22A.C2CC  sw2       Cisco 3750  London, Green Str  10.0.255.2  255

sqlite> select * from switch where model LIKE '%3750%' and ip LIKE '10.0.%';
mac           hostname   model      location      ip       vlan
-----        -----      -----      -----      -----
0010.A11A.C1CC  sw1       Cisco 3750  London, Green Str  10.0.255.1  255
0020.A22A.C2CC  sw2       Cisco 3750  London, Green Str  10.0.255.2  255
```

OR

Оператор OR:

```
sqlite> select * from switch where model = 'Cisco 3750' or model = 'Cisco 3850';
mac           hostname   model      location      ip       vlan
-----        -----      -----      -----      -----
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
0010.A11A.C1CC  sw1       Cisco 3750  London, Green Str  10.0.255.1  255
0020.A22A.C2CC  sw2       Cisco 3750  London, Green Str  10.0.255.2  255
0030.A3AA.C1CC  sw3       Cisco 3850  London, Green Str  10.255.1.3  255
```

IN

Оператор IN:

```
sqlite> select * from switch where model in ('Cisco 3750', 'C3750');
mac           hostname   model      location      ip       vlan
-----        -----      -----      -----      -----
0060.A6AA.C4CC  sw6       C3750     London, Green Str  10.255.1.6  255
0010.A11A.C1CC  sw1       Cisco 3750  London, Green Str  10.0.255.1  255
0020.A22A.C2CC  sw2       Cisco 3750  London, Green Str  10.0.255.2  255
```

NOT

Оператор NOT:

```
sqlite> select * from switch where model not in ('Cisco 3750', 'C3750');
mac           hostname   model      location      ip       vlan
-----  -----
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

sqlite> select * from switch where model LIKE '%3750%' and ip not LIKE '10.0.%';
mac           hostname   model      location      ip       vlan
-----  -----
0060.A6AA.C4CC  sw6        C3750     London, Green Str  10.255.1.6  255
```

Модуль sqlite3

Для работы с SQLite в Python используется модуль sqlite3.

Connection

Объект **Connection** - это подключение к конкретной БД. Можно сказать, что этот объект представляет БД.

Пример создания подключения:

```
import sqlite3

connection = sqlite3.connect('dhcp_snooping.db')
```

Cursor

После создания соединения надо создать объект Cursor - это основной способ работы с БД.

Создается курсор из соединения с БД:

```
connection = sqlite3.connect('dhcp_snooping.db')
cursor = connection.cursor()
```

Выполнение команд SQL

Для выполнения команд SQL в модуле есть несколько методов:

- `execute()` - метод для выполнения одного выражения SQL
- `executemany()` - метод позволяет выполнить одно выражение SQL для последовательности параметров (или для итератора)
- `executescript()` - метод позволяет выполнить несколько выражений SQL за один раз

Метод execute

Метод `execute` позволяет выполнить одну команду SQL.

Сначала надо создать соединение и курсор:

```
In [1]: import sqlite3

In [2]: connection = sqlite3.connect('sw_inventory.db')

In [3]: cursor = connection.cursor()
```

Создание таблицы switch с помощью метода `execute`:

```
In [4]: cursor.execute("create table switch (mac text not NULL primary key, hostname t
ext, model text, location text)")
Out[4]: <sqlite3.Cursor at 0x1085be880>
```

Выражения SQL могут быть параметризованы - вместо данных можно подставлять специальные значения. За счет этого можно использовать одну и ту же команду SQL для передачи разных данных.

Например, таблицу `switch` нужно заполнить данными из списка `data`:

```
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')]
```

Для этого можно использовать запрос вида:

```
In [6]: query = "INSERT into switch values (?, ?, ?, ?, ?)"
```

Знаки вопроса в команде используются для подстановки данных, которые будут передаваться методу `execute`.

Теперь можно передать данные таким образом:

```
In [7]: for row in data:  
...:     cursor.execute(query, row)  
...:
```

Второй аргумент, который передается методу `execute`, должен быть кортежем. Если нужно передать кортеж с одним элементом, используется запись `(value,)`.

Чтобы изменения были применены, нужно выполнить `commit` (обратите внимание, что метод `commit` вызывается у соединения):

```
In [8]: connection.commit()
```

Теперь, при запросе из командной строки `sqlite3`, можно увидеть эти строки в таблице `switch`:

```
$ sqlite3 sw_inventory.db  
  
sqlite> select * from switch;  
mac          hostname    model      location  
-----  
0000.AAAA.CCCC  sw1        Cisco 3750  London, Green Str  
0000.BBBB.CCCC  sw2        Cisco 3780  London, Green Str  
0000.AAAA.DDDD  sw3        Cisco 2960  London, Green Str  
0011.AAAA.CCCC  sw4        Cisco 3750  London, Green Str
```

Метод `executemany`

Метод `executemany` позволяет выполнить одну команду SQL для последовательности параметров (или для итератора).

С помощью метода `executemany` в таблицу `switch` можно добавить аналогичный список данных одной командой.

Например, в таблицу `switch` надо добавить данные из списка `data2`:

```
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')]
```

Для этого нужно использовать аналогичный запрос вида:

```
In [10]: query = "INSERT into switch values (?, ?, ?, ?)"
```

Теперь можно передать данные методу executemany:

```
In [11]: cursor.executemany(query, data2)
Out[11]: <sqlite3.Cursor at 0x10ee5e810>

In [12]: connection.commit()
```

После выполнения commit данные доступны в таблице:

```
sqlite> select * from switch;
mac           hostname    model      location
-----
0000.AAAA.CCCC sw1        Cisco 3750 London, Green Str
0000.BBBB.CCCC sw2        Cisco 3780 London, Green Str
0000.AAAA.DDDD sw3        Cisco 2960 London, Green Str
0011.AAAA.CCCC sw4        Cisco 3750 London, Green Str
0000.1111.0001 sw5        Cisco 3750 London, Green Str
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
```

Метод executemany подставил соответствующие кортежи в команду SQL, и все данные добавились в таблицу.

Метод executescript

Метод executescript позволяет выполнить несколько выражений SQL за один раз.

Особенно удобно использовать этот метод при создании таблиц:

```
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>
```

Получение результатов запроса

Для получения результатов запроса в sqlite3 есть несколько способов:

- использование методов `fetch...()` - в зависимости от метода возвращаются одна, несколько или все строки
- использование курсора как итератора - возвращается итератор

Метод `fetchone`

Метод `fetchone` возвращает одну строку данных.

Пример получения информации из базы данных `sw_inventory.db`:

```
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')
```

Обратите внимание, что хотя запрос SQL подразумевает, что запрашивалось всё содержимое таблицы, метод `fetchone` вернёт только одну строку.

Если повторно вызвать метод, он вернет следующую строку:

```
In [21]: print(cursor.fetchone())
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
```

Аналогичным образом метод будет возвращать следующие строки. После обработки всех строк метод начинает возвращать `None`.

За счет этого метод можно использовать в цикле, например, так:

```
In [22]: cursor.execute('select * from switch')
Out[22]: <sqlite3.Cursor at 0x104eda810>

In [23]: while True:
...:     next_row = cursor.fetchone()
...:     if next_row:
...:         print(next_row)
...:     else:
...:         break
...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

Метод fetchmany

Метод `fetchmany` возвращает список строк данных.

Синтаксис метода:

```
cursor.fetchmany([size=cursor.arraysize])
```

С помощью параметра `size` можно указывать, какое количество строк возвращается.

По умолчанию параметр `size` равен значению `cursor.arraysize`:

```
In [24]: print(cursor.arraysize)
1
```

Например, таким образом можно возвращать по три строки из запроса:

```
In [25]: cursor.execute('select * from switch')
Out[25]: <sqlite3.Cursor at 0x104eda810>

In [26]: from pprint import pprint

In [27]: while True:
...:     three_rows = cursor.fetchmany(3)
...:     if three_rows:
...:         pprint(three_rows)
...:     else:
...:         break
...:

[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')]
[('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')]
[('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Метод выдает нужное количество строк, а если строк осталось меньше, чем параметр size, то оставшиеся строки.

Метод fetchall

Метод fetchall возвращает все строки в виде списка:

```
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')]
```

Важный аспект работы метода - он возвращает все оставшиеся строки.

То есть, если до метода fetchall использовался, например, метод fetchone, то метод fetchall вернет оставшиеся строки запроса:

```
In [30]: cursor.execute('select * from switch')
Out[30]: <sqlite3.Cursor at 0x104eda810>

In [31]: cursor.fetchone()
Out[31]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')

In [32]: cursor.fetchone()
Out[32]: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')

In [33]: cursor.fetchall()
Out[33]:
[('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Метод `fetchmany` в этом аспекте работает аналогично.

Cursor как итератор

Если нужно построчно обрабатывать результирующие строки, лучше использовать курсор как итератор. При этом не нужно использовать методы `fetch`.

При использовании методов `execute` возвращается курсор. А, так как курсор можно использовать как итератор, можно использовать его, например, в цикле `for`:

```
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')
```

И, конечно же, аналогичный вариант отработает и без присваивания переменной:

```
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')
```

Использование модуля sqlite3 без явного создания курсора

Методы execute доступны и в объекте Connection, и в объекте Cursor. А методы fetch доступны только в объекте Cursor.

При использовании методов execute с объектом Connection курсор возвращается как результат выполнения метода execute. Его можно использовать как итератор и получать данные без методов fetch.

За счет этого при работе с модулем sqlite3 можно не создавать курсор.

Пример итогового скрипта (файл 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()
```

Результат выполнения будет таким:

```
$ 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')
```


Обработка исключений

Посмотрим на пример использования метода `execute` при возникновении ошибки.

В таблице `switch` поле `mac` должно быть уникальным. И, если попытаться записать пересекающийся MAC-адрес, возникнет ошибка:

```
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
```

Соответственно, можно перехватить исключение:

```
In [40]: try:
...:     con.execute(query)
...: except sqlite3.IntegrityError as e:
...:     print("Error occured: ", e)
...:
Error occured:  UNIQUE constraint failed: switch.mac
```

Обратите внимание, что надо перехватывать исключение `sqlite3.IntegrityError`, а не `IntegrityError`.

Connection как менеджер контекста

После выполнения операций изменения должны быть сохранены (надо выполнить `commit()`), а затем можно закрыть соединение, если оно больше не нужно.

Python позволяет использовать объект Connection как менеджер контекста. В таком случае, не нужно явно делать `commit`.

При этом:

- при возникновении исключения, транзакция автоматически откатывается
- если исключения не было, автоматически выполняется `commit`

Пример использования соединения с базой как менеджера контекстов (`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 tex
t)''')

try:
    with con:
        query = 'INSERT into switch values (?, ?, ?, ?)'
        con.executemany(query, data)

except sqlite3.IntegrityError as e:
    print('Error occurred: ', e)

for row in con.execute('select * from switch'):
    print(row)

con.close()
```

Обратите внимание, что хотя транзакция будет откатываться при возникновении исключения, само исключение всё равно надо перехватывать.

Для проверки этого функционала надо записать в таблицу данные, в которых MAC-адрес повторяется. Но прежде, чтобы не повторять части кода, лучше разнести код в файле `create_sw_inventory_ver2.py` по функциям (файл `create_sw_inventory_ver2_functions.py`):

```
# -*- 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'),
        ('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 occurred: ', 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.
'''

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()
```

Результат выполнения скрипта выглядит так:

```
$ python create_sw_inventory_ver2_functions.py
Создание таблицы...
Запись данных в БД:
[('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')]
Запись данных прошла успешно

Проверка содержимого БД
[('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')]
```

Теперь проверим, как функция `write_data_to_db` отработает при наличии одинаковых MAC-адресов в данных.

В файле `create_sw_inventory_ver3.py` используются функции из файла `create_sw_inventory_ver2_functions.py` и подразумевается, что скрипт будет запускаться после записи предыдущих данных:

```
# -*- 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()
```

В списке `data2` у коммутатора `sw7` MAC-адрес совпадает с уже существующим в БД коммутатором `sw3`.

Результат выполнения скрипта:

```
$ python create_sw_inventory_ver3.py

Проверка текущего содержимого БД
[('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')]

-----
Попытка записать данные с повторяющимся MAC-адресом:
[('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

Проверка содержимого БД
[('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')]
```

Обратите внимание, что содержимое таблицы `switch` до и после добавления информации одинаково. Это значит, что не записалась ни одна строка из списка `data2`.

Так получилось из-за того, что используется метод `executemany`, и в пределах одной транзакции мы пытаемся записать все 4 строки. Если возникает ошибка с одной из них - откатываются все изменения.

Иногда это именно то поведение, которое нужно. Если же надо, чтобы игнорировались только строки с ошибками, надо использовать метод `execute` и записывать каждую строку отдельно.

В файле `create_sw_inventory_ver4.py` создана функция `write_rows_to_db`, которая уже по очереди пишет данные и, если возникла ошибка, то только изменения для конкретных данных откатываются:

```
# -*- 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

    Функция пытается записать по очереди 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 МАС-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()

```

Теперь результат выполнения будет таким (пропущен только sw7):

```
$ python create_sw_inventory_ver4.py
```

Проверка текущего содержимого БД

```
[('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')]
```

Попытка записать данные с повторяющимся MAC-адресом:

```
[('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')]
```

Запись данных "0055.AAAA.CCCC, sw5, Cisco 3750, London, Green Str" прошла успешно

Запись данных "0066.BBBB.CCCC, sw6, Cisco 3780, London, Green Str" прошла успешно

При записи данных "0000.AAAA.DDDD, sw7, Cisco 2960, London, Green Str" возникла ошибка

Запись данных "0088.AAAA.CCCC, sw8, Cisco 3750, London, Green Str" прошла успешно

Проверка содержимого БД

```
[('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')]
```

Пример использования SQLite

В разделе [Регулярные выражения](#) был пример разбора вывода команды `show ip dhcp snooping binding`. На выходе мы получили информацию о параметрах подключенных устройств (interface, IP, MAC, VLAN).

В таком варианте можно посмотреть только все подключенные устройства к коммутатору. Если же нужно узнать на основании одного из параметров другие, то в таком виде это не очень удобно.

Например, если нужно по IP-адресу получить информацию о том, к какому интерфейсу подключен компьютер, какой у него MAC-адрес и в каком он VLAN, то по выводу скрипта это сделать не очень просто и, главное, не очень удобно.

Запишем информацию, полученную из вывода `sh ip dhcp snooping binding` в SQLite. Это позволит делать запросы по любому параметру и получать недостающие.

Для этого примера достаточно создать одну таблицу, где будет храниться информация.

Определение таблицы прописано в отдельном файле `dhcp_snooping_schema.sql` и выглядит так:

```
create table if not exists dhcp (
    mac      text not NULL primary key,
    ip       text,
    vlan     text,
    interface text
);
```

Для всех полей определен тип данных "текст".

MAC-адрес является первичным ключом нашей таблицы, что вполне логично, так как MAC-адрес должен быть уникальным.

Кроме того, используется выражение `create table if not exists` - SQLite создаст таблицу только в том случае, если она не существует.

Теперь надо создать файл БД, подключиться к базе данных и создать таблицу (файл `create_sqlite_ver1.py`):

```
import sqlite3

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print('Done')

conn.close()
```

Комментарии к файлу:

- при выполнении строки `conn = sqlite3.connect('dhcp_snooping.db')`:
 - создается файл dhcp_snooping.db, если его нет
 - создается объект Connection
- в БД создается таблица (если ее не было) на основании команд, которые указаны в файле dhcp_snooping_schema.sql:
 - открывается файл dhcp_snooping_schema.sql
 - `schema = f.read()` - весь файл считывается в одну строку
 - `conn.executescript(schema)` - метод executescript позволяет выполнять команды SQL, которые прописаны в файле

Выполнение скрипта:

```
$ python create_sqlite_ver1.py
Creating schema...
Done
```

В результате должен быть создан файл БД и таблица dhcp.

Проверить, что таблица создалась, можно с помощью утилиты sqlite3, которая позволяет выполнять запросы прямо в командной строке.

Список созданных таблиц выводится таким образом:

```
$ sqlite3 dhcp_snooping.db "SELECT name FROM sqlite_master WHERE type='table'"
dhcp
```

Теперь нужно записать информацию из вывода команды `sh ip dhcp snooping binding` в таблицу (файл dhcp_snooping.txt):

MacAddress	IpAddress	Lease(sec)	Type	VLAN	Interface
<hr/>					
<hr/>					
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	FastEthernet0/1
0					
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					

Во второй версии скрипта сначала вывод в файле dhcp_snooping.txt обрабатывается регулярными выражениями, а затем записи добавляются в БД (файл create_sqlite_ver2.py):

```
import sqlite3
import re

regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print('Done')

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = '''insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)'''
            conn.execute(query, row)
    except sqlite3.IntegrityError as e:
        print('Error occurred: ', e)

conn.close()
```

Пока что файл БД каждый раз надо удалять, так как скрипт пытается его создать при каждом запуске.

Комментарии к скрипту:

- в регулярном выражении, которое проходится по выводу команды `sh ip dhcp snooping binding`, используются не именованные группы, как в примере раздела [Регулярные выражения](#), а нумерованные
 - группы созданы только для тех элементов, которые нас интересуют
- `result` - это список, в котором хранится результат обработки вывода команды
 - но теперь тут не словари, а кортежи с результатами
 - это нужно для того, чтобы их можно было сразу передавать на запись в БД
- Перебираем в полученном списке кортежей элементы
- В этом скрипте используется еще один вариант записи в БД
 - строка `query` описывает запрос. Но вместо значений указываются знаки вопроса. Такой вариант записи запроса позволяет динамически подставлять значение полей
 - затем методу `execute` передается строка запроса и кортеж `row`, где находятся значения

Выполняем скрипт:

```
$ python create_sqlite_ver2.py
Creating schema...
Done
Inserting DHCP Snooping data
```

Проверим, что данные записались:

```
$ 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
```

Теперь попробуем запросить по определенному параметру:

```
$ 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
```

То есть, теперь на основании одного параметра можно получать остальные.

Переделаем скрипт таким образом, чтобы в нём была проверка на наличие файла dhcp_snooping.db. Если файл БД есть, то не надо создавать таблицу, считаем, что она уже создана.

Файл create_sqlite_ver3.py:

```

import os
import sqlite3
import re

data_filename = 'dhcp_snooping.txt'
db_filename = 'dhcp_snooping.db'
schema_filename = 'dhcp_snooping_schema.sql'

regex = re.compile('(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

db_exists = os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if not db_exists:
    print('Creating schema...')
    with open(schema_filename, 'r') as f:
        schema = f.read()
    conn.executescript(schema)
    print('Done')
else:
    print('Database exists, assume dhcp table does, too.')

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = '''insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)'''
            conn.execute(query, row)
    except sqlite3.IntegrityError as e:
        print('Error occurred: ', e)

conn.close()

```

Теперь есть проверка наличия файла БД, и файл dhcp_snooping.db будет создаваться только в том случае, если его нет. Данные также записываются только в том случае, если не создан файл dhcp_snooping.db.

Разделение процесса создания таблицы и заполнения ее данными вынесено в задания к разделу.

Если файла нет (предварительно его удалить):

```
$ rm dhcp_snooping.db
$ python create_sqlite_ver3.py
Creating schema...
Done
Inserting DHCP Snooping data
```

Проверим. В случае, если файл уже есть, но данные не записаны:

```
$ 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
```

Если есть и БД и данные:

```
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
Error occurred: UNIQUE constraint failed: dhcp.mac
```

Теперь делаем отдельный скрипт, который занимается отправкой запросов в БД и выводом результатов. Он должен:

- ожидать от пользователя ввода параметров:
 - имя параметра
 - значение параметра
- делать нормальный вывод данных по запросу

Файл get_data_ver1.py:

```
# -*- coding: utf-8 -*-
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

key, value = sys.argv[1:]
keys = ['mac', 'ip', 'vlan', 'interface']
keys.remove(key)

conn = sqlite3.connect(db_filename)

#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)
```

Комментарии к скрипту:

- из аргументов, которые передали скрипту,читываются параметры key, value
 - из списка keys удаляется выбранный ключ. Таким образом, в списке остаются только те параметры, которые нужно вывести
- подключаемся к БД
 - conn.row_factory = sqlite3.Row - позволяет далее обращаться к данным в колонках по имени колонки
- из БД выбираются те строки, в которых ключ равен указанному значению
 - в SQL значения можно подставлять через знак вопроса, но нельзя подставлять имя столбца. Поэтому имя столбца подставляется через форматирование строк, а значение - штатным средством SQL.
 - Обратите внимание на (value,) - таким образом передается кортеж с одним элементом
- Полученная информация выводится на стандартный поток вывода:
 - перебираем полученные результаты и выводим только те поля, названия которых находятся в списке keys

Проверим работу скрипта.

Показать параметры хоста с 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
-----
```

Показать хосты в 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
-----
```

Вторая версия скрипта для получения данных с небольшими улучшениями:

- Вместо форматирования строк используется словарь, в котором описаны запросы, соответствующие каждому ключу.
- Выполняется проверка ключа, который был выбран
- Для получения заголовков всех столбцов, который соответствуют запросу, используется метод keys()

Файл 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 = ?'}

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)
```

В этом скрипте есть несколько недостатков:

- не проверяется количество аргументов, которые передаются скрипту
- хотелось бы собирать информацию с разных коммутаторов. А для этого надо добавить поле, которое указывает, на каком коммутаторе была найдена запись

Кроме того, многое нужно доработать в скрипте, который создает БД и записывает данные.

Все доработки будут выполняться в заданиях этого раздела.

Дополнительные материалы

Документация:

- [SQLite Tutorial](#) - подробное описание SQLite
- [Документация модуля sqlite3](#)
- [sqlite3 на сайте PyMOTW](#)

Статьи:

- [A thorough guide to SQLite database operations in Python](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 18.1

На основе файла [create_sqlite_ver3.py](#) из примеров раздела, необходимо создать два скрипта:

- `create_db.py`
 - сюда должна быть вынесена функциональность по созданию БД:
 - должна выполняться проверка наличия файла БД
 - если файла нет, согласно описанию схемы БД в файле `dhcp_snooping_schema.sql`, должна быть создана БД (БД отличается от примера в разделе)
- `add_data.py`
 - с помощью этого скрипта, выполняется добавление данных в БД
 - добавлять надо не только данные из вывода `sh ip dhcp snooping binding`, но и информацию о коммутаторах

Код в скриптах должен быть разбит на функции. Какие именно функции и как разделить код, надо решить самостоятельно. Часть кода может быть глобальной.

В БД теперь две таблицы (схема описана в файле `dhcp_snooping_schema.sql`):

- `switches` - в ней находятся данные о коммутаторах
- `dhcp` - эта таблица осталась такой же как в примере, за исключением поля `switch`
 - это поле ссылается на поле `hostname` в таблице `switches`

Соответственно, в файле `add_data.py` две части:

- информация о коммутаторах добавляется в таблицу switches
 - данные о коммутаторах, находятся в файле switches.yml
- информация на основании вывода sh ip dhcp snooping binding добавляется в таблицу dhcp
 - вывод с трёх коммутаторов:
 - файлы sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt
 - так как таблица dhcp изменилась, и в ней теперь присутствует поле switch, его нужно также заполнять. Имя коммутатора определяется по имени файла с данными

На данном этапе, оба скрипта вызываются без аргументов.

Задание 18.1а

Скопировать скрипт add_data.py из задания 18.1.

Добавить в файл add_data.py, из задания 18.1, проверку на наличие БД:

- если файл БД есть, записать данные
- если файла БД нет, вывести сообщение, что БД нет и её необходимо сначала создать

Задание 18.2

На основе файла get_data_ver1.py из раздела, создать скрипт get_data.py.

Код в скрипте должен быть разбит на функции. Какие именно функции и как разделить код, надо решить самостоятельно. Часть кода может быть глобальной.

В примере из раздела, скрипту передавались два аргумента:

- key - имя столбца, по которому надо найти информацию
- value - значение

Теперь необходимо расширить функциональность таким образом:

- если скрипт был вызван без аргументов, вывести всё содержимое таблицы dhcp
 - отформатировать вывод в виде столбцов
- если скрипт был вызван с двумя аргументами, вывести информацию из таблицы dhcp, которая соответствует полю и значению
- если скрипт был вызван с любым другим количеством аргументов, вывести сообщение, что скрипт поддерживает только два или ноль аргументов

Обработка некорректного ввода аргумента будет выполняться в следующем задании

Файл БД можно скопировать из прошлых заданий

В итоге, вывод должен выглядеть так:

```
$ python get_data.py
```

В таблице dhcp такие записи:

```
-----
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:A6:50 192.168.1.100     100 FastEthernet0/5      SW1
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:A6:50 100.1.1.6         3   FastEthernet0/20    SW3
```

```
$ python get_data.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
switch   : sw1
```

```
$ python get_data.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
switch   : sw1
```

```
-----
mac      : 00:07:BC:3F:A6:50
ip       : 10.1.10.6
interface : FastEthernet0/3
switch   : sw1
```

```
-----
mac      : 00:A9:BB:3D:D6:58
ip       : 10.1.10.20
interface : FastEthernet0/7
switch   : sw2
```

```
$ python get_data.py vlan
```

Пожалуйста, введите два или ноль аргументов

Задание 18.2а

Дополнить скрипт get_data.py из задания 18.2

Теперь должна выполняться проверка не только по количеству аргументов, но и по значению аргументов. Если имя аргумента введено неправильно, надо вывести сообщение об ошибке (пример сообщения ниже).

Файл БД можно скопировать из прошлых заданий

В итоге, вывод должен выглядеть так:

```
$ python get_data.py vln 10
данный параметр не поддерживается.
Допустимые значения параметров: mac, ip, vlan, interface, switch
```

Задание 18.3

В прошлых заданиях информация добавлялась в пустую БД. В этом задании, разбирается ситуация, когда в БД уже есть информация.

Скопируйте скрипт add_data.py и попробуйте выполнить его повторно, на существующей БД. Должна возникнуть ошибка.

При создании схемы БД, было явно указано, что поле MAC-адрес, должно быть уникальным. Поэтому, при добавлении записи с таким же MAC-адресом, возникает ошибка.

Но, нужно каким-то образом обновлять БД, чтобы в ней хранилась актуальная информация.

Например, можно каждый раз, когда записывается информация, предварительно просто удалять всё из таблицы dhcp.

Но, в принципе, старая информация тоже может пригодиться.

Поэтому, мы будем делать немного по-другому. Создадим новое поле active, которое будет указывать является ли запись актуальной.

Поле active должно принимать такие значения:

- 0 - означает False. И используется для того, чтобы отметить запись как неактивную
- 1 - True. Используется чтобы указать, что запись активна

Каждый раз, когда информация из файлов с выводом DHCP snooping добавляется заново, надо пометить все существующие записи (для данного коммутатора), как неактивные (active = 0). Затем можно обновлять информацию и пометить новые записи, как активные (active = 1).

Таким образом, в БД останутся и старые записи, для MAC-адресов, которые сейчас не активны, и появится обновленная информация для активных адресов.

Новая схема БД находится в файле `dhcp_snooping_schema.sql`

Измените скрипт `add_data.py` таким образом, чтобы выполнялись новые условия и заполнялось поле `active`.

Код в скрипте должен быть разбит на функции. Какие именно функции и как разделить код, надо решить самостоятельно. Часть кода может быть глобальной.

Для проверки корректности запроса SQL, можно выполнить его в командной строке, с помощью утилиты `sqlite3`.

Для проверки задания и работы нового поля, попробуйте удалить пару строк из одного из файлов с выводом `dhcp_snooping`. И после этого проверить, что удаление строки отображаются в таблице как неактивные.

Задание 18.4

Обновить файл `get_data` из задания 18.2 или 18.2a. Добавить поддержку столбца `active`, который мы добавили в задании 18.3.

Теперь, при запросе информации, сначала должны отображаться активные записи, а затем, неактивные.

Например:

```
$ python get_data.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
switch   : sw1
-----
```

```
=====
Inactive values:
```

```
-----
mac      : 00:09:23:34:16:18
vlan     : 10
interface : FastEthernet0/4
switch   : sw1
-----
```

```
$ python get_data1.py
```

```
-----
Active values:
```

```
-----
00:09:BB:3D:D6:58 10.1.10.2      10  FastEthernet0/1    sw1    1
00:04:A3:3E:5B:69 10.1.5.2       5   FastEthernet0/10   sw1    1
00:05:B3:7E:9B:60 10.1.5.4       5   FastEthernet0/9    sw1    1
00:07:BC:3F:A6:50 10.1.10.6      10  FastEthernet0/3    sw1    1
00:09:BC:3F:A6:50 192.168.100.100 1   FastEthernet0/7    sw1    1
00:B4:A3:3E:5B:69 10.1.5.20      5   FastEthernet0/5    sw2    1
00:C5:B3:7E:9B:60 10.1.5.40      5   FastEthernet0/9    sw2    1
00:A9:BC:3F:A6:50 10.1.10.60     20  FastEthernet0/2    sw2    1
-----
```

```
Inactive values:
```

```
-----
00:A9:BB:3D:D6:58 10.1.10.20     10  FastEthernet0/7    sw2    0
-----
```

Задание 18.5

Теперь в БД остается и старая информация. И, если какой-то МАС-адрес не появлялся в новых записях, запись с ним, может оставаться в БД очень долго.

И, хотя это может быть полезно, чтобы посмотреть, где МАС-адрес находился в последний раз, постоянно хранить эту информацию не очень полезно.

Например, если запись в БД уже больше месяца, то её можно удалить.

Для того, чтобы сделать такой критерий, нужно ввести новое поле, в которое будет записываться последнее время добавления записи.

Новое поле называется `last_active` и в нём должна находиться строка, в формате:

`YYYY-MM-DD HH:MM:SS`.

В этом задании необходимо:

- изменить, соответственно, таблицу `dhcp` и добавить новое поле.
 - таблицу можно поменять из cli `sqlite`, но файл `dhcp_snooping_schema.sql` тоже необходимо изменить
- изменить скрипт `add_data.py`, чтобы он добавлял к каждой записи время

Как получить строку со временем и датой, в указанном формате, показано в задании. Раскомментируйте строку и посмотрите как она выглядит.

```
import datetime

now = str(datetime.datetime.today().replace(microsecond=0))
#print(now)
```

Задание 18.5a

После выполнения задания 18.5, в таблице `dhcp` есть новое поле `last_active`.

Обновите скрипт `add_data.py`, таким образом, чтобы он удалял все записи, которые были активными более 7 дней назад.

Для того, чтобы получить такие записи, можно просто вручную обновить поле `last_active`.

В файле задания описан пример работы с объектами модуля `datetime`. Обратите внимание, что объекты, как и строки с датой, которые пишутся в БД, можно сравнивать между собой.

```
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))
```

Задание 18.6

В этом задании выложен файл `parse_dhcp_snooping.py`.

В файле созданы несколько функций и описаны аргументы командной строки, которые принимает файл.

В файле `parse_dhcp_snooping.py` нельзя ничего менять.

Есть поддержка аргументов для выполнения всех действий, которые, в предыдущих заданиях, выполнялись в файлах `create_db.py`, `add_data.py` и `get_data.py`.

В файле `parse_dhcp_snooping.py` есть такая строка:

```
import parse_dhcp_snooping_functions as pds
```

И задача этого задания в том, чтобы создать все необходимые функции, в файле `parse_dhcp_snooping_functions.py` на основе информации в файле `parse_dhcp_snooping.py`.

Из файла `parse_dhcp_snooping.py`, необходимо определить:

- какие функции должны быть в файле `parse_dhcp_snooping_functions.py`
- какие параметры создать в этих функциях

Необходимо создать соответствующие функции и перенести в них функционал, который описан в предыдущих заданиях.

Вся необходимая информация, присутствует в функциях `create`, `add`, `get`, в файле `parse_dhcp_snooping.py`.

В принципе, для выполнения задания, не обязательно разбираться с модулем `argparse`. Но, Вы можете почитать о нём в разделе [Модули](#).

Для того, чтобы было проще начать, попробуйте создать необходимые функции в файле `parse_dhcp_snooping_functions.py` и, например, просто выведите аргументы функций, используя `print`.

Потом, можно создать функции, которые запрашивают информацию из БД (базу данных можно скопировать из предыдущих заданий).

Можно создавать любые вспомогательные функции в файле `parse_dhcp_snooping_functions.py`, а не только те, которые вызываются из файла `parse_dhcp_snooping.py`.

Проверьте все операции:

- создание БД
- добавление информации о коммутаторах
- добавление информации на основании вывода `sh ip dhcp snooping binding` из

файлов

- выборку информации из БД (по параметру и всю информацию)

Чтобы было проще понять, как будет выглядеть вызов скрипта, ниже несколько примеров.

```
$ 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}  additional info
create_db           create new db
add                add data to db
get                get data from db

$ 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

$ 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

$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n NAME] [-s SCHEMA]

optional arguments:
-h, --help      show this help message and exit
-n NAME        db filename
-s SCHEMA      db schema filename
```

```
$ python parse_dhcp_snooping.py create_db
Creating DB dhcp_snooping.db with DB schema dhcp_snooping_schema.sql
Creating schema...
Done

$ python parse_dhcp_snooping.py add sw1_dhcp_snooping.txt sw2_dhcp_snooping.txt sw3_dhcp_snooping.txt
Reading info from file(s)
sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt

Adding data to db dhcp_snooping.db

$ python parse_dhcp_snooping.py add -s switches.yml
Adding switch data to database

$ python parse_dhcp_snooping.py get
Showing dhcp_snooping.db content...
-----
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  192.168.1.100   100 FastEthernet0/5      sw1
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  100.1.1.6      3   FastEthernet0/20    sw3

$ python parse_dhcp_snooping.py get -k vlan -v 10
Geting data from DB: dhcp_snooping.db
Request data for host(s) with 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
switch   : sw1
-----
mac      : 00:07:BC:3F:A6:50
ip       : 10.1.10.6
interface : FastEthernet0/3
switch   : sw1
-----
mac      : 00:A9:BB:3D:D6:58
ip       : 10.1.10.20
interface : FastEthernet0/7
switch   : sw2
-----
```

```
$ python parse_dhcp_snooping.py get -k vln -v 10
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]
parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'vln' (choose from 'ma
c', 'ip', 'vlan', 'interface', 'switch')
```

Работа с сетевым оборудованием

В этой части рассматриваются:

- подключение к оборудованию по SSH и Telnet
- одновременное подключение к нескольким устройствам
- создание шаблонов конфигурации с помощью Jinja2
- обработка вывода команд с помощью TextFSM

Подключение к оборудованию

В этом разделе рассматривается как подключиться к оборудованию по протоколам:

- SSH
- Telnet

В Python есть несколько модулей, которые позволяют подключаться к оборудованию и выполнять команды:

- **rexpexpect** - это реализация expect на Python
 - этот модуль позволяет работать с любой интерактивной сессией: ssh, telnet, sftp и др.
 - кроме того, он позволяет выполнять различные команды в ОС (это можно делать и с помощью других модулей)
 - несмотря на то, что rexpexpect может быть менее удобным в использовании, чем другие модули, он реализует более общий функционал и это позволяет использовать его в ситуациях, когда другие модули не работают
- **telnetlib** - этот модуль позволяет подключаться по Telnet
 - в версии 1.0 netmiko также появилась поддержка Telnet, поэтому, если netmiko поддерживает то оборудование, которое используется у Вас, удобней будет использовать его
- **paramiko** - это модуль, который позволяет подключаться по SSHv2
 - он более удобен в использовании, чем rexpexpect, но с более узкой функциональностью (поддерживает только SSH)
- **netmiko** - это модуль, который упрощает использование paramiko для сетевых устройств
 - netmiko это "обертка" вокруг paramiko, которая ориентирована на работу с сетевым оборудованием

В этом разделе рассматриваются все 4 модуля, а также как подключаться к нескольким устройствам параллельно.

В примерах раздела используются три маршрутизатора. К ним нет никаких требований, только настроенный SSH.

Параметры, которые используются в разделе:

- пользователь: cisco
- пароль: cisco
- пароль на режим enable: cisco

- SSH версии 2
- IP-адреса: 192.168.100.1, 192.168.100.2, 192.168.100.3

Ввод пароля

При подключении к оборудованию вручную, как правило, пароль также вводится вручную.

При автоматизации подключения надо решить, каким образом будет передаваться пароль:

- запрашивать пароль при старте скрипта и считывать ввод пользователя
 - минус в том, что будет видно, какие символы вводит пользователь
- записывать логин и пароль в каком-то файле
 - это не очень безопасно

Как правило, один и тот же пользователь использует одинаковый логин и пароль для подключения к оборудованию.

И, как правило, будет достаточно запросить логин и пароль при старте скрипта, а затем использовать их для подключения на разные устройства.

К сожалению, если использовать `input()`, набираемый пароль будет виден. А хотелось бы, чтобы при вводе пароля вводимые символы не отображались.

Модуль `getpass`

Модуль `getpass` позволяет запрашивать пароль, не отображая вводимые символы:

```
In [1]: import getpass  
  
In [2]: password = getpass.getpass()  
Password:  
  
In [3]: print(password)  
testpass
```

Переменные окружения

Еще один вариант хранения пароля (а можно и пользователя) - переменные окружения.

Например, таким образом логин и пароль записываются в переменные:

```
$ export SSH_USER=user  
$ export SSH_PASSWORD=userpass
```

А затем в Python считаются значения в переменные в скрипте:

```
import os  
  
USERNAME = os.environ.get('SSH_USER')  
PASSWORD = os.environ.get('SSH_PASSWORD')
```

Модуль pexpect

Модуль pexpect позволяет автоматизировать интерактивные подключения, такие как:

- telnet
- ssh
- ftp

Для начала, модуль pexpect нужно установить:

```
pip install pexpect
```

Pexpect - это реализация expect на Python.

Логика работы pexpect такая:

- запускается какая-то программа
- pexpect ожидает определенный вывод (приглашение, запрос пароля и подобное)
- получив вывод, он отправляет команды/данные
- последние два действия повторяются столько, сколько нужно

При этом сам pexpect не реализует различные утилиты, а использует уже готовые.

В pexpect есть два основных инструмента:

- функция `run()`
- класс `spawn`

pexpect.run()

Функция `run()` позволяет легко вызвать какую-то программу и вернуть её вывод.

Например:

```
In [1]: import pexpect

In [2]: output = pexpect.run('ls -ls')

In [3]: print(output)
b'total 44\r\n4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py\r\n4 -rw-r--r-- 1 vagrant vagrant 3393 Jul 14 07:15 2_telnetlib.py\r\n4 -rw-r--r-- 1 vagrant vagrant 3452 Jul 14 07:15 3_paramiko.py\r\n4 -rw-r--r-- 1 vagrant vagrant 3127 Jul 14 07:15 4_netmiko.py\r\n4 -rw-r--r-- 1 vagrant vagrant 718 Jul 14 07:15 4_netmiko_telnet.py\r\n4 -rw-r--r-- 1 vagrant vagrant 300 Jul 8 15:31 devices.yaml\r\n4 -rw-r--r-- 1 vagrant vagrant 413 Jul 14 07:15 netmiko_function.py\r\n4 -rw-r--r-- 1 vagrant vagrant 876 Jul 14 07:15 netmiko_multiprocessing.py\r\n4 -rw-r--r-- 1 vagrant vagrant 1147 Jul 14 07:15 netmiko_threading_data_list.py\r\n4 -rw-r--r-- 1 vagrant vagrant 1121 Jul 14 07:15 netmiko_threading_data.py\r\n4 -rw-r--r-- 1 vagrant vagrant 671 Jul 14 07:15 netmiko_threading.py\r\n'

In [4]: print(output.decode('utf-8'))
total 44
4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py
4 -rw-r--r-- 1 vagrant vagrant 3393 Jul 14 07:15 2_telnetlib.py
4 -rw-r--r-- 1 vagrant vagrant 3452 Jul 14 07:15 3_paramiko.py
4 -rw-r--r-- 1 vagrant vagrant 3127 Jul 14 07:15 4_netmiko.py
4 -rw-r--r-- 1 vagrant vagrant 718 Jul 14 07:15 4_netmiko_telnet.py
4 -rw-r--r-- 1 vagrant vagrant 300 Jul 8 15:31 devices.yaml
4 -rw-r--r-- 1 vagrant vagrant 413 Jul 14 07:15 netmiko_function.py
4 -rw-r--r-- 1 vagrant vagrant 876 Jul 14 07:15 netmiko_multiprocessing.py
4 -rw-r--r-- 1 vagrant vagrant 1147 Jul 14 07:15 netmiko_threading_data_list.py
4 -rw-r--r-- 1 vagrant vagrant 1121 Jul 14 07:15 netmiko_threading_data.py
4 -rw-r--r-- 1 vagrant vagrant 671 Jul 14 07:15 netmiko_threading.py
```

pexpect.spawn

Класс `spawn` поддерживает больше возможностей. Он позволяет взаимодействовать с вызванной программой, отправляя данные и ожидая ответ.

Например, таким образом можно инициировать соединение SSH:

```
In [5]: ssh = pexpect.spawn('ssh cisco@192.168.100.1')
```

После выполнения этой строки, подключение готово. Теперь необходимо указать какую строку ожидать. В данном случае, надо дождаться запроса о пароле:

```
In [6]: ssh.expect('[Pp]assword')
Out[6]: 0
```

Обратите внимание как описана строка, которую ожидает `pexpect`: `[Pp]assword`. Это регулярное выражение, которое описывает строку `password` или `Password`. То есть, методу `expect` можно передавать регулярное выражение как аргумент.

Метод `expect` вернул число 0 в результате работы. Это число указывает, что совпадение было найдено и что это элемент с индексом ноль. Индекс тут фигурирует из-за того, что `expect` можно передавать список строк. Например, можно передать список с двумя элементами:

```
In [7]: ssh = pexpect.spawn('ssh cisco@192.168.100.1')

In [8]: ssh.expect(['password', 'Password'])
Out[8]: 1
```

Обратите внимание, что теперь возвращается 1. Это значит, что совпадением было слово `Password`.

Теперь можно отправлять пароль. Для этого используется команда `sendline`:

```
In [9]: ssh.sendline('cisco')
Out[9]: 6
```

Команда `sendline` отправляет строку, автоматически добавляет к ней перевод строки на основе значения `os.linesep`, а затем возвращает число указывающее сколько байт было записано.

В `pexpect` есть несколько вариантов отправки команд, не только `sendline`.

Для того чтобы попасть в режим `enable` цикл `expect-sendline` повторяется:

```
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
```

Теперь можно отправлять команду:

```
In [15]: ssh.sendline('sh ip int br')
Out[15]: 13
```

После отправки команды, pexpect надо указать до какого момента считать вывод.
Указываем, что считать надо до #:

```
In [16]: ssh.expect('#')
Out[16]: 0
```

Вывод команды находится в атрибуте before:

```
In [17]: ssh.before
Out[17]: b'sh ip int br\r\nInterface
          Protocol\r\nEthernet0/0
          up      \r\nEthernet0/1
          up      \r\nEthernet0/2
          up      \r\nEthernet0/3
          up      \r\nEthernet0/3.100
          up      \r\nEthernet0/3.200
          up      \r\nEthernet0/3.300
          up      \r\nR1'
          IP-Address      OK? Method Status
          192.168.100.1   YES NVRAM  up
          192.168.200.1   YES NVRAM  up
          19.1.1.1        YES NVRAM
          192.168.230.1   YES NVRAM
          10.100.0.1      YES NVRAM
          10.200.0.1      YES NVRAM
          10.30.0.1       YES NVRAM
          S NVRAM  up'
```

Так как результат выводится в виде последовательности байтов, надо конвертировать ее в строку:

```
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
Ethernet0/1        192.168.200.1   YES NVRAM  up
Ethernet0/2        19.1.1.1        YES NVRAM  up
Ethernet0/3        192.168.230.1   YES NVRAM  up
Ethernet0/3.100    10.100.0.1      YES NVRAM  up
Ethernet0/3.200    10.200.0.1      YES NVRAM  up
Ethernet0/3.300    10.30.0.1       YES NVRAM  up
R1
```

Завершается сессия вызовом метода close:

```
In [20]: ssh.close()
```

Специальные символы в shell

Pexpect не интерпретирует специальные символы shell, такие как `>`, `|`, `*`.

Для того, чтобы, например, команда `ls -ls | grep SUMMARY` отработала, нужно запустить shell таким образом:

```
In [1]: import pexpect

In [2]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep pexpect"')

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

В предыдущем примере встретилось использование `pexpect.EOF`.

EOF (end of file) — конец файла

Это специальное значение, которое позволяет отреагировать на завершение исполнения команды или сессии, которая была запущена в `spawn`.

При вызове команды `ls -ls` pexpect не получает интерактивный сеанс. Команда выполняется и всё, на этом завершается её работа.

Поэтому если запустить её и указать в `expect` приглашение, возникнет ошибка:

```
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')
...
```

Но, если передать в `expect` EOF, ошибки не будет.

Метод pexpect.expect

В `pexpect.expect` как шаблон может использоваться:

- регулярное выражение
- EOF - этот шаблон позволяет среагировать на исключение EOF
- TIMEOUT - исключение timeout (по умолчанию значение timeout = 30 секунд)
- compiled re

Еще одна очень полезная возможность pexpect.expect: можно передавать не одно значение, а список.

Например:

```
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
```

Тут несколько важных моментов:

- когда pexpect.expect вызывается со списком, можно указывать разные ожидаемые строки
- кроме строк, можно указывать исключения
- pexpect.expect возвращает номер элемента списка, который сработал
 - в данном случае номер 2, так как исключение EOF находится в списке под номером два
- за счет такого формата можно делать ответвления в программе, в зависимости от того, с каким элементом было совпадение

Пример использования pexpect

Пример использования pexpect для подключения к оборудованию и передачи команды show (файл 1_pexpect.py):

```
import pexpect
import getpass
import sys

COMMAND = sys.argv[1]
USER = input('Username: ')
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1', '192.168.100.2', '192.168.100.3']

for IP in DEVICES_IP:
    print('Connection to device {}'.format(IP))
    with pexpect.spawn('ssh {}@{}'.format(USER, IP)) as ssh:

        ssh.expect('Password:')
        ssh.sendline(PASSWORD)

        ssh.expect('#')
        ssh.sendline('enable')

        ssh.expect('Password:')
        ssh.sendline(ENABLE_PASS)

        ssh.expect('#')
        ssh.sendline('terminal length 0')

        ssh.expect('#')
        ssh.sendline(COMMAND)

        ssh.expect('#')
        print(ssh.before.decode('utf-8'))
```

Комментарии к скрипту:

- команда, которую нужно выполнить, передается как аргумент
- затем запрашивается логин, пароль и пароль на режим enable
 - пароли запрашиваются с помощью модуля getpass
- `ip_list` - это список IP-адресов устройств, к которым будет выполняться подключение
- в цикле выполняется подключение к устройствам из списка

- в классе `spawn` выполняется подключение по SSH к текущему адресу, используя указанное имя пользователя
- после этого начинают чередоваться пары методов: `expect` и `sendline`
 - `expect` - ожидание подстроки
 - `sendline` - когда строка появилась, отправляется команда
- так происходит до конца цикла, и только последняя команда отличается:
 - `before` позволяет считать всё, что поймал `rexpect` до предыдущей подстроки в `expect`

Обратите внимание на строку `ssh.expect('#>')`. Метод `expect` ожидает не просто строку, а регулярное выражение.

Выполнение скрипта выглядит так:

```
$ python 1_pexpect.py "sh ip int br"
Username: nata
Password:
Enter enable secret:
Connection to device 192.168.100.1
sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.1   YES NVRAM  up           up
FastEthernet0/1    unassigned     YES NVRAM  up           up
FastEthernet0/1.10 10.1.10.1     YES manual up        up
FastEthernet0/1.20 10.1.20.1     YES manual up        up
FastEthernet0/1.30 10.1.30.1     YES manual up        up
FastEthernet0/1.40 10.1.40.1     YES manual up        up
FastEthernet0/1.50 10.1.50.1     YES manual up        up
FastEthernet0/1.60 10.1.60.1     YES manual up        up
FastEthernet0/1.70 10.1.70.1     YES manual up        up
R1
Connection to device 192.168.100.2
sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.2   YES NVRAM  up           up
FastEthernet0/1    unassigned     YES NVRAM  up           up
FastEthernet0/1.10 10.2.10.1     YES manual up        up
FastEthernet0/1.20 10.2.20.1     YES manual up        up
FastEthernet0/1.30 10.2.30.1     YES manual up        up
FastEthernet0/1.40 10.2.40.1     YES manual up        up
FastEthernet0/1.50 10.2.50.1     YES manual up        up
FastEthernet0/1.60 10.2.60.1     YES manual up        up
FastEthernet0/1.70 10.2.70.1     YES manual up        up
R2
Connection to device 192.168.100.3
sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.3   YES NVRAM  up           up
FastEthernet0/1    unassigned     YES NVRAM  up           up
FastEthernet0/1.10 10.3.10.1     YES manual up        up
FastEthernet0/1.20 10.3.20.1     YES manual up        up
FastEthernet0/1.30 10.3.30.1     YES manual up        up
FastEthernet0/1.40 10.3.40.1     YES manual up        up
FastEthernet0/1.50 10.3.50.1     YES manual up        up
FastEthernet0/1.60 10.3.60.1     YES manual up        up
FastEthernet0/1.70 10.3.70.1     YES manual up        up
R3
```

Обратите внимание, что, так как в последнем expect указано, что надо ожидать подстроку `#`, метод before показал и команду, и имя хоста.

Модуль telnetlib

Модуль `telnetlib` входит в стандартную библиотеку Python. Это реализация клиента `telnet`.

Подключиться по `telnet` можно и используя `rexpert`. Плюс `telnetlib` в том, что этот модуль входит в стандартную библиотеку Python.

Принцип работы `telnetlib` напоминает `rexpert`, но есть несколько отличий. Самое заметное отличие в том, что `telnetlib` требует передачи байтовой строки, а не обычной.

Подключение выполняется таким образом:

```
In [1]: telnet = telnetlib.Telnet('192.168.100.1')
```

С помощью метода `read_until` указывается до какой строки считать вывод. При этом, как аргумент надо передавать не обычную строку, а байты:

```
In [2]: telnet.read_until(b'Username')
Out[2]: b'\r\n\r\nUser Access Verification\r\n\r\nUsername'
```

Метод `read_until` возвращает все, что он считал до указанной строки.

Для передачи данных используется метод `write`. Ему нужно передавать байтовую строку:

```
In [3]: telnet.write(b'cisco\n')
```

Читаем вывод до слова `Password` и передаем пароль:

```
In [4]: telnet.read_until(b'Password')
Out[4]: b': cisco\r\nPassword'

In [5]: telnet.write(b'cisco\n')
```

Теперь можно указать, что надо считать вывод до приглашения, а затем отправить команду:

```
In [6]: telnet.read_until(b'>')
Out[6]: b': \r\nR1>

In [7]: telnet.write(b'sh ip int br\r\n')
```

После отправки команды можно продолжать использовать метод `read_until`:

```
In [8]: telnet.read_until(b'>')
Out[8]: b'sh ip int br\r\nInterface
          Protocol\r\nEthernet0/0
              up      \r\nEthernet0/1
              up      \r\nEthernet0/2
              up      \r\nEthernet0/3
up           up      \r\nEthernet0/3.100
up           up      \r\nEthernet0/3.200
AM up        up      \r\nEthernet0/3.300
VRAM up      up      \r\nEthernet0/3.300
NVRAM up      up      \r\nR1>'
```

Или использовать еще один метод для чтения `read_very_eager`.

При использовании метода `read_very_eager`, можно отправить несколько команд, а затем считать весь доступный вывод:

```
In [9]: telnet.write(b'sh arp\n')

In [10]: telnet.write(b'sh clock\n')

In [11]: telnet.write(b'sh ip int br\n')

In [12]: all_result = telnet.read_very_eager().decode('utf-8')

In [13]: print(all_result)
sh arp
Protocol Address          Age (min) Hardware Addr Type   Interface
Internet 10.30.0.1        -       aabb.cc00.6530 ARPA   Ethernet0/3.300
Internet 10.100.0.1       -       aabb.cc00.6530 ARPA   Ethernet0/3.100
Internet 10.200.0.1       -       aabb.cc00.6530 ARPA   Ethernet0/3.200
Internet 19.1.1.1         -       aabb.cc00.6520 ARPA   Ethernet0/2
Internet 192.168.100.1    -       aabb.cc00.6500 ARPA   Ethernet0/0
Internet 192.168.100.2    124    aabb.cc00.6600 ARPA   Ethernet0/0
Internet 192.168.100.3    143    aabb.cc00.6700 ARPA   Ethernet0/0
Internet 192.168.100.100  160    aabb.cc80.c900 ARPA   Ethernet0/0
Internet 192.168.200.1    -       0203.e800.6510 ARPA   Ethernet0/1
Internet 192.168.200.100  13     0800.27ac.16db ARPA   Ethernet0/1
Internet 192.168.230.1    -       aabb.cc00.6530 ARPA   Ethernet0/3
R1>sh clock
*19:18:57.980 UTC Fri Nov 3 2017
R1>sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
Ethernet0/0        192.168.100.1  YES NVRAM up           up
Ethernet0/1        192.168.200.1  YES NVRAM up           up
Ethernet0/2        19.1.1.1       YES NVRAM up           up
Ethernet0/3        192.168.230.1  YES NVRAM up           up
Ethernet0/3.100    10.100.0.1    YES NVRAM up           up
Ethernet0/3.200    10.200.0.1    YES NVRAM up           up
Ethernet0/3.300    10.30.0.1     YES NVRAM up           up
R1>
```

С `read_until` будет немного другой подход. Можно выполнить те же три команды, но затем получать вывод по одной за счет чтения до строки с приглашением:

```
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 Inte
rface\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\nInt
ernet 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 192.168.1
00.1 - 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 a
abb.cc00.6700 ARPA Ethernet0/0\r\nInternet 192.168.100.100 162 aabb.cc80.c
900 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 Ethernet
0/1\r\nInternet 192.168.230.1 - aabb.cc00.6530 ARPA Ethernet0/3\r\nR1>'
```



```
In [18]: telnet.read_until(b'>')
Out[18]: b'sh clock\r\n*19:20:39.388 UTC Fri Nov 3 2017\r\nR1>'

In [19]: telnet.read_until(b'>')
Out[19]: b'sh ip int br\r\nInterface IP-Address OK? Method Status
s Protocol\r\nEthernet0/0 192.168.100.1 YES NVRAM up
up \r\nEthernet0/1 192.168.200.1 YES NVRAM u
p up \r\nEthernet0/2 19.1.1.1 YES NVRAM
up up \r\nEthernet0/3 192.168.230.1 YES NVRA
M up up \r\nEthernet0/3.100 10.100.0.1 YES NV
RAM up up \r\nEthernet0/3.200 10.200.0.1 YES
NVRAM up up \r\nEthernet0/3.300 10.30.0.1 YE
S NVRAM up up \r\nR1>'
```

Важное отличие между `read_until` и `read_very_eager` заключается в том, как они реагируют на отсутствие вывода.

Метод `read_until` ждет определенную строку. По умолчанию, если ее нет, метод "зависнет". Опциональный параметр `timeout` позволяет указать сколько ждать нужную строку:

```
In [20]: telnet.read_until(b'>', timeout=5)
Out[20]: b''
```

Если за указанное время строка не появилась, возвращается пустая строка.

Метод `read_very_eager` просто вернет пустую строку, если вывода нет:

```
In [21]: telnet.read_very_eager()
Out[21]: b''
```

Метод `expect` позволяет указывать список с регулярными выражениями. Он работает похоже на `reexpect`, но в модуле `telnetlib` всегда надо передавать список регулярных выражений.

При этом, можно передавать байтовые строки или компилированные регулярные выражения:

```
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*n*19:35:10.984 UTC Fri Nov 3 2017\r\nR1>')
```

Метод `expect` возвращает кортеж из трех элементов:

- индекс выражения, которое совпало
- объект Match
- байтовая строка, которая содержит все считанное до регулярного выражения и включая его

Соответственно, при необходимости, с этими элементами можно дальше работать:

```
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>'
```

Закрывается соединение методом close:

```
In [32]: telnet.close()
```

Пример использования telnetlib

Принцип работы telnetlib напоминает rexpert, поэтому пример ниже должен быть понятен.

Файл 2_telnetlib.py:

```
import telnetlib
import time
import getpass
import sys

COMMAND = sys.argv[1].encode('utf-8')
USER = input('Username: ').encode('utf-8')
PASSWORD = getpass.getpass().encode('utf-8')
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ').encode('utf-8')

DEVICES_IP = ['192.168.100.1', '192.168.100.2', '192.168.100.3']

for IP in DEVICES_IP:
    print('Connection to device {}'.format(IP))
    with telnetlib.Telnet(IP) as t:

        t.read_until(b'Username:')
        t.write(USER + b'\n')

        t.read_until(b'Password:')
        t.write(PASSWORD + b'\n')
        t.write(b'enable\n')

        t.read_until(b'Password:')
        t.write(ENABLE_PASS + b'\n')
        t.write(b'terminal length 0\n')
        t.write(COMMAND + b'\n')

        time.sleep(5)

        output = t.read_very_eager().decode('utf-8')
        print(output)
```

Использование объекта Telnet как менеджера контекса добавлено в версии 3.6

telnetlib очень похож на rexpert:

- `with telnetlib.Telnet(ip) as t` - класс Telnet представляет соединение к серверу.
 - в данном случае ему передается только IP-адрес, но можно передать и порт, к которому нужно подключаться
- `read_until` - похож на метод `expect` в модуле rexpert. Указывает, до какой строки

следует считывать вывод

- `write` - передать строку
- `read_very_eager` - считать всё, что получается

Выполнение скрипта:

```
$ python 2_telnetlib.py "sh ip int br"
Username: cisco
Password:
Enter enable secret:
Connection to device 192.168.100.1

R1#terminal length 0
R1#sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.1   YES NVRAM up           up
FastEthernet0/1    unassigned      YES NVRAM up           up
FastEthernet0/1.10 10.1.10.1     YES manual up         up
FastEthernet0/1.20 10.1.20.1     YES manual up         up
FastEthernet0/1.30 10.1.30.1     YES manual up         up
FastEthernet0/1.40 10.1.40.1     YES manual up         up
FastEthernet0/1.50 10.1.50.1     YES manual up         up
FastEthernet0/1.60 10.1.60.1     YES manual up         up
FastEthernet0/1.70 10.1.70.1     YES manual up         up
R1#
Connection to device 192.168.100.2

R2#terminal length 0
R2#sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.2   YES NVRAM up           up
FastEthernet0/1    unassigned      YES NVRAM up           up
FastEthernet0/1.10 10.2.10.1     YES manual up         up
FastEthernet0/1.20 10.2.20.1     YES manual up         up
FastEthernet0/1.30 10.2.30.1     YES manual up         up
FastEthernet0/1.40 10.2.40.1     YES manual up         up
FastEthernet0/1.50 10.2.50.1     YES manual up         up
FastEthernet0/1.60 10.2.60.1     YES manual up         up
FastEthernet0/1.70 10.2.70.1     YES manual up         up
R2#
Connection to device 192.168.100.3

R3#terminal length 0
R3#sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.3   YES NVRAM up           up
FastEthernet0/1    unassigned      YES NVRAM up           up
FastEthernet0/1.10 10.3.10.1     YES manual up         up
FastEthernet0/1.20 10.3.20.1     YES manual up         up
FastEthernet0/1.30 10.3.30.1     YES manual up         up
FastEthernet0/1.40 10.3.40.1     YES manual up         up
FastEthernet0/1.50 10.3.50.1     YES manual up         up
FastEthernet0/1.60 10.3.60.1     YES manual up         up
FastEthernet0/1.70 10.3.70.1     YES manual up         up
R3#
```


Модуль paramiko

Paramiko - это реализация протокола SSHv2 на Python. Paramiko предоставляет функциональность клиента и сервера. Мы будем рассматривать только функциональность клиента.

Так как Paramiko не входит в стандартную библиотеку модулей Python, его нужно установить:

```
pip install paramiko
```

Пример использования Paramiko (файл 3_paramiko.py):

```
import paramiko
import getpass
import sys
import time

COMMAND = sys.argv[1]
USER = input('Username: ')
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1', '192.168.100.2', '192.168.100.3']

for IP in DEVICES_IP:
    print('Connection to device {}'.format( IP ))
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    client.connect(hostname=IP, username=USER, password=PASSWORD,
                   look_for_keys=False, allow_agent=False)

    with client.invoke_shell() as ssh:
        ssh.send('enable\n')
        ssh.send(ENABLE_PASS + '\n')
        time.sleep(1)

        ssh.send('terminal length 0\n')
        time.sleep(1)
        ssh.recv(1000).decode('utf-8')

        ssh.send(COMMAND + '\n')
        time.sleep(2)
        result = ssh.recv(5000).decode('utf-8')
        print(result)
```

Комментарии к скрипту:

- `client = paramiko.SSHClient()`
 - этот класс представляет соединение к SSH-серверу. Он выполняет аутентификацию клиента.
- `client.set_missing_host_key_policy(paramiko.AutoAddPolicy())`
 - `set_missing_host_key_policy` - устанавливает, какую политику использовать, когда выполняется подключение к серверу, ключ которого неизвестен.
 - `paramiko.AutoAddPolicy()` - политика, которая автоматически добавляет новое имя хоста и ключ в локальный объект HostKeys.
- `client.connect(IP, username=USER, password=PASSWORD, look_for_keys=False, allow_agent=False)`
 - `client.connect` - метод, который выполняет подключение к SSH-серверу и аутентифицирует подключение
 - `hostname` - имя хоста или IP-адрес
 - `username` - имя пользователя
 - `password` - пароль
 - `look_for_keys` - по умолчанию paramiko выполняет аутентификацию по ключам. Чтобы отключить это, надо поставить флаг в `False`
 - `allow_agent` - paramiko может подключаться к локальному SSH агенту ОС. Это нужно при работе с ключами, а так как в данном случае аутентификация выполняется по логину/паролю, это нужно отключить.
 - `with client.invoke_shell() as ssh`
 - после выполнения предыдущей команды уже есть подключение к серверу. Метод `invoke_shell` позволяет установить интерактивную сессию SSH с сервером.
 - Внутри установленной сессии выполняются команды и получаются данные:
 - `ssh.send` - отправляет указанную строку в сессию
 - `ssh.recv` - получает данные из сессии. В скобках указывается максимальное значение в байтах, которое можно получить. Этот метод возвращает считанную строку
 - Кроме этого, между отправкой команды и считыванием кое-где стоит строка `time.sleep`
 - с помощью неё указывается пауза - сколько времени подождать, прежде чем скрипт продолжит выполнятся. Это делается для того, чтобы дождаться выполнения команды на оборудовании

Так выглядит результат выполнения скрипта:

```
$ python 3_paramiko.py "sh ip int br"
Username: cisco
Password:
```

```

Enter enable secret:
Connection to device 192.168.100.1

R1>enable
Password:
R1#terminal length 0

R1#
sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.1   YES NVRAM  up           up
FastEthernet0/1    unassigned      YES NVRAM  up           up
FastEthernet0/1.10 10.1.10.1     YES manual up        up
FastEthernet0/1.20 10.1.20.1     YES manual up        up
FastEthernet0/1.30 10.1.30.1     YES manual up        up
FastEthernet0/1.40 10.1.40.1     YES manual up        up
FastEthernet0/1.50 10.1.50.1     YES manual up        up
FastEthernet0/1.60 10.1.60.1     YES manual up        up
FastEthernet0/1.70 10.1.70.1     YES manual up        up

R1#
Connection to device 192.168.100.2

R2>enable
Password:
R2#terminal length 0

R2#
sh ip int br
FastEthernet0/0    192.168.100.2   YES NVRAM  up           up
FastEthernet0/1    unassigned      YES NVRAM  up           up
FastEthernet0/1.10 10.2.10.1     YES manual up        up
FastEthernet0/1.20 10.2.20.1     YES manual up        up
FastEthernet0/1.30 10.2.30.1     YES manual up        up
FastEthernet0/1.40 10.2.40.1     YES manual up        up
FastEthernet0/1.50 10.2.50.1     YES manual up        up
FastEthernet0/1.60 10.2.60.1     YES manual up        up
FastEthernet0/1.70 10.2.70.1     YES manual up        up

R2#
Connection to device 192.168.100.3

R3>enable
Password:
R3#terminal length 0

R3#
sh ip int br
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.3   YES NVRAM  up           up
FastEthernet0/1    unassigned      YES NVRAM  up           up
FastEthernet0/1.10 10.3.10.1     YES manual up        up
FastEthernet0/1.20 10.3.20.1     YES manual up        up
FastEthernet0/1.30 10.3.30.1     YES manual up        up
FastEthernet0/1.40 10.3.40.1     YES manual up        up
FastEthernet0/1.50 10.3.50.1     YES manual up        up

```

```
FastEthernet0/1.60      10.3.60.1        YES manual up          up
FastEthernet0/1.70      10.3.70.1        YES manual up          up
R3#
```

Обратите внимание, что в вывод попал и процесс ввода пароля enable, и команда terminal length.

Это связано с тем, что paramiko собирает весь вывод в буфер. И, при вызове метода `recv` (например, `ssh.recv(1000)`), paramiko возвращает всё, что есть в буфере. После выполнения `recv` буфер пуст.

Поэтому, если нужно получить только вывод команды `sh ip int br`, то надо оставить `recv`, но не делать `print`:

```
ssh.send('enable\n')
ssh.send(ENABLE_PASS + '\n')
time.sleep(1)

ssh.send('terminal length 0\n')
time.sleep(1)
#Тут мы вызываем recv, но не выводим содержимое буфера
ssh.recv(1000)

ssh.send(COMMAND + '\n')
time.sleep(3)
result = ssh.recv(5000).decode('utf-8')
print(result)
```

Модуль netmiko

Netmiko - это модуль, который позволяет упростить использование paramiko для сетевых устройств.

Грубо говоря, netmiko - это такая "обертка" для paramiko.

Сначала netmiko нужно установить:

```
pip install netmiko
```

Пример использования netmiko (файл 4_netmiko.py):

```
import getpass
import sys

from netmiko import ConnectHandler


COMMAND = sys.argv[1]
USER = input('Username: ')
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1', '192.168.100.2', '192.168.100.3']


for IP in DEVICES_IP:
    print('Connection to device {}'.format(IP))
    DEVICE_PARAMS = {'device_type': 'cisco_ios',
                     'ip': IP,
                     'username': USER,
                     'password': PASSWORD,
                     'secret': ENABLE_PASS}

    with ConnectHandler(**DEVICE_PARAMS) as ssh:
        ssh.enable()

        result = ssh.send_command(COMMAND)
        print(result)
```

Посмотрите, насколько проще выглядит этот пример с netmiko.

Разберемся с содержимым скрипта:

- DEVICE_PARAMS - это словарь, в котором указываются параметры устройства

- device_type - это предопределенные значения, которые понимает netmiko
 - в данном случае, так как подключение выполняется к устройству с Cisco IOS, используется значение 'cisco_ios'
- `with ConnectHandler(**DEVICE_PARAMS) as ssh` - устанавливается соединение с устройством на основе параметров, которые находятся в словаре
 - две звездочки перед словарем - это распаковка словаря (подробнее в разделе [Распаковка аргументов](#))
- `ssh.enable()` - переход в режим enable
 - пароль передается автоматически
 - используется значение ключа secret, который указан в словаре DEVICE_PARAMS
- `result = ssh.send_command(COMMAND)` - отправка команды и получение вывода

В этом примере не передается команда terminal length, так как netmiko по умолчанию выполняет эту команду.

Так выглядит результат выполнения скрипта:

```
$ python 4_netmiko.py "sh ip int br"
Username: cisco
Password:
Enter enable password:
Connection to device 192.168.100.1
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.1  YES NVRAM up           up
FastEthernet0/1    unassigned     YES NVRAM up           up
FastEthernet0/1.10 10.1.10.1    YES manual up          up
FastEthernet0/1.20 10.1.20.1    YES manual up          up
FastEthernet0/1.30 10.1.30.1    YES manual up          up
FastEthernet0/1.40 10.1.40.1    YES manual up          up
FastEthernet0/1.50 10.1.50.1    YES manual up          up
FastEthernet0/1.60 10.1.60.1    YES manual up          up
FastEthernet0/1.70 10.1.70.1    YES manual up          up
Connection to device 192.168.100.2
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.2  YES NVRAM up           up
FastEthernet0/1    unassigned     YES NVRAM up           up
FastEthernet0/1.10 10.2.10.1    YES manual up          up
FastEthernet0/1.20 10.2.20.1    YES manual up          up
FastEthernet0/1.30 10.2.30.1    YES manual up          up
FastEthernet0/1.40 10.2.40.1    YES manual up          up
FastEthernet0/1.50 10.2.50.1    YES manual up          up
FastEthernet0/1.60 10.2.60.1    YES manual up          up
FastEthernet0/1.70 10.2.70.1    YES manual up          up
Connection to device 192.168.100.3
Interface          IP-Address      OK? Method Status      Protocol
FastEthernet0/0    192.168.100.3  YES NVRAM up           up
FastEthernet0/1    unassigned     YES NVRAM up           up
FastEthernet0/1.10 10.3.10.1    YES manual up          up
FastEthernet0/1.20 10.3.20.1    YES manual up          up
FastEthernet0/1.30 10.3.30.1    YES manual up          up
FastEthernet0/1.40 10.3.40.1    YES manual up          up
FastEthernet0/1.50 10.3.50.1    YES manual up          up
FastEthernet0/1.60 10.3.60.1    YES manual up          up
FastEthernet0/1.70 10.3.70.1    YES manual up          up
```

В выводе нет никаких лишних приглашений, только вывод команды sh ip int br.

Так как netmiko наиболее удобный модуль для подключения к сетевому оборудования, разберемся с ним подробней.

Поддерживаемые типы устройств

Netmiko поддерживает несколько типов устройств:

- Arista vEOS
- Cisco ASA
- Cisco IOS
- Cisco IOS-XR
- Cisco SG300
- HP Comware7
- HP ProCurve
- Juniper Junos
- Linux
- и другие

Актуальный список можно посмотреть в [репозитории](#) модуля.

Словарь, определяющий параметры устройств

В словаре могут указываться такие параметры:

```
cisco_router = {'device_type': 'cisco_ios', # предопределенный тип устройства
                 'ip': '192.168.1.1', # адрес устройства
                 'username': 'user', # имя пользователя
                 'password': 'userpass', # пароль пользователя
                 'secret': 'enablepass', # пароль режима enable
                 'port': 20022, # порт SSH, по умолчанию 22
                 }
```

Подключение по SSH

```
ssh = ConnectHandler(**cisco_router)
```

Режим enable

Перейти в режим enable:

```
ssh.enable()
```

Выйти из режима enable:

```
ssh.exit_enable_mode()
```

Отправка команд

В netmiko есть несколько способов отправки команд:

- `send_command` - отправить одну команду
- `send_config_set` - отправить список команд или команду в конфигурационном режиме
- `send_config_from_file` - отправить команды из файла (использует внутри метод `send_config_set`)
- `send_command_timing` - отправить команду и подождать вывод на основании таймера

send_command

Метод `send_command` позволяет отправить одну команду на устройство.

Например:

```
result = ssh.send_command('show ip int br')
```

Метод работает таким образом:

- отправляет команду на устройство и получает вывод до строки с приглашением или до указанной строки
 - приглашение определяется автоматически
 - если на вашем устройстве оно не определилось, можно просто указать строку, до которой считывать вывод
 - ранее так работал метод `send_command_expect`, но с версии 1.0.0 так работает `send_command`, а метод `send_command_expect` оставлен для совместимости
- метод возвращает вывод команды
- методу можно передавать такие параметры:
 - `command_string` - команда
 - `expect_string` - до какой строки считывать вывод
 - `delay_factor` - параметр позволяет увеличить задержку до начала поиска

строки

- `max_loops` - количество итераций, до того как метод выдаст ошибку (исключение). По умолчанию 500
- `strip_prompt` - удалить приглашение из вывода. По умолчанию удаляется
- `strip_command` - удалить саму команду из вывода

В большинстве случаев достаточно будет указать только команду.

send_config_set

Метод `send_config_set` позволяет отправить команду или несколько команд конфигурационного режима.

Пример использования:

```
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)
```

Метод работает таким образом:

- заходит в конфигурационный режим,
- затем передает все команды
- и выходит из конфигурационного режима
 - в зависимости от типа устройства, выхода из конфигурационного режима может и не быть. Например, для IOS-XR выхода не будет, так как сначала надо закоммитить изменения

send_config_from_file

Метод `send_config_from_file` отправляет команды из указанного файла в конфигурационный режим.

Пример использования:

```
result = ssh.send_config_from_file('config_ospf.txt')
```

Метод открывает файл, считывает команды и передает их методу `send_config_set`.

Дополнительные методы

Кроме перечисленных методов для отправки команд, netmiko поддерживает такие методы:

- `config_mode` - перейти в режим конфигурации
 - `ssh.config_mode()`
- `exit_config_mode` - выйти из режима конфигурации
 - `ssh.exit_config_mode()`
- `check_config_mode` - проверить, находится ли netmiko в режиме конфигурации (возвращает True, если в режиме конфигурации, и False - если нет)
 - `ssh.check_config_mode()`
- `find_prompt` - возвращает текущее приглашение устройства
 - `ssh.find_prompt()`
- `commit` - выполнить commit на IOS-XR и Juniper
 - `ssh.commit()`
- `disconnect` - завершить соединение SSH

Тут `ssh` - это созданное предварительно соединение SSH: `ssh = ConnectHandler(**cisco_router)`

Telnet

С версии 1.0.0 netmiko поддерживает подключения по Telnet, пока что только для Cisco IOS устройств.

Внутри netmiko использует `telnetlib` для подключения по Telnet. Но, при этом, предоставляет тот же интерфейс для работы, что и подключение по SSH.

Для того, чтобы подключиться по Telnet, достаточно в словаре, который определяет параметры подключения, указать тип устройства '`cisco_ios_telnet`':

```
DEVICE_PARAMS = {'device_type': 'cisco_ios_telnet',
                 'ip': IP,
                 'username': USER,
                 'password': PASSWORD,
                 'secret': ENABLE_PASS }
```

В остальном, методы, которые применимы к SSH, применимы и к Telnet. Пример, аналогичный примеру с SSH (файл `4_netmiko_telnet.py`):

```
import getpass
import sys
import time

from netmiko import ConnectHandler


COMMAND = sys.argv[1]
USER = input('Username: ')
PASSWORD = getpass.getpass()
ENABLE_PASS = getpass.getpass(prompt='Enter enable password: ')

DEVICES_IP = ['192.168.100.1', '192.168.100.2', '192.168.100.3']


for IP in DEVICES_IP:
    print('Connection to device {}'.format(IP))
    DEVICE_PARAMS = {'device_type': 'cisco_ios_telnet',
                     'ip': IP,
                     'username': USER,
                     'password': PASSWORD,
                     'secret': ENABLE_PASS,
                     'verbose': True}
    with ConnectHandler(**DEVICE_PARAMS) as telnet:
        telnet.enable()

        result = telnet.send_command(COMMAND)
        print(result)
```

Аналогично работают и методы:

- send_command_timing()
- find_prompt()
- send_config_set()
- send_config_from_file()
- check_enable_mode()
- disconnect()

Дополнительные материалы

Документация:

- [pexpect](#)
- [telnetlib](#)
- [paramiko Client](#)
- [paramiko Channel](#)
- [netmiko](#)
- [threading](#)
- [multiprocessing](#)
- [queue](#)
- [time](#)
- [datetime](#)
- [getpass](#)

Статьи:

- [Netmiko Library](#)
- [Automate SSH connections with netmiko](#)
- [Network Automation Using Python: BGP Configuration](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 19.1

Создать функцию `send_show_command`.

Функция подключается по SSH (с помощью `netmiko`) к устройству и выполняет указанную команду.

Параметры функции:

- `device` - словарь с параметрами подключения к устройству
- `command` - команда, которую надо выполнить

Функция возвращает словарь с результатами выполнения команды:

- ключ - IP устройства
- значение - результат выполнения команды

Отправить команду `command` на все устройства из файла `devices.yaml` (для этого надо считать информацию из файла) с помощью функции `send_show_command`.

```
command = "sh ip int br"
```

Задание 19.1a

Переделать функцию send_show_command из задания 19.1 таким образом, чтобы обрабатывалось исключение, которое генерируется при ошибке аутентификации на устройстве.

При возникновении ошибки, должно выводиться сообщение исключения.

Для проверки измените пароль на устройстве или в файле devices.yaml.

Задание 19.1b

Дополнить функцию send_show_command из задания 19.1а таким образом, чтобы обрабатывалось не только исключение, которое генерируется при ошибке аутентификации на устройстве, но и исключение, которое генерируется, когда IP-адрес устройства недоступен.

При возникновении ошибки, должно выводиться сообщение исключения.

Для проверки измените IP-адрес на устройстве или в файле devices.yaml.

Задание 19.2

Создать функцию send_config_commands

Функция подключается по SSH (с помощью netmiko) к устройству и выполняет перечень команд в конфигурационном режиме на основании переданных аргументов.

Параметры функции:

- device - словарь с параметрами подключения к устройству
- config_commands - список команд, которые надо выполнить

Функция возвращает словарь с результатами выполнения команды:

- ключ - IP устройства
- значение - вывод с выполнением команд

Отправить список команд commands на все устройства из файла devices.yaml (для этого надо считать информацию из файла) с помощью функции send_config_commands.

```
commands = [ 'logging 10.255.255.1',
              'logging buffered 20010',
              'no logging console' ]
```

Задание 19.2a

Дополнить функцию `send_config_commands` из задания 19.2

Добавить аргумент `verbose`, который контролирует будет ли результат выполнения команд выводиться на стандартный поток вывода.

По умолчанию, результат должен выводиться.

Задание 19.2b

В этом задании необходимо переделать функцию `send_config_commands` из задания 19.2a или 19.2 и добавить проверку на ошибки.

При выполнении каждой команды, скрипт должен проверять результат на такие ошибки:

- Invalid input detected, Incomplete command, Ambiguous command

Если при выполнении какой-то из команд возникла ошибка, функция должна выводить сообщение на стандартный поток вывода с информацией о том, какая ошибка возникла, при выполнении какой команды и на каком устройстве.

При этом, параметр `verbose` также должен работать, но теперь он отвечает за вывод только тех команд, которые выполнились корректно.

Функция `send_config_commands` теперь должна возвращать кортеж из двух словарей:

- первый словарь с выводом команд, которые выполнились без ошибки
- второй словарь с выводом команд, которые выполнились с ошибками

Оба словаря в формате:

- ключ - команда
- значение - вывод с выполнением команд

Отправить список команд `commands` на все устройства из файла `devices.yaml` (для этого надо считать информацию из файла) с помощью функции `send_config_commands`.

Примеры команд с ошибками:

```
R1(config)#logging 0255.255.1
^
% Invalid input detected at '^' marker.

R1(config)#logging
% Incomplete command.

R1(config)#i
% Ambiguous command: "i"
```

В файле задания заготовлены команды с ошибками и без:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'i']
correct_commands = ['logging buffered 20010', 'ip http server']

commands = commands_with_errors+correct_commands
```

Задание 19.2c

Переделать функцию `send_config_commands` из задания 19.2b

Если при выполнении команды возникла ошибка, спросить пользователя надо ли выполнять остальные команды.

Варианты ответа [y]/n:

- у - выполнять остальные команды (значение по умолчанию)
- н - не выполнять остальные команды

Функция `send_config_commands` по-прежнему должна возвращать кортеж из двух словарей:

- первый словарь с выводом команд, которые выполнились без ошибки
- второй словарь с выводом команд, которые выполнились с ошибками

Оба словаря в формате

- ключ - команда
- значение - вывод с выполнением команд

Проверить функцию на командах с ошибкой.

Задание 19.2d

В этом задании надо создать функцию `send_cfg_to_devices`, которая выполняет команды на нескольких устройствах последовательно и при этом выполняет проверку на ошибки в командах.

Параметры функции:

- `devices_list` - список словарей с параметрами подключения к устройствам, которым надо передать команды
- `config_commands` - список команд, которые надо выполнить

Функция должна проверять результат на такие ошибки:

- Invalid input detected, Incomplete command, Ambiguous command

Если при выполнении какой-то из команд возникла ошибка, функция должна выводить сообщение на стандартный поток вывода с информацией о том, какая ошибка возникла, при выполнении какой команды и на каком устройстве.

После обнаружения ошибки, функция должна спросить пользователя надо ли выполнять эту команду на других устройствах.

Варианты ответа [y]/n:

- `y` - выполнять команду на оставшихся устройствах (значение по умолчанию)
- `n` - не выполнять команду на оставшихся устройствах

Функция `send_cfg_to_devices` должна возвращать кортеж из двух словарей:

- первый словарь с выводом команд, которые выполнились без ошибки
- второй словарь с выводом команд, которые выполнились с ошибками

Оба словаря в формате

- ключ - IP устройства
- значение - вложенный словарь:
 - ключ - команда
 - значение - вывод с выполнением команд

В файле задания заготовлены команды с ошибками и без:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'i']
correct_commands = ['logging buffered 20010', 'ip http server']

commands = commands_with_errors+correct_commands
```

Задание 19.3

Создать функцию `send_commands` (для подключения по SSH используется `netmiko`).

Параметры функции:

- `device` - словарь с параметрами подключения к устройству, которому надо передать команды
- `show` - одна команда `show` (строка)
- `filename` - имя файла, в котором находятся команды, которые надо выполнить (строка)
- `config` - список с командами, которые надо выполнить в конфигурационном режиме

В зависимости от того, какой аргумент был передан, функция вызывает разные функции внутри. При вызове функции, всегда будет передаваться только один из аргументов `show`, `config`, `filename`.

Далее комбинация из аргумента и соответствующей функции:

- `show` - функция `send_show_command` из задания 19.1
- `config` - функция `send_config_commands` из задания 19.2
- `filename` - функция `send_commands_from_file` (ее надо написать по аналогии с предыдущими)

Функция возвращает словарь с результатами выполнения команды:

- `ключ` - IP устройства
- `значение` - вывод с выполнением команд

Проверить работу функции на примере:

- устройств из файла `devices.yaml` (для этого надо считать информацию из файла)
- и различных комбинаций аргумента с командами:
 - списка команд `commands`
 - команды `command`
 - файла `config.txt`

```
commands = ['logging 10.255.255.1',
            'logging buffered 20010',
            'no logging console']
command = "sh ip int br"
```

Задание 19.4

Создать функцию `send_commands_to_devices` (для подключения по SSH используется `netmiko`).

Параметры функции:

- devices_list - список словарей с параметрами подключения к устройствам, которым надо передать команды
- show - одна команда show (строка)
- filename - имя файла, в котором находятся команды, которые надо выполнить (строка)
- config - список с командами, которые надо выполнить в конфигурационном режиме

В этой функции должен использоваться список словарей, в котором не указаны имя пользователя, пароль, и пароль на enable (файл devices2.yaml).

Функция должна запрашивать имя пользователя, пароль и пароль на enable при старте. Пароль не должен отображаться при наборе.

Функция send_commands_to_devices должна использовать функцию send_commands из задания 19.3.

Задание 19.4а

Дополнить функцию send_commands_to_devices таким образом, чтобы перед подключением к устройствам по SSH, выполнялась проверка доступности устройства pingом (можно вызвать команду ping в ОС).

Как выполнять команды ОС, описано в разделе [subprocess](#). Там же есть пример функции с отправкой ping.

Если устройство доступно, можно выполнять подключение. Если не доступно, вывести сообщение о том, что устройство с определенным IP-адресом недоступно и не выполнять подключение к этому устройству.

Для удобства можно сделать отдельную функцию для проверки доступности и затем использовать ее в функции send_commands_to_devices.

Одновременное подключение к нескольким устройствам

Когда нужно опросить много устройств, выполнение подключений поочередно будет достаточно долгим. Конечно, это будет быстрее, чем подключение вручную, но хотелось бы получать отклик как можно быстрее.

Все эти "долго" и "быстрее" относительные понятия, но в этом разделе мы научимся и конкретно измерять, сколько отрабатывал скрипт, чтобы сравнить, насколько быстрее будет выполняться подключение.

Для параллельного подключения к устройствам в этом разделе используется модуль `concurrent.futures`.

Также в разделе [Дополнительная информация](#) рассматриваются основы работы с модулями:

- `threading`
- `multiprocessing`

Измерение времени выполнения скрипта

Для оценки времени выполнения скрипта есть несколько вариантов. В курсе используются самые простые варианты:

- утилита Linux time
- и модуль Python datetime

Рассматриваются оба варианта, на тот случай, если используется Windows.

При оценке времени выполнения скрипта в данном случае не важна высокая точность. Главное - сравнить время выполнения скрипта в разных вариантах.

time

Утилита time в Linux позволяет замерить время выполнения скрипта. Например:

```
$ time python thread_paramiko.py
...
real    0m4.712s
user    0m0.336s
sys     0m0.064s
```

Нас интересует real время. В данном случае это 4.7 секунд.

Для использования утилиты time достаточно написать time перед строкой запуска скрипта.

datetime

Второй вариант - модуль datetime. Этот модуль позволяет работать со временем и датами в Python.

Пример использования:

```
from datetime import datetime
import time

start_time = datetime.now()

#Тут выполняются действия
time.sleep(5)

print(datetime.now() - start_time)
```

Результат выполнения:

```
$ python test.py
0:00:05.004949
```

Процессы и потоки в Python (CPython)

Для начала нам нужно разобраться с терминами:

- процесс (process) - это, грубо говоря, запущенная программа. Процессу выделяются отдельные ресурсы: память, процессорное время
- поток (thread) - это единица исполнения в процессе. Потоки разделяют ресурсы процесса, к которому они относятся.

Python (а точнее, CPython - реализация, которая используется в курсе) оптимизирован для работы в однопоточном режиме. Это хорошо, если в программе используется только один поток.

И, в то же время, у Python есть определенные нюансы работы в многопоточном режиме. Связаны они с тем, что CPython использует GIL (global interpreter lock).

GIL не дает нескольким потокам исполнять одновременно код Python. Если не вдаваться в подробности, то GIL можно представить как некий переходящий флаг, который разрешает потокам выполнятся. У кого флаг, тот может выполнять работу.

Флаг передается либо каждые сколько-то инструкций Python, либо, например, когда выполняются какие-то операции ввода-вывода.

Поэтому получается, что разные потоки не будут выполняться параллельно, а программа просто будет между ними переключаться, выполняя их в разное время.

Но не всё так плохо. Если в программе есть некое "ожидание": пакетов из сети, запроса пользователя, пауза типа sleep - то в такой программе потоки будут выполнятся как будто параллельно. А всё потому, что во время таких пауз флаг (GIL) можно передать другому потоку.

Но тут также нужно быть осторожным, так как такой результат может наблюдаться на небольшом количестве сессий, но может ухудшиться с ростом количества сессий.

Потоки отлично подходят для задач, которые связаны с операциями ввода-вывода. Подключение к оборудованию входит в число подобных задач.

В следующих разделах рассматривается, как использовать потоки для подключения по Telnet/SSH. И проверяется, какое суммарное время будет занимать исполнение скрипта, по сравнению с последовательным исполнением и с использованием процессов.

Процессы

Процессы позволяют выполнять задачи на разных ядрах компьютера. Это важно для задач, которые завязаны на операции ввода-вывода.

Для каждого процесса создается своя копия ресурсов, выделяется память, у каждого процесса свой GIL. Это же делает процессы более тяжеловесными, по сравнению с потоками.

Кроме того, количество процессов, которые запускаются параллельно, зависит от количества ядер и CPU и обычно исчисляется в десятках, тогда как количество потоков для операций ввода-вывода может исчисляться в сотнях.

Процессы и потоки можно совмещать, но это усложняет программу и на базовом уровне для операций ввода-вывода лучше остановиться на потоках.

Модуль concurrent.futures

Модуль concurrent.futures предоставляет высокоуровневый интерфейс для работы с процессами и потоками. При этом и для потоков, и для процессов используется одинаковый интерфейс, что позволяет легко переключаться между ними.

Если сравнивать этот модуль с threading или multiprocessing, то у него меньше возможностей. Но зато с concurrent.futures работать проще и интерфейс более понятный.

Модуль concurrent.futures позволяет легко решить задачу запуска нескольких потоков/процессов и получения из них данных.

Модуль предоставляет два класса:

- **ThreadPoolExecutor** - для работы с потоками
- **ProcessPoolExecutor** - для работы с процессами

Оба класса используют одинаковый интерфейс, поэтому достаточно разобраться с одним и затем просто переключиться на другой при необходимости.

Модуль использует понятие future. [Future](#) - это объект, который представляет отложенное вычисление. Этот объект можно запрашивать о состоянии (завершена работа или нет), можно получать результаты или исключения, которые возникли в процессе работы, по мере возникновения.

При этом нет необходимости создавать их вручную. Эти объекты создаются ThreadPoolExecutor и ProcessPoolExecutor.

Метод map

Метод map - это самый простой вариант работы с concurrent.futures.

Пример использования функции map с ThreadPoolExecutor (файл netmiko_threads_map_ver1.py):

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint

import yaml
from netmiko import ConnectHandler


def connect_ssh(device_dict, command='sh clock'):
    print('Connection to device: {}'.format(device_dict['ip']))
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices)
    return list(f_result)

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh, devices['routers'])
    pprint(all_done)
```

Это первый ограниченный пример, так как сейчас в функции connect_ssh команда задана вручную, а не передается как аргумент. В следующей версии это будет исправлено.

Сейчас нас интересует функция threads_conn:

```
def threads_conn(function, devices, limit=2):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices)
    return list(f_result)
```

- `with ThreadPoolExecutor(max_workers=limit) as executor:` - класс ThreadPoolExecutor инициируется в блоке with с указанием количества потоков

- `f_result = executor.map(function, devices)` - метод `map` похож на функцию `map`, но тут функция `function` вызывается в разных потоках. При этом в разных потоках функция будет вызываться с разными аргументами - элементами итерируемого объекта `devices`.
- метод `map` возвращает генератор. В этом генераторе содержатся результаты выполнения функций

Обратите внимание, что функция занимает всего 4 строки, и для получения данных не надо создавать очередь и передавать ее в функцию `connect_ssh`.

Результат выполнения:

```
$ python netmiko_threads_map_ver1.py
Connection to device: 192.168.100.1
Connection to device: 192.168.100.2
Connection to device: 192.168.100.3
[{'192.168.100.1': '*04:43:01.629 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*04:43:01.648 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*04:43:07.291 UTC Mon Aug 28 2017'}]
```

Важная особенность метода `map` - он возвращает результаты в том же порядке, в котором они указаны в итерируемом объекте.

Для демонстрации этой особенности в функции `connect_ssh` добавлены сообщения с выводом информации о том, когда функция начала работать и когда закончила. А также для маршрутизатора с IP 192.168.100.1 добавлен `sleep`, чтобы задержать выполнение функции (файл `netmiko_threads_map_ver2.py`):

```

from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time

import yaml
from netmiko import ConnectHandler

start_msg = '==> {} Connection to device: {}'
received_msg = '<== {} Received result from device: {}'

def connect_ssh(device_dict, command='sh clock'):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices)
    return list(f_result)

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh, devices['routers'])
    pprint(all_done)

```

Результат выполнения:

```

$ python netmiko_threads_map_ver2.py
==> 04:50:50.175076 Connection to device: 192.168.100.1
==> 04:50:50.175553 Connection to device: 192.168.100.2
<== 04:50:55.582707 Received result from device: 192.168.100.2
==> 04:50:55.689248 Connection to device: 192.168.100.3
<== 04:51:01.135640 Received result from device: 192.168.100.3
<== 04:51:05.568037 Received result from device: 192.168.100.1
[{'192.168.100.1': '*04:51:05.395 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*04:50:55.411 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*04:51:00.964 UTC Mon Aug 28 2017'}]

```

Обратите внимание на фактический порядок выполнения задач: 192.168.100.2, 192.168.100.3, 192.168.100.1. Но в итоговом списке все равно соблюдается порядок на основе списка devices['routers'].

Еще один момент, который тут хорошо заметен, это то, что как только одна задача выполнилась, сразу берется следующая. То есть, ограничение в два потока влияет на количество потоков, которые выполняются одновременно.

Теперь осталось изменить функцию таким образом, чтобы ей можно было передавать команду как аргумент.

Для этого мы воспользуемся функцией repeat из модуля itertools (файл netmiko_threads_map_final.py):

```

from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler


start_msg = '====> {} Connection to device: {}'
received_msg = '<==== {} Received result from device: {}'

def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2, command=''):
    with ThreadPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices, repeat(command))
    return list(f_result)

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                            devices['routers'],
                            command='sh clock')
    pprint(all_done)

```

Функция `repeat` тут нужна для того, чтобы команда передавалась при каждом вызове функции `connect_ssh`.

Результат выполнения:

```
$ python netmiko_threads_map_final.py
==> 05:01:08.314962 Connection to device: 192.168.100.1
==> 05:01:08.315114 Connection to device: 192.168.100.2
<== 05:01:13.693083 Received result from device: 192.168.100.2
==> 05:01:13.799002 Connection to device: 192.168.100.3
<== 05:01:19.363250 Received result from device: 192.168.100.3
<== 05:01:23.685859 Received result from device: 192.168.100.1
[{'192.168.100.1': '*05:01:23.513 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*05:01:13.522 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*05:01:19.189 UTC Mon Aug 28 2017'}]
```

Использование ProcessPoolExecutor с map

Для того чтобы предыдущий пример использовал процессы вместо потоков, достаточно сменить ThreadPoolExecutor на ProcessPoolExecutor ():

```

from concurrent.futures import ProcessPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat

import yaml
from netmiko import ConnectHandler


start_msg = '====> {} Connection to device: {}'
received_msg = '<==== {} Received result from device: {}'

def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2, command=''):
    with ProcessPoolExecutor(max_workers=limit) as executor:
        f_result = executor.map(function, devices, repeat(command))
    return list(f_result)

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                           devices['routers'],
                           command='sh clock')
    pprint(all_done)

```

Результат выполнения:

```
$ python netmiko_processes_map_final.py
==> 05:26:42.974505 Connection to device: 192.168.100.1
==> 05:26:42.975733 Connection to device: 192.168.100.2
<== 05:26:48.389420 Received result from device: 192.168.100.2
==> 05:26:48.495598 Connection to device: 192.168.100.3
<== 05:26:54.104585 Received result from device: 192.168.100.3
<== 05:26:58.367981 Received result from device: 192.168.100.1
[{'192.168.100.1': '*05:26:58.195 UTC Mon Aug 28 2017'},
 {'192.168.100.2': '*05:26:48.218 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*05:26:53.932 UTC Mon Aug 28 2017'}]
```

Метод submit и работа с futures

При использовании метода `map` объект `future` использовался внутри, но в итоге мы получали уже готовый результат функции.

В модуле `concurrent.futures` есть метод `submit`, который позволяет запускать `future`, и функция `as_completed`, которая ожидает как аргумент итерируемый объект с `futures` и возвращает `future` по мере завершения. В этом случае порядок не будет соблюдаться, как с `map`.

Теперь функция `threads_conn` выглядит немного по-другому (файл `netmiko_threads_submit.py`):

```

from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time

import yaml
from netmiko import ConnectHandler


start_msg = '====> {} Connection to device: {}'
received_msg = '<==== {} Received result from device: {}'

def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2, command=''):
    all_results = []
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            all_results.append(f.result())
    return all_results

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                           devices['routers'],
                           command='sh clock')
    pprint(all_done)

```

Остальной код не изменился, поэтому разобраться надо только с функцией `threads_conn`:

```
def threads_conn(function, devices, limit=2, command=''):
    all_results = []
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            all_results.append(f.result())
    return all_results
```

Теперь в блоке `with` два цикла:

- `future_ssh` - это список объектов `future`, который создается с помощью `list comprehensions`
- для создания `future` используется функция `submit`
 - ей как аргументы передаются: имя функции, которую надо выполнить, и ее аргументы
- следующий цикл проходится по списку `future` с помощью функции `as_completed`. Эта функция возвращает `future` только когда они завершили работу или были отменены. При этом `future` возвращаются по мере завершения работы

Результат выполнения:

```
$ python netmiko_threads_submit.py
==> 06:02:14.582011 Connection to device: 192.168.100.1
==> 06:02:14.582155 Connection to device: 192.168.100.2
<== 06:02:20.155865 Received result from device: 192.168.100.2
==> 06:02:20.262584 Connection to device: 192.168.100.3
<== 06:02:25.864270 Received result from device: 192.168.100.3
<== 06:02:29.962225 Received result from device: 192.168.100.1
[{'192.168.100.2': '*06:02:19.983 UTC Mon Aug 28 2017'},
 {'192.168.100.3': '*06:02:25.691 UTC Mon Aug 28 2017'},
 {'192.168.100.1': '*06:02:29.789 UTC Mon Aug 28 2017'}]
```

Обратите внимание, что порядок не сохраняется и зависит от того, какие функции раньше завершили работу.

Future

Чтобы посмотреть на `future`, в скрипт добавлены несколько строк с выводом информации (`netmiko_threads_submit_verbose.py`):

```

from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time

import yaml
from netmiko import ConnectHandler


start_msg = '==> {} Connection to device: {}'
received_msg = '<== {} Received result from device: {}'

def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2, command=''):
    all_results = {}
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = []
        for device in devices:
            future = executor.submit(function, device, command)
            future_ssh.append(future)
            print('Future: {} for device {}'.format(future, device['ip']))
        for f in as_completed(future_ssh):
            result = f.result()
            print('Future done {}'.format(f))
            all_results.update(result)
    return all_results

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                           devices['routers'],
                           command='sh clock')
    pprint(all_done)

```

Так как в прошлом варианте мы уже проверили, что результат возвращается в порядке выполнения, тут функция threads_conn возвращает словарь, а не список.

Результат выполнения:

```
$ python netmiko_threads_submit_verbose.py
==> 06:16:56.059256 Connection to device: 192.168.100.1
Future: <Future at 0xb68427cc state=running> for device 192.168.100.1
==> 06:16:56.059434 Connection to device: 192.168.100.2
Future: <Future at 0xb68483ac state=running> for device 192.168.100.2
Future: <Future at 0xb6848b4c state=pending> for device 192.168.100.3
<==> 06:17:01.482761 Received result from device: 192.168.100.2
<==> 06:17:01.589605 Connection to device: 192.168.100.3
Future done <Future at 0xb68483ac state=finished returned dict>
<==> 06:17:07.226815 Received result from device: 192.168.100.3
Future done <Future at 0xb6848b4c state=finished returned dict>
<==> 06:17:11.444831 Received result from device: 192.168.100.1
Future done <Future at 0xb68427cc state=finished returned dict>
{'192.168.100.1': '*06:17:11.273 UTC Mon Aug 28 2017',
 '192.168.100.2': '*06:17:01.310 UTC Mon Aug 28 2017',
 '192.168.100.3': '*06:17:07.055 UTC Mon Aug 28 2017'}
```

Так как по умолчанию используется ограничение в два потока, только два из трех future показывают статус running. Третий находится в состоянии pending и ждет, пока до него дойдет очередь.

Обработка исключений

Если при выполнении функции возникло исключение, оно будет сгенерировано при получении результата

Например, в файле devices.yaml пароль для устройства 192.168.100.2 изменен на неправильный:

```
$ 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 491, in establish_connection
    self.remote_conn_pre.connect(**ssh_connect_params)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/client.py",
  line 394, in connect
    look_for_keys, gss_auth, gss_kex, gss_deleg_creds, gss_host)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/client.py",
  line 649, in _auth
    raise saved_exception
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/client.py",
  line 636, in _auth
    self._transport.auth_password(username, password)
```

```
File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/transport.py", line 1329, in auth_password
    return self.auth_handler.wait_for_response(my_event)
File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/paramiko/auth_handler.py", line 217, in wait_for_response
    raise e
paramiko.ssh_exception.AuthenticationException: Authentication failed.
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "netmiko_threads_submit.py", line 40, in <module>
    command='sh clock')
  File "netmiko_threads_submit.py", line 32, in threads_conn
    all_results.append(f.result())
  File "/usr/local/lib/python3.6/concurrent/futures/_base.py", line 398, in result
    return self._get_result()
  File "/usr/local/lib/python3.6/concurrent/futures/_base.py", line 357, in _get_result
    raise self._exception
  File "/usr/local/lib/python3.6/concurrent/futures/thread.py", line 55, in run
    result = self.fn(*self.args, **self.kwargs)
  File "netmiko_threads_submit.py", line 19, in connect_ssh
    with ConnectHandler(**device_dict) as ssh:
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/netmiko/ssh_dispatcher.py", line 122, in ConnectHandler
    return ConnectionClass(*args, **kwargs)
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/netmiko/base_connection.py", line 145, in __init__
    self._establish_connection()
  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.
```

Так как исключение возникает при получении результата, легко добавить обработку исключений (файл netmiko_threads_submit_exception.py):

```

from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException

start_msg = '====> {} Connection to device: {}'
received_msg = '<==== {} Received result from device: {}'

def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def threads_conn(function, devices, limit=2, command=''):
    all_results = {}
    with ThreadPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                all_results.update(result)
    return all_results

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = threads_conn(connect_ssh,
                           devices['routers'],
                           command='sh clock')
    pprint(all_done)

```

Результат выполнения:

```
$ python netmiko_threads_submit_exception.py
==> 06:45:56.327892 Connection to device: 192.168.100.1
==> 06:45:56.328190 Connection to device: 192.168.100.2
==> 06:45:58.964806 Connection to device: 192.168.100.3
Authentication failure: unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
<== 06:46:04.325812 Received result from device: 192.168.100.3
<== 06:46:11.731541 Received result from device: 192.168.100.1
{'192.168.100.1': '*06:46:11.556 UTC Mon Aug 28 2017',
 '192.168.100.3': '*06:46:04.154 UTC Mon Aug 28 2017'}
```

Конечно, обработка исключения может выполняться и внутри функции `connect_ssh`, но это просто пример того, как можно работать с исключениями при использовании `future`.

ProcessPoolExecutor

Так как все работает аналогичным образом и для процессов, тут приведет последний вариант (файл `netmiko_processes_submit_exception.py`):

```

from concurrent.futures import ProcessPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException

start_msg = '====> {} Connection to device: {}'
received_msg = '<==== {} Received result from device: {}'

def connect_ssh(device_dict, command):
    print(start_msg.format(datetime.now().time(), device_dict['ip']))
    if device_dict['ip'] == '192.168.100.1':
        time.sleep(10)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print(received_msg.format(datetime.now().time(), device_dict['ip']))
    return {device_dict['ip']: result}

def processes_conn(function, devices, limit=2, command=''):
    all_results = {}
    with ProcessPoolExecutor(max_workers=limit) as executor:
        future_ssh = [executor.submit(function, device, command)
                      for device in devices]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                all_results.update(result)
    return all_results

if __name__ == '__main__':
    devices = yaml.load(open('devices.yaml'))
    all_done = processes_conn(connect_ssh,
                               devices['routers'],
                               command='sh clock')
    pprint(all_done)

```

Результат выполнения:

```
$ python netmiko_processes_submit_exception.py
==> 06:40:43.828249 Connection to device: 192.168.100.1
==> 06:40:43.828664 Connection to device: 192.168.100.2
Authentication failure: unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
==> 06:40:46.292613 Connection to device: 192.168.100.3
<== 06:40:51.890816 Received result from device: 192.168.100.3
<== 06:40:59.231330 Received result from device: 192.168.100.1
{'192.168.100.1': '*06:40:59.056 UTC Mon Aug 28 2017',
 '192.168.100.3': '*06:40:51.719 UTC Mon Aug 28 2017'}
```

Дополнительные материалы

GIL

- [Can't we get rid of the Global Interpreter Lock?](#)
- [GIL \(на русском\)](#)
- [Understanding the Python GIL](#)
- [Python threads and the GIL](#)

concurrent.futures

Документация Python:

- [concurrent.futures — Launching parallel tasks](#)
- [PEP 3148](#)
- [PyMOTW. concurrent.futures — Manage Pools of Concurrent Tasks](#)

Статьи:

- [A quick introduction to the concurrent.futures module](#)
- [Python - paralellizing CPU-bound tasks with concurrent.futures](#)
- [concurrent.futures in Python 3](#)

Полезные вопросы и ответы на stackoverflow

- [How many processes should I run in parallel?](#)
- [How many threads is too many?](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 20.1

Переделать задание 19.4а таким образом, чтобы проверка доступности устройств выполнялась не последовательно, а параллельно.

Для этого, можно взять за основу функцию `check_ip_addresses` из задания 11.3. Функцию надо переделать таким образом, чтобы проверка IP-адресов выполнялась параллельно в разных потоках.

Задание 20.1a

Переделать функцию из задания 20.1 таким образом, чтобы она позволяла контролировать количество параллельных проверок IP.

Для этого, необходимо добавить новый параметр `limit`, со значением по умолчанию - 2.

Функция должна проверять адреса из списка таким образом, чтобы в любой момент времени максимальное количество параллельных проверок было равным `limit`.

Задание 20.2

Создать функцию `send_commands_threads`, которая запускает функцию `send_commands` из задания 19.3 на разных устройствах в параллельных потоках.

Параметры функции `send_commands_threads` надо определить самостоятельно. Должна быть возможность передавать параметры `show`, `config`, `filename` функции `send_commands`.

Функция `send_commands_threads` возвращает словарь с результатами выполнения команд на устройствах:

- ключ - IP устройства
- значение - вывод с выполнением команд

Задание 20.2а

Переделать функцию `send_commands_threads` из задания 20.2 таким образом, чтобы с помощью аргумента `limit`, можно было указывать сколько подключений будут выполняться параллельно.

По умолчанию, значение аргумента должно быть 2.

Шаблоны конфигураций с Jinja

Jinja2 - это язык шаблонов, который используется в Python. Jinja - это не единственный язык шаблонов (шаблонизатор) для Python и, конечно же, не единственный язык шаблонов в целом.

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

Примеры использования:

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

Установить Jinja2 можно с помощью pip:

```
pip install jinja2
```

Далее термины Jinja и Jinja2 используются взаимозаменяюще.

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

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

Пример шаблона Jinja:

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

Комментарии к шаблону:

- В Jinja переменные записываются в двойных фигурных скобках.
- При выполнении скрипта эти переменные заменяются нужными значениями.

Этот шаблон может использоваться для генерации конфигурации разных устройств с помощью подстановки других наборов переменных.

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

```
from jinja2 import Template

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

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

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

Комментарии к файлу basic_generator.py:

- в первой строке из Jinja2 импортируется класс Template
- создается объект template, которому передается шаблон
 - в шаблоне используются переменные в синтаксисе Jinja
- в словаре liverpool ключи должны быть такими же, как имена переменных в шаблоне
 - значения, которые соответствуют ключам - это те данные, которые будут подставлены на место переменных
- последняя строка рендерит шаблон, используя словарь liverpool, то есть, подставляет значения в переменные.

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

```
$ python basic_generator.py

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

Пример использования **Jinja2**

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

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

Файл router_template.py

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

template_r1 = Template('''
hostname {{name}}
!
interface Loopback10
description MPLS loopback
ip address 10.10.{{id}}.1 255.255.255.255
!
interface GigabitEthernet0/0
description WAN to {{name}} sw1 G0/1
!
interface GigabitEthernet0/0.1{{id}}1
description MPLS to {{to_name}}
encapsulation dot1Q 1{{id}}1
ip address 10.{{id}}.1.2 255.255.255.252
ip ospf network point-to-point
ip ospf hello-interval 1
ip ospf cost 10
!
interface GigabitEthernet0/1
description LAN {{name}} to sw1 G0/2 !
interface GigabitEthernet0/1.{{IT}}
description PW IT {{name}} - {{to_name}}
encapsulation dot1Q {{IT}}
xconnect 10.10.{{to_id}}.1 {{id}}1 encapsulation mpls
backup peer 10.10.{{to_id}}.2 {{id}}21
    backup delay 1 1
!
interface GigabitEthernet0/1.{{BS}}
description PW BS {{name}} - {{ to_name}}
encapsulation dot1Q {{BS}}
xconnect 10.10.{{to_id}}.1 {{to_id}}{{id}}1 encapsulation mpls
    backup peer 10.10.{{to_id}}.2 {{to_id}}{{id}}21
    backup delay 1 1
!
router ospf 10
    router-id 10.10.{{id}}.1
    auto-cost reference-bandwidth 10000
    network 10.0.0.0 0.255.255.255 area 0
!
'''')
```

Файл routers_info.yml

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

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

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

Файл router_config_generator.py

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

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

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

Файл router_config_generator.py:

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

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

- название файла, в который записывается итоговая конфигурация, состоит из поля name в словаре и строки _r1.txt
 - например, Liverpool_r1.txt
- файл с таким именем открывается в режиме для записи
- в файл записывается результат рендеринга шаблона с использованием текущего словаря
- конструкция with сама закрывает файл

- управление возвращается в начало цикла (пока не переберутся все словари)

Запускаем файл router_config_generator.py:

```
$ python router_config_generator.py
```

В результате получатся три конфигурационных файла:

Liverpool_r1.txt

Bristol_r1.txt

Coventry_r1.txt

```
hostname Liverpool
!
interface Loopback10
  description MPLS loopback
  ip address 10.10.11.1 255.255.255.255
!
interface GigabitEthernet0/0
  description WAN to Liverpool sw1 G0/1
!
interface GigabitEthernet0/0.1111
  description MPLS to LONDON
  encapsulation dot1Q 1111
  ip address 10.11.1.2 255.255.255.252
  ip ospf network point-to-point
  ip ospf hello-interval 1
  ip ospf cost 10
!
interface GigabitEthernet0/1
  description LAN Liverpool to sw1 G0/2
!
interface GigabitEthernet0/1.791
  description PW IT Liverpool - LONDON
  encapsulation dot1Q 791
  xconnect 10.10.1.1 1111 encapsulation mpls
    backup peer 10.10.1.2 1121
    backup delay 1 1
!
interface GigabitEthernet0/1.1550
  description PW BS Liverpool - LONDON
  encapsulation dot1Q 1550
  xconnect 10.10.1.1 11111 encapsulation mpls
    backup peer 10.10.1.2 11121
    backup delay 1 1
!
router ospf 10
  router-id 10.10.11.1
  auto-cost reference-bandwidth 10000
  network 10.0.0.0 0.255.255.255 area 0
!
```

```
hostname Bristol
!
interface Loopback10
  description MPLS loopback
  ip address 10.10.12.1 255.255.255.255
!
interface GigabitEthernet0/0
  description WAN to Bristol sw1 G0/1
!
interface GigabitEthernet0/0.1121
  description MPLS to LONDON
  encapsulation dot1Q 1121
  ip address 10.12.1.2 255.255.255.252
  ip ospf network point-to-point
  ip ospf hello-interval 1
  ip ospf cost 10
!
interface GigabitEthernet0/1
  description LAN Bristol to sw1 G0/2
!
interface GigabitEthernet0/1.793
  description PW IT Bristol - LONDON
  encapsulation dot1Q 793
  xconnect 10.10.1.1 1211 encapsulation mpls
  backup peer 10.10.1.2 1221
  backup delay 1 1
!
interface GigabitEthernet0/1.1510
  description PW BS Bristol - LONDON
  encapsulation dot1Q 1510
  xconnect 10.10.1.1 11211 encapsulation mpls
  backup peer 10.10.1.2 11221
  backup delay 1 1
!
router ospf 10
  router-id 10.10.12.1
  auto-cost reference-bandwidth 10000
  network 10.0.0.0 0.255.255.255 area 0
!
```

```
hostname Coventry
!
interface Loopback10
  description MPLS loopback
  ip address 10.10.14.1 255.255.255.255
!
interface GigabitEthernet0/0
  description WAN to Coventry sw1 G0/1
!
interface GigabitEthernet0/0.1141
  description MPLS to Manchester
  encapsulation dot1Q 1141
  ip address 10.14.1.2 255.255.255.252
  ip ospf network point-to-point
  ip ospf hello-interval 1
  ip ospf cost 10
!
interface GigabitEthernet0/1
  description LAN Coventry to sw1 G0/2
!
interface GigabitEthernet0/1.892
  description PW IT Coventry - Manchester
  encapsulation dot1Q 892
  xconnect 10.10.2.1 1411 encapsulation mpls
  backup peer 10.10.2.2 1421
  backup delay 1 1
!
interface GigabitEthernet0/1.1650
  description PW BS Coventry - Manchester
  encapsulation dot1Q 1650
  xconnect 10.10.2.1 21411 encapsulation mpls
  backup peer 10.10.2.2 21421
  backup delay 1 1
!
router ospf 10
  router-id 10.10.14.1
  auto-cost reference-bandwidth 10000
  network 10.0.0.0 0.255.255.255 area 0
!
```

Пример использования Jinja с корректным использованием программного интерфейса

Для того, чтобы разобраться с Jinja2, лучше использовать предыдущие примеры. В этом разделе описывается корректное использование Jinja. В таком варианте данные, шаблон и скрипт, который генерирует итоговую информацию, разделены.

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

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

Шаблон templates/router_template.txt - это обычный текстовый файл:

```
hostname {{name}}
!
interface Loopback10
  description MPLS loopback
  ip address 10.10.{{id}}.1 255.255.255.255
!
interface GigabitEthernet0/0
  description WAN to {{name}} sw1 G0/1
!
interface GigabitEthernet0/0.1{{id}}1
  description MPLS to {{to_name}}
  encapsulation dot1Q 1{{id}}1
  ip address 10.{{id}}.1.2 255.255.255.252
  ip ospf network point-to-point
  ip ospf hello-interval 1
  ip ospf cost 10
!
interface GigabitEthernet0/1
  description LAN {{name}} to sw1 G0/2 !
interface GigabitEthernet0/1.{{IT}}
  description PW IT {{name}} - {{ to_name}}
  encapsulation dot1Q {{IT}}
  xconnect 10.10.{{to_id}}.1 {{id}}11 encapsulation mpls
  backup peer 10.10.{{to_id}}.2 {{id}}21
  backup delay 1 1
!
interface GigabitEthernet0/1.{{BS}}
  description PW BS {{name}} - {{ to_name}}
  encapsulation dot1Q {{BS}}
  xconnect 10.10.{{to_id}}.1 {{to_id}}{{id}}11 encapsulation mpls
  backup peer 10.10.{{to_id}}.2 {{to_id}}{{id}}21
  backup delay 1 1
!
router ospf 10
  router-id 10.10.{{id}}.1
  auto-cost reference-bandwidth 10000
  network 10.0.0.0 0.255.255.255 area 0
!
```

Файл с данными routers_info.yml

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

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

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

Скрипт для генерации конфигураций router_config_generator_ver2.py

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

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

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

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

Файл router_config_generator.py импортирует из модуля jinja2:

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

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

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

```
import os

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

Переменная `__file__` - это специальная переменная модуля, которая выставляется равной имени скрипта, который был запущен напрямую, и равна полному пути к модулю, когда он импортируется. [Подробнее о специальных переменных и методах](#).

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

Последняя часть осталась неизменной.

Синтаксис шаблонов Jinja2

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

В шаблонах Jinja2 можно использовать:

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

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

Мы разберемся с основами этих возможностей. Подробнее о шаблонах Jinja2 можно почитать в [документации](#).

Все файлы, которые используются как примеры в этом подразделе, находятся в каталоге 3_template_syntax/

Для генерации шаблонов будет использоваться скрипт cfg_gen.py

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

TEMPLATE_DIR, template = os.path.split(sys.argv[1])
VARS_FILE = sys.argv[2]

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

vars_dict = yaml.load(open(VARS_FILE))

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

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

- шаблон
- файл с переменными в формате YAML

Результат будет выведен на стандартный поток вывода.

Пример запуска скрипта:

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

Параметры `trim_blocks` и `lstrip_blocks` описаны в следующем подразделе.

Контроль символов whitespace

`trim_blocks, lstrip_blocks`

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

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

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

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

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

Вывод будет таким:

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

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

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

Из-за блока `{% for ibgp in bgp.ibgp_neighbors %}` появляются переводы строк. По умолчанию такое же поведение будет с любыми другими блоками Jinja.

При добавлении флага `trim_blocks` таким образом:

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

Результат выполнения будет таким:

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

Были удалены пустые строки после блока.

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

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

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

Если добавить аргумент `lstrip_blocks=True` таким образом:

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

Результат выполнения будет таким:

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

Отключение `lstrip_blocks` для блока

Иногда нужно отключить функциональность `lstrip_blocks` для блока.

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

```

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

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

```

Результат будет таким:

```

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

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

```

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

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

```

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

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

```

Он будет отключен и для конца блока:

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

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

Удаление whitespace в блоке

Аналогичным образом можно контролировать удаление whitespace для блока.

Для этого примера в окружении не выставлены флаги:

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

Шаблон templates/env_flags3.txt:

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

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

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

Результат будет таким:

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

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

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


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

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

Если добавить минут в конец блока:

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

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

Удаляется пустая строка и в конце блока:

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

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

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


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

Попробуйте добавить минус в конце выражений, описывающих блок, и посмотреть на результат:

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

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

Переменные

Переменные в шаблоне указываются в двойных фигурных скобках:

```
hostname {{ name }}

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

Значения переменных подставляются на основе словаря, который передается шаблону.

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

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

```
hostname {{ name }}

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

vlan {{ vlans[0] }}

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

И соответствующий файл data_files/vars.yml с переменными:

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

Обратите внимание на использование переменной `vlans` в шаблоне:

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

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

- `ospf.network` или `ospf['network']`

Результат запуска скрипта будет таким:

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

interface Loopback0
    ip address 10.0.0.3 255.255.255.255

vlan 10

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

Цикл for

Цикл for позволяет проходиться по элементам последовательности.

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

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

Пример шаблона templates/for.txt с использованием цикла:

```
hostname {{ name }}

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

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

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

Файл data_files/for.yml с переменными:

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

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

Элементы словаря не упорядочены. Поэтому, если нужно получить упорядоченные элементы, можно либо использовать отсортированный список кортежей, либо использовать упорядоченный словарь `collections.OrderedDict`.

Результат выполнения будет таким:

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

interface Loopback0
    ip address 10.0.0.3 255.255.255.255

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

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

if/elif/else

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

Конструкция if также должна находиться внутри `{% %}`. Нужно явно указывать окончание условия:

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

Пример шаблона templates/if.txt:

```
hostname {{ name }}

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

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

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

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

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

Конфигурация будет генерироваться с двумя вариантами данных.

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

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

Результат будет таким:

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

hostname R3

interface Loopback0
  ip address 10.0.0.3 255.255.255.255

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

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

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

Теперь результат выполнения будет таким:

```
hostname R3

interface Loopback0
    ip address 10.0.0.3 255.255.255.255

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

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

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

Пример шаблона templates/if_vlans.txt:

```
{% for intf, params in trunks.items() %}
interface {{ intf }}
    {% if params.action == 'add' %}
        switchport trunk allowed vlan add {{ params.vlans }}
    {% elif params.action == 'delete' %}
        switchport trunk allowed vlan remove {{ params.vlans }}
    {% else %}
        switchport trunk allowed vlan {{ params.vlans }}
    {% endif %}
{% endfor %}
```

Файл data_files/if_vlans.yml с данными:

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

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

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

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

В итоге будет сгенерирована такая конфигурация:

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

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

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

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

Файл с данными (data_files/if_for.yml):

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

Результат выполнения:

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

Фильтры

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

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

Jinja поддерживает большое количество встроенных фильтров. Мы рассмотрим лишь несколько из них. Остальные фильтры можно найти в [документации](#).

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

default

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

Пример шаблона `templates/filter_default.txt`:

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

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

Файл с данными (`data_files/filter_default.yml`):

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

Результат выполнения:

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

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

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

Например, если файл данных был бы таким:

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

То в итоге сгенерировался такой результат:

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

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

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

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

Результат уже будет таким (значение по умолчанию подставится):

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

dictsort

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

Синтаксис фильтра:

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

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

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

```
{% for intf, params in trunks | dictsort %}
interface {{ intf }}
  {% if params.action == 'add' %}
    switchport trunk allowed vlan add {{ params.vlans }}
  {% elif params.action == 'delete' %}
    switchport trunk allowed vlan remove {{ params.vlans }}
  {% else %}
    switchport trunk allowed vlan {{ params.vlans }}
  {% endif %}
{% endfor %}
```

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

Файл с данными (`data_files/filter_dictsort.yml`):

```

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

```

Результат выполнения будет таким (интерфейсы упорядочены):

```

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

```

join

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

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

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

```

{% for intf, params in trunks | dictsort %}
interface {{ intf }}
  {% if params.action == 'add' %}
    switchport trunk allowed vlan add {{ params.vlans | join(',') }}
  {% elif params.action == 'delete' %}
    switchport trunk allowed vlan remove {{ params.vlans | join(',') }}
  {% else %}
    switchport trunk allowed vlan {{ params.vlans | join(',') }}
  {% endif %}
{% endfor %}

```

Файл с данными (data_files/filter_join.yml):

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

Результат выполнения:

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

Тесты

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

Jinja поддерживает большое количество встроенных тестов. Мы рассмотрим лишь несколько из них. Остальные тесты Вы можете найти в [документации](#).

Тесты, как и фильтры, можно создавать самостоятельно.

defined

Тест `defined` позволяет проверить, есть ли переменная в словаре данных.

Пример шаблона `templates/test_defined.txt`:

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

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

Файл с данными (`data_files/test_defined.yml`):

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

Результат выполнения:

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

iterable

Тест iterable проверяет, является ли объект итератором.

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

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

```
{% for intf, params in trunks | dictsort %}
interface {{ intf }}
{% if params.vlans is iterable %}
  {% if params.action == 'add' %}
    switchport trunk allowed vlan add {{ params.vlans | join(',') }}
  {% elif params.action == 'delete' %}
    switchport trunk allowed vlan remove {{ params.vlans | join(',') }}
  {% else %}
    switchport trunk allowed vlan {{ params.vlans | join(',') }}
  {% endif %}
{% else %}
  {% if params.action == 'add' %}
    switchport trunk allowed vlan add {{ params.vlans }}
  {% elif params.action == 'delete' %}
    switchport trunk allowed vlan remove {{ params.vlans }}
  {% else %}
    switchport trunk allowed vlan {{ params.vlans }}
  {% endif %}
{% endif %}
{% endfor %}
```

Файл с данными (data_files/test_iterable.yml):

```

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

```

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

Результат выполнения:

```

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

```

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

set

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

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

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

```
% for intf, params in trunks | dictsort %
  {% set vlans = params.vlans %}
  {% set action = params.action %}

  interface {{ intf }}
    {% if vlans is iterable %}
      {% if action == 'add' %}
        switchport trunk allowed vlan add {{ vlans | join(',') }}
      {% elif action == 'delete' %}
        switchport trunk allowed vlan remove {{ vlans | join(',') }}
      {% else %}
        switchport trunk allowed vlan {{ vlans | join(',') }}
      {% endif %}
    {% else %}
      {% if action == 'add' %}
        switchport trunk allowed vlan add {{ vlans }}
      {% elif action == 'delete' %}
        switchport trunk allowed vlan remove {{ vlans }}
      {% else %}
        switchport trunk allowed vlan {{ vlans }}
      {% endif %}
    {% endif %}
  {% endfor %}
```

Обратите внимание на вторую и третью строки:

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

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

Файл с данными (data_files/set.yml):

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

Результат выполнения:

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

include

Выражение `include` позволяет добавить один шаблон в другой.

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

Шаблон `templates/vlans.txt`:

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

Шаблон `templates/ospf.txt`:

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

Шаблон `templates/bgp.txt`:

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

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

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

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

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

```

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

```

Результат выполнения скрипта:

```

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

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

```

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

Шаблон templates/router.txt:

```

{% include 'ospf.txt' %}

{% include 'bgp.txt' %}

logging {{ log_server }}

```

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

Файл с данными (data_files/router.yml):

```

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

bgp:
  local_as: 100
  loopback: lo100
  ibgp_neighbors:
    - 10.0.0.2
    - 10.0.0.3
  ebgp_neighbors:
    90.1.1.1: 500
    80.1.1.1: 600
  log_server: 10.1.1.1

```

Результат выполнения скрипта будет таким:

```

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

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

logging 10.1.1.1

```

Благодаря `include`, шаблон `templates/ospf.txt` используется и в шаблоне `templates/switch.txt`, и в шаблоне `templates/router.txt`, вместо того, чтобы повторять одно и то же дважды.

Наследование шаблонов

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

При использовании наследования различают:

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

Пример базового шаблона templates/base_router.txt:

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

Обратите внимание на четыре блока, которые созданы в шаблоне:

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

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

Дочерний шаблон templates/hq_router.txt:

```
% extends "base_router.txt" %

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

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

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

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

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

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

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

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

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

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

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

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

В данном случае конфигурация дополняется. Именно поэтому в дочернем шаблоне `templates/hq_router.txt` блок `ospf` начинается с выражения `{{ super() }}`:

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

`{{ super() }}` переносит в дочерний шаблон содержимое этого блока из родительского шаблона. За счет этого в дочерний шаблон перенесутся строки из родительского.

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

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

Подытожим правила работы с блоками. Если в родительском шаблоне создан блок:

- без содержимого - в дочернем шаблоне можно заполнить этот блок или игнорировать. Если блок заполнен, в нём будет только то, что было написано в дочернем шаблоне (пример - блок alias)
- с содержимым - то в дочернем шаблоне можно выполнить такие действия:
 - игнорировать блок - в таком случае в дочерний шаблон попадет содержимое, которое находилось в этом блоке в родительском шаблоне (пример - блок services)
 - переписать блок - тогда в дочернем шаблоне будет только то, что указано в нём
 - перенести содержимое блока из родительского шаблона и дополнить его - тогда в дочернем шаблоне будет и содержимое блока из родительского шаблона, и содержимое из дочернего шаблона. Для переноса содержимого из родительского шаблона используется выражение `{{ super() }}` (пример - блок ospf)

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

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

Результат выполнения будет таким:

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

    network 10.0.1.0 0.0.0.255 area 0
    network 10.0.2.0 0.0.0.255 area 2
    network 10.1.1.0 0.0.0.255 area 0
!
!
alias configure sh do sh
alias exec ospf sh run | s ^router ospf
alias exec bri show ip int bri | exc unass
alias exec id show int desc
alias exec top sh proc cpu sorted | excl 0.00%__0.00%__0.00%
alias exec c conf t
alias exec diff sh archive config differences nvram:startup-config system:running-configuration
alias exec desc sh int desc | ex down
!
line con 0
    logging synchronous
    history size 100
line vty 0 4
    logging synchronous
    history size 100
    transport input ssh
!
```

Обратите внимание, что в блоке ospf есть и команды из базового шаблона, и команды из дочернего шаблона.

Дополнительные материалы

Документация:

- [Общая документация Jinja2](#)
- [Синтаксис шаблонов](#)

Статьи:

- [Network Configuration Templates Using Jinja2. Matt Oswalt](#)
- [Python And Jinja2 Tutorial. Jeremy Schulman](#)
- [Configuration Generator with Python and Jinja2](#)
- [Custom filters for a Jinja2 based Config Generator](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 21.1

Переделать скрипт cfg_gen.py в функцию generate_cfg_from_template.

Функция ожидает два аргумента:

- путь к шаблону
- файл с переменными в формате YAML

Функция должна возвращать конфигурацию, которая была сгенерирована.

Проверить работу функции на шаблоне templates/for.txt и данных data_files/for.yml.

```
from jinja2 import Environment, FileSystemLoader
import yaml
import sys
import os

#$ python cfg_gen.py templates/for.txt data_files/for.yml
TEMPLATE_DIR, template = os.path.split(sys.argv[1])
VARS_FILE = sys.argv[2]

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

vars_dict = yaml.load(open(VARS_FILE))

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

Задание 21.1a

Дополнить функцию generate_cfg_from_template из задания 21.1:

Функция generate_cfg_from_template должна принимать любые аргументы, которые принимает класс Environment и просто передавать их ему.

То есть, надо добавить возможность контролировать аргументы trim_blocks, lstrip_blocks и любые другие аргументы Environment через функцию generate_cfg_from_template.

Проверить функциональность на аргументах:

- trim_blocks
- lstrip_blocks

Задание 21.1b

Дополнить функцию generate_cfg_from_template из задания 21.1 или 21.1a:

- добавить поддержку разных форматов для файла с данными

Должны поддерживаться такие форматы:

- YAML
- JSON
- словарь Python

Сделать для каждого формата свой параметр функции. Например:

- YAML - yaml_file
- JSON - json_file
- словарь Python - py_dict

Проверить работу функции на шаблоне templates/for.txt и данных:

- data_files/for.yml
- data_files/for.json
- словаре data_dict

```
data_dict = {'vlans': {  
    10: 'Marketing',  
    20: 'Voice',  
    30: 'Management'},  
'ospf': [{ 'network': '10.0.1.0 0.0.0.255', 'area': 0},  
         { 'network': '10.0.2.0 0.0.0.255', 'area': 2},  
         { 'network': '10.1.1.0 0.0.0.255', 'area': 0}],  
'id': 3,  
'name': 'R3'}
```

Задание 21.1с

Переделать функцию generate_cfg_from_template из задания 21.1, 21.1а или 21.1б:

- сделать автоматическое распознавание разных форматов для файла с данными
- для передачи разных типов данных, должен использоваться один и тот же параметр data

Должны поддерживаться такие форматы:

- YAML - файлы с расширением yml или yaml
- JSON - файлы с расширением json
- словарь Python

Если не получилось определить тип данных, вывести сообщение error_message (перенести текст сообщения в тело функции), завершить работу функции и вернуть None .

Проверить работу функции на шаблоне templates/for.txt и данных:

- data_files/for.yml
- data_files/for.json
- словаре data_dict

```
error_message = ''
Не получилось определить формат данных.
Поддерживаются файлы с расширением .json, .yml, .yaml и словари Python
'''

data_dict = {'vlans': {
    10: 'Marketing',
    20: 'Voice',
    30: 'Management'},
'ospf': [{ 'network': '10.0.1.0 0.0.0.255', 'area': 0},
          { 'network': '10.0.2.0 0.0.0.255', 'area': 2},
          { 'network': '10.1.1.0 0.0.0.255', 'area': 0}],
'id': 3,
'name': 'R3'}
```

Задание 21.2

На основе конфигурации config_r1.txt, создать шаблоны:

- templates/cisco_base.txt - в нём должны быть все строки, кроме настройки alias и event manager
 - имя хоста должно быть переменной hostname
- templates/alias.txt - в этот шаблон перенести все alias
- templates/eem_int_desc.txt - в этом шаблоне должен быть event manager applet

В шаблонах templates/alias.txt и templates/eem_int_desc.txt переменных нет.

Создать шаблон templates/cisco_router_base.txt.

В шаблон должно быть включено содержимое шаблонов:

- templates/cisco_base.txt
- templates/alias.txt
- templates/eem_int_desc.txt

При этом, нельзя копировать текст шаблонов.

Проверьте шаблон templates/cisco_router_base.txt, с помощью функции generate_cfg_from_template из задания 21.1-21.1c. Не копируйте код функции.

В качестве данных, используйте файл data_files/router_info.yml

Задание 21.3

Создайте шаблон templates/ospf.txt на основе конфигурации OSPF в файле cisco_ospf.txt. Пример конфигурации дан, чтобы напомнить синтаксис.

Какие значения должны быть переменными:

- номер процесса. Имя переменной - `process`
- router-id. Имя переменной - `router_id`
- reference-bandwidth. Имя переменной - `ref_bw`
- интерфейсы, на которых нужно включить OSPF. Имя переменной - `ospf_intf`
 - на месте этой переменной ожидается список словарей с такими ключами:
 - `name` - имя интерфейса, вида Fa0/1, VLan10, Gi0/0
 - `ip` - IP-адрес интерфейса, вида 10.0.1.1
 - `area` - номер зоны
 - `passive` - является ли интерфейс пассивным. Допустимые значения: True или False

Для всех интерфейсов в списке `ospf_intf`, надо сгенерировать строки:

```
network x.x.x.x 0.0.0.0 area x
```

Если интерфейс пассивный, для него должна быть добавлена строка:

```
passive-interface x
```

Для интерфейсов, которые не являются пассивными, в режиме конфигурации интерфейса, надо добавить строку:

```
ip ospf hello-interval 1
```

Все команды должны быть в соответствующих режимах.

Проверьте получившийся шаблон `templates/ospf.txt`, на данных в файле `data_files/ospf.yml`, с помощью функции `generate_cfg_from_template` из задания 21.1-21.1с. Не копируйте код функции.

Задание 21.3а

Измените шаблон `templates/ospf.txt` таким образом, чтобы для перечисленных переменных были указаны значения по умолчанию, которые используются в том случае, если переменная не задана.

Не использовать для этого выражения `if/else`.

Задать в шаблоне значения по умолчанию для таких переменных:

- `process` - значение по умолчанию 1

- `ref_bw` - значение по умолчанию 10000

Проверьте получившийся шаблон `templates/ospf.txt`, на данных в файле `data_files/ospf2.yml`, с помощью функции `generate_cfg_from_template` из задания 21.1-21.1с. Не копируйте код функции.

Задание 21.3b

Измените шаблон `templates/ospf.txt` из задания 21.3а таким образом, чтобы для перечисленных переменных были указаны значения по умолчанию, которые используются в том случае, если переменная не задана или, если в переменной пустое значение.

Не использовать для этого выражения `if/else`.

Задать в шаблоне значения по умолчанию для таких переменных:

- `process` - значение по умолчанию 1
- `ref_bw` - значение по умолчанию 10000

Проверьте получившийся шаблон `templates/ospf.txt`, на данных в файле `data_files/ospf3.yml`, с помощью функции `generate_cfg_from_template` из задания 21.1-21.1с. Не копируйте код функции.

Задание 21.4

Создайте шаблон `templates/add_vlan_to_switch.txt`, который будет использоваться при необходимости добавить VLAN на коммутатор.

В шаблоне должны поддерживаться возможности:

- добавления VLAN и имени VLAN
- добавления VLAN как access, на указанном интерфейсе
- добавления VLAN в список разрешенных, на указанные транки

Если VLAN необходимо добавить как access, то надо настроить и режим интерфейса и добавить его в VLAN:

```
interface Gi0/1
switchport mode access
switchport access vlan 5
```

Для транков, необходимо только добавить VLAN в список разрешенных:

```
interface Gi0/10
switchport trunk allowed vlan add 5
```

Имена переменных надо выбрать на основании примера данных, в файле `data_files/add_vlan_to_switch.yaml`.

Проверьте шаблон `templates/add_vlan_to_switch.txt` на данных в файле `data_files/add_vlan_to_switch.yaml`, с помощью функции `generate_cfg_from_template` из задания 21.1-21.1с. Не копируйте код функции.

Обработка вывода команд с TextFSM

На оборудовании, которое не поддерживает какого-то программного интерфейса, вывод команд `show` возвращается в виде строки. И, хотя отчасти она структурирована, но всё же это просто строка. И её надо как-то обработать, чтобы получить объекты Python, например, словарь или список.

Например, можно построчно обрабатывать вывод команды и, используя, например, регулярные выражения, получить объекты Python. Но есть более удобный вариант, чем просто обрабатывать каждый вывод построчно: TextFSM.

TextFSM - это библиотека, созданная Google для обработки вывода с сетевых устройств. Она позволяет создавать шаблоны, по которым будет обрабатываться вывод команды.

Использование TextFSM лучше, чем простая построчная обработка, так как шаблоны дают лучшее представление о том, как вывод будет обрабатываться, и шаблонами проще поделиться. А значит, проще найти уже созданные шаблоны и использовать их, или поделиться своими.

Для начала библиотеку надо установить:

```
pip install textfsm
```

Для использования TextFSM надо создать шаблон, по которому будет обрабатываться вывод команды.

Пример вывода команды `traceroute`:

```
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
 1 10.0.12.1 1 msec 0 msec 0 msec
 2 15.0.0.5 0 msec 5 msec 4 msec
 3 57.0.0.7 4 msec 1 msec 4 msec
 4 79.0.0.9 4 msec * 1 msec
```

Например, из вывода надо получить хопы, через которые прошел пакет.

В таком случае шаблон TextFSM будет выглядеть так (файл `traceroute.template`):

```

Value ID (\d+)
Value Hop (\d+(\.\d+)\{3\})

Start
  ^ ${ID} ${Hop} -> Record

```

Первые две строки определяют переменные:

- `Value ID (\d+)` - эта строка определяет переменную ID, которая описывает регулярное выражение: `(\d+)` - одна или более цифр
 - сюда попадут номера хопов
- `Value Hop (\d+(\.\d+)\{3\})` - эта строка определяет переменную Hop, которая описывает IP-адрес таким регулярным выражением: `(\d+(\.\d+)\{3\})`

После строки `Start` начинается сам шаблон. В данном случае он очень простой:

- `^ ${ID} ${Hop} -> Record`
 - сначала идет символ начала строки, затем два пробела и переменные ID и Hop
 - в TextFSM переменные описываются таким образом: `${имя переменной}`
 - слово `Record` в конце означает, что строки, которые попадут под описанный шаблон, будут обработаны и выведены в результаты TextFSM (с этим подробнее мы разберемся в [следующем разделе](#))

Скрипт для обработки вывода команды `traceroute` с помощью TextFSM (`parse_traceroute.py`):

```

import textfsm

traceroute = """
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
  1 10.0.12.1 1 msec 0 msec 0 msec
  2 15.0.0.5 0 msec 5 msec 4 msec
  3 57.0.0.7 4 msec 1 msec 4 msec
  4 79.0.0.9 4 msec * 1 msec
"""

template = open('traceroute.template')
fsm = textfsm.TextFSM(template)
result = fsm.ParseText(traceroute)

print(fsm.header)
print(result)

```

Результат выполнения скрипта:

```
$ python parse_traceroute.py
['ID', 'Hop']
[['1', '10.0.12.1'], ['2', '15.0.0.5'], ['3', '57.0.0.7'], ['4', '79.0.0.9']]
```

Строки, которые совпали с описанным шаблоном, возвращаются в виде списка списков. Каждый элемент - это список, который состоит из двух элементов: номера хопа и IP-адреса.

Разберемся с содержимым скрипта:

- traceroute - это переменная, которая содержит вывод команды traceroute
- template = open('traceroute.template') - содержимое файла с шаблоном TextFSM считывается в переменную template
- fsm = textfsm.TextFSM(template) - класс, который обрабатывает шаблон и создает из него объект в TextFSM
- result = fsm.ParseText(traceroute) - метод, который обрабатывает переданный вывод согласно шаблону и возвращает список списков, в котором каждый элемент - это обработанная строка
- В конце выводится заголовок: print(fsm.header), который содержит имена переменных и результат обработки

В этом выводом можно работать дальше. Например, периодически выполнять команду traceroute и сравнивать, изменилось ли количество хопов и их порядок.

Для работы с TextFSM нужны вывод команды и шаблон:

- для разных команд нужны разные шаблоны
- TextFSM возвращает результат обработки в табличном виде (в виде списка списков)
 - этот вывод легко преобразовать в csv формат или в список словарей

Синтаксис шаблонов TextFSM

В этом разделе описан синтаксис шаблонов на основе документации TextFSM. В следующем разделе показаны примеры использования синтаксиса. Поэтому, в принципе, можно перейти сразу к следующему разделу, а к этому возвращаться по необходимости, для тех ситуаций, для которых нет примера, и когда нужно перечитать, что означает какой-то параметр.

Шаблон TextFSM описывает, каким образом данные должны обрабатываться.

Любой шаблон состоит из двух частей:

- определения переменных
 - эти переменные описывают, какие столбцы будут в табличном представлении
- определения состояний

Пример разбора команды traceroute:

```
# Определение переменных:
Value ID (\d+)
Value Hop (\d+(\.\.\d+)\{3\})

# Секция с определением состояний всегда должна начинаться с состояния Start
Start
#   Переменные      действие
^ ${ID} ${Hop} -> Record
```

Определение переменных

В секции с переменными должны идти только определения переменных. Единственное исключение - в этом разделе могут быть комментарии.

В этом разделе не должно быть пустых строк. Для TextFSM пустая строка означает завершение секции определения переменных.

Формат описания переменных:

```
Value [option[,option...]] name regex
```

Синтаксис описания переменных (для каждой опции ниже мы рассмотрим примеры):

- `Value` - это ключевое слово, которое указывает, что создается переменная. Его обязательно нужно указывать

- option - опции, которые определяют, как работать с переменной. Если нужно указать несколько опций, они должны быть отделены запятой, без пробелов.
Поддерживаются такие опции:
 - **Filldown** - значение, которое ранее совпало с регулярным выражением, запоминается до следующей обработки строки (если не было явно очищено или снова совпало регулярное выражение).
 - это значит, что последнее значение столбца, которое совпало с регулярным выражением, запоминается и используется в следующих строках, если в них не присутствовал этот столбец.
 - **Key** - определяет, что это поле содержит уникальный идентификатор строки
 - **Required** - строка, которая обрабатывается, будет записана только в том случае, если эта переменная присутствует.
 - **List** - значение - это список, и каждое совпадение с регулярным выражением будет добавлять в список элемент. По умолчанию каждое следующее совпадение перезаписывает предыдущее.
 - **Fillup** - работает как Filldown, но заполняет пустые значение выше до тех пор, пока не найдет совпадение. Не совместимо с Required.
- `name` - имя переменной, которое будет использоваться как имя колонки. Зарезервированные имена не должны использоваться как имя переменной.
- `regex` - регулярное выражение, которое описывает переменную. Регулярное выражение должно быть в скобках.

Определение состояний

После определения переменных нужно описать состояния:

- каждое определение состояния должно быть отделено пустой строкой (как минимум, одной)
- первая строка - имя состояния
- затем идут строки, которые описывают правила
 - правила должны начинаться с пробела и символа `^`

Начальное состояние всегда **Start**. Входные данные сравниваются с текущим состоянием, но в строке правила может быть указано, что нужно перейти к другому состоянию.

Проверка выполняется построчно, пока не будет достигнут **EOF**(конец файла), или текущее состояние перейдет в состояние **End**.

Зарезервированные состояния

Зарезервированы такие состояния:

- **Start** - это состояние обязательно должно быть указано. Без него шаблон не будет работать.
- **End** - это состояние завершает обработку входящих строк и не выполняет состояние **EOF**.
- **EOF** - это неявное состояние, которое выполняется всегда, когда обработка дошла до конца файла. Выглядит оно таким образом:

```
EOF
^.* -> Record
```

EOF записывает текущую строку, прежде чем обработка завершается. Если это поведение нужно изменить, надо явно в конце шаблона написать EOF:

```
EOF
```

Правила состояний

Каждое состояние состоит из одного или более правил:

- TextFSM обрабатывает входящие строки и сравнивает их с правилами
- если правило (регулярное выражение) совпадает со строкой, выполняются действия, которые описаны в правиле, и для следующей строки процесс повторяется заново, с начала состояния.

Правила должны быть описаны в таком формате:

```
^regex [-> action]
```

В правиле:

- каждое правило должно начинаться с пробела и символа `^`
 - символ `^` означает начало строки и всегда должен указываться явно
- `regex` - это регулярное выражение, в котором могут использоваться переменные
 - для указания переменной, \ может использоваться синтаксис `$ValueName` или `${ValueName}` (этот формат предпочтителен)
 - в правиле на место переменных подставляются регулярные выражения, которые они описывают
 - если нужно явно указать символ конца строки, используется значение `$$`

Действия в правилах

После регулярного выражения в правиле могут указываться действия:

- между регулярным выражением и действием должен быть символ `->`
- действия могут состоять из трех частей в таком формате: **L.R S**
 - **L - Line Action** - действия, которые применяются к входящей строке
 - **R - Record Action** - действия, которые применяются к собранным значениям
 - **S - State Transition** - переход в другое состояние
- по умолчанию используется **Next.NoRecord**

Line Actions

Line Actions:

- **Next** - обработать строку, прочитать следующую и начать проверять её с начала состояния. Это действие используется по умолчанию, если не указано другое
- **Continue** - продолжить обработку правил, как будто совпадения не было, при этом значения присваиваются

Record Action

Record Action - опциональное действие, которое может быть указано после Line Action. Они должны быть разделены точкой. Типы действий:

- **NoRecord** - не выполнять ничего. Это действие по умолчанию, когда другое не указано
- **Record** - запомнить значения, которые совпали с правилом. Все переменные, кроме тех, где указана опция `Filldown`, обнуляются.
- **Clear** - обнулить все переменные, кроме тех, где указана опция `Filldown`.
- **Clearall** - обнулить все переменные.

Разделять действия точкой нужно только в том случае, если нужно указать и Line, и Record действия. Если нужно указать только одно из них, точку ставить не нужно.

State Transition

После действия может быть указано новое состояние:

- состояние должно быть одним из зарезервированных или определенных в шаблоне
- если входная строка совпала:
 - все действия выполняются,
 - считывается следующая строка,

- затем текущее состояние меняется на новое, и обработка продолжается в новом состоянии.

Если в правиле используется действие , то в нём нельзя использовать переход в другое состояние. Это правило нужно для того, чтобы в последовательности состояний не было петель.

Error Action

Специальное действие **Error** останавливает всю обработку строк, отбрасывает все строки, которые были собраны до сих пор, и возвращает исключение.

Синтаксис этого действия такой:

```
^regex -> Error [word|"string"]
```

Примеры использования TextFSM

В этом разделе рассматриваются примеры шаблонов и использования TextFSM.

Для обработки вывода команд по шаблону в разделе используется скрипт `parse_output.py`. Он не привязан к конкретному шаблону и выводу: шаблон и вывод команды будут передаваться как аргументы:

```
import sys
import textfsm
from tabulate import tabulate

template = sys.argv[1]
output_file = sys.argv[2]

f = open(template)
output = open(output_file).read()

re_table = textfsm.TextFSM(f)

header = re_table.header
result = re_table.ParseText(output)

print(tabulate(result, headers=header))
```

Пример запуска скрипта:

```
$ python parse_output.py template command_output
```

Модуль `tabulate` используется для отображения данных в табличном виде (его нужно установить, если хотите использовать этот скрипт). Аналогичный вывод можно было сделать и с помощью форматирования строк, но с `tabulate` это сделать проще.

Обработка данных по шаблону всегда выполняется одинаково. Поэтому скрипт будет одинаковый, только шаблон и данные будут отличаться.

Начиная с простого примера, разберемся с тем, как использовать TextFSM.

show clock

Первый пример - разбор вывода команды `sh clock` (файл `output/sh_clock.txt`):

```
15:10:44.867 UTC Sun Nov 13 2016
```

Для начала в шаблоне надо определить переменные:

- в начале каждой строки должно быть ключевое слово Value
 - каждая переменная определяет столбец в таблице
- следующее слово - название переменной
- после названия, в скобках - регулярное выражение, которое описывает значение переменной

Определение переменных выглядит так:

```
Value Time (.:....)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)
```

Подсказка по спецсимволам:

- . - любой символ
- + - одно или более повторений предыдущего символа
- \S - все символы, кроме whitespace
- \w - любая буква или цифра
- \d - любая цифра

После определения переменных должна идти пустая строка и состояние **Start**, а после, начиная с пробела и символа ^, идет правило (файл templates/sh_clock.template):

```
Value Time (.:....)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)

Start
^${Time}.* ${Timezone} ${WeekDay} ${Month} ${MonthDay} ${Year} -> Record
```

Так как в данном случае в выводе всего одна строка, можно не писать в шаблоне действие Record. Но лучше его использовать в ситуациях, когда надо записать значения, чтобы привыкать к этому синтаксису и не ошибиться, когда нужна обработка нескольких строк.

Когда TextFSM обрабатывает строки вывода, он подставляет вместо переменных их значения. В итоге правило будет выглядеть так:

```
^(...:....).*(\S+)(\w+)(\w+)(\d+)(\d+)
```

Когда это регулярное выражение применяется в выводу show clock, в каждой группе регулярного выражения будет находиться соответствующее значение:

- 1 группа: 15:10:44
- 2 группа: UTC
- 3 группа: Sun
- 4 группа: Nov
- 5 группа: 13
- 6 группа: 2016

В правиле, кроме явного действия Record, которое указывает, что запись надо поместить в финальную таблицу, по умолчанию также используется правило Next. Оно указывает, что надо перейти к следующей строке текста. Так как в выводе команды sh clock только одна строка, обработка завершается.

Результат отработки скрипта будет таким:

```
$ python parse_output.py templates/sh_clock.template output/sh_clock.txt
Time      Timezone     WeekDay     Month      MonthDay     Year
-----  -----  -----  -----  -----  -----
15:10:44    UTC        Sun       Nov        13      2016
```

show cdp neighbors detail

Теперь попробуем обработать вывод команды show cdp neighbors detail.

Особенность этой команды в том, что данные находятся не в одной строке, а в разных.

В файле output/sh_cdp_n_det.txt находится вывод команды show cdp neighbors detail:

```
SW1#show cdp neighbors detail
-----
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
```

```
Platform: cisco WS-C2960-8TC-L, Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16, Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE S
OFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
    IP address: 10.1.1.2

-----
Device ID: R1
Entry address(es):
    IP address: 10.1.1.1
Platform: Cisco 3825, Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/22, Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1, RELEA
SE 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, RELEAS
E 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):
```

Из вывода команды надо получить такие поля:

- LOCAL_HOST - имя устройства из приглашения
- DEST_HOST - имя соседа
- MGMNT_IP - IP-адрес соседа
- PLATFORM - модель соседнего устройства
- LOCAL_PORT - локальный интерфейс, который соединен с соседом
- REMOTE_PORT - порт соседнего устройства
- IOS_VERSION - версия IOS соседа

Шаблон выглядит таким образом (файл templates/sh_cdp_n_det.template):

```
Value LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
^${LOCAL_HOST}[>#].
^Device ID: ${DEST_HOST}
^.*IP address: ${MGMNT_IP}
^Platform: ${PLATFORM},
^Interface: ${LOCAL_PORT}, Port ID \(outgoing port\): ${REMOTE_PORT}
^.*Version ${IOS_VERSION},
```

Результат выполнения скрипта:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST      DEST_HOST      MGMNT_IP      PLATFORM      LOCAL_PORT      REMOTE_PORT
IOS_VERSION
-----
-----
SW1            R2            10.2.2.2      Cisco 2911    GigabitEthernet1/0/21  GigabitEther
net0/0  15.2(2)T1
```

Несмотря на то, что правила с переменными описаны в разных строках, и, соответственно, работают с разными строками, TextFSM собирает их в одну строку таблицы. То есть, переменные, которые определены в начале шаблона, задают строку

итоговой таблицы.

Обратите внимание, что в файле sh_cdp_n_det.txt находится вывод с тремя соседями, а в таблице только один сосед, последний.

Record

Так получилось из-за того, что в шаблоне не указано действие **Record**. И в итоге в финальной таблице осталась только последняя строка.

Исправленный шаблон:

```
Value LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
^${LOCAL_HOST}[>#].
^Device ID: ${DEST_HOST}
^.*IP address: ${MGMNT_IP}
^Platform: ${PLATFORM},
^Interface: ${LOCAL_PORT}, Port ID \(outgoing port\): ${REMOTE_PORT}
^.*Version ${IOS_VERSION}, -> Record
```

Теперь результат запуска скрипта выглядит так:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST      DEST_HOST      MGMNT_IP      PLATFORM      LOCAL_PORT      RE
MOTE_PORT      IOS_VERSION
-----  -----  -----  -----  -----  -----
-----  -----
SW1           SW2           10.1.1.2      cisco WS-C2960-8TC-L GigabitEthernet1/0/16 Gi
gabitEthernet0/1 12.2(55)SE9
                    R1           10.1.1.1      Cisco 3825      GigabitEthernet1/0/22 Gi
gabitEthernet0/0 12.4(24)T1
                    R2           10.2.2.2      Cisco 2911      GigabitEthernet1/0/21 Gi
gabitEthernet0/0 15.2(2)T1
```

Вывод получен со всех трёх устройств. Но переменная LOCAL_HOST отображается не в каждой строке, а только в первой.

Filldown

Это связано с тем, что приглашение, из которого взято значение переменной, появляется только один раз. И для того, чтобы оно появлялось и в последующих строках, надо использовать действие **Filldown** для переменной LOCAL_HOST:

```
Value Filldown LOCAL_HOST (\S+)
Value DEST_HOST (\S+)
Value MGMT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
^${LOCAL_HOST}[>#].
^Device ID: ${DEST_HOST}
^.*IP address: ${MGMT_IP}
^Platform: ${PLATFORM},
^Interface: ${LOCAL_PORT}, Port ID \(outgoing port\): ${REMOTE_PORT}
^.*Version ${IOS_VERSION}, -> Record
```

Теперь мы получили такой вывод:

```
$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST      DEST_HOST      MGMT_IP      PLATFORM          LOCAL_PORT      RE
MOTE_PORT      IOS_VERSION
-----  -----
-----  -----
SW1           SW2           10.1.1.2      cisco WS-C2960-8TC-L GigabitEthernet1/0/16 Gi
gabitEthernet0/1 12.2(55)SE9
SW1           R1            10.1.1.1      Cisco 3825        GigabitEthernet1/0/22 Gi
gabitEthernet0/0 12.4(24)T1
SW1           R2            10.2.2.2      Cisco 2911        GigabitEthernet1/0/21 Gi
gabitEthernet0/0 15.2(2)T1
SW1
```

Теперь значение переменной LOCAL_HOST появилось во всех трёх строках. Но появился ещё один странный эффект - последняя строка, в которой заполнена только колонка LOCAL_HOST.

Required

Дело в том, что все переменные, которые мы определили, опциональны. К тому же, одна переменная с параметром Filldown. И, чтобы избавиться от последней строки, нужно сделать хотя бы одну переменную обязательной с помощью параметра **Required**:

```

Value Filldown LOCAL_HOST (\S+)
Value Required DEST_HOST (\S+)
Value MGMNT_IP (.*)
Value PLATFORM (.*)
Value LOCAL_PORT (.*)
Value REMOTE_PORT (.*)
Value IOS_VERSION (\S+)

Start
^${LOCAL_HOST}[>#].
^Device ID: ${DEST_HOST}
^.*IP address: ${MGMNT_IP}
^Platform: ${PLATFORM},
^Interface: ${LOCAL_PORT}, Port ID \(outgoing port\): ${REMOTE_PORT}
^.*Version ${IOS_VERSION}, -> Record

```

Теперь мы получим корректный вывод:

```

$ python parse_output.py templates/sh_cdp_n_det.template output/sh_cdp_n_det.txt
LOCAL_HOST      DEST_HOST      MGMNT_IP      PLATFORM          LOCAL_PORT      RE
MOTE_PORT      IOS_VERSION
-----
-----      -----      -----      -----      -----
SW1            SW2           10.1.1.2    cisco WS-C2960-8TC-L GigabitEthernet1/0/16 Gi
gabitEthernet0/1 12.2(55)SE9
SW1            R1            10.1.1.1    Cisco 3825        GigabitEthernet1/0/22 Gi
gabitEthernet0/0 12.4(24)T1
SW1            R2            10.2.2.2    Cisco 2911        GigabitEthernet1/0/21 Gi
gabitEthernet0/0 15.2(2)T1

```

show ip interface brief

В случае, когда нужно обработать данные, которые выведены столбцами, шаблон TextFSM наиболее удобен.

Шаблон для вывода команды show ip interface brief (файл templates/sh_ip_int_br.template):

```

Value INTF (\S+)
Value ADDR (\S+)
Value STATUS (up|down|administratively down)
Value PROTO (up|down)

Start
^${INTF}\s+\${ADDR}\s+\w+\s+\w+\s+\${STATUS}\s+\${PROTO} -> Record

```

В этом случае правило можно описать одной строкой.

Вывод команды (файл output/sh_ip_int_br.txt):

```
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
```

Результат выполнения будет таким:

```
$ python parse_output.py templates/sh_ip_int_br.template output/sh_ip_int_br.txt
INT          ADDR      STATUS     PROTO
-----
FastEthernet0/0 15.0.15.1   up        up
FastEthernet0/1 10.0.12.1   up        up
FastEthernet0/2 10.0.13.1   up        up
FastEthernet0/3 unassigned  up        up
Loopback0       10.1.1.1   up        up
Loopback100     100.0.0.1  up        up
```

show ip route ospf

Рассмотрим случай, когда нам нужно обработать вывод команды show ip route ospf, и в таблице маршрутизации есть несколько маршрутов к одной сети.

Для маршрутов к одной и той же сети вместо нескольких строк, где будет повторяться сеть, будет создана одна запись, в которой все доступные next-hop адреса собраны в список.

Пример вывода команды show ip route ospf (файл output/sh_ip_route_ospf.txt):

```
R1#sh ip route ospf
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
      D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area
      N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2
      E1 - OSPF external type 1, E2 - OSPF external type 2
      i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2
      ia - IS-IS inter area, * - candidate default, U - per-user static route
      o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP
      + - replicated route, % - next hop override

Gateway of last resort is not set

      10.0.0.0/8 is variably subnetted, 10 subnets, 2 masks
0        10.0.24.0/24 [110/20] via 10.0.12.2, 1w2d, Ethernet0/1
0        10.0.34.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
0        10.2.2.2/32 [110/11] via 10.0.12.2, 1w2d, Ethernet0/1
0        10.3.3.3/32 [110/11] via 10.0.13.3, 1w2d, Ethernet0/2
0        10.4.4.4/32 [110/21] via 10.0.13.3, 1w2d, Ethernet0/2
                  [110/21] via 10.0.12.2, 1w2d, Ethernet0/1
                  [110/21] via 10.0.14.4, 1w2d, Ethernet0/3
0        10.5.35.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
```

Для этого примера упрощаем задачу и считаем, что маршруты могут быть только OSPF и с обозначением только О (то есть, только внутризональные маршруты).

Первая версия шаблона выглядит так:

```
Value Network ([[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}))
Value Mask (\^\d{1,2})
Value Distance (\d+)
Value Metric (\d+)
Value NextHop ([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})

Start
^O +${Network}${Mask}\s\[ ${Distance}\}/${Metric}\]\svia\s${NextHop}, -> Record
```

Результат получился такой:

Network	Mask	Distance	Metric	NextHop
10.0.24.0	/24	110	20	10.0.12.2
10.0.34.0	/24	110	20	10.0.13.3
10.2.2.2	/32	110	11	10.0.12.2
10.3.3.3	/32	110	11	10.0.13.3
10.4.4.4	/32	110	21	10.0.13.3
10.5.35.0	/24	110	20	10.0.13.3

Всё нормально, но потерялись варианты путей для маршрута 10.4.4.4/32. Это логично, ведь нет правила, которое подошло бы для такой строки.

List

Воспользуемся опцией **List** для переменной NextHop:

```
Value Network (([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}))
Value Mask (\^\d{1,2})
Value Distance (\d+)
Value Metric (\d+)
Value List NextHop ([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})

Start
^0 +${Network}${Mask}\s\[$Distance\]\/${Metric}\]\svia\s${NextHop}, -> Record
```

Теперь вывод получился таким:

Network	Mask	Distance	Metric	NextHop
10.0.24.0	/24	110	20	['10.0.12.2']
10.0.34.0	/24	110	20	['10.0.13.3']
10.2.2.2	/32	110	11	['10.0.12.2']
10.3.3.3	/32	110	11	['10.0.13.3']
10.4.4.4	/32	110	21	['10.0.13.3']
10.5.35.0	/24	110	20	['10.0.13.3']

Изменилось то, что в столбце NextHop отображается список, но пока с одним элементом.

Так как перед записью маршрута, для которого есть несколько путей, надо добавить к нему все доступные адреса NextHop, надо перенести действие **Record**.

Для этого, запись переносится на момент, когда встречается следующая строка с маршрутом. В этот момент надо записать предыдущую строку, и только после этого уже записывать текущую. Для этого используется такая запись:

```
^0 -> Continue.Record
```

В ней действие **Record** говорит, что надо записать текущее значение переменных. А, так как в этом правиле нет переменных, записывается то, что было в предыдущих значениях.

Действие **Continue** говорит, что надо продолжить работать с текущей строкой так, как будто совпадения не было. За счет этого сработает следующая строка.

Остается добавить правило, которое будет описывать дополнительные маршруты к сети (в них нет сети и маски):

```
^\s+\[${Distance}\}/${Metric}\]\svia\s${NextHop},
```

Итоговый шаблон выглядит так (файл `templates/sh_ip_route_ospf.template`):

```
Value Network (([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}))
Value Mask (\d{1,2})
Value Distance (\d+)
Value Metric (\d+)
Value List NextHop ([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})

Start
^0 -> Continue.Record
^0 +${Network}${Mask}\s\[${Distance}\}/${Metric}\]\svia\s${NextHop},
^\s+\[${Distance}\}/${Metric}\]\svia\s${NextHop},
```

Этот пример сложнее предыдущих, чтобы его лучше понять, попробуйте постепенно перейти с прошлого варианта шаблона к последнему.

В результате мы получим такой вывод:

Network	Mask	Distance	Metric	NextHop
10.0.24.0	/24	110	20	['10.0.12.2']
10.0.34.0	/24	110	20	['10.0.13.3']
10.2.2.2	/32	110	11	['10.0.12.2']
10.3.3.3	/32	110	11	['10.0.13.3']
10.4.4.4	/32	110	21	['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0	/24	110	20	['10.0.13.3']

show etherchannel summary

TextFSM удобно использовать для разбора вывода, который отображается столбцами, или для обработки вывода, который находится в разных строках. Менее удобными получаются шаблоны, когда надо получить несколько однотипных элементов из одной строки.

Пример вывода команды `show etherchannel summary` (файл `output/sh_etherchannel_summary.txt`):

```

sw1# sh etherchannel summary
Flags: D - down      P - bundled in port-channel
      I - stand-alone S - suspended
      H - Hot-standby (LACP only)
      R - Layer3       S - Layer2
      U - in use       f - failed to allocate aggregator

      M - not in use, minimum links not met
      u - unsuitable for bundling
      w - waiting to be aggregated
      d - default port

Number of channel-groups in use: 2
Number of aggregators: 2

Group Port-channel Protocol Ports
-----+-----+-----+
1     Po1(SU)        LACP    Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3     Po3(SU)        -       Fa0/11(P)  Fa0/12(P)  Fa0/13(P)  Fa0/14(P)

```

В данном случае нужно получить:

- имя и номер port-channel. Например, Po1
- список всех портов в нём. Например, ['Fa0/1', 'Fa0/2', 'Fa0/3']

Сложность тут в том, что порты находятся в одной строке, а в TextFSM нельзя указывать одну и ту же переменную несколько раз в строке. Но есть возможность несколько раз искать совпадение в строке.

Первая версия шаблона выглядит так:

```

Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
^\$CHANNEL\(\S+[\w-]+\w+\$MEMBERS\)\-> Record

```

В шаблоне две переменные:

- CHANNEL - имя и номер агрегированного порта
- MEMBERS - список портов, которые входят в агрегированный порт. Для этой переменной указан тип - List

Результат:

```

CHANNEL      MEMBERS
-----  -----
Po1          ['Fa0/1']
Po3          ['Fa0/11']

```

Пока что в выводе только первый порт, а нужно, чтобы попали все порты. В данном случае надо продолжить обработку строки с портами после найденного совпадения. То есть, использовать действие Continue и описать следующее выражение.

Единственная строка, которая есть в шаблоне, описывает первый порт. Надо добавить строку, которая описывает следующий порт.

Следующая версия шаблона:

```

Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
^\\d+ +${CHANNEL}\\\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\\\(-> Continue
^\\d+ +${CHANNEL}\\\(\S+ +[\w-]+ +[\w ]+ +\\S+ +${MEMBERS}\\\(-> Record

```

Вторая строка описывает такое же выражение, но переменная MEMBERS смещается на следующий порт.

Результат:

```

CHANNEL      MEMBERS
-----  -----
Po1          ['Fa0/1', 'Fa0/2']
Po3          ['Fa0/11', 'Fa0/12']

```

Аналогично надо дописать в шаблон строки, которые описывают третий и четвертый порт. Но, так как в выводе может быть переменное количество портов, надо перенести правило Record на отдельную строку, чтобы оно не было привязано к конкретному количеству портов в строке.

Если Record будет находиться, например, после строки, в которой описаны четыре порта, для ситуации, когда портов в строке меньше, запись не будет выполняться.

Итоговый шаблон (файл templates/sh_etherchannel_summary.txt):

```

Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
^d+.* -> Continue.Record
^d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\$+ +${MEMBERS}\)\( -> Continue
^d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\$+ +){2} +${MEMBERS}\)\( -> Continue
^d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\$+ +){3} +${MEMBERS}\)\( -> Continue

```

Результат обработки:

CHANNEL	MEMBERS
Po1	['Fa0/1', 'Fa0/2', 'Fa0/3']
Po3	['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14']

Теперь все порты попали в вывод.

Шаблон предполагает, что в одной строке будет максимум четыре порта. Если портов может быть больше, надо добавить соответствующие строки в шаблон.

Возможен ещё один вариант вывода команды sh etherchannel summary (файл output/sh_etherchannel_summary2.txt):

```

sw1# sh etherchannel summary
Flags: D - down      P - bundled in port-channel
      I - stand-alone s - suspended
      H - Hot-standby (LACP only)
      R - Layer3       S - Layer2
      U - in use       f - failed to allocate aggregator

      M - not in use, minimum links not met
      u - unsuitable for bundling
      w - waiting to be aggregated
      d - default port

Number of channel-groups in use: 2
Number of aggregators: 2

Group Port-channel Protocol Ports
-----+-----+-----+
1    Po1(SU)      LACP    Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3    Po3(SU)      -       Fa0/11(P)  Fa0/12(P)  Fa0/13(P)  Fa0/14(P)
                           Fa0/15(P)  Fa0/16(P)

```

В таком выводе появляется новый вариант - строки, в которых находятся только порты.

Для того, чтобы шаблон обрабатывал и этот вариант, надо его модифицировать (файл templates/sh_etherchannel_summary2.txt):

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
^\d+.* -> Continue.Record
^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\)\( -> Continue
^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\$S+ +${MEMBERS}\)\( -> Continue
^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\$S+ +){2} +${MEMBERS}\)\( -> Continue
^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\$S+ +){3} +${MEMBERS}\)\( -> Continue
^ +${MEMBERS} -> Continue
^ +\$S+ +${MEMBERS} -> Continue
^ +(\$S+ +){2} +${MEMBERS} -> Continue
^ +(\$S+ +){3} +${MEMBERS} -> Continue
```

Результат будет таким:

CHANNEL	MEMBERS
-----	-----
Ro1	['Fa0/1', 'Fa0/2', 'Fa0/3']
Ro3	['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14', 'Fa0/15', 'Fa0/16']

На этом мы заканчиваем разбираться с шаблонами TextFSM.

Примеры шаблонов для Cisco и другого оборудования можно посмотреть в проекте [ntc-ansible](#).

TextFSM CLI Table

Благодаря TextFSM можно обрабатывать вывод команд и получать структурированный результат. Однако, всё ещё надо вручную прописывать, каким шаблоном обрабатывать команды `show`, каждый раз, когда используется TextFSM.

Было бы намного удобней иметь какое-то соответствие между командой и шаблоном, чтобы можно было написать общий скрипт, который выполняет подключения к устройствам, отправляет команды, сам выбирает шаблон и парсит вывод в соответствии с шаблоном.

В TextFSM есть такая возможность.

Для того, чтобы это можно было воспользоваться, надо создать файл, в котором описаны соответствия между командами и шаблонами. В TextFSM он называется `index`.

Этот файл должен находиться в каталоге с шаблонами и должен иметь такой формат:

- первая строка - названия колонок
- каждая следующая строка - это соответствие шаблона команде
- обязательные колонки, местоположение которых фиксировано (должны быть обязательно первой и последней, соответственно):
 - первая колонка - имена шаблонов
 - последняя колонка - соответствующая команда
 - в этой колонке используется специальный формат, чтобы описать то, что команда может быть написана не полностью
- остальные колонки могут быть любыми
 - например, в примере ниже будут колонки `Hostname`, `Vendor`. Они позволяют уточнить информацию об устройстве, чтобы определить, какой шаблон использовать.
 - например, команда `show version` может быть у оборудования Cisco и HP. Соответственно, только команды недостаточно, чтобы определить, какой шаблон использовать. В таком случае можно передать информацию о том, какой тип оборудования используется, вместе с командой, и тогда получится определить правильный шаблон.
- во всех столбцах, кроме первого, поддерживаются регулярные выражения
 - в командах внутри `[[[]]]` регулярные выражения не поддерживаются

Пример файла `index`:

```
Template, Hostname, Vendor, Command
sh_cdp_n_det.template, .*, Cisco, sh[[ow]] cdp ne[[ighbors]] de[[tail]]
sh_clock.template, .*, Cisco, sh[[ow]] clo[[ck]]
sh_ip_int_br.template, .*, Cisco, sh[[ow]] ip int[[erface]] br[[ief]]
sh_ip_route_ospf.template, .*, Cisco, sh[[ow]] ip rou[[te]] o[[spf]]
```

Обратите внимание на то, как записаны команды:

- `sh[[ow]] ip int[[erface]] br[[ief]]`
 - эта запись будет преобразована в выражение `sh((ow)?)? ip int((erface)?)? br((ief)?)?`
 - это значит, что TextFSM сможет определить, какой шаблон использовать, даже если команда набрана не полностью
 - например, такие варианты команды сработают:
 - `sh ip int br`
 - `show ip inter bri`

Как использовать CLI table

Посмотрим, как пользоваться классом `clitable` и файлом `index`.

В каталоге `templates` такие шаблоны и файл `index`:

```
sh_cdp_n_det.template
sh_clock.template
sh_ip_int_br.template
sh_ip_route_ospf.template
index
```

Сначала попробуем поработать с `CLI Table` в `ipython`, чтобы посмотреть, какие возможности есть у этого класса, а затем посмотрим на финальный скрипт.

Для начала импортируем класс `clitable`:

```
In [1]: import clitable
```

Проверять работу `clitable` будем на последнем примере из прошлого раздела - выводе команды `show ip route ospf`. Считываем вывод, который хранится в файле `output/sh_ip_route_ospf.txt`, в строку:

```
In [2]: output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()
```

Сначала надо инициализировать класс, передав ему имя файла, в котором хранится соответствие между шаблонами и командами, и указать имя каталога, в котором хранятся шаблоны:

```
In [3]: cli_table = clitable.CliTable('index', 'templates')
```

Надо указать, какая команда передается, и указать дополнительные атрибуты, которые помогут идентифицировать шаблон. Для этого нужно создать словарь, в котором ключи - имена столбцов, которые определены в файле index. В данном случае не обязательно указывать название вендора, так как команде sh ip route ospf соответствует только один шаблон.

```
In [4]: attributes = {'Command': 'show ip route ospf', 'Vendor': 'Cisco'}
```

Методу ParseCmd надо передать вывод команды и словарь с параметрами:

```
In [5]: cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
```

В результате в объекте cli_table получаем обработанный вывод команды sh ip route ospf.

Методы cli_table (чтобы посмотреть все методы, надо вызвать dir(cli_table)):

```
In [6]: cli_table.  
cli_table.AddColumn      cli_table.NewRow      cli_table.index      cli_t  
able.size  
cli_table.AddKeys      cli_table.ParseCmd    cli_table.index_file  cli_t  
able.sort  
cli_table.Append      cli_table.ReadIndex   cli_table.next       cli_t  
able.superkey  
cli_table.CsvToTable  cli_table.Remove     cli_table.raw        cli_t  
able.synchronised  
cli_table.FormattedTable cli_table.Reset    cli_table.row         cli_t  
able.table  
cli_table.INDEX       cli_table.RowWith   cli_table.row_class   cli_t  
able.template_dir  
cli_table.KeyValue     cli_table.extend    cli_table.row_index  cli_t  
cli_table.LabelValueTable cli_table.header  cli_table.separator
```

Например, если вызвать `print cli_table`, получим такой вывод:

```
In [7]: print(cli_table)
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']
```

Метод `FormattedTable` позволяет получить вывод в виде таблицы:

```
In [8]: print(cli_table.FormattedTable())
Network      Mask   Distance  Metric  NextHop
=====
10.0.24.0  /24    110        20      10.0.12.2
10.0.34.0  /24    110        20      10.0.13.3
10.2.2.2   /32    110        11      10.0.12.2
10.3.3.3   /32    110        11      10.0.13.3
10.4.4.4   /32    110        21      10.0.13.3, 10.0.12.2, 10.0.14.4
10.5.35.0  /24    110        20      10.0.13.3
```

Такой вывод может пригодиться для отображения информации.

Чтобы получить из объекта `cli_table` структурированный вывод, например, список списков, надо обратиться к объекту таким образом:

```
In [9]: data_rows = [list(row) for row in cli_table]

In [11]: data_rows
Out[11]:
[['10.0.24.0', '/24', '110', '20', ['10.0.12.2']],
 ['10.0.34.0', '/24', '110', '20', ['10.0.13.3']],
 ['10.2.2.2', '/32', '110', '11', ['10.0.12.2']],
 ['10.3.3.3', '/32', '110', '11', ['10.0.13.3']],
 ['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']],
 ['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]
```

Отдельно можно получить названия столбцов:

```
In [12]: header = list(cli_table.header)

In [14]: header
Out[14]: ['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
```

Теперь вывод аналогичен тому, который был получен в прошлом разделе.

Соберем всё в один скрипт (файл `textfsm_clitable.py`):

```
import clitable

output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()

cli_table = clitable.CliTable('index', 'templates')

attributes = {'Command': 'show ip route ospf' , 'Vendor': 'Cisco'}

cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
print('CLI Table output:\n', cli_table)

print('Formatted Table:\n', cli_table.FormattedTable())

data_rows = [list(row) for row in cli_table]
header = list(cli_table.header)

print(header)
for row in data_rows:
    print(row)
```

В упражнениях к этому разделу будет задание, в котором надо объединить описанную процедуру в функцию, а также вариант с получением списка словарей.

Выход будет таким:

```
$ python textfsm_clitable.py
CLI Table output:
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']

Formatted Table:
Network      Mask     Distance   Metric    NextHop
=====
10.0.24.0   /24     110        20       10.0.12.2
10.0.34.0   /24     110        20       10.0.13.3
10.2.2.2    /32     110        11       10.0.12.2
10.3.3.3    /32     110        11       10.0.13.3
10.4.4.4    /32     110        21       10.0.13.3, 10.0.12.2, 10.0.14.4
10.5.35.0   /24     110        20       10.0.13.3

['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
['10.0.24.0', '/24', '110', '20', ['10.0.12.2']]
['10.0.34.0', '/24', '110', '20', ['10.0.13.3']]
['10.2.2.2', '/32', '110', '11', ['10.0.12.2']]
['10.3.3.3', '/32', '110', '11', ['10.0.13.3']]
['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']]
['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]
```

Теперь с помощью TextFSM можно не только получать структурированный вывод, но и автоматически определять, какой шаблон использовать, по команде и optionalным аргументам.

Дополнительные материалы

Документация:

- [TextFSM](#)

Статьи:

- [Programmatic Access to CLI Devices with TextFSM. Jason Edelman \(26.02.2015\)](#) - основы TextFSM и идеи о развитии, которые легли в основу модуля ntc-ansible
- [Parse CLI outputs with TextFSM. Henry Ölsner \(24.08.2015\)](#) - пример использования TextFSM для разбора большого файла с выводом sh inventory. Подробнее объясняется синтаксис TextFSM
- [Creating Templates for TextFSM and ntc_show_command. Jason Edelman \(27.08.2015\)](#) - подробнее рассматривается синтаксис TextFSM, и показаны примеры использования модуля ntc-ansible (обратите внимание, что синтаксис модуля уже немного изменился)
- [TextFSM and Structured Data. Kirk Byers \(22.10.2015\)](#) - вводная статья о TextFSM. Тут не описывается синтаксис, но дается общее представление о том, что такое TextFSM, и пример его использования

Проекты, которые используют TextFSM:

- [Модуль ntc-ansible](#)

Шаблоны TextFSM (из модуля ntc-ansible):

- [ntc-templates](#)

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 22.1

Переделать пример, который использовался в разделе TextFSM, в функцию.

Функция должна называться `parse_output`. Параметры функции:

- `template` - шаблон TextFSM (это должно быть имя файла, в котором находится шаблон)
- `output` - вывод соответствующей команды `show` (строка)

Функция должна возвращать список:

- первый элемент - это список с названиями столбцов (в примере ниже, находится в переменной `header`)
- остальные элементы это списки, в которых находятся результаты обработки вывода (в примере ниже, находится в переменной `result`)

Проверить работу функции на каком-то из примеров раздела.

Пример из раздела:

```

import sys
import textfsm
from tabulate import tabulate

template = sys.argv[1]
output_file = sys.argv[2]

with open(template) as f, open(output_file) as output:
    re_table = textfsm.TextFSM(f)
    header = re_table.header
    result = re_table.ParseText(output.read())
    print(result)
    print(tabulate(result, headers=header))

```

Задание 22.1а

Переделать функцию `parse_output` из задания 22.1 таким образом, чтобы, вместо списка списков, она возвращала один список словарей:

- ключи - названия столбцов,
- значения, соответствующие значения в столбцах.

То есть, для каждой строки будет один словарь в списке.

Задание 22.2

В этом задании нужно использовать функцию `parse_output` из задания 22.1. Она используется для того, чтобы получить структурированный вывод в результате обработки вывода команды.

Полученный вывод нужно записать в CSV формате.

Для записи вывода в CSV, нужно создать функцию `list_to_csv`, которая ожидает как аргументы:

- список:
 - первый элемент - это список с названиями заголовков
 - остальные элементы это списки, в котором находятся результаты обработки вывода
- имя файла, в который нужно записать данные в CSV формате

Проверить работу функции на примере обработки команды `sh ip int br` (шаблон и вывод есть в разделе).

Задание 22.3

Сделать шаблон TextFSM для обработки вывода sh ip dhcp snooping binding. Вывод команды находится в файле output/sh_ip_dhcp_snooping.txt.

Шаблон должен обрабатывать и возвращать значения таких столбцов:

- MacAddress
- IpAddress
- VLAN
- Interface

Проверить работу шаблона с помощью функции из задания 22.1.

Задание 22.4

На основе примера из раздела [clitable](#) сделать функцию parse_command_dynamic.

Параметры функции:

- словарь атрибутов, в котором находятся такие пары ключ: значение:
 - 'Command': команда
 - 'Vendor': вендор (обратите внимание, что файл index отличается от примера, который использовался в разделе, поэтому Вам нужно подставить тут правильное значение)
- имя файла, где хранится соответствие между командами и шаблонами (строка)
 - значение по умолчанию аргумента - index
- каталог, где хранятся шаблоны и файл с соответствиями (строка)
 - значение по умолчанию аргумента - templates
- вывод команды (строка)

Функция должна возвращать список словарей с результатами обработки вывода команды (как в задании 22.1а):

- ключи - названия столбцов
- значения - соответствующие значения в столбцах

Проверить работу функции на примере вывода команды sh ip int br.

Пример из раздела:

```

import clitable

output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()

cli_table = clitable.CliTable('index', 'templates')
attributes = {'Command': 'show ip route ospf', 'Vendor': 'Cisco'}

cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)

print('CLI Table output:\n', cli_table)
print('Formatted Table:\n', cli_table.FormattedTable())

data_rows = [list(row) for row in cli_table]
header = list(cli_table.header)

print(header)
for row in data_rows:
    print(row)

```

Задание 22.4а

Переделать функцию из задания 22.4:

- добавить аргумент `show_output`, который контролирует будет ли выводиться результат обработки команды на стандартный поток вывода
 - по умолчанию `False`, что значит результат не будет выводиться
- результат должен отображаться с помощью `FormattedTable` (пример есть в разделе)

Задание 22.5

В этом задании соединяется функциональность `TextFSM` и подключение к оборудованию.

Задача такая:

- подключиться к оборудованию
- выполнить команду `show`
- полученный вывод передавать на обработку `TextFSM`
- вернуть результат обработки

Для этого, воспользуемся функциями, которые были созданы ранее:

- функцией `send_show_command` из упражнения 19.1
- функцией `parse_command_dynamic` из упражнения 22.4

В этом упражнении нужно создать функцию `send_and_parse_command`:

- функция должна использовать внутри себя функции `parse_command_dynamic` и `send_show_command`.
- какие аргументы должны быть у функции `send_and_parse_command`, нужно решить самостоятельно
 - но, надо иметь в виду, какие аргументы ожидают две готовые функции, которые мы используем
- функция `send_and_parse_command` должна возвращать:
 - функция `send_show_command` возвращает словарь с результатами выполнения команды:
 - ключ - IP устройства
 - значение - результат выполнения команды
 - затем, нужно отправить полученный вывод на обработку функции `parse_command_dynamic`
 - в результате, должен получиться словарь, в котором:
 - ключ - IP устройства
 - значение - список словарей (то есть, тот вывод, который был получен из функции `parse_command_dynamic`)

Для функции `send_show_command` создан файл `devices.yaml`, в котором находятся параметры подключения к устройствам.

Проверить работу функции `send_and_parse_command` на команде `sh ip int br`.

Задание 22.6

Это задание похоже на задание 22.5, но в этом задании подключения надо выполнять параллельно с помощью потоков. Для параллельного подключения использовать модуль `concurrent.futures`.

В этом упражнении нужно создать функцию `send_and_parse_command_parallel`:

- она должна использовать внутри себя функцию `send_and_parse_command`
- какие аргументы должны быть у функции `send_and_parse_command_parallel`, нужно решить самостоятельно
- функция `send_and_parse_command_parallel` должна возвращать словарь, в котором:
 - ключ - IP устройства
 - значение - список словарей

Проверить работу функции `send_and_parse_command_parallel` на команде `sh ip int br`.

```
import yaml

test_command = "sh ip int br"
devices = yaml.load(open('devices.yaml'))
```

Ansible

Ansible - это система управления конфигурациями. Ansible позволяет автоматизировать и упростить настройку, обслуживание и развертывание серверов, служб, ПО и др.

На данный момент существует несколько [систем управления конфигурациями](#).

Однако для работы с сетевым оборудованием чаще всего используется Ansible. Связано это с тем, что Ansible не требует установки агента на управляемые хосты. Особенно актуально это для устройств, которые позволяют работать с ними только через CLI.

Кроме того, Ansible активно развивается в сторону поддержки сетевого оборудования, и в нём постоянно появляются новые возможности и модули для работы с сетевым оборудованием.

Некоторое сетевое оборудование поддерживает другие системы управления конфигурациями (позволяет установить агента).

Одно из важных преимуществ Ansible заключается в том, что он очень прост в использовании, и с ним легко начать работать.

Примеры задач, которые поможет решить Ansible:

- подключение по SSH к устройствам
 - параллельное подключение к устройствам по SSH (можно указывать, сколько устройствам подключаться одновременно)
- отправка команд на устройства
- удобный синтаксис описания устройств:
 - можно разбивать устройства на группы и затем отправлять какие-то команды на всю группу
- поддержка шаблонов конфигураций с Jinja2

Это всего лишь несколько возможностей Ansible, которые относятся к сетевому оборудованию. Они перечислены для того, чтобы показать, что эти задачи Ansible сразу снимает, и можно не использовать для этого какие-то скрипты.

Установка Ansible

Ansible нужно устанавливать только на той машине, с которой будет выполняться управление устройствами.

Требования к управляемому хосту:

- поддержка Python 3 (тестировалось на 3.6)
- Windows не может быть управляемым хостом

В книге используется Ansible версии 2.4

Если Вы используете Ansible в работе, не только для тестов, возможно, следует использовать стабильную версию и просто использовать Python 2.7. Это легко сделать, установив Ansible в виртуальном окружении, где по умолчанию используется Python 2.7.

Если Вы хотите использовать Ansible с Python 2.7, для этого раздела лучше переключиться на книгу для Python 2.7, так как они могут отличаться.

Ansible довольно часто обновляется, поэтому лучше установить его в виртуальном окружении. Новые версии выходят примерно раз в полгода.

Установить Ansible можно [по-разному](#).

С помощью pip Ansible можно установить таким образом:

```
$ pip install ansible
```

Параметры оборудования

В примерах раздела используются три маршрутизатора и один коммутатор. К ним нет никаких требований, только настроенный SSH.

Параметры, которые используются в разделе:

- пользователь: cisco
- пароль: cisco
- пароль на режим enable: cisco
- SSH версии 2
- IP-адреса:
 - R1: 192.168.100.1
 - R2: 192.168.100.2
 - R3: 192.168.100.3
 - SW1: 192.168.100.100

Если Вы будете использовать другие параметры, измените соответственно инвентарный файл, конфигурационный файл Ansible и файл group_vars/all.yml.

Основы Ansible

Ansible:

- Работает без установки агента на управляемые хосты
- Использует SSH для подключения к управляемым хостам
- Выполняет изменения с помощью модулей Python, которые выполняются на управляемых хостах
- Может выполнять действия локально на управляющем хосте
- Использует YAML для описания сценариев
- Содержит множество модулей (их количество постоянно растет)
- Легко писать свои модули

Терминология

- **Control machine** — управляющий хост. Сервер Ansible, с которого происходит управление другими хостами
- **Manage node** — управляемые хосты
- **Inventory** — инвентарный файл. В этом файле описываются хосты, группы хостов, а также могут быть созданы переменные
- **Playbook** — файл сценариев
- **Play** — сценарий (набор задач). Связывает задачи с хостами, для которых эти задачи надо выполнить
- **Task** — задача. Вызывает модуль с указанными параметрами и переменными
- **Module** — модуль Ansible. Реализует определенные функции

Список терминов в [документации](#).

Quick start

С Ansible очень просто начать работать. Минимум, который нужен для начала работы:

- инвентарный файл - в нём описываются устройства
- изменить конфигурацию Ansible для работы с сетевым оборудованием
- разобраться с ad-hoc командами - это возможность выполнять простые действия с устройствами из командной строки
 - например, с помощью ad-hoc команд можно отправить команду show на несколько устройств

Намного больше возможностей появится при использовании playbook (файлы сценариев). Но ad-hoc команды намного проще начать использовать. И с ними легче начать разбираться с Ansible.

Инвентарный файл

Инвентарный файл - это файл, в котором описываются устройства, к которым Ansible будет подключаться.

Хосты и группы

В инвентарном файле устройства могут указываться используя IP-адреса или имена. Устройства могут быть указаны по одному или разбиты на группы.

Файл описывается в формате INI. Пример файла:

```
r5.example.com

[cisco-routers]
192.168.255.1
192.168.255.2
192.168.255.3
192.168.255.4

[cisco-edge-routers]
192.168.255.1
192.168.255.2
```

Название, которое указано в квадратных скобках - это название группы. В данном случае, созданы две группы устройств: `cisco-routers` и `cisco-edge-routers`.

Обратите внимание, что адреса `192.168.255.1` и `192.168.255.2` находятся в двух группах. Это нормальная ситуация, один и тот же адрес или имя хоста, можно помещать в разные группы.

Таким образом можно применять отдельно какие-то политики для группы `cisco-edge-routers`, но в то же время, когда необходимо настроить что-то, что касается всех маршрутизаторов, можно использовать группу `cisco-routers`.

К разбиению на группы надо подходить внимательно. Ansible это еще и, в какой-то мере, система описания инфраструктуры. Позже мы будем рассматривать групповые переменные и роли, где значение групп будет заметно в полной мере.

По умолчанию, файл находится в `/etc/ansible/hosts`.

Но можно создавать свой инвентарный файл и использовать его. Для этого нужно, либо указать его при запуске `ansible`, используя опцию `-i <путь>`, либо указать файл в конфигурационном файле Ansible.

Часто инвентарный файл размещают в каталоге `inventories`, который создают в корне каталога с `playbook`. Это дает возможность хранить информацию про хосты вместе с остальной информацией в системе контроля версий.

Если инфраструктура большая и хостов много, то имеет смысл разбить инвентарный файл на несколько частей:

```
inventories/
├── branch-A
│   ├── cisco-routers
│   └── cisco-switches
├── branch-B
│   ├── cisco-routers
│   └── cisco-switches
└── headquarter
    ├── cisco-routers
    ├── cisco-switches
    └── juniper-routers
```

Если какое-то из устройств использует нестандартный порт SSH, порт можно указать после имени или адреса устройства, через двоеточие (ниже показан пример).

Такой вариант указания порта работает только с подключениями OpenSSH и не работает с paramiko.

Пример инвентарного файла, с использованием нестандартных портов для SSH:

```
[cisco-routers]
192.168.255.1:22022
192.168.255.2:22022
192.168.255.3:22022

[cisco-switches]
192.168.254.1
192.168.254.2
```

Если в группу надо добавить несколько устройств с однотипными именами, можно использовать такой вариант записи:

```
[cisco-routers]
192.168.255.[1:5]
```

Такая запись означает, что в группу попадут устройства с адресами 192.168.255.1-192.168.255.5. Этот формат записи поддерживается и для имен хостов:

```
[cisco-routers]
router[A:D].example.com
```

Группа из групп

Ansible также позволяет объединять группы устройств в общую группу. Для этого используется специальный синтаксис:

```
[cisco-routers]
192.168.255.1
192.168.255.2
192.168.255.3

[cisco-switches]
192.168.254.1
192.168.254.2

[cisco-devices:children]
cisco-routers
cisco-switches
```

Группы по-умолчанию

По-умолчанию, в Ansible существует две группы: `all` и `ungrouped`. Первая включает в себя все хосты, а вторая, соответственно, хосты, которые не принадлежат ни одной из групп.

Ad Hoc команды

Ad-hoc команды - это возможность запустить какое-то действие Ansible из командной строки.

Такой вариант используется, как правило, в тех случаях, когда надо что-то проверить, например, работу модуля. Или просто выполнить какое-то разовое действие, которое не нужно сохранять.

В любом случае, это простой и быстрый способ начать использовать Ansible.

Сначала нужно создать в локальном каталоге инвентарный файл. Назовем его myhosts:

```
[cisco-routers]
192.168.100.1
192.168.100.2
192.168.100.3

[cisco-switches]
192.168.100.100
```

При подключении к устройствам первый раз, сначала лучше подключиться к ним вручную, чтобы ключи устройств были сохранены локально. В Ansible есть возможность отключить эту первоначальную проверку ключей. В разделе о конфигурационном файле мы посмотрим, как это делать (такой вариант может понадобиться, если надо подключаться к большому количеству устройств).

Пример ad-hoc команды:

```
$ ansible cisco-routers -i myhosts -m raw -a "sh ip int br" -u cisco --ask-pass
```

Разберемся с параметрами команды:

- `cisco-routers` - группа устройств, к которым нужно применить действия
 - эта группа должна существовать в инвентарном файле
 - это может быть конкретное имя или адрес
 - если нужно указать все хосты из файла, можно использовать значение `all` или *
 - Ansible поддерживает более сложные варианты указания хостов, с регулярными выражениями и разными шаблонами. Подробнее об этом в [документации](#)
- `-i myhosts` - параметр `-i` позволяет указать инвентарный файл

- `-m raw -a "sh ip int br"` - параметр `-m raw` означает, что используется модуль `raw`
 - этот модуль позволяет отправлять команды в SSH сессии, но при этом не загружает на хост модуль Python. То есть, этот модуль просто отправляет указанную команду как строку и всё
 - плюс модуля `raw` в том, что он может использоваться для любой системы, которую поддерживает Ansible
 - `-a "sh ip int br"` - параметр `-a` указывает, какую команду отправить
- `-u cisco` - подключение выполняется от имени пользователя `cisco`
- `--ask-pass` - параметр, который нужно указать, чтобы аутентификация была по паролю, а не по ключам

Результат выполнения будет таким:

```
$ ansible cisco-routers -i myhosts -m raw -a "sh ip int br" -u cisco --ask-pass
```

```
SSH password:  
192.168.100.1 | FAILED | rc=0 >>  
to use the 'ssh' connection type with passwords, you must install the sshpass program  
  
192.168.100.2 | FAILED | rc=0 >>  
to use the 'ssh' connection type with passwords, you must install the sshpass program  
  
192.168.100.3 | FAILED | rc=0 >>  
to use the 'ssh' connection type with passwords, you must install the sshpass program
```

Ошибка значит, что нужно установить программу `sshpass`. Эта особенность возникает, только когда используется аутентификация по паролю.

Установка `sshpass`:

```
$ sudo apt-get install sshpass
```

Команду надо выполнить повторно:

```
$ ansible cisco-routers -i myhosts -m raw -a "sh ip int br" -u cisco --ask-pass
```

SSH password:

192.168.100.1 | SUCCESS | rc=0 >>

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	unassigned	YES	manual	administratively down	down
Ethernet0/3	unassigned	YES	manual	up	up
Loopback0	10.1.1.1	YES	manual	up	up

Shared connection to 192.168.100.1 closed.

192.168.100.2 | SUCCESS | rc=0 >>

Interface	IP-Address	OK?	Method	Status	Protocol
Ethernet0/0	192.168.100.2	YES	manual	up	up
Ethernet0/1	unassigned	YES	unset	administratively down	down
Ethernet0/2	192.168.200.1	YES	manual	administratively down	down
Ethernet0/3	unassigned	YES	manual	up	up
Loopback0	10.1.1.1	YES	manual	up	up

Connection to 192.168.100.2 closed by remote host.
Shared connection to 192.168.100.2 closed.

192.168.100.3 | SUCCESS | rc=0 >>

Interface	IP-Address	OK?	Method	Status	Protocol
Ethernet0/0	192.168.100.3	YES	manual	up	up
Ethernet0/1	unassigned	YES	unset	administratively down	down
Ethernet0/2	192.168.200.1	YES	manual	administratively down	down
Ethernet0/3	unassigned	YES	manual	up	up
Loopback0	10.1.1.1	YES	manual	up	up
Loopback10	10.255.3.3	YES	manual	up	up

Shared connection to 192.168.100.3 closed.

Теперь всё прошло успешно. Команда выполнилась, и отобразился вывод с каждого устройства.

Аналогичным образом можно попробовать выполнять и другие команды и/или на других комбинациях устройств.

Конфигурационный файл

Настройки Ansible можно менять в конфигурационном файле.

Конфигурационный файл Ansible может храниться в разных местах (файлы перечислены в порядке уменьшения приоритета):

- ANSIBLE_CONFIG (переменная окружения)
- ansible.cfg (в текущем каталоге)
- .ansible.cfg (в домашнем каталоге пользователя)
- /etc/ansible/ansible.cfg

Ansible ищет файл конфигурации в указанном порядке и использует первый найденный (конфигурация из разных файлов не совмещается).

В конфигурационном файле можно менять множество параметров. Полный список параметров и их описание можно найти в [документации](#).

В текущем каталоге должен быть инвентарный файл myhosts:

```
[cisco-routers]
192.168.100.1
192.168.100.2
192.168.100.3

[cisco-switches]
192.168.100.100
```

В текущем каталоге надо создать такой конфигурационный файл ansible.cfg:

```
[defaults]

inventory = ./myhosts
remote_user = cisco
ask_pass = True
```

Настройки в конфигурационном файле:

- [defaults] - эта секция конфигурации описывает общие параметры по умолчанию
- inventory = ./myhosts - параметр inventory позволяет указать местоположение инвентарного файла.
 - Если настроить этот параметр, не придется указывать, где находится файл, при каждом запуске Ansible

- `remote_user = cisco` - от имени какого пользователя будет подключаться Ansible
- `ask_pass = True` - этот параметр аналогичен опции `--ask-pass` в командной строке. Если он выставлен в конфигурации Ansible, то уже не нужно указывать его в командной строке.

Теперь вызов ad-hoc команды будет выглядеть так:

```
$ ansible cisco-routers -m raw -a "sh ip int br"
```

Теперь не нужно указывать инвентарный файл, пользователя и опцию `--ask-pass`.

gathering

По умолчанию Ansible собирает факты об устройствах.

Факты - это информация о хостах, к которым подключается Ansible. Эти факты можно использовать в playbook и шаблонах как переменные.

Сбором фактов, по умолчанию, занимается модуль [setup](#).

Но для сетевого оборудования модуль `setup` не подходит, поэтому сбор фактов надо отключить. Это можно сделать в конфигурационном файле Ansible или в playbook.

Для сетевого оборудования нужно использовать отдельные модули для сбора фактов (если они есть). Это рассматривается в разделе [ios_facts](#).

Отключение сбора фактов в конфигурационном файле:

```
gathering = explicit
```

host_key_checking

Параметр `host_key_checking` отвечает за проверку ключей при подключении по SSH. Если указать в конфигурационном файле `host_key_checking=False`, проверка будет отключена.

Это полезно, когда с управляющего хоста Ansible надо подключиться к большому количеству устройств первый раз.

Чтобы проверить этот функционал, надо удалить сохраненные ключи для устройств Cisco, к которым уже выполнялось подключение.

В линукс они находятся в файле `~/.ssh/known_hosts`.

Если выполнить ad-hoc команду после удаления ключей, вывод будет таким:

```
$ ansible cisco-routers -m raw -a "sh ip int br"
```

```
SSH password:  
192.168.100.1 | FAILED | rc=0 >>  
Using a SSH password instead of a key is not possible because Host Key checking is enabled  
and sshpass does not support this. Please add this host's fingerprint to your known_hosts  
file to manage this host.  
  
192.168.100.2 | FAILED | rc=0 >>  
Using a SSH password instead of a key is not possible because Host Key checking is enabled  
and sshpass does not support this. Please add this host's fingerprint to your known_hosts  
file to manage this host.  
  
192.168.100.3 | FAILED | rc=0 >>  
Using a SSH password instead of a key is not possible because Host Key checking is enabled  
and sshpass does not support this. Please add this host's fingerprint to your known_hosts  
file to manage this host.
```

Добавляем в конфигурационный файл параметр host_key_checking:

```
[defaults]  
  
inventory = ./myhosts  
  
remote_user = cisco  
ask_pass = True  
  
host_key_checking=False
```

И повторим ad-hoc команду:

```
$ ansible cisco-routers -m raw -a "sh ip int br"
```

```

SSH password:
192.168.100.1 | SUCCESS | rc=0 >>

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        unassigned     YES manual administratively down down
Ethernet0/3        unassigned     YES manual up            up
Tunnel0            unassigned     YES unset up            down
Tunnel1            unassigned     YES unset up            down
Tunnel3            unassigned     YES unset up            down
Tunnel9            unassigned     YES unset up            down
Tunnel10           unassigned     YES unset up            down
Tunnel11           unassigned     YES unset up            down
Tunnel15           unassigned     YES unset up            down
Warning: Permanently added '192.168.100.1' (RSA) to the list of known hosts.
Shared connection to 192.168.100.1 closed.

192.168.100.3 | SUCCESS | rc=0 >>

Interface          IP-Address      OK? Method Status      Protocol
Ethernet0/0        192.168.100.3  YES manual up           up
Ethernet0/1        unassigned     YES unset administratively down down
Ethernet0/2        192.168.200.1  YES manual administratively down down
Ethernet0/3        unassigned     YES manual up            up
Loopback10         10.255.3.3    YES manual up           up
Warning: Permanently added '192.168.100.3' (RSA) to the list of known hosts.
Shared connection to 192.168.100.3 closed.

192.168.100.2 | SUCCESS | rc=0 >>

Interface          IP-Address      OK? Method Status      Protocol
Ethernet0/0        192.168.100.2  YES manual up           up
Ethernet0/1        unassigned     YES unset administratively down down
Ethernet0/2        unassigned     YES manual administratively down down
Ethernet0/3        unassigned     YES manual up            up
Loopback0          10.0.0.2      YES manual up           up
Warning: Permanently added '192.168.100.2' (RSA) to the list of known hosts.
Connection to 192.168.100.2 closed by remote host.
Shared connection to 192.168.100.2 closed.

```

Обратите внимание на строки:

```
Warning: Permanently added '192.168.100.1' (RSA) to the list of known hosts.
```

Ansible сам добавил ключи устройств в файл `~/.ssh/known_hosts`. При подключении в следующий раз этого сообщения уже не будет.

Другие параметры конфигурационного файла можно посмотреть в документации.

Пример конфигурационного файла в [репозитории Ansible](#).

Модули Ansible

Вместе с установкой Ansible устанавливается также большое количество модулей (библиотека модулей). В текущей библиотеке модулей находится порядка 200 модулей.

Модули отвечают за действия, которые выполняет Ansible. При этом каждый модуль, как правило, отвечает за свою конкретную и небольшую задачу.

Модули можно выполнять отдельно, в ad-hoc командах или собирать в определенный сценарий (play), а затем в playbook.

Как правило, при вызове модуля ему нужно передать аргументы. Какие-то аргументы будут управлять поведением и параметрами модуля, а какие-то передавать, например, команду, которую надо выполнить.

Например, мы уже выполняли ad-hoc команды, используя модуль raw, и передавали ему аргументы:

```
$ ansible cisco-routers -i myhosts -m raw -a "sh ip int br" -u cisco --ask-pass
```

Выполнение такой же задачи в playbook будет выглядеть так (playbook рассматривается в следующем разделе):

```
- name: run sh ip int br
  raw: sh ip int br | ex unass
```

После выполнения модуль возвращает результаты в формате JSON.

Модули Ansible, как правило, идемпотентны. Это означает, что модуль можно выполнять сколько угодно раз, но при этом модуль будет выполнять изменения, только если система не находится в желаемом состоянии.

В Ansible модули разделены на две категории:

- **core** - это модули, которые всегда устанавливаются вместе с Ansible. Их поддерживает основная команда разработчиков Ansible.
- **extra** - это модули на данный момент устанавливаются с Ansible, но нет гарантии, что они и дальше будут устанавливаться с Ansible. Возможно, в будущем их нужно будет устанавливать отдельно. Большинство этих модулей поддерживается сообществом.

Также в Ansible модули разделены по функциональности. Список всех категорий находится в [документации](#).

Основы playbooks

Playbook (файл сценариев) — это файл, в котором описываются действия, которые нужно выполнить на какой-то группе хостов.

Внутри playbook:

- play - это набор задач, которые нужно выполнить для группы хостов
- task - это конкретная задача. В задаче есть как минимум:
 - описание (название задачи можно не писать, но очень рекомендуется)
 - модуль и команда (действие в модуле)

Синтаксис playbook

Playbook описываются в формате YAML.

Синтаксис YAML описан в [разделе YAML](#) или в [документации Ansible](#).

Пример синтаксиса playbook

Все примеры этого раздела находятся в каталоге 2_playbook_basics

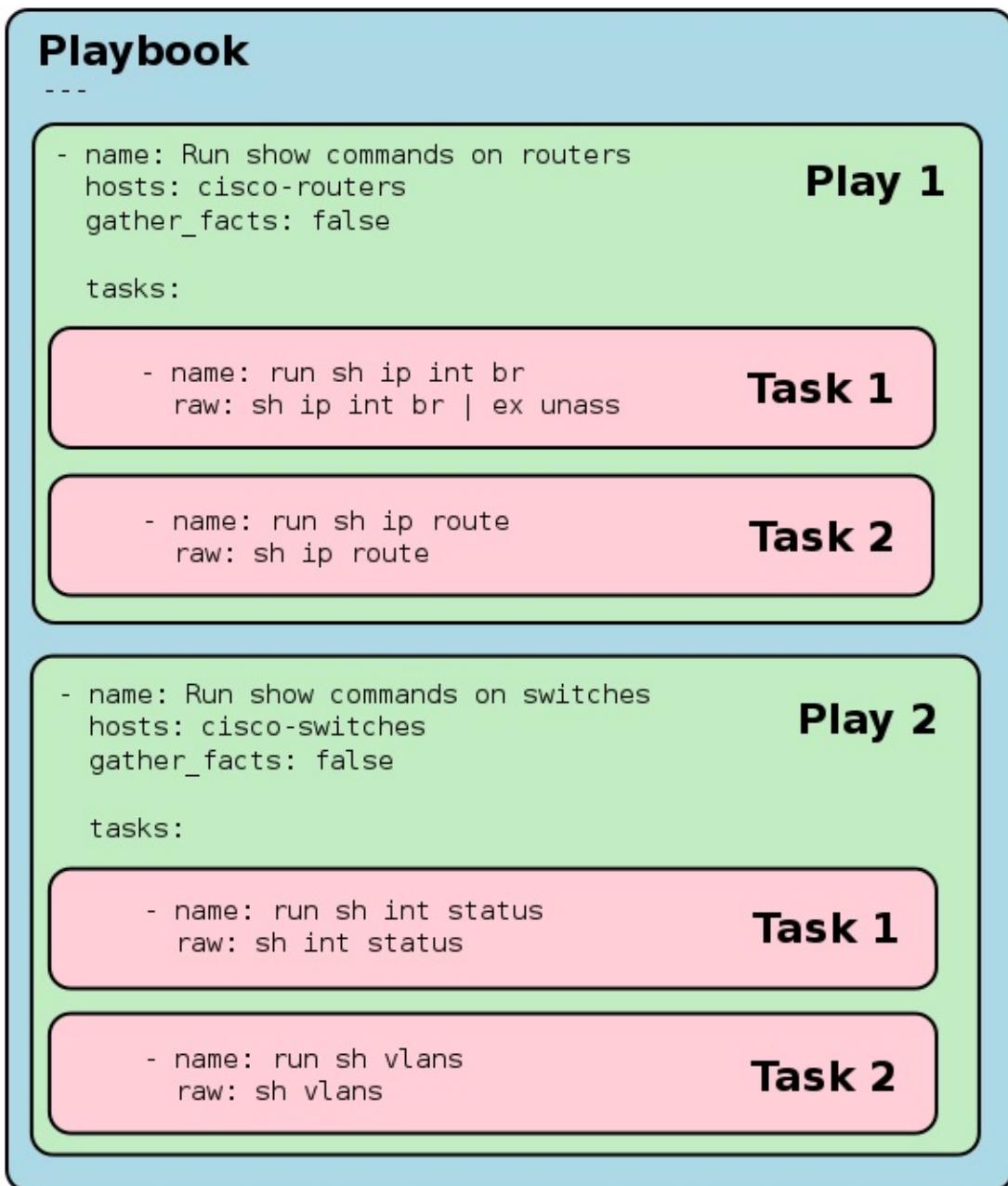
Пример playbook 1_show_commands_with_raw.yml:

```
---  
- name: Run show commands on routers  
  hosts: cisco-routers  
  gather_facts: false  
  
  tasks:  
  
    - name: run sh ip int br  
      raw: sh ip int br | ex unass  
  
    - name: run sh ip route  
      raw: sh ip route  
  
- name: Run show commands on switches  
  hosts: cisco-switches  
  gather_facts: false  
  
  tasks:  
  
    - name: run sh int status  
      raw: sh int status  
  
    - name: run sh vlan  
      raw: show vlan
```

В playbook два сценария (play):

- `name: Run show commands on routers` - имя сценария (play). Этот параметр обязательно должен быть в любом сценарии
- `hosts: cisco-routers` - сценарий будет применяться к устройствам в группе `cisco-routers`
 - тут может быть указано и несколько групп, например, таким образом: `hosts: cisco-routers:cisco-switches`. Подробнее в [документации](#)
- обычно, в play надо указывать параметр `remote_user`. Но, так как мы указали его в конфигурационном файле Ansible, можно не указывать его в play.
- `gather_facts: false` - отключение сбора фактов об устройстве, так как для сетевого оборудования надо использовать отдельные модули для сбора фактов.
 - в разделе [конфигурационный файл](#) рассматривалось, как отключить сбор фактов по умолчанию. Если он отключен, то параметр `gather_facts` в play не нужно указывать.
- `tasks:` - дальше идет перечень задач
 - в каждой задаче настроено имя (опционально) и действие. Действие может быть только одно.
 - в действии указывается, какой модуль использовать, и параметры модуля.

И тот же playbook с отображением элементов:



Так выглядит выполнение playbook:

```
$ ansible-playbook 1_show_commands_with_raw.yml
```

```
PLAY [Run show commands on routers] *****  
  
TASK [run sh ip int br] *****  
changed: [192.168.100.1]  
changed: [192.168.100.3]  
changed: [192.168.100.2]  
  
TASK [run sh ip route] *****  
changed: [192.168.100.1]  
changed: [192.168.100.3]  
changed: [192.168.100.2]  
  
PLAY [Run show commands on switches] *****  
  
TASK [run sh int status] *****  
changed: [192.168.100.100]  
  
TASK [run sh vlans] *****  
changed: [192.168.100.100]  
  
PLAY RECAP *****  
192.168.100.1 : ok=2    changed=2    unreachable=0    failed=0  
192.168.100.100 : ok=2    changed=2    unreachable=0    failed=0  
192.168.100.2 : ok=2    changed=2    unreachable=0    failed=0  
192.168.100.3 : ok=2    changed=2    unreachable=0    failed=0
```

Обратите внимание, что для запуска playbook используется другая команда. Для ad-hoc команды использовалась команда ansible. А для playbook - ansible-playbook.

Для того, чтобы убедиться, что команды, которые указаны в задачах, выполнились на устройствах, запустите playbook с опцией -v (вывод сокращен):

```
$ ansible-playbook 1_show_commands_with_raw.yml -v
```

```
SSH password:  
  
PLAY [Run show commands on routers] *****  
  
TASK [run sh ip int br] *****  
changed: [192.168.100.1] => {"changed": true, "rc": 0, "stderr": "Shared connection  
to 192.168.100.1 closed.\r\n", "stdout": "\r\nInterface IP-Address  
s 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\nLoopback0  
10.1.1.1 YES manual up up \r\n", "stdout_lines":  
": ["", "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 ", "Loopback0 10.1.1.1 YES manual up  
up "]}
```

В следующих разделах мы научимся отображать эти данные в нормальном формате и посмотрим, что с ними можно делать.

Порядок выполнения задач и сценариев

Сценарии (play) и задачи (task) выполняются последовательно, в том порядке, в котором они описаны в playbook.

Если в сценарии, например, две задачи, то сначала первая задача должна быть выполнена для всех устройств, которые указаны в параметре hosts. Только после того, как первая задача была выполнена для всех хостов, начинается выполнение второй задачи.

Если в ходе выполнения playbook возникла ошибка в задаче на каком-то устройстве, это устройство исключается, и другие задачи на нём выполняться не будут.

Например, заменим пароль пользователя cisco на cisco123 (правильный cisco) на маршрутизаторе 192.168.100.1 и запустим playbook заново:

```
$ ansible-playbook 1_show_commands_with_raw.yml
```

```
SSH password: *****

PLAY [Run show commands on routers] *****
TASK [run sh ip int br] *****
changed: [192.168.100.2]
changed: [192.168.100.3]
fatal: [192.168.100.1]: FAILED! => {"changed": true, "failed": true, "msg": "non-zero return code", "rc": 5, "stderr": "", "stdout": "", "stdout_lines": []}

TASK [run sh ip route] *****
changed: [192.168.100.2]
changed: [192.168.100.3]

PLAY [Run show commands on switches] *****
TASK [run sh int status] *****
changed: [192.168.100.100]

TASK [run sh vlan] *****
changed: [192.168.100.100]
      to retry, use: --limit @/home/vagrant/repos/pyneng-examples-exercises/examples/15_ansible/2_playbook_basics/1_show_commands_with_raw.retry

PLAY RECAP *****
192.168.100.1      : ok=0    changed=0    unreachable=0    failed=1
192.168.100.100    : ok=2    changed=2    unreachable=0    failed=0
192.168.100.2      : ok=2    changed=2    unreachable=0    failed=0
192.168.100.3      : ok=2    changed=2    unreachable=0    failed=0
```

Обратите внимание на ошибку в выполнении первой задачи для маршрутизатора 192.168.100.1.

Во второй задаче 'TASK [run sh ip route]', Ansible уже исключил маршрутизатор и выполняет задачу только для маршрутизаторов 192.168.100.2 и 192.168.100.3.

Еще один важный аспект - Ansible выдал сообщение:

```
to retry, use: --limit @/home/vagrant/repos/pyneng-examples-exercises/examples/23_ansible/2_playbook_basics/1_show_commands_with_raw.retry
```

Если при выполнении playbook, на каком-то устройстве возникла ошибка, Ansible создает специальный файл, который называется точно так же, как playbook, но расширение меняется на `retry`. (Если Вы выполняете задания параллельно, то этот файл должен появиться у Вас)

В этом файле хранится имя или адрес устройства, на котором возникла ошибка. Так выглядит файл `1_show_commands_with_raw.retry` сейчас:

```
192.168.100.1
```

Создается этот файл для того, чтобы можно было перезапустить playbook заново только для проблемного устройства (устройств). То есть, надо исправить проблему с устройством и заново запустить playbook.

Настраиваем правильный пароль на маршрутизаторе 192.168.100.1, а затем перезапускаем playbook таким образом:

```
$ ansible-playbook 1_show_commands_with_raw.yml --limit @/home/vagrant/repos/pyneng-examples-exercises/examples/23_ansible_basics/2_playbook_basics/1_show_commands_with_raw.retry
```

```
SSH password:
```

```
PLAY [Run show commands on routers] *****
TASK [run sh ip int br] *****
changed: [192.168.100.1]

TASK [run sh ip route] *****
changed: [192.168.100.1]

PLAY RECAP *****
192.168.100.1 : ok=2    changed=2    unreachable=0    failed=0
```

Ansible взял список устройств, которые перечислены в файле `retry`, и выполнил playbook только для них.

Можно было запустить playbook и так (то есть, писать не полный путь к файлу `retry`):

```
$ ansible-playbook 1_show_commands_with_raw.yml --limit @1_show_commands_with_raw.retr  
у
```

Параметр `--limit` очень полезная вещь. Он позволяет ограничивать, для каких хостов или групп будет выполняться playbook, при этом не меняя сам playbook.

Например, таким образом playbook можно запустить только для маршрутизатора 192.168.100.1:

```
$ ansible-playbook 1_show_commands_with_raw.yml --limit 192.168.100.1
```

Идемпотентность

Модули Ansible идемпотентны. Это означает, что модуль можно выполнять сколько угодно раз, но при этом модуль будет выполнять изменения, только если система не находится в желаемом состоянии.

Но есть исключения из такого поведения. Например, модуль `raw` всегда вносит изменения. Поэтому при выполнении playbook выше всегда отображалось состояние `changed`.

Но, если, например, в задаче указано, что на сервер Linux надо установить пакет `httpd`, то он будет установлен только в том случае, если его нет. То есть, действие не будет повторяться снова и снова при каждом запуске, а лишь тогда, когда пакета нет.

Аналогично и с сетевым оборудованием. Если задача модуля - выполнить команду в конфигурационном режиме, а она уже есть на устройстве, модуль не будет вносить изменения.

Переменные

Переменной может быть, например:

- информация об устройстве, которая собрана как факт, а затем используется в шаблоне.
- в переменные можно записывать полученный вывод команды.
- переменная может быть указана вручную в playbook

Имена переменных

В Ansible есть определенные ограничения по формату имен переменных:

- Переменные могут состоять из букв, чисел и символа _
- Переменные должны начинаться с буквы

Кроме того, можно создавать словари с переменными (в формате YAML):

```
R1:  
  IP: 10.1.1.1/24  
  DG: 10.1.1.100
```

Обращаться к переменным в словаре можно двумя вариантами:

```
R1['IP']  
R1.IP
```

Правда, при использовании второго варианта могут быть проблемы, если название ключа совпадает с зарезервированным словом (методом или атрибутом) в Python или Ansible.

Где можно определять переменные

Переменные можно создавать:

- в инвентарном файле
- в playbook
- в специальных файлах для группы/устройства
- в отдельных файлах, которые добавляются в playbook через include (как в Jinja2)

- в ролях, которые затем используются
- можно даже передавать переменные при вызове playbook

Также можно использовать факты, которые были собраны про устройство, как переменные.

Переменные в инвентарном файле

В инвентарном файле можно указывать переменные для группы:

```
[cisco-routers]
192.168.100.1
192.168.100.2
192.168.100.3

[cisco-switches]
192.168.100.100

[cisco-routers:vars]
ntp_server=192.168.255.100
log_server=10.255.100.1
```

Переменные `ntp_server` и `log_server` относятся к группе `cisco-routers` и могут использоваться, например, при генерации конфигурации на основе шаблона.

Переменные в playbook

Переменные можно задавать прямо в playbook. Это может быть удобно тем, что переменные находятся там же, где все действия.

Например, можно задать переменные `ntp_server` и `log_server` в playbook таким образом:

```

---
- name: Run show commands on routers
hosts: cisco-routers
gather_facts: false

vars:
  ntp_server: 192.168.255.100
  log_server: 10.255.100.1

tasks:
  - name: run sh ip int br
    raw: sh ip int br | ex unass

  - name: run sh ip route
    raw: sh ip route

```

Переменные в специальных файлах для группы/устройства

Ansible позволяет хранить переменные для группы/устройства в специальных файлах:

- Для групп устройств, переменные должны находиться в каталоге `group_vars`, в файлах, которые называются, как имя группы.
 - Кроме того, можно создавать в каталоге `group_vars` файл `all`, в котором будут находиться переменные, которые относятся ко всем группам.
- Для конкретных устройств, переменные должны находиться в каталоге `host_vars`, в файлах, которые соответствуют имени или адресу хоста.
- Все файлы с переменными должны быть в формате YAML. Расширение файла может быть таким: `yml`, `yaml`, `json` или без расширения
- каталоги `group_vars` и `host_vars` должны находиться в том же каталоге, что и `playbook`, или могут находиться внутри каталога `inventory` (первый вариант более распространенный).
 - если каталоги и файлы названы правильно и расположены в указанных каталогах, Ansible сам распознает файлы и будет использовать переменные

Например, если инвентарный файл `myhosts` выглядит так:

```
[cisco-routers]
192.168.100.1
192.168.100.2
192.168.100.3

[cisco-switches]
192.168.100.100
```

Можно создать такую структуру каталогов:

```
└── group_vars
    ├── all.yml
    ├── cisco-routers.yml      | Каталог с переменными для групп устройств
    └── cisco-switches.yml
    |
    └── host_vars
        ├── 192.168.100.1
        ├── 192.168.100.2
        ├── 192.168.100.3      | Каталог с переменными для устройств
        └── 192.168.100.100
        |
        └── myhosts            | Инвентарный файл
```

Ниже пример содержимого файлов переменных для групп устройств и для отдельных хостов.

group_vars/all.yml (в этом файле указываются значения по умолчанию, которые относятся ко всем устройствам):

```
---
cli:
  host: "{{ inventory_hostname }}"
  username: "cisco"
  password: "cisco"
  authorize: yes
  auth_pass: "cisco"
```

В данном случае указываются переменные, которые предопределены самим Ansible.

В файле group_vars/all.yml создан словарь cli. В этом словаре перечислены те аргументы, которые должны задаваться для работы с сетевым оборудованием через встроенные модули Ansible (рассматривается в разделе [сетевые модули](#))

Интересный момент в этом файле - переменная host: "{{ inventory_hostname }}":

- inventory_hostname - это специальная переменная, которая указывает на тот хост,

для которого Ansible выполняет действия.

- синтаксис {{ inventory_hostname }} - это подстановка переменных. Используется формат Jinja

group_vars/cisco-routers.yml

```
---  
  
log_server: 10.255.100.1  
ntp_server: 10.255.100.1  
users:  
    user1: pass1  
    user2: pass2  
    user3: pass3
```

В файле group_vars/cisco-routers.yml находятся переменные, которые указывают IP-адреса Log и NTP серверов и нескольких пользователей. Эти переменные могут использоваться, например, в шаблонах конфигурации.

group_vars/cisco-switches.yml

```
---  
  
vlans:  
    - 10  
    - 20  
    - 30
```

В файле group_vars/cisco-switches.yml указана переменная vlans со списком VLANов.

Файлы с переменными для хостов однотипны, и в них меняются только адреса и имена:

Файл host_vars/192.168.100.1

```
---  
  
hostname: london_r1  
mgmnt_loopback: 100  
mgmnt_ip: 10.0.0.1  
ospf_ints:  
    - 192.168.100.1  
    - 10.0.0.1  
    - 10.255.1.1
```

Файл host_vars/192.168.100.2

```
---  
  
hostname: london_r2  
mgmnt_loopback: 100  
mgmnt_ip: 10.0.0.2  
ospf_ints:  
  - 192.168.100.2  
  - 10.0.0.2  
  - 10.255.2.2
```

Файл host_vars/192.168.100.3

```
---  
  
hostname: london_r3  
mgmnt_loopback: 100  
mgmnt_ip: 10.0.0.3  
ospf_ints:  
  - 192.168.100.3  
  - 10.0.0.3  
  - 10.255.3.3
```

Файл host_vars/192.168.100.100

```
---  
  
hostname: london_sw1  
mgmnt_int: VLAN100  
mgmnt_ip: 10.0.0.100
```

Приоритет переменных

В этом разделе не рассматривается размещение переменных:

- в отдельных файлах, которые добавляются в playbook через include (как в Jinja2)
- в ролях, которые затем используются
- передача переменных при вызове playbook

Это рассматривается в курсе [Ansible для сетевых инженеров](#)

Чаще всего, переменная с определенным именем только одна. Но иногда может понадобиться создать переменную в разных местах, и тогда нужно понимать, в каком порядке Ansible перезаписывает переменные.

Приоритет переменных (последние значения переписывают предыдущие):

- Значения переменных в ролях
 - задачи в ролях будут видеть собственные значения. Задачи, которые определены вне роли, будут видеть последние значения переменных роли
- переменные в инвентарном файле
- переменные для группы хостов в инвентарном файле
- переменные для хостов в инвентарном файле
- переменные в каталоге `group_vars`
- переменные в каталоге `host_vars`
- факты хоста
- переменные сценария (`play`)
- переменные сценария, которые запрашиваются через `vars_prompt`
- переменные, которые передаются в сценарий через `vars_files`
- переменные, полученные через параметр `register`
- `set_facts`
- переменные из роли и помещенные через `include`
- переменные блока (переписывают другие значения только для блока)
- переменные задачи (`task`) (переписывают другие значения только для задачи)
- переменные, которые передаются при вызове `playbook` через параметр `--extra-vars` (всегда наиболее приоритетные)

Работа с результатами выполнения модуля

В этом разделе рассматриваются несколько способов, которые позволяют посмотреть на вывод, полученный с устройств.

Примеры используют модуль raw, но аналогичные принципы работают и с другими модулями.

verbose

В предыдущих разделах один из способов отобразить результат выполнения команд уже использовался - флаг verbose.

Конечно, вывод не очень удобно читать, но, как минимум, он позволяет увидеть, что команды выполнились. Также этот флаг позволяет подробно посмотреть, какие шаги выполняет Ansible.

Пример запуска playbook с флагом verbose (вывод сокращен):

```
ansible-playbook 1_show_commands_with_raw.yml -v
```

```
SSH password:

PLAY [Run show commands on routers] ****
TASK [run sh ip int br] ****
changed: [192.168.100.1] => {"changed": true, "rc": 0, "stderr": "Shared connection\n      to 192.168.100.1 closed.\r\n", "stdout": "\r\nInterface          IP-Address\nOK? Method Status      Protocol\r\nEthernet0/0        192.\n168.100.1    YES NVRAM up           \r\nEthernet0/1        192.168.200.1\n10.1.1.1      YES manual up       \r\n                           up      \r\nLoopback0         192.168.100.1\n                           up      \r\n                           OK? Method Status\nProtocol", "Ethernet0/0\n                           up      ", "Ethernet0/1\n                           up      ", "Loopback0\n                           up      "]}
```

При увеличении количества букв v в флаге, вывод становится более подробным. Попробуйте вызывать этот же playbook и добавлять к флагу буквы v (5 и больше показывают одинаковый вывод) таким образом:

```
ansible-playbook 1_show_commands_with_raw.yml -vvv
```

В выводе видны результаты выполнения задачи, они возвращаются в формате JSON:

- **changed** - ключ, который указывает, были ли внесены изменения
- **rc** - return code. Это поле появляется в выводе тех модулей, которые выполняют какие-то команды
- **stderr** - ошибки при выполнении команды. Это поле появляется в выводе тех модулей, которые выполняют какие-то команды
- **stdout** - вывод команды
- **stdout_lines** - вывод в виде списка команд, разбитых построчно

register

Параметр **register** сохраняет результат выполнения модуля в переменную. Затем эта переменная может использоваться в шаблонах, в принятии решений о ходе сценария или для отображения вывода.

Попробуем сохранить результат выполнения команды.

В playbook `2_register_vars.yml` с помощью `register` вывод команды `sh ip int br` сохранен в переменную `sh_ip_int_br_result`:

```
---
```

```
- name: Run show commands on routers
  hosts: cisco-routers
  gather_facts: false

  tasks:
    - name: run sh ip int br
      raw: sh ip int br | ex unass
      register: sh_ip_int_br_result
```

Если запустить этот playbook, вывод не будет отличаться, так как вывод только записан в переменную, но с переменной не выполняется никаких действий. Следующий шаг - отобразить результат выполнения команды с помощью модуля `debug`.

debug

Модуль `debug` позволяет отображать информацию на стандартный поток вывода. Это может быть произвольная строка, переменная, факты об устройстве.

Для отображения сохраненных результатов выполнения команды, в playbook `2_register_vars.yml` добавлена задача с модулем `debug`:

```
---  
- name: Run show commands on routers  
hosts: cisco-routers  
gather_facts: false  
  
tasks:  
  
- name: run sh ip int br  
  raw: sh ip int br | ex unass  
  register: sh_ip_int_br_result  
  
- name: Debug registered var  
  debug: var=sh_ip_int_br_result.stdout_lines
```

Обратите внимание, что выводится не всё содержимое переменной `sh_ip_int_br_result`, а только содержимое `stdout_lines`. В `sh_ip_int_br_result.stdout_lines` находится список строк, поэтому вывод будет структурирован.

Результат запуска playbook выглядит так:

```
$ ansible-playbook 2_register_vars.yml
```

```

SSH password:

PLAY [Run show commands on routers] ****
TASK [run sh ip int br] ****
changed: [192.168.100.1]
changed: [192.168.100.2]
changed: [192.168.100.3]

TASK [Debug registered var] ****
ok: [192.168.100.1] => {
    "sh_ip_int_br_result.stdout_lines": [
        "",
        "Interface          IP-Address      OK? Method Status      Protocol",
        "Ethernet0/0        192.168.100.1  YES NVRAM   up           ",
        "Ethernet0/1        192.168.200.1  YES NVRAM   up           ",
        "Loopback0          10.1.1.1      YES manual  up           "
    ]
}
ok: [192.168.100.2] => {
    "sh_ip_int_br_result.stdout_lines": [
        "",
        "Interface          IP-Address      OK? Method Status      Protocol",
        "Ethernet0/0        192.168.100.2  YES manual  up           ",
        "Ethernet0/2        192.168.200.1  YES manual administratively down  down   ",
        "Loopback0          10.1.1.1      YES manual  up           "
    ]
}
ok: [192.168.100.3] => {
    "sh_ip_int_br_result.stdout_lines": [
        "",
        "Interface          IP-Address      OK? Method Status      Protocol",
        "Ethernet0/0        192.168.100.3  YES manual  up           ",
        "Ethernet0/2        192.168.200.1  YES manual administratively down  down   ",
        "Loopback0          10.1.1.1      YES manual  up           ",
        "Loopback10         10.255.3.3    YES manual  up           "
    ]
}

PLAY RECAP ****
192.168.100.1      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.3      : ok=2    changed=1    unreachable=0    failed=0

```

register, debug, when

С помощью ключевого слова **when** можно указать условие, при выполнении которого задача выполняется. Если условие не выполняется, то задача пропускается.

when в Ansible используется, как if в Python.

Пример playbook 3_register_debug_when.yml:

```
---  
- name: Run show commands on routers  
  hosts: cisco-routers  
  gather_facts: false  
  
  tasks:  
  
    - name: run sh ip int br  
      raw: sh ip int bri | ex unass  
      register: sh_ip_int_br_result  
  
    - name: Debug registered var  
      debug:  
        msg: "Error in command"  
        when: "'invalid' in sh_ip_int_br_result.stdout"
```

В последнем задании несколько изменений:

- модуль `debug` отображает не содержимое сохраненной переменной, а сообщение, которое указано в переменной `msg`.
- условие `when` указывает, что данная задача выполнится только при выполнении условия
 - `when: "'invalid' in sh_ip_int_br_result.stdout"` - это условие означает, что задача будет выполнена только в том случае, если в выводе `sh_ip_int_br_result.stdout` будет найдена строка `invalid` (например, когда неправильно введена команда)

Модули, которые работают с сетевым оборудованием, автоматически проверяют ошибки при выполнении команд. Тут этот пример используется для демонстрации возможностей Ansible.

Выполнение playbook:

```
$ ansible-playbook 3_register_debug_when.yml
```

SSH password:

```
PLAY [Run show commands on routers] ****
TASK [run sh ip int br] ****
changed: [192.168.100.2]
changed: [192.168.100.1]
changed: [192.168.100.3]

TASK [Debug registered var] ****
skipping: [192.168.100.1]
skipping: [192.168.100.2]
skipping: [192.168.100.3]

PLAY RECAP ****
192.168.100.1      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=1    unreachable=0    failed=0
```

Обратите внимание на сообщения skipping - это означает, что задача не выполнялась для указанных устройств. Не выполнилась она потому, что условие в when не было выполнено.

Выполнение того же playbook, но с ошибкой в команде:

```
---
- name: Run show commands on routers
  hosts: cisco-routers
  gather_facts: false

  tasks:
    - name: run sh ip int br
      raw: shh ip int bri | ex unass
      register: sh_ip_int_br_result

    - name: Debug registered var
      debug:
        msg: "Error in command"
      when: "'invalid' in sh_ip_int_br_result.stdout"
```

Теперь результат выполнения такой:

```
$ ansible-playbook 3_register_debug_when.yml
```

```
SSH password:
```

```
PLAY [Run show commands on routers] *****

TASK [run sh ip int br] *****
changed: [192.168.100.1]
changed: [192.168.100.2]
changed: [192.168.100.3]

TASK [Debug registered var] *****
ok: [192.168.100.1] => {
    "msg": "Error in command"
}
ok: [192.168.100.2] => {
    "msg": "Error in command"
}
ok: [192.168.100.3] => {
    "msg": "Error in command"
}

PLAY RECAP *****
192.168.100.1      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.3      : ok=2    changed=1    unreachable=0    failed=0
```

Так как команда была с ошибкой, сработало условие, которое описано в `when`, и задача вывела сообщение с помощью модуля `debug`.

Модули для работы с сетевым оборудованием

В предыдущих разделах для отправки команд на оборудование использовался модуль `raw`. Он универсален, и с его помощью можно отправлять команды на любое устройство.

В этом разделе рассматриваются модули, которые работают с сетевым оборудованием.

Глобально модули для работы с сетевым оборудованием можно разделить на две части:

- модули для оборудования с поддержкой API
- модули для оборудования, которое работает только через CLI

Если оборудование поддерживает API, как, например, [NXOS](#), то для него создано большое количество модулей, которые выполняют конкретные действия по настройке функционала (например, для NXOS создано более 60 модулей).

Для оборудования, которое работает только через CLI, Ansible поддерживает, как минимум, такие три типа модулей:

- `os_command` - выполняет команды `show`
- `os_facts` - собирает факты об устройствах
- `os_config` - выполняет команды конфигурации

Соответственно, для разных операционных систем будут разные модули. Например, для Cisco IOS модули будут называться:

- `ios_command`
- `ios_config`
- `ios_facts`

Аналогичные три модуля доступны для таких ОС:

- `Dellos10`
- `Dellos6`
- `Dellos9`
- `EOS`
- `IOS`
- `IOS XR`
- `JUNOS`

- SR OS
- VyOS

Полный список всех сетевых модулей, которые поддерживает Ansible, в [документации](#).

Обратите внимание, что Ansible очень активно развивается в сторону поддержки работы с сетевым оборудованием, и в следующей версии Ansible, могут быть дополнительные модули. Поэтому, если на момент чтения книги уже есть следующая версия Ansible (версия в книге 2.5), используйте её и посмотрите в документации, какие новые возможности и модули появились.

В этом разделе все рассматривается на примере модулей для работы с Cisco IOS:

- ios_command
- ios_config
- ios_facts

Аналогичные модули command, config и facts для других вендоров и ОС работают одинаково, поэтому, если разобраться, как работать с модулями для IOS, с остальными всё будет аналогично.

Кроме того, рассматривается модуль ntc-ansible, который не входит в core модули Ansible.

Особенности подключения к сетевому оборудованию

При работе с сетевым оборудованием надо указать, что должно использоваться подключение типа network_cli. Это можно указывать в инвентарном файле, файлах с переменными и т.д.

Пример настройки для сценария (play):

```
---
- name: Run show commands on routers
  hosts: cisco-routers
  connection: network_cli
```

В Ansible переменные можно указывать в разных местах, поэтому те же настройки можно указать по-другому.

Например, в инвентарном файле:

```
[cisco-routers]
192.168.100.1
192.168.100.2
192.168.100.3

[cisco-switches]
192.168.100.100

[cisco-routers:vars]
ansible_connection=network_cli
```

Или в файлах переменных, например, в group_vars/all.yml:

```
---
ansible_connection: network_cli
```

Модули, которые используются для работы с сетевым оборудованием, требуют задания нескольких параметров.

Все описание и примеры относятся к модулям ios_x и могут отличаться для других модулей.

Для каждой задачи должны быть доступны такие параметры:

- ansible_network_os - например, ios, eos
- ansible_user - имя пользователя
- ansible_password - пароль
- ansible_become - нужно ли переходить в привилегированный режим (enable, для Cisco)
- ansible_become_method - каким образом надо переходить в привилегированный режим
- ansible_become_pass - пароль для привилегированного режима

Пример указания всех параметров в group_vars/all.yml:

```
---
ansible_connection: network_cli
ansible_network_os: ios
ansible_user: cisco
ansible_password: cisco
ansible_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

Подготовка к работе с сетевыми модулями

В следующих разделах рассматривается работа с модулями `ios_command`, `ios_facts` и `ios_config`. Для того, чтобы все примеры `playbook` работали, надо создать несколько файлов (проверить, что они есть).

Инвентарный файл `myhosts`:

```
[cisco-routers]
192.168.100.1
192.168.100.2
192.168.100.3

[cisco-switches]
192.168.100.100
```

Конфигурационный файл `ansible.cfg`:

```
[defaults]

inventory = ./myhosts
```

В файле `group_vars/all.yml` надо создать параметры для подключения к оборудованию:

```
---

ansible_connection: network_cli
ansible_network_os: ios
ansible_user: cisco
ansible_password: cisco
ansible_become: yes
ansible_become_method: enable
ansible_become_pass: cisco
```

Модуль ios_command

Модуль **ios_command** отправляет команду show на устройство под управлением IOS и возвращает результат выполнения команды.

Модуль ios_command не поддерживает отправку команд в конфигурационном режиме. Для этого используется отдельный модуль - ios_config.

Модуль ios_command поддерживает такие параметры:

- **commands** - список команд, которые надо отправить на устройство
- **wait_for** (или waitfor) - список условий, на которые надо проверить вывод команды. Задача ожидает выполнения всех условий. Если после указанного количества попыток выполнения команды условия не выполняются, будет считаться, что задача выполнена неудачно.
- **match** - этот параметр используется вместе с wait_for для указания политики совпадения. Если параметр match установлен в all, должны выполниться все условия в wait_for. Если параметр равен any, достаточно, чтобы выполнилось одно из условий.
- **retries** - указывает количество попыток выполнения команды, прежде чем она будет считаться невыполненной. По умолчанию - 10 попыток.
- **interval** - интервал в секундах между повторными попытками выполнить команду. По умолчанию - 1 секунда.

Перед отправкой самой команды модуль:

- выполняет аутентификацию по SSH
- переходит в режим enable
- выполняет команду `terminal length 0`, чтобы вывод команд show отражался полностью, а не постранично
- выполняет команду `terminal width 512`

Пример использования модуля ios_command (playbook 1_ios_command.yml):

```
---  
- name: Run show commands on routers  
  hosts: cisco-routers  
  
  tasks:  
  
    - name: run sh ip int br  
      ios_command:  
        commands: show ip int br  
        register: sh_ip_int_br_result  
  
    - name: Debug registered var  
      debug: var=sh_ip_int_br_result.stdout_lines
```

Модуль `ios_command` ожидает, как минимум, такие аргументы:

- `commands` - список команд, которые нужно отправить на устройство

Обратите внимание, что параметр `register` находится на одном уровне с именем задачи и модулем, а не на уровне параметров модуля `ios_command`.

Результат выполнения playbook:

```
$ ansible-playbook 1_ios_command.yml
```

```

SSH password:

PLAY [Run show commands on routers] ****
TASK [run sh ip int br] ****
ok: [192.168.100.1]
ok: [192.168.100.2]
ok: [192.168.100.3]

TASK [Debug registered var] ****
ok: [192.168.100.1] => {
  "sh_ip_int_br_result.stdout_lines": [
    [
      "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        unassigned     YES manual  administratively down  down  ,
      "Ethernet0/3        unassigned     YES manual  up            up   ,
      "Loopback0          10.1.1.1      YES manual  up            up   "
    ]
  ]
}
ok: [192.168.100.2] => {
  "sh_ip_int_br_result.stdout_lines": [
    [
      "Interface          IP-Address      OK? Method Status      Protocol",
      "Ethernet0/0        192.168.100.2  YES manual  up           up   ",
      "Ethernet0/1        unassigned     YES unset   administratively down  down  ,
      "Ethernet0/2        192.168.200.1  YES manual  administratively down  down  ,
      "Ethernet0/3        unassigned     YES manual  up            up   ,
      "Loopback0          10.1.1.1      YES manual  up            up   "
    ]
  ]
}
ok: [192.168.100.3] => {
  "sh_ip_int_br_result.stdout_lines": [
    [
      "Interface          IP-Address      OK? Method Status      Protocol",
      "Ethernet0/0        192.168.100.3  YES manual  up           up   ",
      "Ethernet0/1        unassigned     YES unset   administratively down  down  ,
      "Ethernet0/2        192.168.200.1  YES manual  administratively down  down  ,
      "Ethernet0/3        unassigned     YES manual  up            up   ,
      "Loopback0          10.1.1.1      YES manual  up            up   ,
      "Loopback10         10.255.3.3    YES manual  up            up   "
    ]
  ]
}

PLAY RECAP ****
192.168.100.1      : ok=2    changed=0    unreachable=0    failed=0
192.168.100.2      : ok=2    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=2    changed=0    unreachable=0    failed=0

```

В отличие от использования модуля raw, playbook не указывает, что были выполнены изменения.

Выполнение нескольких команд

Модуль ios_command позволяет выполнять несколько команд.

Playbook 2_ios_command.yml выполняет несколько команд и получает их вывод:

```
---  
- name: Run show commands on routers  
  hosts: cisco-routers  
  
  tasks:  
  
    - name: run show commands  
      ios_command:  
        commands:  
          - show ip int br  
          - sh ip route  
        register: show_result  
  
    - name: Debug registered var  
      debug: var=show_result.stdout_lines
```

В первой задаче указываются две команды, поэтому синтаксис должен быть немного другим - команды должны быть указаны как список, в формате YAML.

Результат выполнения playbook (вывод сокращен):

```
$ ansible-playbook 2_ios_command.yml
```

```
SSH password:
PLAY [Run show commands on routers] *****
TASK [run show commands] *****
ok: [192.168.100.3]
ok: [192.168.100.1]
ok: [192.168.100.2]

TASK [Debug registered var] *****
ok: [192.168.100.1] => {
  "show_result.stdout_lines": [
    [
      "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        unassigned     YES manual  administratively down  down  ,
      "Ethernet0/3        unassigned     YES manual  up            up     ,
      "Loopback0          10.1.1.1      YES manual  up            up     "
    ],
    [
      "Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP",
      "D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area",
      "N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2",
      "E1 - OSPF external type 1, E2 - OSPF external type 2",
      "i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2",
      "ia - IS-IS inter area, * - candidate default, U - per-user static route",
      "o - ODR, P - periodic downloaded static route, H - NHRP, l - LISPs",
      "+ - replicated route, % - next hop override",
      "",
      "Gateway of last resort is not set",
      "",
      "10.0.0.0/32 is subnetted, 2 subnets",
      "C    10.1.1.1 is directly connected, Loopback0",
      "D    10.255.3.3 [90/409600] via 192.168.100.3, 02:04:51, Ethernet0/0",
      "192.168.100.0/24 is variably subnetted, 2 subnets, 2 masks",
      "C    192.168.100.0/24 is directly connected, Ethernet0/0",
      "L    192.168.100.1/32 is directly connected, Ethernet0/0",
      "192.168.200.0/24 is variably subnetted, 2 subnets, 2 masks",
      "C    192.168.200.0/24 is directly connected, Ethernet0/1",
      "L    192.168.200.1/32 is directly connected, Ethernet0/1"
    ]
  ]
}

```

Обе команды выполнились на всех устройствах.

Если модулю передаются несколько команд, результат выполнения команд находится в переменных `stdout` и `stdout_lines` в списке. Вывод будет в том порядке, в котором команды описаны в задаче.

За счет этого, например, можно вывести результат выполнения первой команды, указав:

```
- name: Debug registered var
  debug: var=show_result.stdout_lines[0]
```

Обработка ошибок

В модуле встроено распознание ошибок. Поэтому, если команда выполнена с ошибкой, модуль отобразит, что возникла ошибка.

Например, если сделать ошибку в команде и запустить playbook еще раз

```
$ ansible-playbook 2_ios_command.yml
```

```
PLAY [Run show commands on routers] ****
TASK [run show commands] ****
fatal: [192.168.100.2]: FAILED! => {"changed": false, "failed": true, "msg": "shw ip int br\r\n      ^\r\n% Invalid input detected at '^' marker.\r\n\r\n\r\nR2#", "rc": 1}
fatal: [192.168.100.3]: FAILED! => {"changed": false, "failed": true, "msg": "shw ip int br\r\n      ^\r\n% Invalid input detected at '^' marker.\r\n\r\n\r\nR3#", "rc": 1}
fatal: [192.168.100.1]: FAILED! => {"changed": false, "failed": true, "msg": "shw ip int br\r\n      ^\r\n% Invalid input detected at '^' marker.\r\n\r\n\r\nR1#", "rc": 1}
      to retry, use: --limit @/home/vagrant/repos/pyneng-online-jun-jul-2017/examples/15_ansible/3_network_modules/ios_command/2_ios_command.retry

PLAY RECAP ****
192.168.100.1      : ok=0    changed=0    unreachable=0    failed=1
192.168.100.2      : ok=0    changed=0    unreachable=0    failed=1
192.168.100.3      : ok=0    changed=0    unreachable=0    failed=1
```

Ansible обнаружил ошибку и возвращает сообщение ошибки. В данном случае - 'Invalid input'.

Аналогичным образом модуль обнаруживает ошибки:

- Ambiguous command
- Incomplete command

wait_for

Параметр `wait_for` (или `waitfor`) позволяет указывать список условий, на которые надо проверить вывод команды.

Пример playbook (файл `3_ios_command_wait_for.yml`):

```

---
- name: Run show commands on routers
  hosts: cisco-routers

  tasks:

    - name: run show commands
      ios_command:
        commands: ping 192.168.100.100
        wait_for:
          - result[0] contains 'Success rate is 100 percent'

```

В playbook всего одна задача, которая отправляет команду ping 192.168.100.100, и проверяет, есть ли в выводе команды фраза 'Success rate is 100 percent'.

Если в выводе команды содержится эта фраза, задача считается корректно выполненной.

Запуск playbook:

```
$ ansible-playbook 3_ios_command_wait_for.yml -v
```

```

Using /home/vagrant/repos/pyneng-online-jun-jul-2017/examples/15_ansible/3_network_modules/ios_command/ansible.cfg as config file

PLAY [Run show commands on routers] ****
TASK [run show commands] ****
ok: [192.168.100.1] => {"changed": false, "failed": false, "stdout": ["Type escape sequence to abort.\nSending 5, 100-byte ICMP Echos to 192.168.100.100, timeout is 2 seconds:\n!!!!\nSuccess rate is 100 percent (5/5), round-trip min/avg/max = 1/1/1 ms"], "stdout_lines": [[["Type escape sequence to abort.", "Sending 5, 100-byte ICMP Echos to 192.168.100.100, timeout is 2 seconds:", "!!!!", "Success rate is 100 percent (5/5), round-trip min/avg/max = 1/1/1 ms"]]]}
ok: [192.168.100.2] => {"changed": false, "failed": false, "stdout": ["Type escape sequence to abort.\nSending 5, 100-byte ICMP Echos to 192.168.100.100, timeout is 2 seconds:\n!!!!\nSuccess rate is 100 percent (5/5), round-trip min/avg/max = 1/1/1 ms"], "stdout_lines": [[["Type escape sequence to abort.", "Sending 5, 100-byte ICMP Echos to 192.168.100.100, timeout is 2 seconds:", "!!!!", "Success rate is 100 percent (5/5), round-trip min/avg/max = 1/1/1 ms"]]]}
ok: [192.168.100.3] => {"changed": false, "failed": false, "stdout": ["Type escape sequence to abort.\nSending 5, 100-byte ICMP Echos to 192.168.100.100, timeout is 2 seconds:\n!!!!\nSuccess rate is 100 percent (5/5), round-trip min/avg/max = 1/1/1 ms"], "stdout_lines": [[["Type escape sequence to abort.", "Sending 5, 100-byte ICMP Echos to 192.168.100.100, timeout is 2 seconds:", "!!!!", "Success rate is 100 percent (5/5), round-trip min/avg/max = 1/1/1 ms"]]]}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.2 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3 : ok=1    changed=0    unreachable=0    failed=0

```

Если указан IP-адрес, который не доступен, результат будет таким:

```
$ ansible-playbook 3_ios_command_wait_for.yml -v
```

```
PLAY [Run show commands on routers] ****
TASK [run show commands] ****
fatal: [192.168.100.1]: FAILED! => {"changed": false, "failed": true, "msg": "timeout trying to send command: b'ping 192.168.100.10'", "rc": 1}
fatal: [192.168.100.2]: FAILED! => {"changed": false, "failed": true, "msg": "timeout trying to send command: b'ping 192.168.100.10'", "rc": 1}
fatal: [192.168.100.3]: FAILED! => {"changed": false, "failed": true, "msg": "timeout trying to send command: b'ping 192.168.100.10'", "rc": 1}
      to retry, use: --limit @/home/vagrant/repos/pyneng-online-jun-jul-2017/examples/15_ansible/3_network_modules/ios_command/3_ios_command_wait_for.retry

PLAY RECAP ****
192.168.100.1 : ok=0    changed=0    unreachable=0    failed=1
192.168.100.2 : ok=0    changed=0    unreachable=0    failed=1
192.168.100.3 : ok=0    changed=0    unreachable=0    failed=1
```

Такой вывод из-за того, что по умолчанию таймаут для каждого пакета 2 секунды, и за время выполнения playbook команда еще не выполнена.

По умолчанию есть 10 попыток выполнить команду, при этом между каждыми двумя попытками интервал - секунда. В реальной ситуации при проверке доступности адреса лучше сделать хотя бы две попытки.

Playbook 3_ios_command_wait_for_interval.yml выполняет две попытки, на каждую попытку 12 секунд:

```
---
- name: Run show commands on routers
  hosts: cisco-routers

  tasks:
    - name: run show commands
      ios_command:
        commands: ping 192.168.100.5 timeout 1
        wait_for:
          - result[0] contains 'Success rate is 100 percent'
        retries: 2
        interval: 12
```

В этом случае вывод будет таким:

```
$ ansible-playbook 3_ios_command_wait_for_interval.yml
```

```
PLAY [Run show commands on routers] ****
TASK [run show commands] ****
fatal: [192.168.100.2]: FAILED! => {"changed": false, "failed": true, "failed_conditions": ["result[0] contains 'Success rate is 100 percent'"], "msg": "One or more conditional statements have not be satisfied"}
fatal: [192.168.100.1]: FAILED! => {"changed": false, "failed": true, "failed_conditions": ["result[0] contains 'Success rate is 100 percent'"], "msg": "One or more conditional statements have not be satisfied"}
fatal: [192.168.100.3]: FAILED! => {"changed": false, "failed": true, "failed_conditions": ["result[0] contains 'Success rate is 100 percent'"], "msg": "One or more conditional statements have not be satisfied"}
      to retry, use: --limit @/home/vagrant/repos/pyneng-online-jun-jul-2017/examples/15_ansible/3_network_modules/ios_command/3_ios_command_wait_for_interval.retry

PLAY RECAP ****
192.168.100.1 : ok=0    changed=0    unreachable=0    failed=1
192.168.100.2 : ok=0    changed=0    unreachable=0    failed=1
192.168.100.3 : ok=0    changed=0    unreachable=0    failed=1
```

Модуль ios_facts

Модуль `ios_facts` собирает информацию с устройств под управлением IOS.

Информация берется из таких команд:

- `dir`
- `show version`
- `show memory statistics`
- `show interfaces`
- `show ipv6 interface`
- `show lldp`
- `show lldp neighbors detail`
- `show running-config`

Чтобы видеть, какие команды Ansible выполняет на оборудовании, можно настроить [EEM applet](#), который будет генерировать лог сообщения о выполненных командах.

В модуле можно указывать, какие параметры собирать - можно собирать всю информацию, а можно только подмножество. По умолчанию модуль собирает всю информацию, кроме конфигурационного файла.

Какую информацию собирать, указывается в параметре `gather_subset`. Поддерживаются такие варианты (указаны также команды, которые будут выполняться на устройстве):

- **all**
- **hardware**
 - `dir`
 - `show version`
 - `show memory statistics`
- **config**
 - `show version`
 - `show running-config`
- **interfaces**
 - `dir`
 - `show version`
 - `show interfaces`
 - `show ip interface`
 - `show ipv6 interface`

- show lldp
- show lldp neighbors detail

Собрать все факты:

```
- ios_facts:  
  gather_subset: all
```

Собрать только подмножество interfaces:

```
- ios_facts:  
  gather_subset:  
    - interfaces
```

Собрать всё, кроме hardware:

```
- ios_facts:  
  gather_subset:  
    - "!hardware"
```

Ansible собирает такие факты:

- ansible_net_all_ipv4_addresses - список IPv4 адресов на устройстве
- ansible_net_all_ipv6_addresses - список IPv6 адресов на устройстве
- ansible_net_config - конфигурация (для Cisco sh run)
- ansible_net_filesystems - файловая система устройства
- ansible_net_gather_subset - какая информация собирается (hardware, default, interfaces, config)
- ansible_net_hostname - имя устройства
- ansible_net_image - имя и путь ОС
- ansible_net_interfaces - словарь со всеми интерфейсами устройства. Имена интерфейсов - ключи, а данные - параметры каждого интерфейса
- ansible_net_memfree_mb - сколько свободной памяти на устройстве
- ansible_net_memtotal_mb - сколько памяти на устройстве
- ansible_net_model - модель устройства
- ansible_net_serialnum - серийный номер
- ansible_net_version - версия IOS

Использование модуля

Пример playbook 1_ios_facts.yml с использованием модуля ios_facts (собираются все факты):

```
---
- name: Collect IOS facts
  hosts: cisco-routers

  tasks:
    - name: Facts
      ios_facts:
        gather_subset: all
```

```
$ ansible-playbook 1_ios_facts.yml
```

```
SSH password:

PLAY [Collect IOS facts] ****
TASK [Facts] ****
ok: [192.168.100.1]
ok: [192.168.100.2]
ok: [192.168.100.3]

PLAY RECAP ****
192.168.100.1      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=0    unreachable=0    failed=0
```

Для того, чтобы посмотреть, какие именно факты собираются с устройства, можно добавить флаг -v (информация сокращена):

```
$ ansible-playbook 1_ios_facts.yml -v
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
```

```
SSH password:

PLAY [Collect IOS facts] ****
TASK [Facts] ****
ok: [192.168.100.1] => {"ansible_facts": {"ansible_net_all_ipv4_addresses": ["192.168.200.1", "192.168.100.1", "10.1.1.1"], "ansible_net_all_ipv6_addresses": [], "ansible_net_config": "Building configuration...\n\nCurrent configuration : 6716 bytes\n!\nLast configuration change at 09:09:04 UTC Sun Dec 18 2016\nversion 15.2\nno service times"}}
```

После того, как Ansible собрал факты с устройства, все факты доступны как переменные в playbook, шаблонах и т.д.

Например, можно отобразить содержимое факта с помощью debug (playbook 2_ios_facts_debug.yml):

```
---  
- name: Collect IOS facts  
  hosts: 192.168.100.1  
  
  tasks:  
  
    - name: Facts  
      ios_facts:  
        gather_subset: all  
  
    - name: Show ansible_net_all_ipv4_addresses fact  
      debug: var=ansible_net_all_ipv4_addresses  
  
    - name: Show ansible_net_interfaces fact  
      debug: var=ansible_net_interfaces['Ethernet0/0']
```

Результат выполнения playbook:

```
$ ansible-playbook 2_ios_facts_debug.yml
```

```
SSH password:

PLAY [Collect IOS facts] *****

TASK [Facts] *****
ok: [192.168.100.1]

TASK [Show ansible_net_all_ipv4_addresses fact] *****
ok: [192.168.100.1] => {
    "ansible_net_all_ipv4_addresses": [
        "192.168.200.1",
        "192.168.100.1"
    ]
}

TASK [Show fact] *****
ok: [192.168.100.1] => {
    "ansible_net_interfaces['Ethernet0/0']": {
        "bandwidth": 10000,
        "description": null,
        "duplex": null,
        "ipv4": {
            "address": "192.168.100.1",
            "masklen": 24
        },
        "lineprotocol": "up",
        "macaddress": "aabb.cc00.6500",
        "mediatype": null,
        "mtu": 1500,
        "operstatus": "up",
        "type": "AmdPZ"
    }
}

PLAY RECAP *****
192.168.100.1 : ok=3    changed=0    unreachable=0    failed=0
```

Сохранение фактов

В том виде, в котором информация отображается в режиме `verbose`, довольно сложно понять какая информация собирается об устройствах. Для того, чтобы лучше понять, какая информация собирается об устройствах и в каком формате, скопируем полученную информацию в файл.

Для этого будет использоваться модуль `copy`.

Playbook `3_ios_facts.yml` собирает всю информацию об устройствах и записывает в разные файлы (создайте каталог `all_facts` перед запуском playbook или раскомментируйте задачу `Create all_facts dir`, и Ansible создаст каталог сам):

```

---
- name: Collect IOS facts
  hosts: cisco-routers

  tasks:

    - name: Facts
      ios_facts:
        gather_subset: all
        register: ios_facts_result

    #- name: Create all_facts dir
    #  file:
    #    path: ./all_facts/
    #    state: directory
    #    mode: 0755

    - name: Copy facts to files
      copy:
        content: "{{ ios_facts_result | to_nice_json }}"
        dest: "all_facts/{{inventory_hostname}}_facts.json"

```

Модуль `copy` позволяет копировать файлы с управляющего хоста (на котором установлен Ansible) на удаленный хост. Но так как в этом случае, указан параметр `connection: local`, файлы будут скопированы на локальный хост.

Чаще всего, модуль `copy` используется таким образом:

```

- copy:
  src: /srv/myfiles/foo.conf
  dest: /etc/foo.conf

```

Но в данном случае нет исходного файла, содержимое которого нужно скопировать. Вместо этого, есть содержимое переменной `ios_facts_result`, которое нужно перенести в файл `all_facts/{{inventory_hostname}}_facts.json`.

Для того, чтобы перенести содержимое переменной в файл, в модуле `copy` вместо `src` используется параметр `content`.

В строке `content: "{{ ios_facts_result | to_nice_json }}"`

- параметр `to_nice_json` - это фильтр `Jinja2`, который преобразует информацию переменной в формат, в котором удобней читать информацию
- переменная в формате `Jinja2` должна быть заключена в двойные фигурные скобки, а также указана в двойных кавычках

Так как в пути dest используются имена устройств, будут сгенерированы уникальные файлы для каждого устройства.

Результат выполнения playbook:

```
$ ansible-playbook 3_ios_facts.yml
```

```
SSH password:

PLAY [Collect IOS facts] *****

TASK [Facts] *****
ok: [192.168.100.1]
ok: [192.168.100.2]
ok: [192.168.100.3]

TASK [Copy facts to files] *****
changed: [192.168.100.3]
changed: [192.168.100.1]
changed: [192.168.100.2]

PLAY RECAP *****
192.168.100.1      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.3      : ok=2    changed=1    unreachable=0    failed=0
```

После этого в каталоге all_facts находятся такие файлы:

```
192.168.100.1_facts.json
192.168.100.2_facts.json
192.168.100.3_facts.json
```

Содержимое файла all_facts/192.168.100.1_facts.json:

```
{
  "ansible_facts": {
    "ansible_net_all_ipv4_addresses": [
      "192.168.200.1",
      "192.168.100.1",
      "10.1.1.1"
    ],
    "ansible_net_all_ipv6_addresses": [],
    "ansible_net_config": "Building configuration...\n\nCurrent configuration :
```

Сохранение информации об устройствах не только поможет разобраться, какая информация собирается, но и может быть полезным для дальнейшего использования информации. Например, можно использовать факты об устройстве в шаблоне.

При повторном выполнении playbook Ansible не будет изменять информацию в файлах, если факты об устройстве не изменились

Если информация изменилась, для соответствующего устройства будет выставлен статус changed. Таким образом, по выполнению playbook всегда понятно, когда какая-то информация изменилась.

Повторный запуск playbook (без изменений):

```
$ ansible-playbook 3_ios_facts.yml
```

```
SSH password:
```

```
PLAY [Collect IOS facts] ****
TASK [Facts] ****
ok: [192.168.100.1]
ok: [192.168.100.3]
ok: [192.168.100.2]

TASK [Copy facts to files] ****
ok: [192.168.100.2]
ok: [192.168.100.1]
ok: [192.168.100.3]

PLAY RECAP ****
192.168.100.1      : ok=2    changed=0    unreachable=0    failed=0
192.168.100.2      : ok=2    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=2    changed=0    unreachable=0    failed=0
```

Модуль ios_config

Модуль ios_config позволяет настраивать устройства под управлением IOS, а также генерировать шаблоны конфигураций или отправлять команды на основании шаблона.

Параметры модуля:

- **after** - какие действия выполнить после команд
- **before** - какие действия выполнить до команд
- **backup** - параметр, который указывает, нужно ли делать резервную копию текущей конфигурации устройства перед внесением изменений. Файл будет копироваться в каталог backup относительно каталога, в котором находится playbook
- **config** - параметр, который позволяет указать базовый файл конфигурации, с которым будут сравниваться изменения. Если он указан, модуль не будет скачивать конфигурацию с устройства.
- **defaults** - параметр указывает, нужно ли собирать всю информацию с устройства, в том числе, значения по умолчанию. Если включить этот параметр, то модуль будет собирать текущую конфигурацию с помощью команды sh run all. По умолчанию этот параметр отключен, и конфигурация проверяется командой sh run
- **lines (commands)** - список команд, которые должны быть настроены. Команды нужно указывать без сокращений и ровно в том виде, в котором они будут в конфигурации.
- **match** - параметр указывает, как именно нужно сравнивать команды
- **parents** - название секции, в которой нужно применить команды. Если команда находится внутри вложенной секции, нужно указывать весь путь. Если этот параметр не указан, то считается, что команда должна быть в глобальном режиме конфигурации
- **replace** - параметр указывает, как выполнять настройку устройства
- **save_when** - сохранять ли текущую конфигурацию в стартовую. По умолчанию конфигурация не сохраняется
- **src** - параметр указывает путь к файлу, в котором находится конфигурация или шаблон конфигурации. Взаимоисключающий параметр с lines (то есть, можно указывать или lines, или src). Заменяет модуль ios_template, который скоро будет удален.
- **diff_against, diff_ignore_lines, intended_config** - параметры, которые указывают, какие конфигурации надо сравнивать

lines (commands)

Самый простой способ использовать модуль ios_config - отправлять команды глобального конфигурационного режима с параметром lines.

Для параметра lines есть alias commands, то есть, можно вместо lines писать commands.

Пример playbook 1_ios_config_lines.yml:

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:
    - name: Config password encryption
      ios_config:
        lines:
          - service password-encryption
```

Используется переменная cli, которая указана в файле group_vars/all.yml.

Результат выполнения playbook:

```
$ ansible-playbook 1_ios_config_lines.yml
```

```
PLAY [Run cfg commands on routers] ****
TASK [Config password encryption] ****
changed: [192.168.100.1]
changed: [192.168.100.3]
changed: [192.168.100.2]

PLAY RECAP ****
192.168.100.1      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=1    unreachable=0    failed=0
```

Ansible выполняет такие команды:

- terminal length 0
- enable
- show running-config - чтобы проверить, есть ли эта команда на устройстве. Если

команда есть, задача выполняться не будет. Если команды нет, задача выполнится

- если команды, которая указана в задаче, нет в конфигурации:
 - configure terminal
 - service password-encryption
 - end

Так как модуль каждый раз проверяет конфигурацию, прежде чем применит команду, модуль идемпотентен. То есть, если ещё раз запустить playbook, изменения не будут выполнены:

```
$ ansible-playbook 1_ios_config_lines.yml
```

```
SSH password:  
  
PLAY [Run cfg commands on routers] *****  
  
TASK [Config password encryption] *****  
ok: [192.168.100.2]  
ok: [192.168.100.1]  
ok: [192.168.100.3]  
  
PLAY RECAP *****  
192.168.100.1      : ok=1    changed=0    unreachable=0    failed=0  
192.168.100.2      : ok=1    changed=0    unreachable=0    failed=0  
192.168.100.3      : ok=1    changed=0    unreachable=0    failed=0
```

Обязательно пишите команды полностью, а не сокращенно. И обращайте внимание, что для некоторых команд IOS сам добавляет параметры. Если писать команду не в том виде, в котором она реально видна в конфигурационном файле, модуль не будет идемпотентен. Он будет всё время считать, что команды нет, и вносить изменения каждый раз.

Параметр `lines` позволяет отправлять и несколько команд (playbook `1_ios_config_mult_lines.yml`):

```
---  
- name: Run cfg commands on routers  
  hosts: cisco-routers  
  
  tasks:  
  
    - name: Send config commands  
      ios_config:  
        lines:  
          - service password-encryption  
          - no ip http server  
          - no ip http secure-server  
          - no ip domain lookup
```

Результат выполнения:

```
$ ansible-playbook 1_ios_config_mult_lines.yml
```

```
SSH password:  
  
PLAY [Run cfg commands on routers] *****  
  
TASK [Send config commands] *****  
changed: [192.168.100.3]  
changed: [192.168.100.1]  
changed: [192.168.100.2]  
  
PLAY RECAP *****  
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0  
192.168.100.2 : ok=1    changed=1    unreachable=0    failed=0  
192.168.100.3 : ok=1    changed=1    unreachable=0    failed=0
```

parents

Параметр parents используется, чтобы указать, в каком подрежиме применить команды.

Например, необходимо применить такие команды:

```
line vty 0 4
login local
transport input ssh
```

В таком случае, playbook 2_ios_config_parents_basic.yml будет выглядеть так:

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
```

Запуск будет выполняться аналогично предыдущим playbook:

```
$ ansible-playbook 2_ios_config_parents_basic.yml
```

SSH password:

```
PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
changed: [192.168.100.1]
changed: [192.168.100.2]
changed: [192.168.100.3]

PLAY RECAP ****
192.168.100.1      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=1    unreachable=0    failed=0
```

Если команда находится в нескольких вложенных режимах, подрежимы указываются в списке parents.

Например, необходимо выполнить такие команды:

```
policy-map OUT_QOS
  class class-default
    shape average 1000000000 1000000
```

Тогда playbook 2_ios_config_parents_mult.yml будет выглядеть так:

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:
    - name: Config QoS policy
      ios_config:
        parents:
          - policy-map OUT_QOS
          - class class-default
        lines:
          - shape average 1000000000 1000000
```

Отображение обновлений

В этом разделе рассматриваются варианты отображения информации об обновлениях, которые выполнил модуль ios_config.

Playbook 2_ios_config_parents_basic.yml:

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
```

Для того, чтобы playbook что-то менял, нужно сначала отменить команды - либо вручную, либо изменив playbook. Например, на маршрутизаторе 192.168.100.1 вместо строки transport input ssh вручную прописать строку transport input all.

Например, можно выполнить playbook с флагом verbose:

```
$ ansible-playbook 2_ios_config_parents_basic.yml -v
```

```
Using /home/vagrant/repos/pyneng-online-jun-jul-2017/examples/15_ansible/3_network_modules/ios_config/ansible.cfg as config file

PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
ok: [192.168.100.2] => {"changed": false, "failed": false}
ok: [192.168.100.3] => {"changed": false, "failed": false}
changed: [192.168.100.1] => {"banners": {}, "changed": true, "commands": ["line vty 0 4", "transport input ssh"], "failed": false, "updates": ["line vty 0 4", "transport input ssh"]}
PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
192.168.100.2 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3 : ok=1    changed=0    unreachable=0    failed=0
```

В выводе в поле `updates` видно, какие именно команды Ansible отправил на устройство. Изменения были выполнены только на маршрутизаторе 192.168.100.1.

Обратите внимание, что команда `login local` не отправлялась, так как она настроена.

Поле `updates` в выводе есть только в том случае, когда есть изменения.

В режиме `verbose` информация видна обо всех устройствах. Но было бы удобней, чтобы информация отображалась только для тех устройств, для которых произошли изменения.

Новый playbook `3_ios_config_debug.yml` на основе `2_ios_config_parents_basic.yml`:

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
      register: cfg

    - name: Show config updates
      debug: var=cfg.updates
      when: cfg.changed
```

Изменения в playbook:

- результат работы первой задачи сохраняется в переменную `cfg`.
- в следующей задаче модуль `debug` выводит содержимое поля `updates`.
 - но так как поле `updates` в выводе есть только в том случае, когда есть изменения, ставится условие `when`, которое проверяет, были ли изменения
 - задача будет выполняться, только если на устройстве были внесены изменения.
 - вместо `when: cfg.changed` можно написать `when: cfg.changed == true`

Если запустить повторно playbook, когда изменений не было, задача `Show config updates` пропускается:

```
$ ansible-playbook 3_ios_config_debug.yml
```

```

SSH password:

PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
ok: [192.168.100.2]
ok: [192.168.100.3]
ok: [192.168.100.1]

TASK [Show config updates] ****
skipping: [192.168.100.1]
skipping: [192.168.100.2]
skipping: [192.168.100.3]

PLAY RECAP ****
192.168.100.1      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=0    unreachable=0    failed=0

```

Если внести изменения в конфигурацию маршрутизатора 192.168.100.1 (изменить transport input ssh на transport input all):

```
$ ansible-playbook 3_ios_config_debug.yml
```

```

SSH password:

PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
ok: [192.168.100.2]
changed: [192.168.100.1]
ok: [192.168.100.3]

TASK [Show config updates] ****
ok: [192.168.100.1] => {
  "cfg.updates": [
    "line vty 0 4",
    "transport input all"
  ]
}
skipping: [192.168.100.2]
skipping: [192.168.100.3]

PLAY RECAP ****
192.168.100.1      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=0    unreachable=0    failed=0

```

Теперь второе задание отображает информацию о том, какие именно изменения были внесены на маршрутизаторе.

save_when

Параметр **save_when** позволяет указать, нужно ли сохранять текущую конфигурацию в стартовую.

Доступные варианты значений:

- always - всегда сохранять конфигурацию (в этом случае флаг modified будет равен True)
- never (по умолчанию) - не сохранять конфигурацию
- modified - в этом случае конфигурация сохраняется только при наличии изменений

К сожалению, на данный момент (версия ansible 2.4) этот параметр не отрабатывает корректно, так как на устройство отправляется команда copy running-config startup-config, но при этом не отправляется подтверждение на сохранение. Из-за этого при запуске playbook с параметром save_when, выставленным в always или modified, появляется такая ошибка:

```
fatal: [192.168.100.2]: FAILED! => {"changed": false, "failed": true,
"msg": "timeout trying to send command: b'copy running-config startup-config'", "rc": 1}
```

Исправить это достаточно легко, настроив в IOS:

```
file prompt quiet
```

По умолчанию настроено `file prompt alert`

Если такая настройка не подходит на постоянной основе, можно ее настраивать до и исправлять после использования Ansible.

Еще один вариант - самостоятельно сделать сохранение, используя модуль `ios_command`.

Playbook `4_ios_config_save_when.yml`:

```

---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
        #save_when: modified
      register: cfg

    - name: Save config
      ios_command:
        commands:
          - write
      when: cfg.changed

```

Надо внести изменения на маршрутизаторе 192.168.100.1. Например, изменить строку transport input all на transport input ssh.

Выполнение playbook:

```
$ ansible-playbook 4_ios_config_save_when.yml
```

SSH password:

```

PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
ok: [192.168.100.3]
ok: [192.168.100.2]
changed: [192.168.100.1]

TASK [Save config] ****
skipping: [192.168.100.2]
skipping: [192.168.100.3]
ok: [192.168.100.1]

PLAY RECAP ****
192.168.100.1      : ok=2    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=0    unreachable=0    failed=0

```


backup

Параметр **backup** указывает, нужно ли делать резервную копию текущей конфигурации устройства перед внесением изменений. Файл будет копироваться в каталог backup относительно каталога, в котором находится playbook (если каталог не существует, он будет создан).

Playbook 5_ios_config_backup.yml:

```
---
```

```
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:
    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
      backup: yes
```

Теперь каждый раз, когда выполняется playbook (даже если не нужно вносить изменения в конфигурацию), в каталог backup будет копироваться текущая конфигурация:

```
$ ansible-playbook 5_ios_config_backup.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on routers] *****

TASK [Config line vty] *****
ok: [192.168.100.1] => {"backup_path": "/home/nata/pyneng_course/chapter15/backup/192.168.100.1_config.2016-12-10@12:35:38", "changed": false, "warnings": []}
ok: [192.168.100.3] => {"backup_path": "/home/nata/pyneng_course/chapter15/backup/192.168.100.3_config.2016-12-10@12:35:38", "changed": false, "warnings": []}
ok: [192.168.100.2] => {"backup_path": "/home/nata/pyneng_course/chapter15/backup/192.168.100.2_config.2016-12-10@12:35:38", "changed": false, "warnings": []}

PLAY RECAP *****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.2 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3 : ok=1    changed=0    unreachable=0    failed=0
```

В каталоге backup теперь находятся файлы такого вида (при каждом запуске playbook они перезаписываются):

```
192.168.100.1_config.2016-12-10@10:42:34
192.168.100.2_config.2016-12-10@10:42:34
192.168.100.3_config.2016-12-10@10:42:34
```

При работе с Python 3, может возникнуть ошибка "RuntimeError: dictionary changed size during iteration". Ее можно исправить вручную. Для этого надо запустить playbook с опцией -vvv и посмотреть где находится модуль ios_config. В выводе также будет информация о том в какой строке ошибка.

Пример ошибки:

```
File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/ansible/plugins/action/ios_config.py", line 57, in run
    for key in result.keys():
RuntimeError: dictionary changed size during iteration
```

Для исправления, надо в указанной строке сменить `for key in result.keys():` на `for key in list(result.keys()): .`

defaults

Параметр **defaults** указывает, нужно ли собирать всю информацию с устройства, в том числе и значения по умолчанию. Если включить этот параметр, модуль будет собирать текущую конфигурацию с помощью команды `sh run all`. По умолчанию этот параметр отключен, и конфигурация проверяется командой `sh run`.

Этот параметр полезен в том случае, если в настройках указывается команда, которая не видна в конфигурации. Например, такое может быть, когда указан параметр, который и так используется по умолчанию.

Если не использовать параметр `defaults` и указать команду, которая настроена по умолчанию, то при каждом запуске playbook будут вноситься изменения.

Происходит это потому, что Ansible каждый раз вначале проверяет наличие команд в соответствующем режиме. Если команд нет, то соответствующая задача выполняется.

Например, в таком playbook каждый раз будут вноситься изменения (попробуйте запустить его самостоятельно):

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:
    - name: Config interface
      ios_config:
        parents:
          - interface Ethernet0/2
      lines:
        - ip address 192.168.200.1 255.255.255.0
        - ip mtu 1500
```

Если добавить параметр `defaults: yes`, изменения уже не будут внесены, если не хватало только команды `ip mtu 1500` (playbook `6_ios_config_defaults.yml`):

```
---  
- name: Run cfg commands on routers  
  hosts: cisco-routers  
  
  tasks:  
  
    - name: Config interface  
      ios_config:  
        parents:  
          - interface Ethernet0/2  
        lines:  
          - ip address 192.168.200.1 255.255.255.0  
          - ip mtu 1500  
        defaults: yes
```

Запуск playbook:

```
$ ansible-playbook 6_ios_config_defaults.yml
```

```
SSH password:  
  
PLAY [Run cfg commands on routers] *****  
  
TASK [Config interface] *****  
ok: [192.168.100.3]  
ok: [192.168.100.1]  
ok: [192.168.100.2]  
  
PLAY RECAP *****  
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0  
192.168.100.2 : ok=1    changed=0    unreachable=0    failed=0  
192.168.100.3 : ok=1    changed=0    unreachable=0    failed=0
```

after

Параметр **after** указывает, какие команды выполнить после команд в списке `lines` (или `commands`).

Команды, которые указаны в параметре `after`:

- выполняются, только если должны быть внесены изменения.
- при этом они будут выполнены независимо от того, есть они в конфигурации или нет.

Параметр `after` очень полезен в ситуациях, когда необходимо выполнить команду, которая не сохраняется в конфигурации.

Например, команда `no shutdown` не сохраняется в конфигурации маршрутизатора, и если добавить её в список `lines`, изменения будут вноситься каждый раз при выполнении playbook.

Но, если написать команду `no shutdown` в списке `after`, то она будет применена только в том случае, если нужно вносить изменения (согласно списка `lines`).

Пример использования параметра `after` в playbook `7_ios_config_after.yml`:

```
---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:
    - name: Config interface
      ios_config:
        parents:
          - interface Ethernet0/3
      lines:
        - ip address 192.168.230.1 255.255.255.0
      after:
        - no shutdown
```

Первый запуск playbook, с внесением изменений:

```
$ ansible-playbook 7_ios_config_after.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on routers] ****
TASK [Config interface] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["interface Ethernet0/3", "ip address 192.168.230.1 255.255.255.0", "no shutdown"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Второй запуск playbook (изменений нет, поэтому команда no shutdown не выполняется):

```
$ ansible-playbook 7_ios_config_after.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on routers] ****
TASK [Config interface] ****
ok: [192.168.100.1] => {"changed": false, "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0
```

Рассмотрим ещё один пример использования after.

С помощью after можно сохранять конфигурацию устройства (playbook 7_ios_config_after_save.yml):

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
      after:
        - end
        - write
```

Результат выполнения playbook (изменения только на маршрутизаторе 192.168.100.1):

```
$ ansible-playbook 7_ios_config_after_save.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
ok: [192.168.100.2] => {"changed": false, "warnings": []}
ok: [192.168.100.3] => {"changed": false, "warnings": []}
changed: [192.168.100.1] => {"changed": true, "updates": ["line vty 0 4", "transpo
rt input ssh", "end", "write"], "warnings": []}

PLAY RECAP ****
192.168.100.1      : ok=1    changed=1    unreachable=0    failed=0
192.168.100.2      : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3      : ok=1    changed=0    unreachable=0    failed=0
```

before

Параметр **before** указывает, какие действия выполнить до команд в списке lines.

Команды, которые указаны в параметре before:

- выполняются, только если должны быть внесены изменения.
- при этом они будут выполнены независимо от того, есть они в конфигурации или нет.

Параметр before полезен в ситуациях, когда какие-то действия необходимо выполнить перед выполнением команд в списке lines.

При этом, как и after, параметр before не влияет на то, какие команды сравниваются с конфигурацией. То есть, по-прежнему сравниваются только команды в списке lines.

Playbook 8_ios_config_before.yml:

```
---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:
    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
```

В playbook 8_ios_config_before.yml ACL IN_to_OUT сначала удаляется с помощью параметра before, а затем создается заново.

Таким образом, в ACL всегда находятся только те строки, которые заданы в списке lines.

Запуск playbook с изменениями:

```
$ ansible-playbook 8_ios_config_before.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["no ip access-list extended IN_to_OUT", "ip access-list extended IN_to_OUT", "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"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Запуск playbook без изменений (команда в списке before не выполняется):

```
$ ansible-playbook 8_ios_config_before.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
ok: [192.168.100.1] => {"changed": false, "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0
```

match

Параметр **match** указывает, как именно нужно сравнивать команды (что считается изменением):

- **line** - команды проверяются построчно. Этот режим используется по умолчанию
- **strict** - должны совпасть не только сами команды, но и их положение относительно друг друга
- **exact** - команды должны в точности совпадать с конфигурацией, и не должно быть никаких лишних строк
- **none** - модуль не будет сравнивать команды с текущей конфигурацией

match: line

Режим `match: line` используется по умолчанию.

В этом режиме модуль проверяет только наличие строк, перечисленных в списке `lines` в соответствующем режиме. При этом не проверяется порядок строк.

На маршрутизаторе 192.168.100.1 настроен такой ACL:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
 permit tcp 10.0.1.0 0.0.0.255 any eq 22
```

Пример использования playbook `9_ios_config_match_line.yml` в режиме `line`:

```
---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:
    - name: Config ACL
      ios_config:
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
```

Результат выполнения playbook:

```
$ ansible-playbook 9_ios_config_match_line.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["ip access-list extended IN_to_OUT", "permit tcp 10.0.1.0 0.0.0.255 any eq www", "permit icmp any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Обратите внимание, что в списке `updates` только две из трёх строк ACL. Так как в режиме `lines` модуль сравнивает команды независимо друг от друга, он обнаружил, что не хватает только двух команд из трех.

В итоге конфигурация на маршрутизаторе выглядит так:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
 permit tcp 10.0.1.0 0.0.0.255 any eq 22
 permit tcp 10.0.1.0 0.0.0.255 any eq www
 permit icmp any any
```

То есть, порядок команд поменялся. И хотя в этом случае это не важно, иногда это может привести совсем не к тем результатам, которые ожидались.

Если повторно запустить playbook при такой конфигурации, он не будет выполнять изменения, так как все строки были найдены.

match: exact

Пример, в котором порядок команд важен.

ACL на маршрутизаторе:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
 permit tcp 10.0.1.0 0.0.0.255 any eq 22
 permit tcp 10.0.1.0 0.0.0.255 any eq www
 deny   ip any any
```

Playbook `9_ios_config_match_exact.yml` (будет постепенно дополняться):

```

---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:

    - name: Config ACL
      ios_config:
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
          - deny ip any any

```

Если запустить playbook, результат будет таким:

```
$ ansible-playbook 9_ios_config_match_exact.yml -v
```

```

Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["ip access-list extended IN_to_OUT", "permit icmp any any", "deny ip any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0

```

Теперь ACL выглядит так:

```

R1#sh run | s access
ip access-list extended IN_to_OUT
  permit tcp 10.0.1.0 0.0.0.255 any eq 22
  permit tcp 10.0.1.0 0.0.0.255 any eq www
  deny   ip any any
  permit icmp any any

```

Конечно же, в таком случае последнее правило никогда не сработает.

Можно добавить к этому playbook параметр before и сначала удалить ACL, а затем применять команды:

```

---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:

    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
          - deny   ip any any

```

Если применить playbook к последнему состоянию маршрутизатора, то изменений не будет никаких, так как все строки уже есть.

Попробуем начать с такого состояния ACL:

```

R1#sh run | s access
ip access-list extended IN_to_OUT
permit tcp 10.0.1.0 0.0.0.255 any eq 22
permit tcp 10.0.1.0 0.0.0.255 any eq www
deny   ip any any

```

Результат будет таким:

```
$ ansible-playbook 9_ios_config_match_exact.yml -v
```

```

Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["no ip access-list extended IN_to_OUT", "ip access-list extended IN_to_OUT", "permit icmp any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0

```

И, соответственно, на маршрутизаторе:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
 permit icmp any any
```

Теперь в ACL осталась только одна строка:

- Модуль проверил, каких команд не хватает в ACL (так как режим по умолчанию `match: line`),
- обнаружил, что не хватает команды `permit icmp any any`, и добавил её

Но, так как в playbook ACL сначала удаляется, а затем применяется список команд `lines`, получилось, что в итоге в ACL одна строка.

Поможет в такой ситуации вариант `match: exact`:

```
---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:
    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
          - deny ip any any
      match: exact
```

Применение playbook `9_ios_config_match_exact.yml` к текущему состоянию маршрутизатора (в ACL одна строка):

```
$ ansible-playbook 9_ios_config_match_exact.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["no ip access-list exten
ded IN_to_OUT", "ip access-list extended IN_to_OUT", "permit tcp 10.0.1.0 0.0.0.25
5 any eq www", "permit tcp 10.0.1.0 0.0.0.255 any eq 22", "permit icmp any any", "d
eny ip any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Теперь результат такой:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
 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
 deny ip any any
```

То есть, теперь ACL выглядит точно так же, как и строки в списке `lines`, и в том же порядке.

И для того, чтобы окончательно разобраться с параметром `match: exact`, ещё один пример.

Закомментируем в playbook строки с удалением ACL:

```

---  

- name: Run cfg commands on router  

  hosts: 192.168.100.1  

  tasks:  

  - name: Config ACL  

    ios_config:  

      #before:  

      # - no ip access-list extended IN_to_OUT  

      parents:  

      - ip access-list extended IN_to_OUT  

      lines:  

      - 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  

      - deny ip any any  

      match: exact

```

В начало ACL добавлена строка:

```

ip access-list extended IN_to_OUT
permit udp any any
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
deny ip any any

```

То есть, последние 4 строки выглядят так, как нужно, и в том порядке, котором нужно. Но, при этом, есть лишняя строка. Для варианта match: exact - это уже несовпадение.

В таком варианте, playbook будет выполняться каждый раз и пытаться применить все команды из списка lines, что не будет влиять на содержимое ACL:

```
$ ansible-playbook 9_ios_config_match_exact.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["ip access-list extended IN_to_OUT", "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", "deny ip any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Это значит, что при использовании `match:exact` важно, чтобы был какой-то способ удалить конфигурацию, если она не соответствует тому, что должно быть (или чтобы команды перезаписывались). Иначе эта задача будет выполняться каждый раз при запуске playbook.

match: strict

Вариант `match: strict` не требует, чтобы объект был в точности как указано в задаче, но команды, которые указаны в списке `lines`, должны быть в том же порядке.

Если указан список `parents`, команды в списке `lines` должны идти сразу за командами `parents`.

На маршрутизаторе такой ACL:

```
ip access-list extended IN_to_OUT
 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
 deny ip any any
```

Playbook 9_ios_config_match_strict.yml:

```

---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:

    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
        match: strict

```

Выполнение playbook:

```
$ ansible-playbook 9_ios_config_match_strict.yml -v
```

```

Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
ok: [192.168.100.1] => {"changed": false, "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0

```

Так как изменений не было, ACL остался таким же.

В такой же ситуации, при использовании `match: exact`, было бы обнаружено изменение, и ACL бы состоял только из строк в списке `lines`.

match: none

Использование `match: none` отключает идемпотентность задачи: каждый раз при выполнении playbook будут отправляться команды, которые указаны в задаче.

Пример playbook `9_ios_config_match_none.yml`:

```

---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:

    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
        match: none

```

Каждый раз при запуске playbook результат будет таким:

```
$ ansible-playbook 9_ios_config_match_none.yml -v
```

```

Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["no ip access-list extended IN_to_OUT", "ip access-list extended IN_to_OUT", "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"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0

```

Использование `match: none` подходит в тех случаях, когда, независимо от текущей конфигурации, нужно отправить все команды.

replace

Параметр replace указывает, как именно нужно заменять конфигурацию:

- **line** - в этом режиме отправляются только те команды, которых нет в конфигурации. Этот режим используется по умолчанию
- **block** - в этом режиме отправляются все команды, если хотя бы одной команды нет

replace: line

Режим `replace: line` - это режим работы по умолчанию. В этом режиме, если были обнаружены изменения, отправляются только недостающие строки.

Например, на маршрутизаторе такой ACL:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
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
```

Попробуем запустить такой playbook `10_ios_config_replace_line.yml`:

```
---
- name: Run cfg commands on router
  hosts: 192.168.100.1
  tasks:
    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
          - deny ip any any
```

Выполнение playbook:

```
$ ansible-playbook 10_ios_config_replace_line.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["no ip access-list extended IN_to_OUT", "ip access-list extended IN_to_OUT", "deny ip any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

После этого на маршрутизаторе такой ACL:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
  deny ip any any
```

В данном случае модуль проверил, каких команд не хватает в ACL (так как режим по умолчанию `match: line`), обнаружил, что не хватает команды `deny ip any any`, и добавил её. Но, так как ACL сначала удаляется, а затем применяется список команд `lines`, получилось, что у нас теперь ACL с одной строкой.

В таких ситуациях подходит режим `replace: block`.

replace: block

В режиме `replace: block` отправляются все команды из списка `lines` (и `parents`), если на устройстве нет хотя бы одной из этих команд.

Повторим предыдущий пример.

ACL на маршрутизаторе:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
  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
```

Playbook 10_ios_config_replace_block.yml:

```

---
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:

    - name: Config ACL
      ios_config:
        before:
          - no ip access-list extended IN_to_OUT
        parents:
          - ip access-list extended IN_to_OUT
        lines:
          - 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
          - deny ip any any
      replace: block

```

Выполнение playbook:

```
$ ansible-playbook 10_ios_config_replace_block.yml -v
```

```

Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"changed": true, "updates": ["no ip access-list extended IN_to_OUT", "ip access-list extended IN_to_OUT", "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", "deny ip any any"], "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0

```

В результате на маршрутизаторе такой ACL:

```

R1#sh run | s access
ip access-list extended IN_to_OUT
  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
  deny ip any any

```


src

Параметр **src** позволяет указывать путь к файлу конфигурации или шаблону конфигурации, которую нужно загрузить на устройство.

Этот параметр взаимоисключающий с **lines** (то есть, можно указывать или **lines**, или **src**). Он заменяет модуль **ios_template**, который скоро будет удален.

Конфигурация

Пример playbook `11_ios_config_src.yml`:

```
---
```

```
- name: Run cfg commands on router
  hosts: 192.168.100.1

  tasks:
    - name: Config ACL
      ios_config:
        src: templates/acl_cfg.txt
```

В файле `templates/acl_cfg.txt` находится такая конфигурация:

```
ip access-list extended IN_to_OUT
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
deny ip any any
```

Удаляем на маршрутизаторе этот ACL, если он остался с прошлых разделов, и запускаем playbook:

```
$ ansible-playbook 11_ios_config_src.yml -v
```

```
PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
changed: [192.168.100.1] => {"banners": {}, "changed": true, "commands": ["ip access-list extended IN_to_OUT", "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", "deny ip any any"], "failed": false, "updates": [{"ip access-list extended IN_to_OUT", "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", "deny ip any any"}]}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Теперь на маршрутизаторе настроен ACL:

```
R1#sh run | s access
ip access-list extended IN_to_OUT
 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
 deny ip any any
```

Если запустить playbook ещё раз, то никаких изменений не будет, так как этот параметр также идемпотентен:

```
$ ansible-playbook 11_ios_config_src.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config ACL] ****
ok: [192.168.100.1] => {"changed": false, "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0
```

Шаблон Jinja2

В параметре src можно указывать шаблон Jinja2.

Пример шаблона (файл templates/ospf.j2):

```

router ospf 1
  router-id {{ mgmnt_ip }}
  ispf
  auto-cost reference-bandwidth 10000
{% for ip in ospf_ints %}
  network {{ ip }} 0.0.0.0 area 0
{% endfor %}

```

В шаблоне используются две переменные:

- mgmnt_ip - IP-адрес, который будет использоваться как router-id
- ospf_ints - список IP-адресов интерфейсов, на которых нужно включить OSPF

Для настройки OSPF на трёх маршрутизаторах нужно иметь возможность использовать разные значения этих переменных для разных устройств. Для таких задач используются файлы с переменными в каталоге host_vars.

В каталоге host_vars нужно создать такие файлы (если они ещё не созданы):

Файл host_vars/192.168.100.1:

```

---
hostname: london_r1
mgmnt_loopback: 100
mgmnt_ip: 10.0.0.1
ospf_ints:
  - 192.168.100.1
  - 10.0.0.1
  - 10.255.1.1

```

Файл host_vars/192.168.100.2:

```

---
hostname: london_r2
mgmnt_loopback: 100
mgmnt_ip: 10.0.0.2
ospf_ints:
  - 192.168.100.2
  - 10.0.0.2
  - 10.255.2.2

```

Файл host_vars/192.168.100.3:

```
---  
  
hostname: london_r3  
mgmnt_loopback: 100  
mgmnt_ip: 10.0.0.3  
ospf_ints:  
  - 192.168.100.3  
  - 10.0.0.3  
  - 10.255.3.3
```

Теперь можно создавать playbook 11_ios_config_src_jinja.yml:

```
---  
  
- name: Run cfg commands on router  
  hosts: cisco-routers  
  
  tasks:  
  
    - name: Config OSPF  
      ios_config:  
        src: templates/ospf.j2
```

Так как Ansible сам найдет переменные в каталоге host_vars, их не нужно указывать.
Можно сразу запускать playbook:

```
$ ansible-playbook 11_ios_config_src_jinja.yml -v
```

```

PLAY [Run cfg commands on router] ****
TASK [Config OSPF] ****
changed: [192.168.100.1] => {"banners": {}, "changed": true, "commands": ["router ospf 1", "router-id 10.0.0.1", "ispf", "auto-cost reference-bandwidth 10000", "network 192.168.100.1 0.0.0.0 area 0", "network 10.0.0.1 0.0.0.0 area 0", "network 10.255.1.1 0.0.0.0 area 0"], "failed": false, "updates": ["router ospf 1", "router-id 10.0.0.1", "ispf", "auto-cost reference-bandwidth 10000", "network 192.168.100.1 0.0.0.0 area 0", "network 10.0.0.1 0.0.0.0 area 0", "network 10.255.1.1 0.0.0.0 area 0"]}
changed: [192.168.100.2] => {"banners": {}, "changed": true, "commands": ["router ospf 1", "router-id 10.0.0.2", "ispf", "auto-cost reference-bandwidth 10000", "network 192.168.100.2 0.0.0.0 area 0", "network 10.0.0.2 0.0.0.0 area 0", "network 10.255.2.2 0.0.0.0 area 0"], "failed": false, "updates": ["router ospf 1", "router-id 10.0.0.2", "ispf", "auto-cost reference-bandwidth 10000", "network 192.168.100.2 0.0.0.0 area 0", "network 10.0.0.2 0.0.0.0 area 0", "network 10.255.2.2 0.0.0.0 area 0"]}
changed: [192.168.100.3] => {"banners": {}, "changed": true, "commands": ["router ospf 1", "router-id 10.0.0.3", "ispf", "auto-cost reference-bandwidth 10000", "network 192.168.100.3 0.0.0.0 area 0", "network 10.0.0.3 0.0.0.0 area 0", "network 10.255.3.3 0.0.0.0 area 0"], "failed": false, "updates": ["router ospf 1", "router-id 10.0.0.3", "ispf", "auto-cost reference-bandwidth 10000", "network 192.168.100.3 0.0.0.0 area 0", "network 10.0.0.3 0.0.0.0 area 0", "network 10.255.3.3 0.0.0.0 area 0"]}
PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
192.168.100.2 : ok=1    changed=1    unreachable=0    failed=0
192.168.100.3 : ok=1    changed=1    unreachable=0    failed=0

```

Теперь на всех маршрутизаторах настроен OSPF:

```

R1#sh run | s ospf
router ospf 1
router-id 10.0.0.1
ispf
auto-cost reference-bandwidth 10000
network 10.0.0.1 0.0.0.0 area 0
network 10.255.1.1 0.0.0.0 area 0
network 192.168.100.1 0.0.0.0 area 0

R2#sh run | s ospf
router ospf 1
router-id 10.0.0.2
ispf
auto-cost reference-bandwidth 10000
network 10.0.0.2 0.0.0.0 area 0
network 10.255.2.2 0.0.0.0 area 0
network 192.168.100.2 0.0.0.0 area 0

router ospf 1
router-id 10.0.0.3
ispf
auto-cost reference-bandwidth 10000
network 10.0.0.3 0.0.0.0 area 0
network 10.255.3.3 0.0.0.0 area 0
network 192.168.100.3 0.0.0.0 area 0

```

Если запустить playbook ещё раз, то никаких изменений не будет:

```
$ ansible-playbook 11_ios_config_src_jinja.yml -v
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config OSPF] ****
ok: [192.168.100.3] => {"changed": false, "warnings": []}
ok: [192.168.100.2] => {"changed": false, "warnings": []}
ok: [192.168.100.1] => {"changed": false, "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.2 : ok=1    changed=0    unreachable=0    failed=0
192.168.100.3 : ok=1    changed=0    unreachable=0    failed=0
```

Совмещение с другими параметрами

Параметр **src** совместим с такими параметрами:

- backup
- config
- defaults
- save (но у самого save в Ansible 2.2 проблемы с работой)

ntc-ansible

ntc-ansible - это модуль для работы с сетевым оборудованием, который не только выполняет команды на оборудовании, но и обрабатывает вывод команд и преобразует с помощью [TextFSM](#).

Этот модуль не входит в число core модулей Ansible, поэтому его нужно установить.

Но прежде нужно указать Ansible, где искать сторонние модули. Указывается путь в файле ansible.cfg:

```
[defaults]

inventory = ./myhosts

remote_user = cisco
ask_pass = True

library = ./library
```

После этого нужно клонировать репозиторий ntc-ansible, находясь в каталоге library:

```
[~/pyneng_course/chapter15/library]
$ git clone https://github.com/networktocode/ntc-ansible --recursive
Cloning into 'ntc-ansible'...
remote: Counting objects: 2063, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 2063 (delta 1), reused 0 (delta 0), pack-reused 2058
Receiving objects: 100% (2063/2063), 332.15 KiB | 334.00 KiB/s, done.
Resolving deltas: 100% (1157/1157), done.
Checking connectivity... done.
Submodule 'ntc-templates' (https://github.com/networktocode/ntc-templates) registered
for path 'ntc-templates'
Cloning into 'ntc-templates'...
remote: Counting objects: 902, done.
remote: Compressing objects: 100% (34/34), done.
remote: Total 902 (delta 16), reused 0 (delta 0), pack-reused 868
Receiving objects: 100% (902/902), 161.11 KiB | 0 bytes/s, done.
Resolving deltas: 100% (362/362), done.
Checking connectivity... done.
Submodule path 'ntc-templates': checked out '89c57342b47c9990f0708226fb3f268c6b8c1549'
```

А затем установить зависимости модуля:

```
pip install ntc-ansible
```

При установке зависимостей может появиться ошибка:

```
No matching distribution found for textfsm==1.0.1 (from pyntc->ntc-ansible)
```

Ее можно игнорировать, если модуль textfsm установлен.

Если при установке возникнут другие проблемы, посмотрите другие варианты установки в [репозитории проекта](#).

Так как в текущей версии Ansible уже есть модули, которые работают с сетевым оборудованием и позволяют выполнять команды, из всех возможностей ntc-ansible наиболее полезной будет отправка команд show и получение структурированного вывода. За это отвечает модуль ntc_show_command.

ntc_show_command

Модуль использует netmiko для подключения к оборудованию (netmiko должен быть установлен) и, после выполнения команды, преобразует вывод команды show с помощью TextFSM в структурированный вывод (список словарей).

Преобразование будет выполняться в том случае, если в файле index была найдена команда, и для команды был найден шаблон.

Как и с предыдущими сетевыми модулями, в ntc-ansible нужно указывать ряд параметров для подключения:

- **connection** - тут возможны два варианта: ssh (подключение netmiko) или offline (чтение из файла для тестовых целей)
- **platform** - платформа, которая существует в index файле (library/ntc-ansible/ntc-templates/templates/index)
- **command** - команда, которую нужно выполнить на устройстве
- **host** - IP-адрес или имя устройства
- **username** - имя пользователя
- **password** - пароль
- **template_dir** - путь к каталогу, в котором находятся шаблоны (в текущем варианте установки они находятся в каталоге library/ntc-ansible/ntc-templates/templates)

Пример playbook 1_ntc_ansible.yml:

```
---  
- name: Run show commands on router  
  hosts: 192.168.100.1  
  gather_facts: false  
  connection: local  
  
  tasks:  
  
    - name: Run sh ip int br  
      ntc_show_command:  
        connection: ssh  
        platform: "cisco_ios"  
        command: "sh ip int br"  
        host: "{{ inventory_hostname }}"  
        username: "cisco"  
        password: "cisco"  
        template_dir: "library/ntc-ansible/ntc-templates/templates"  
      register: result  
  
    - debug: var=result
```

Результат выполнения playbook:

```
$ ansible-playbook 1_ntc-ansible.yml
```

```

SSH password:

PLAY [Run show commands on router] ****
TASK [Run sh ip int br] ****
ok: [192.168.100.1]

TASK [debug] ****
ok: [192.168.100.1] => {
    "result": [
        {
            "changed": false,
            "response": [
                {
                    "intf": "Ethernet0/0",
                    "ipaddr": "192.168.100.1",
                    "proto": "up",
                    "status": "up"
                },
                {
                    "intf": "Ethernet0/1",
                    "ipaddr": "192.168.200.1",
                    "proto": "up",
                    "status": "up"
                },
                {
                    "intf": "Ethernet0/2",
                    "ipaddr": "unassigned",
                    "proto": "down",
                    "status": "administratively down"
                },
                {
                    "intf": "Ethernet0/3",
                    "ipaddr": "unassigned",
                    "proto": "up",
                    "status": "up"
                },
                {
                    "intf": "Loopback0",
                    "ipaddr": "10.1.1.1",
                    "proto": "up",
                    "status": "up"
                }
            ],
            "response_list": []
        }
    ]
}

PLAY RECAP ****
192.168.100.1 : ok=2      changed=0      unreachable=0      failed=0

```

В переменной `response` находится структурированный вывод в виде списка словарей. Ключи в словарях получены на основании переменных, которые описаны в шаблоне `library/ntc-ansible/ntc-templates/templates/cisco_ios_show_ip_int_brief.template` (единственное отличие - регистр):

```

Value INTF (\S+)
Value IPADDR (\S+)
Value STATUS (up|down|administratively down)
Value PROTO (up|down)

Start
^${INTF}\s+${IPADDR}\s+\w+\s+\w+\s+${STATUS}\s+${PROTO} -> Record

```

Для того, чтобы получить вывод про первый интерфейс, можно поменять вывод модуля debug таким образом:

```
- debug: var=result.response[0]
```

Сохранение результатов выполнения команды

Для того, чтобы сохранить вывод, можно использовать тот же прием, который использовался для модуля ios_facts.

Пример playbook 2_ntc_ansible_save.yml с сохранением результатов команды:

```

---
- name: Run show commands on routers
  hosts: cisco-routers
  gather_facts: false
  connection: local

  tasks:

    - name: Run sh ip int br
      ntc_show_command:
        connection: ssh
        platform: "cisco_ios"
        command: "sh ip int br"
        host: "{{ inventory_hostname }}"
        username: "cisco"
        password: "cisco"
        template_dir: "library/ntc-ansible/ntc-templates/templates"
      register: result

    - name: Copy facts to files
      copy:
        content: "{{ result.response | to_nice_json }}"
        dest: "all_facts/{{inventory_hostname}}_sh_ip_int_br.json"

```

Результат выполнения:

```
$ ansible-playbook 2_ntc-ansible_save.yml
```

```
SSH password:
```

```
PLAY [Run show commands on routers] *****  
TASK [Run sh ip int br] *****  
ok: [192.168.100.3]  
ok: [192.168.100.1]  
ok: [192.168.100.2]  
  
TASK [Copy facts to files] *****  
changed: [192.168.100.2]  
changed: [192.168.100.1]  
changed: [192.168.100.3]  
  
PLAY RECAP *****  
192.168.100.1 : ok=2    changed=1    unreachable=0    failed=0  
192.168.100.2 : ok=2    changed=1    unreachable=0    failed=0  
192.168.100.3 : ok=2    changed=1    unreachable=0    failed=0
```

В результате, в каталоге all_facts появляются соответствующие файлы для каждого маршрутизатора. Пример файла all_facts/192.168.100.1_sh_ip_int_br.json:

```
[  
  {  
    "intf": "Ethernet0/0",  
    "ipaddr": "192.168.100.1",  
    "proto": "up",  
    "status": "up"  
,  
  {  
    "intf": "Ethernet0/1",  
    "ipaddr": "192.168.200.1",  
    "proto": "up",  
    "status": "up"  
,  
  {  
    "intf": "Ethernet0/2",  
    "ipaddr": "unassigned",  
    "proto": "down",  
    "status": "administratively down"  
,  
  {  
    "intf": "Ethernet0/3",  
    "ipaddr": "unassigned",  
    "proto": "up",  
    "status": "up"  
,  
  {  
    "intf": "Loopback0",  
    "ipaddr": "10.1.1.1",  
    "proto": "up",  
    "status": "up"  
  }  
]
```

Шаблоны Jinja2

Для Cisco IOS в ntc-ansible есть такие шаблоны:

```
cisco_ios_dir.template
cisco_ios_show_access-list.template
cisco_ios_show_aliases.template
cisco_ios_show_archive.template
cisco_ios_show_capability_feature_routing.template
cisco_ios_show_cdp_neighbors_detail.template
cisco_ios_show_cdp_neighbors.template
cisco_ios_show_clock.template
cisco_ios_show_interfaces_status.template
cisco_ios_show_interfaces.template
cisco_ios_show_interface_transceiver.template
cisco_ios_show_inventory.template
cisco_ios_show_ip_arp.template
cisco_ios_show_ip_bgp_summary.template
cisco_ios_show_ip_bgp.template
cisco_ios_show_ip_int_brief.template
cisco_ios_show_ip_ospf_neighbor.template
cisco_ios_show_ip_route.template
cisco_ios_show_lldp_neighbors.template
cisco_ios_show_mac-address-table.template
cisco_ios_show_processes_cpu.template
cisco_ios_show_snmp_community.template
cisco_ios_show_spanning-tree.template
cisco_ios_show_standby_brief.template
cisco_ios_show_version.template
cisco_ios_show_vlan.template
cisco_ios_show_vtp_status.template
```

Список всех шаблонов можно посмотреть локально, если ntc-ansible установлен:

```
ls -ls library/ntc-ansible/ntc-templates/templates/
```

Или в [репозитории проекта](#).

Используя TextFSM, можно самостоятельно создавать дополнительные шаблоны.

И для того, чтобы ntc-ansible их использовал автоматически, добавить их в файл index (library/ntc-ansible/ntc-templates/templates/index):

```
# First line is the header fields for columns and is mandatory.
# Regular expressions are supported in all fields except the first.
# Last field supports variable length command completion.
# abc[[xyz]] is expanded to abc(x(y(z)?))?, regexp inside [[]] is not supported
#
Template, Hostname, Platform, Command
cisco_asa_dir.template, .*, cisco_asa, dir
cisco_ios_show_archive.template, .*, cisco_ios, sh[[ow]] arc[[hive]]
cisco_ios_show_capability_feature_routing.template, .*, cisco_ios, sh[[ow]] cap[[ability]] f[[eature]] r[[outing]]
cisco_ios_show_aliases.template, .*, cisco_ios, sh[[ow]] alia[[ses]]
...
```

Синтаксис шаблонов и файла index описаны в разделе [TextFSM](#).

Подробнее об Ansible

Мы рассмотрели основные аспекты Ansible, которые нужны для работы с сетевым оборудованием. Их достаточно, чтобы начать работать с Ansible, но, скорее всего, в процессе работы Вам понадобится больше информации.

Например, как сделать так, чтобы не нужно было повторять одни и те же задачи или сценарии снова и снова, или как организовывать более сложные playbook.

А, возможно, на каком-то этапе понадобится написать свой модуль.

Всё это Ansible позволяет сделать, но это выходит за рамки этого курса. Эта информация вынесена в отдельный курс [Ansible для сетевых инженеров](#). Основы, которые рассматриваются тут, в том курсе повторяются, поэтому, если Вы прочитали весь раздел Ansible в этом курсе, можете начать сразу с четвертого раздела [Playbook](#).

Если какие-то темы не рассмотрены в курсе "Ansible для сетевых инженеров", не забывайте, что у Ansible отличная [документация](#).

Дополнительные материалы

Ansible без привязки к сетевому оборудованию

- У Ansible очень хорошая документация
- Очень хорошая серия видео с транскриптом и хорошими ссылками
- Примеры использования Ansible
- Примеры Playbook с демонстрацией различных возможностей

Ansible for network devices

Документация:

- Networking Support
- Network Modules
- Network Debug and Troubleshooting Guide
- ios_command
- ios_facts
- ios_config

Отличные видео от Ansible:

- AUTOMATING YOUR NETWORK. Репозиторий с примерами из вебинара

Проекты, которые используют TextFSM:

- Модуль ntc-ansible

Шаблоны TextFSM (из модуля ntc-ansible):

- ntc-templates

Статьи:

Обращайте внимание на время написания статьи. В Ansible существенно изменились модули для работы с сетевым оборудованием. И в статьях могут быть ещё старые примеры.

Network Config Templating using Ansible (Kirk Byers):

- <https://pynet.twb-tech.com/blog/ansible/ansible-cfg-template.html>
- <https://pynet.twb-tech.com/blog/ansible/ansible-cfg-template-p2.html>
- <https://pynet.twb-tech.com/blog/ansible/ansible-cfg-template-p3.html>

Очень хорошая серия статей. Постепенно повышается уровень сложности:

- <http://networkop.github.io/blog/2015/06/24/ansible-intro/>
- <http://networkop.github.io/blog/2015/07/03/parser-modules/>
- <http://networkop.github.io/blog/2015/07/10/test-verification/>
- <http://networkop.github.io/blog/2015/07/17/tdd-quickstart/>
- <http://networkop.github.io/blog/2015/08/14/automating-legacy-networks/>
- <http://networkop.github.io/blog/2015/08/26/automating-network-build-p1/>
- <http://networkop.github.io/blog/2015/09/03/automating-bgp-config/>
- <http://networkop.github.io/blog/2015/11/13/automating-flexvpn-config/>
- <http://jedelman.com/home/ansible-for-networking/>
- <http://jedelman.com/home/network-automation-with-ansible-dynamically-configuring-interface-descriptions/>
- <http://www.packetgeek.net/2015/08/using-ansible-to-push-cisco-ios-configuration/>

Задания

Все задания и вспомогательные файлы можно скачать в [репозитории](#).

Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.

Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.

Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 24.1

Создайте playbook task_24_1.yml, который выполняет такие задачи:

- подключается к маршрутизаторам и выполняет команду sh arp
 - результат записывает в переменную sh_arp_output
- отображает содержимое переменной sh_arp_output

Проверьте работу playbook на маршрутизаторах.

Задание 24.1a

Создайте playbook task_24_1a.yml, который выполняет такие задачи:

- подключается к маршрутизаторам и выполняет команду sh arp
 - результат записывает в переменную sh_arp_output
- отображает результат выполнения команды, в виде списка строк, где каждая строка это одна строка вывода команды

Проверьте работу playbook на маршрутизаторах.

Задание 24.1b

Создайте playbook task_24_1b.yml, который выполняет такие задачи:

- подключается к маршрутизаторам и выполняет команды sh arp и sh ip int br

- обе команды должны выполняться в одной задаче
- результат записывает в переменную result
- вторая задача отображает результат выполнения команд

Проверьте работу playbook на маршрутизаторах.

Задание 24.1с

Создайте playbook task_24_1c.yml, который выполняет такие задачи:

- подключается к маршрутизаторам и выполняет команды sh arp и sh ip int br
 - обе команды должны выполняться в одной задаче
 - результат записывает в переменную result
- вторая задача отображает результат выполнения команды sh arp
- третья задача отображает результат выполнения команды sh ip int br

Вторая и третья задачи должны отображать вывод команды в виде списка строк.

Проверьте работу playbook на маршрутизаторах.

Задание 24.2

Создайте playbook task_24_2.yml, который выполняет такие задачи:

- собирает все факты с маршрутизаторов
 - результат нельзя записывать в переменную
- отображает содержимое факта об интерфейсах (в факте находится словарь с интерфейсами и их параметрами)

Проверьте работу playbook на маршрутизаторах.

Задание 24.2а

Создайте playbook task_24_2a.yml, который выполняет такие задачи:

- собирает все факты с маршрутизаторов
 - результат нельзя записывать в переменную
- записывает содержимое факта об интерфейсах в файл в каталог all_facts:
 - имя файла должно быть такого вида: hostname_intf_facts.yaml
 - hostname - это имя текущего устройства, для которого собираются факты
 - файл должен быть в формате YAML, в виде, который удобней для чтения человеком

Проверьте работу playbook на маршрутизаторах.

Задание 24.2b

Создайте playbook task_24_2b.yml, который выполняет такие задачи:

- собирает все факты с маршрутизаторов
 - результат не записывать в переменную
- выполняет команду sh ipv6 int br
 - вывод команды записывает в переменную show_result
- отображает содержимое переменной show_result, но только в том случае, когда факт, в котором содержатся IPv6 адреса в виде списка, не пустой

Проверьте работу playbook на маршрутизаторах.

Задание 24.3

В playbook task_15_3.yml описана одна задача.

Попробуйте выполнить его, как минимум, два раза. Обратите внимание, что изменения вносились каждый раз.

Измените playbook таким образом, чтобы изменения вносились только в том случае, когда настройка логирования на устройстве не соответствует указанной команде.

Задание 24.4

Создайте playbook task_24_4.yml, который выполняет такие задачи:

- создает ACL INET-to-LAN и применяет его к интерфейсу Ethernet0/1 для входящего трафика

При этом, подразумевается, что настройка ACL выполняется только с помощью Playbook. Поэтому, в ACL должны быть только те строки, которые указаны в задаче playbook.

Задача должна выполнять такие действия:

- удалить ACL с интерфейса
- удалить ACL
- создать ACL и настроить правила ACL
- применить ACL к интерфейсу

ACL должен быть таким:

```
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
```

Проверьте работу playbook на маршрутизаторе R1.

Задание 24.4a

Проверьте работу playbook из задания 24.4, в ситуации, когда в ACL добавлена ещё одна строка.

Если, после добавления строки в задаче и выполнения playbook, ACL на маршрутизаторе выглядит так же, как описано в playbook, значит задание выполнено.

Если нет, исправьте соответственно задачу.

Добавьте, например, такую строку в ACL:

```
permit tcp 10.0.1.0 0.0.0.255 any eq telnet
```

Проверьте работу playbook на маршрутизаторе R1.

Задание 24.4b

Добавьте в playbook из задания 24.4a ещё одну задачу:

- она должна отображать, какие команды были отправлены на оборудование, в первой задаче
 - команды должны отображаться только в том случае, если были выполнены изменения
- если нужно, можно изменять и первую задачу

Проверьте работу playbook на маршрутизаторе R1.

Задание 24.4c

Измените playbook из задания 24.4b таким образом, чтобы имя интерфейса, который указывается в задаче, указывалось как переменная `outside_intf`.

Создайте переменную для маршрутизатора R1, в соответствующем файле каталога `host_vars`.

Проверьте работу playbook на маршрутизаторе R1.

Дополнительная информация

В этом разделе собрана информация, которая не вошла в основные разделы курса, но которая, тем не менее, может быть полезна.

Соглашение об именах

В Python есть определенные соглашения об именовании объектов.

В целом, лучше придерживаться этих соглашений. Однако, если в определенной библиотеке или модуле используются другие соглашения, то стоит придерживаться того стиля, который используется в них.

В этом разделе описаны не все правила. Подробнее можно почитать в документе PEP8 на [английском](#) или на [русском](#).

Имена переменных

Имена переменных не должны пересекаться с операторами и названиями модулей или других зарезервированных значений.

Имена переменных обычно пишутся полностью большими или маленькими буквами. В пределах одного скрипта/модуля/пакета лучше придерживаться одного из вариантов.

Если переменные - константы для модуля, то лучше использовать имена, написанные заглавными буквами:

```
DB_NAME = 'dhcp_snooping.db'  
TESTING = True
```

Для обычных переменных лучше использовать имена в нижнем регистре:

```
db_name = 'dhcp_snooping.db'  
testing = True
```

Имена модулей и пакетов

Имена модулей и пакетов задаются маленькими буквами.

Модули могут использовать подчеркивания между словами для того, чтобы имена были более понятными. Для пакетов лучше выбирать короткие имена.

Имена функций

Имена функций задаются маленькими буквами, с подчеркиваниями между словами.

```
def ignore_command(command, ignore):  
  
    ignore_command = False  
  
    for word in ignore:  
        if word in command:  
            return True  
    return ignore_command
```

Имена классов

Имена классов задаются словами с заглавными буквами, без пробелов.

```
class CiscoSwitch:  
  
    def __init__(self, name, vendor = 'cisco', model = '3750'):  
        self.name = name  
        self.vendor = vendor  
        self.model = model
```

Подчеркивание в именах

В Python подчеркивание в начале или в конце имени указывает на специальные имена. Чаще всего это всего лишь договоренность, но иногда это действительно влияет на поведение объекта.

Подчеркивание как имя

В Python одно подчеркивание используется для обозначения того, что данные просто выбрасываются.

Например, если из строки line надо получить MAC-адрес, IP-адрес, VLAN и интерфейс и отбросить остальные поля, можно использовать такой вариант:

```
In [1]: line = '00:09:BB:3D:D6:58  10.1.10.2  86250  dhcp-snooping  10  FastEthernet0/1'

In [2]: mac, ip, _, _, vlan, intf = line.split()

In [3]: print(mac, ip, vlan, intf)
00:09:BB:3D:D6:58 10.1.10.2 10 FastEthernet0/1
```

Такая запись говорит о том, что нам не нужны третий и четвертый элементы.

Можно сделать так:

```
In [4]: mac, ip, lease, entry_type, vlan, intf = line.split()
```

Но тогда может быть непонятно, почему переменные `lease` и `entry_type` не используются дальше. Если понятней использовать имена, то лучше назвать переменные именами вроде `ignored`.

Аналогичный прием может использоваться, когда переменная цикла не нужна:

```
In [5]: [0 for _ in range(10)]
Out[5]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Подчеркивание в интерпретаторе

В интерпретаторе `python` и `ipython` подчеркивание используется для получения результата последнего выражения

```
In [6]: [0 for _ in range(10)]
Out[6]: [0, 0, 0, 0, 0, 0, 0, 0, 0]

In [7]: _
Out[7]: [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]
```

Одно подчеркивание

Одно подчеркивание перед именем

Одно подчеркивание перед именем указывает, что имя используется как внутреннее.

Например, если одно подчеркивание указано в имени функции или метода, это означает, что этот объект является внутренней особенностью реализации и не стоит его использовать напрямую.

Но, кроме того, при импорте вида `from module import *` не будут импортироваться объекты, которые начинаются с подчеркивания.

Например, в файле `example.py` такие переменные и функции:

```
db_name = 'dhcp_snooping.db'
_path = '/home/nata/pyneng/'

def func1(arg):
    print arg

def _func2(arg):
    print arg
```

Если импортировать все объекты из модуля, то те, которые начинаются с подчеркивания, не будут импортированы:

```
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
```

Одно подчеркивание после имени

Одно подчеркивание после имени используется в том случае, когда имя объекта или параметра пересекается со встроенными именами.

Пример:

```
In [12]: line = '00:09:BB:3D:D6:58  10.1.10.2  86250  dhcp-snooping  10  FastEthernet
0/1'

In [13]: mac, ip, lease, type_, vlan, intf = line.split()
```

Два подчеркивания

Два подчеркивания перед именем

Два подчеркивания перед именем метода используются не просто как договоренность. Такие имена трансформируются в формат "имя класса + имя метода". Это позволяет создавать уникальные методы и атрибуты классов.

Такое преобразование выполняется только в том случае, если в конце менее двух подчеркиваний или нет подчеркиваний.

```
In [14]: class Switch(object):
...:     __quantity = 0
...:     def __configure(self):
...:         pass
...:

In [15]: dir(Switch)
Out[15]:
['_Switch__configure', '_Switch__quantity', ...]
```

Хотя методы создавались без приставки `_Switch`, она была добавлена.

Если создать подкласс, то метод `__configure` не перепишет метод родительского класса `Switch`:

```
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', ...]
```

Два подчеркивания перед и после имени

Таким образом обозначаются специальные переменные и методы.

Например, в модуле Python есть такие специальные переменные:

- `__name__` - эта переменная равна строке `__main__`, когда скрипт запускается напрямую, и равна имени модуля, когда импортируется
- `__file__` - эта переменная равна имени скрипта, который был запущен напрямую, и равнаному пути к модулю, когда он импортируется

Переменная `__name__` чаще всего используется, чтобы указать, что определенная часть кода должна выполняться, только когда модуль выполняется напрямую:

```
def multiply(a, b):  
  
    return a * b  
  
if __name__ == '__main__':  
    print(multiply(3, 5))
```

А переменная `__file__` может быть полезна в определении текущего пути к файлу скрипта:

```
import os  
  
print('__file__', __file__)  
print(os.path.abspath(__file__))
```

Вывод будет таким:

```
__file__ example2.py  
/home/vagrant/repos/tests/example2.py
```

Кроме того, таким образом в Python обозначаются специальные методы. Эти методы вызываются при использовании функций и операторов Python и позволяют реализовать определенный функционал.

Как правило, такие методы не нужно вызывать напрямую. Но, например, при создании своего класса может понадобиться описать такой метод, чтобы объект поддерживал какие-то операции в Python.

Например, для того, чтобы можно было получить длину объекта, он должен поддерживать метод `__len__`.

Ещё один специальный метод `__str__` вызывается, когда используется оператор `print` или вызывается функция `str()`. Если необходимо, чтобы при этом отображение было в определенном виде, надо создать этот метод в классе:

```
In [10]: class Switch(object):
...:
...:     def set_name(self, name):
...:         self.name = name
...:
...:     def __configure(self):
...:         pass
...:
...:     def __str__(self):
...:         return 'Switch {}'.format(self.name)
...:

In [11]: sw1 = Switch()

In [12]: sw1.set_name('sw1')

In [13]: print sw1
Switch sw1

In [14]: str(sw1)
Out[14]: 'Switch sw1'
```

Таких специальных методов в Python очень много. Несколько полезных ссылок, где можно почитать про конкретный метод:

- [документация](#)
- [Dive Into Python 3](#)

Полезные встроенные функции

В этом подразделе рассматриваются такие функции:

- lambda
- map
- filter

Анонимная функция lambda

В Python выражение `lambda` позволяет создавать анонимные функции - функции, которые не привязаны к имени.

В анонимной функции `lambda`:

- может содержаться только одно выражение
- аргументов может передаваться сколько угодно

Стандартная функция:

```
In [1]: def sum_arg(a, b): return a + b  
  
In [2]: sum_arg(1, 2)  
Out[2]: 3
```

Аналогичная анонимная функция `lambda`:

```
In [3]: sum_arg = lambda a, b: a + b  
  
In [4]: sum_arg(1, 2)  
Out[4]: 3
```

Обратите внимание, что в определении `lambda` нет оператора `return`, так как в этой функции может быть только одно выражение, которое всегда возвращает значение и завершает работу функции.

Функцию `lambda` удобно использовать в выражениях, где требуется написать небольшую функцию для обработки данных.

Например, в функции `sorted` `lambda` можно использовать для указания ключа для сортировки:

```
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)]
```

Также функция `lambda` пригодится в функциях `map` и `filter`, которые будут рассматриваться в следующих разделах.

Функция map

Функция map применяет функцию к каждому элементу последовательности и возвращает итератор с результатами.

Например, с помощью map можно выполнять преобразования элементов. Перевести все строки в верхний регистр:

```
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']
```

Конвертация в числа:

```
In [3]: list_of_str = ['1', '2', '5', '10']

In [4]: list(map(int, list_of_str))
Out[4]: [1, 2, 5, 10]
```

Вместе с map удобно использовать lambda:

```
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']
```

Если функция, которую использует map(), ожидает два аргумента, то передаются два списка:

```
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 вместо map

Как правило, вместо map можно использовать list comprehension. Чаще всего, вариант с list comprehension более понятный, а в некоторых случаях даже быстрее.

Ответ Alex Martelli со сравнением map и list comprehension

Но map может быть эффективней в том случае, когда надо сгенерировать большое количество элементов, так как map - итератор, а list comprehension генерирует список.

Примеры, аналогичные приведенным выше, в варианте с list comprehension.

Перевести все строки в верхний регистр:

```
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']
```

Конвертация в числа:

```
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]
```

Форматирование строк:

```
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']
```

Для получения пар элементов используется zip:

```
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]
```

Функция filter

Функция `filter()` применяет функцию ко всем элементам последовательности и возвращает итератор с теми объектами, для которых функция вернула `True`.

Например, вернуть только те строки, в которых находятся числа:

```
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']
```

Из списка чисел оставить только нечетные:

```
In [3]: list(filter(lambda x: x%2, [10, 111, 102, 213, 314, 515]))
Out[3]: [111, 213, 515]
```

Аналогично, только четные:

```
In [4]: list(filter(lambda x: not x%2, [10, 111, 102, 213, 314, 515]))
Out[4]: [10, 102, 314]
```

Из списка слов оставить только те, у которых количество букв больше двух:

```
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 вместо filter

Как правило, вместо `filter` можно использовать `list comprehension`.

Примеры, аналогичные приведенным выше, в варианте с `list comprehension`.

Вернуть только те строки, в которых находятся числа:

```
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']
```

Нечетные/четные числа:

```
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]
```

Из списка слов оставить только те, у которых количество букв больше двух:

```
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']
```

Основы **threading** и **multiprocessing**

В разделе [Одновременное подключение к нескольким устройствам](#) рассматривался модуль concurrent.futures. Он предоставляет высокоуровневый интерфейс для асинхронного выполнения задач.

В этом разделе рассматриваются основы модулей `threading` и `multiprocessing`. Они предоставляют больше возможностей, чем `concurrent.futures`, но при этом и сложнее в использовании.

Цель этого раздела - показать как с помощью модулей `threading` и `multiprocessing` распараллелить выполнение задачи.

Модуль **threading**

Модуль `threading` может быть полезен для таких задач:

- фоновое выполнение каких-то задач:
 - например, отправка почты во время ожидания ответа от пользователя
- параллельное выполнение задач, связанных со вводом/выводом
 - ожидание ввода от пользователя
 - чтение/запись файлов
- задачи, где присутствуют паузы:
 - например, паузы с помощью `sleep`

Однако следует учитывать, что в ситуациях, когда требуется повышение производительности за счет использования нескольких процессоров или ядер, нужно использовать модуль `multiprocessing`, а не модуль `threading`.

Рассмотрим пример использования модуля `threading` вместе с последним примером с `netmiko`.

Так как для работы с `threading` удобнее использовать функции, код изменен:

- код подключения по SSH перенесён в функцию
- параметры устройств перенесены в отдельный файл в формате YAML

Файл `netmiko_function.py`:

```
import sys
import yaml
from netmiko import ConnectHandler

#COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, commands):

    print('Connection to device {}'.format( device_dict['ip'] ))

    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()

        result = ssh.send_config_set(commands)
        print(result)

commands_to_send = ['logg 10.1.12.3', 'ip access-li ext TESST2', 'permit ip any any']

for router in devices['routers']:
    connect_ssh(router, commands_to_send)
```

Файл devices.yaml с параметрами подключения к устройствам:

```
routers:
- device_type: cisco_ios
  ip: 192.168.100.1
  username: cisco
  password: cisco
  secret: cisco
- device_type: cisco_ios
  ip: 192.168.100.2
  username: cisco
  password: cisco
  secret: cisco
- device_type: cisco_ios
  ip: 192.168.100.3
  username: cisco
  password: cisco
  secret: cisco
```

Время выполнения скрипта (вывод скрипта удален):

```
$ time python netmiko_function.py "sh ip int br"
...
real    0m6.189s
user    0m0.336s
sys     0m0.080s
```

Пример использования модуля `threading` для подключения по SSH с помощью `netmiko` (файл `netmiko_threading.py`):

```
import sys
import yaml
import threading

from netmiko import ConnectHandler


COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))


def connect_ssh(device_dict, command):
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)

        print('Connection to device {}'.format( device_dict['ip'] ))
        print(result)


def conn_threads(function, devices, command):
    threads = []
    for device in devices:
        th = threading.Thread(target = function, args = (device, command))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

    conn_threads(connect_ssh, devices['routers'], COMMAND)
```

Время выполнения кода:

```
$ time python netmiko_function_threading.py "sh ip int br"
...
real    0m2.229s
user    0m0.408s
sys     0m0.068s
```

Время почти в три раза меньше. Но надо учесть, что такая ситуация не будет повторяться при большом количестве подключений.

Комментарии к функции `conn_threads`:

- `threading.Thread` - класс, который создает поток
 - ему передается функция, которую надо выполнить, и её аргументы
- `th.start()` - запуск потока
- `threads.append(th)` - поток добавляется в список
- `th.join()` - метод ожидает завершения работы потока
 - метод `join` выполняется для каждого потока в списке. Таким образом, основная программа завершится, только когда завершат работу все потоки
 - по умолчанию `join` ждет завершения работы потока бесконечно. Но можно ограничить время ожидания, передав `join` время в секундах. В таком случае `join` завершится после указанного количества секунд.

Получение данных из потоков

В предыдущем примере данные выводились на стандартный поток вывода. Для полноценной работы с потоками необходимо также научиться получать данные из потоков. Чаще всего для этого используется очередь.

В Python есть модуль `queue`, который позволяет создавать разные типы очередей.

Очередь - это структура данных, которая используется и в работе с сетевым оборудованием. Объект `queue.Queue()` - это FIFO очередь.

Очередь передается как аргумент в функцию `connect_ssh`, которая подключается к устройству по SSH. Результат выполнения команды добавляется в очередь.

Пример использования потоков с получением данных (файл `netmiko_threading_data.py`):

```
# -*- coding: utf-8 -*-
import sys
import yaml
import threading
from queue import Queue
from pprint import pprint
from netmiko import ConnectHandler

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print('Connection to device {}'.format(device_dict['ip']))

    #Добавляем словарь в очередь
    queue.put({device_dict['ip']: result})

def conn_threads(function, devices, command):
    threads = []
    q = Queue()

    for device in devices:
        # Передаем очередь как аргумент, функции
        th = threading.Thread(target=function, args=(device, command, q))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

    results = []
    # Берем результаты из очереди и добавляем их в список results
    for t in threads:
        results.append(q.get())

    return results

pprint(conn_threads(connect_ssh, devices['routers'], COMMAND))
```

Обратите внимание, что в функции `connect_ssh` добавился аргумент `queue`.

Очередь вполне можно воспринимать как список:

- метод `queue.put()` равнозначен `list.append()`
- метод `queue.get()` равнозначен `list.pop(0)`

Для работы с потоками и модулем `threading` лучше использовать очередь.

Очередь лучше тем, что она поддерживает только две операции по изменению содержимого:

- добавить элемент - `queue.put()`
- взять элемент - `queue.get()`

А список, кроме этих операций, поддерживает изменение элементов, переприсваивание значений. И при работе с потоками, используя эти операции, можно получить совсем не тот результат, который ожидался.

Но пример со списком, скорее всего, будет проще понять. И при использовании методов `append` и `pop` никаких проблем не будет.

Ниже аналогичный код, но с использованием обычного списка вместо очереди (файл `netmiko_threading_data_list.py`):

```
# -*- coding: utf-8 -*-
import sys
import yaml
import threading
from pprint import pprint

from netmiko import ConnectHandler


COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))


def connect_ssh(device_dict, command, queue):
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        print('Connection to device {}'.format( device_dict['ip'] ))

    #Добавляем словарь в список
    queue.append({ device_dict['ip']: result })


def conn_threads(function, devices, command):
    threads = []
    q = []

    for device in devices:
        # Передаем список как аргумент, функции
        th = threading.Thread(target = function, args = (device, command, q))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

    return q


result = conn_threads(connect_ssh, devices['routers'], COMMAND)
pprint(result)
```

Модуль multiprocessing

Модуль `multiprocessing` использует интерфейс, подобный модулю `threading`. Поэтому перенести код с использования потоков на использование процессов обычно достаточно легко.

Каждому процессу выделяются свои ресурсы. Кроме того, у каждого процесса свой GIL, а значит, нет тех проблем, которые были с потоками, и код может выполняться параллельно и задействовать ядра/процессоры компьютера.

Пример использования модуля `multiprocessing` (файл `netmiko_multiprocessing.py`):

```

import multiprocessing
import sys
import yaml
from pprint import pprint

from netmiko import ConnectHandler


COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))


def connect_ssh(device_dict, command, queue):
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)

        print('Connection to device {}'.format(device_dict['ip']))
        queue.put({device_dict['ip']: result})


def conn_processes(function, devices, command):
    processes = []
    queue = multiprocessing.Queue()

    for device in devices:
        p = multiprocessing.Process(target=function,
                                   args=(device, command, queue))
        p.start()
        processes.append(p)

    for p in processes:
        p.join()

    results = []
    for p in processes:
        results.append(queue.get())

    return results


pprint(conn_processes(connect_ssh, devices['routers'], COMMAND))

```

Обратите внимание, что этот пример аналогичен последнему примеру, который использовался с модулем `threading`. Единственное отличие в том, что в модуле `multiprocessing` есть своя реализация очереди, поэтому нет необходимости использовать модуль `Queue`.

Если проверить время исполнения этого скрипта, аналогичного для модуля `threading` и последовательного подключения, то получаем такую картину:

```
последовательное: 5.833s
threading:          2.225s
multiprocessing:   2.365s
```

Время выполнения для модуля `multiprocessing` немного больше. Но это связано с тем, что на создание процессов уходит больше времени, чем на создание потоков. Если бы скрипт был сложнее и выполнялось больше задач, или было бы больше подключений, тогда бы `multiprocessing` начал бы существенно выигрывать у модуля `threading`.

Дополнительные материалы

Документация:

- [threading](#)
- [multiprocessing](#)
- [queue](#)
- [time](#)
- [datetime](#)

GIL

- [Can't we get rid of the Global Interpreter Lock?](#)
- [GIL \(на русском\)](#)
- [Understanding the Python GIL](#)
- [Python threads and the GIL](#)

Полезные вопросы и ответы на stackoverflow

- [Multiprocessing vs Threading Python](#)
- [Python: what are the differences between the threading and multiprocessing modules?](#)
- [How many processes should I run in parallel?](#)
- [How many threads is too many?](#)

Отличия Python 2.7 и Python 3.6

На данный момент есть две версии книги: для [Python 2.7](#) и Python 3.6. Чтобы облегчить переход с версии 2.7 на 3.6, тут перечислены отличия между этими версиями книги.

Кроме отличий, которые сделаны из-за изменений в Python 3, в книге также [обновлены многие разделы](#).

Со временем, эти обновления будут перенесены и в книгу по Python 2.7

Unicode

В Python 2.7 было два типа строк: str и unicode:

```
In [1]: line = 'test'  
  
In [2]: line2 = u'тест'
```

В Python 3 строка - это тип str, но, кроме этого, в Python 3 появился тип bytes:

```
In [3]: line = 'тест'  
  
In [4]: line.encode('utf-8')  
Out[4]: b'\xd1\x82\xd0\xb5\xd1\x81\xd1\x82'  
  
In [5]: byte_str = b'test'
```

Функция print

В Python 2.7 print был оператором:

```
In [6]: print 1, 'test'  
1 test
```

В Python 3 print - функция:

```
In [7]: print(1, 'test')  
1 test
```

В Python 2.7 можно брать аргументы в скобки, но от этого print не становится функцией и, кроме того, print возвращает другой результат (кортеж):

```
In [8]: print(1, 'test')
(1, 'test')
```

В Python 3, использование синтаксиса Python 2.7 приведет к ошибке:

```
In [9]: print 1, 'test'
File "<ipython-input-2-328abb6b105d>", line 1
    print 1, 'test'
    ^
SyntaxError: Missing parentheses in call to 'print'
```

input вместо raw_input

В Python 2.7 для получения информации от пользователя в виде строки использовалась функция raw_input:

```
In [10]: number = raw_input('Number: ')
Number: 55

In [11]: number
Out[11]: '55'
```

В Python 3 используется input:

```
In [12]: number = input('Number: ')
Number: 55

In [13]: number
Out[13]: '55'
```

range вместо xrange

В Python 2.7 были две функции

- range - возвращает список
- xrange - возвращает итератор

Пример range и xrange в 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]
```

В Python 3 есть только функция `range`, и она возвращает итератор:

```
In [17]: range(5)
Out[17]: range(0, 5)

In [18]: list(range(5))
Out[18]: [0, 1, 2, 3, 4]
```

Методы словарей

Несколько изменений произошло в методах словарей.

`dict.keys()`, `values()`, `items()`

Методы `keys()`, `values()`, `items()` в Python 3 возвращают "views" вместо списков. Особенность view заключается в том, что они меняются вместе с изменением словаря. И фактически они лишь дают способ посмотреть на соответствующие объекты, но не создают их копию.

В Python 3 нет методов:

- `viewitems`, `viewkeys`, `viewvalues`
- `iteritems`, `iterkeys`, `itervalues`

Для сравнения, методы словаря в Python 2.7:

```
In [19]: d = {1:100, 2:200, 3:300}

In [20]: d.
          d.clear      d.get       d.iteritems   d.keys        d.setdefault d.viewitems
          d.copy       d.has_key    d.iterkeys     d.pop         d.update      d.viewkeys
          d.fromkeys   d.items     d.itervalues  d.popitem    d.values      d.viewvalues
```

И в 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()
```

Распаковка переменных

В Python 3 появилась возможность использовать `*` при распаковке переменных:

```
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
```

В Python 2.7 этот синтаксис не поддерживается:

```
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
```

Итератор вместо списка

В Python 2.7 `map`, `filter` и `zip` возвращали список:

```
In [28]: map(str, [1,2,3,4,5])
Out[28]: ['1', '2', '3', '4', '5']

In [29]: filter(lambda x: x>3, [1,2,3,4,5])
Out[29]: [4, 5]

In [30]: zip([1,2,3], [100,200,300])
Out[30]: [(1, 100), (2, 200), (3, 300)]
```

В Python 3 они возвращают итератор:

```
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

В версии Python 3.5 в модуле subprocess появилась новая функция - run. Она предоставляет более удобный интерфейс для работы с модулем и получения вывода команд.

Соответственно, вместо функций call и check_output используется функция run. Но функции call и check_output остались.

Jinja2

В модуле Jinja2 больше не нужно использовать такой код, так как кодировка по умолчанию и так utf-8:

```
import sys
reload(sys)
sys.setdefaultencoding('utf-8')
```

В самих шаблонах, как и в Python, изменились методы словарей. Тут, аналогично, вместо iteritems надо использовать items.

Модули repect, telnetlib, paramiko

Модули repect, telnetlib, paramiko отправляют и получают байты, поэтому надо делать encode/decode соответственно.

В netmiko эта конвертация выполняется автоматически.

Мелочи

- Название модуля Queue сменилось на queue
- С версии Python 3.6 объект csv.DictReader возвращает OrderedDict вместо обычного словаря.

Дополнительная информация

Ниже приведены ссылки на ресурсы с информацией об изменениях в Python 3.

Документация:

- [What's New In Python 3.0](#)
- [Should I use Python 2 or Python 3 for my development activity?](#)

Статьи:

- [The key differences between Python 2.7.x and Python 3.x with examples](#)
- [Supporting Python 3: An in-depth guide](#)

Продолжение обучения

Как правило, информацию тяжело усвоить с первого раза. Особенно, новую информацию.

Если делать практические задания и пометки, в ходе изучения, то усвоится намного больше информации, чем, если просто читать книгу. Но, скорее всего, в каком-то виде, надо будет читать о той же информации несколько раз.

Книга дает лишь основы Python и поэтому надо обязательно продолжать учиться и повторять уже пройденные темы и изучать новое. И тут есть множество вариантов:

- автоматизировать что-то в работе
- изучать дальше Python для автоматизации работы с сетью
- изучать Python без привязки к сетевому оборудованию

Тут ресурсы перечислены выборочно, с учетом того, что Вы уже прочитали книгу. Но, кроме этого, я сделала [подборку ресурсов](#) в которой можно найти и другие материалы.

Написание скриптов для автоматизации рабочих процессов

Скорее всего, после прочтения книги, появятся идеи, что можно автоматизировать на работе. Это отличный вариант, так как на реальной задаче всегда проще учиться и изучать новое. Но лучше не ограничиваться только рабочими задачами и изучать Python дальше.

Python позволяет делать достаточно многое обладая только базовыми знаниями. Поэтому всегда рабочие задачи позволяют принципиально повысить уровень знаний или подтолкнуть к этому.

Но зная Python лучше, те же задачи можно решать, как правило, намного проще. Поэтому лучше не останавливаться и учиться дальше.

Ниже описаны ресурсы с привязкой к сетевому оборудованию и в целом по Python. В зависимости от того, по каким материалам Вы лучше учитесь, можно выбрать книги или видео курсы из списка

Python для автоматизации работы с сетевым оборудованием

Книги:

- [Mastering Python Networking \(Eric Chou\)](#) - отчасти перекликается с тем, что рассматривалось в этой книге, но в ней есть и много новых тем. Плюс, рассматриваются примеры не только на оборудовании Cisco, но Juniper и Arista.

Блоги - позволяют быть в курсе новостей в этой сфере:

- [Kirk Byers](#)
- [Jason Edelman](#)
- [Matt Oswalt](#)
- [Michael Kashin](#)
- [Henry Ölsner](#)
- [Mat Wood](#)

У Packet Pushers достаточно часто выходят подкасты об автоматизации:

- [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](#)

Проекты:

- [CiscoConfParse](#) - библиотека, которая парсит конфигурации типа Cisco IOS. С ее помощью можно: проверять существующие конфигурации маршрутизаторов/коммутаторов, получать определенную часть конфигурации, изменять конфигурацию
- [NAPALM](#) - NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) это библиотека, которая позволяет работать с сетевым оборудованием разных вендоров, используя унифицированный API
- [NOC Project](#) - NOC is the scalable, high-performance and open-source OSS system for ISP, service and content providers
- [Requests](#) - библиотека для работы с HTTP
- [SaltStack](#) - аналог Ansible
- [Scapy](#) - сетевая утилита, которая позволяет манипулировать сетевыми пакетами
- [StackStorm](#) - StackStorm is event-driven automation commonly used for auto-remediation, security responses, facilitated troubleshooting, complex deployments, and

more

Python без привязки к сетевому оборудованию

Книги:

- [A Byte of Python](#) - книга по основам Python. [На русском](#)
- [Dive Into Python 3](#) - в этой книге рассматриваются более продвинутые темы и она отлично подойдет для 2-3 книги по Python
- [Problem Solving with Algorithms and Data Structures using Python](#) - отличная книга по структурам данных и алгоритмам. Много примеров и домашних заданий. Написана простым, понятным языком. [На русском](#)
- [Automate the Boring Stuff with Python](#) - в этой книге можно найти много идей по автоматизации ежедневной работы. Тут рассматриваются такие темы: работа с файлами PDF, Excel, Word, отправка писем, работа с картинками, работа в веб. [На русском](#)

Курсы:

- [MITx - 6.00.1x Introduction to Computer Science and Programming Using Python](#) - очень хороший курс по Python. Отличный вариант для продолжения обучения после книги. В нём Вы и повторите пройденный материал по основам Python, но под другим углом и узнаете много нового. В курсе много практических заданий и он достаточно интенсивный.
- [Python от Computer Science Center](#) - отличные видеолекции по Python. Тут есть и немного основ и более продвинутые темы

Сайты с задачами:

- [HackerRank](#) - на этом сайте задачи разбиты по областям: алгоритмы, регулярные выражения, базы данных и другие. Но есть и базовые задачи
- [CheckIO - online game for Python and JavaScript coders](#)

Подкасты позволяют в целом расширить кругозор и получить представление о разных проектах, модулях и библиотеках Python:

- [Talk Python To Me](#)
- [Best Python Podcasts](#)

Документация:

- [Официальная документация Python](#)

- [Python Module of the Week](#)
- [Tiny-Python-3.6-Notebook](#) - Отличная шпаргалка по Python 3.6

Отзывы читателей книги и слушателей курса

Ян Коробов

Курс Python для сетевых инженеров.

Вот и пролетели 3 месяца с первого занятия и пора писать отзыв. Как только я услышал про этот курс и что его ведет та самая Наташа Самойленко, я подумал вот оно! Как было бы здорово у нее поучится и ожидания оправдались на 200%!

Как и многих, кто вообще не занимался раньше программированием, перед курсом меня еще одолевали сомнения, а смогу ли я? Наверное, надо хорошо знать математику, итд. Все стереотипы быстро развеиваются. На деле программирование под руководством Наташи превращается в увлекательное, затягивающее дело. Серьезно, как по мне так это очень весело! Но это не значит, что Вас ждет легкая прогулка, сразу советую работать на максимум и делать все задания. Результат прямо пропорционален затраченным усилиям. На курсе попадаешь в атмосферу разработчика, подготовлены все необходимые инструменты для этого. Интенсивность курса средняя. Очень много практики и заданий в контексте сетевого администрирования. Как по мне так отличная структура, Одна, две лекции в неделю, остальная неделя на практические задания. В slack всегда можно рассчитывать на помочь Наташи по заданиям и конечно заряд мотивации на неделю обеспечен;) Иногда в чате можно встретить таких "монстров" как Эмиль Гарипов! Помимо этого, огромное количество дополнительного материала (помимо лекций, заданий и книги) И так, незаметно для себя ты уже между делом пишешь код что бы получить словарь из Английской книги потому что тебе не хватает словарного запаса. Или опрашивашь 1000-и устройств, складывая результат в базу, применяешь шаблоны jinja, конфигурируешь, пишешь playbook в Ansible. И Наташа не может и не хочет стоять на месте поэтому появились дополнительные занятия после курса по ООП, что невероятно расширяет ваши возможности.

Что бы чему-то научиться нужно идти к лучшим, людям страстно одержимым своим делом, на наше счастье Наташа одинаково увлечена и программированием, и желанием научить нас, огромное тебе спасибо за это!!!

Сергей Забабурин

Осталось три темы и я закончу воркшоп "Питон для сетевых инженеров" от Наташи Самойленко. Курс шел с сентября по ноябрь, 13 недель.

Я как всегда отжег: оплатил, ничего не делал и начал заниматься, когда группа прошла половину. Однако, начав, уже не смог оторваться и топлю до финала. Наташа, если вдруг не знаете, написала чуть меньше чем весь сетевой раздел XGU.ru , автоматизировала проект CCIE за год и написала для него первую "большую лабу". Послушать ее чарующий голос можно в клубе 256-TTL

В курсе сам Python, базы SQL, форматы YAML, JSON, шаблоны Jinja2 и, конечно, Ansible. 50 часов видео и 114 заданий для самостоятельной работы.

Почему этот курс?

Во-первых он продуман и оснащен всем необходимым инструментарием. Книга, лекции, записи лекций, домашние задания, тесты на повторение, графики "кто сдал а кто еще ЛОЛ" и конечно чатик, где всегда помогут а Наташа не спит вовсе.

Во-вторых это атмосфера. Шутки-прибаутки на лекциях. В чатике тебе не дадут решение сразу но заглумят в нужном направлении. Если Вы когда то в общаге готовились к сессии, то должны понять. Что важно для меня - Наташа профессионал и очень увлеченый человек. Это подкупает в начале и очень поддерживает весь курс. Письма с ее фирменным "Все отлично! Но вот можно еще так и вот так и еще 3 другими способами...." ждешь как праздника.

В третьих Наташа коварна. Пройденные темы интервально повторяются в заданиях в разных вариациях, сами задания для каждой темы усложняются от номера к номеру, обрастают проверками, функционалом и проч. Например у Вас позади остались базовые структуры данных, функции и форматы данных. И сейчас у Вас подключение к оборудованию через netmiko. Что проще - два метода. Рано радуетесь - задание будет таким: возьмите файлик YAML с параметрами устройств, напишите пару функций, которые проходят по оборудованию из этого YAML и вводят команды. Сделайте скрипт таким, чтобы команды можно было ввести как угодно: поштучно или файлом или списком . Заодно проверьте ошибки (вдруг у Вас командочка кривая) и на выходе сделайте два словаря в которых вложенные словари для команд которые прошли и которые нет. И не забудьте использовать регулярки)

И наконец, что действительно важно. Наташа заявляет: питон это просто и после курса Вы будете его использовать в повседневной работе. Так и происходит. Начиная с азов и включая серьезные вещи (например потоки, производительность кода) шаг за шагом она выращивает в участнике навыки и знания с которыми можно идти и работать. Это курс для практиков, примеры из реальной жизни, материал подобран и сверстан так,

что все нужное и всего достаточно. Если Вам требуется эффективно, в понятные сроки и за умеренную плату (сейчас это 250\$) закрыть вопрос с питоном - то Вам сюда: natenka.github.io/pyneng-online

ЗЫ: Если Вам не нужно быстро и Вы ходите как я два года сидеть и сожалеть на тему "как плохо не уметь писать скрипты для Juniper" то совсем даром на сайте выложена книга и Git с примерами и готовыми виртуалками для самостоятельных штудий.

Александр Романов

Когда у меня стали отнимать слишком много времени рутинные задачи. Когда коэффициент подошёл к восьмидесяти процентам я понял что нужно что то менять. Я пробовал изучать сам , но из за недостатка времени у меня не получалось. И в один прекрасный день я натолкнулся на курс который предлагала Наташа Самойленко. С первых занятий я понял что этот курс ни в какой сравнение с теми которые у меня бывали раньше. Имея большой опыт в администрировании сетей показывала подходящие примеры для быстрейшего усвоения нами материалов и применения их в работе. И всегда старалась консультировать и отвечать на вопросы в отличии от других преподавателей на курсах которых я проходил ранее. Думаю этот курс прочно займет первое место среди всех курсов которые я проходил (наверное пока Наташа не выпустит что то новое). Думаю после этого запатентовать бренд Natasha Samoylenko как будущий сертификат качества.

Денис Карпушин

Все что Вы хотели знать о Python, но боялись спросить.

Основная проблема самостоятельного обучения это дисциплина. Сложно себя организовать и заставить вообще что-то делать, тем более довести все до конца. Я записался на курс что бы была мотивация и оформленная программа обучения.

Все обучение построено в очень удобной форме, и каждый найдет для себя подходящий вариант. Доступны живые лекции, записи, книга, непосредственно общение с преподавателем. Можно комбинировать как Вам удобно, чтобы обучение было в радость. Атмосфера почти домашняя (Спасибо Наташе).

Не имея никакого опыта в программировании за пару месяцев начал использовать Python для решения рабочих задач. Сложно представить что-то лучше этого курса если Вы решили начать изучать программирование. Буду скучать по беседам, домашке, лекциям, и завидовать тем кому только предстоит всё это испытать.

Евгений Овчинников

В октябре 2016 года я решил стать программистом и начал учиться. Покупал платные курсы от одного очень навязчивого своей рекламой образовательного российского интернет-ресурса. После нескольких месяцев обучения стало понятно, что прогресс хоть и есть, но довольно скромный.

Ключевой проблемой было то, что изучаемые материалы либо были слабо применимы к моей работе (инженер техподдержки в небольшом провайдере), либо неприменимы совсем. Оставалось только время в вечернее время и выходные дни. Постепенно я уставал и терял мотивацию что-то делать. Самовнушение не помогало. Вероятно я ошибся с выбором направления (веб-разработка), или не сумел выделить нормальное количество времени на самостоятельную работу. На все это накладывалась сильная загруженность на новой работе - необходимо было быстро усваивать массу материала. К лету 2017 процесс стал совсем - я пытался что-то делать, но уже не было сил и желания. Пройдя примерно половину курса по специальности Веб-разработчик я понял, что больше не хочу этим заниматься и забил. Было грустно

Затем увидел на linkmeup.ru объявление о курсе "Питон для сетевых инженеров". Почитал описание курса и понял, что такой подход (ориентация на сетевую сферу) может резко увеличить мои шансы на успех. Ведь я изучаю язык программирования, который могу применить как раз на своем рабочем месте. Мне нужно опрашивать сетевое оборудование? Нужно. Мне нужно массово менять конфигурации? Нужно (хотя пока не решаюсь это делать скриптами, страшно). Вот тебе Python, товарищ! Бери и пользуйся.

Тааак. Что у нас с вакансиями по Питону? Да их куча! Кроме чисто программистских вакансий есть те же самые сетевые инженеры со знанием Питона, сетевые администраторы со знанием Питона... Даже научный сотрудник со знанием Питона на момент написание этих строк куда-то требуется. Читатель мой, ты понимаешь, что это означает? Научившись основам Питона и начав применять их на практике, ты еще не становишься программистом, но ты резко увеличиваешь количество доступных тебе вакансий. Твоя ценность на рынке труда существенно повышается.

В конце августа начались занятия. Наташа очень аккуратно подводила нас к мысли, что Питона бояться не надо. Что она за нас уже все инструменты подготовила. Что на сайте курса есть расписание курса, ссылки на задания по курсу, ссылки на материалы по занятиям в книге, ссылки на презентации по курсу, ссылки на репозиторий курса, ссылки на статьи о правильном обучении, ссылки на полезные ссылки в интернете... В общем студенту осталось только сесть перед монитором в назначенное время. Которое Наташа обозначила в календаре Google и предложила всем себе его добавить.

После многих лекций студентам рассылаются тесты, которые позволяют быстро проверить, что ты запомнил из нового материала. Это очень приятный и неожиданный для меня момент. Он еще раз показывает, насколько Наташа серьезно готовилась к проведению занятий.

Домашняя работа. Тут Наташа тоже подошла необычно: есть необходимый минимум заданий, которые нужно выполнить для получения сертификата. И есть максимум, который студенты делают по желанию. В любом случае практика программирования у студентов набирается достаточно большой. А уж если Вы осилите максимум, то честь Вам и почет! Я пока еще не осилил. Надежда тает с каждым днем, но пока есть :)

Теперь о том, что не понравилось. Часть материала можно было бы серьезно сократить. Я бы это сделал с разделами про параллельное подключение к оборудованию и материалами про Ансибл. Параллельное подключение довольно сложно идет для новичка. Я заснул в самом начале раздела и почти не просыпался до конца. Бродя как тема важная, но настолько мозг еще к такому материалу не готов, что он просто его игнорирует. Во время лекции я открывал вкладки в браузере, серфил, слушал музыку и иногда запускал видеоролики. В общем, вел себя как типичный студент, который пришел на лекцию чисто для галочки.

Ансибл. Он интересен, но я против его присутствия на курсе программирования. Изучение программирования забирает очень много времени. Это время нужно использовать по-максимуму. Мы же часть этого времени отбираем на изучение инструмента для работы с сетями. Пусть этот инструмент написан на Питоне, но работать на Ансибле это не программирование. Это работать на ПО, которое само по себе нужно изучать. Мы же пришли изучать язык программирования.

Все остальные материалы мне понравились. Может быть надо можно углубить Jinja и TextFSM. Материала по ним немало, но мне показалось, что можно сделать побольше, т.к. шаблонизация в современном программировании очень важная вещь.

Трехмесячный курс по Питону закончился и сейчас в моей жизни происходит следующее: я поговорил с начальником отдела и получил задание, связанное с настройкой и допиливанием модуля на Питоне для нашей системы мониторинга. То есть я не просто сотрудник техподдержки, который умеет настраивать сетевое оборудование. Я теперь могу делать новые задачи, получать новый опыт и добавлять в "Избранное" много интересных вакансий. Слава великому Питону!

ЗЫ. Курсы с того образовательного портала надо бы пройти до конца - оплачено ведь. Но так неохота и лень 😊

Олег Востриков

Курс "Python для сетевых инженеров" подкупает обилием примеров, понятных сетевым инженерам. Т.е. название не маркетинговый ход, а отвечает содержанию. Материал подобран и скомпонован очень удачно, что позволит любому сетевику, независимо от уровня навыков программирования, начать погружение в Python и перейти от азов к решению реальных задач.

Лично мне курс пришелся как нельзя кстати, уже к середине курса я начал пытаться автоматизировать рабочие задачи. К окончанию курса это вылилось в настройку десятков тысяч устройств. Думаю не трудно представить, сколько времени потребовалось бы, чтобы повторить это вручную.

Отдельное спасибо Наташе за попытку привить слушателям "чистописание" кода и дополнительные варианты решенных задач, это помогает расширить кругозор и улучшить код как с эстетической, так и с практической точки зрения.

Эмиль Гарипов

Чтобы понять почему именно этот курс и почему именно с Наташей мне надо упомянуть о себе любимом. Я не разу не программист и никогда им не был. Программисты всегда вызывали у меня уважение и долю здоровой зависти, когда за пару часов работы могли реализовать все твои хотелки. У меня всегда возникала ужас, когда дело касалось программирования в школе, в университете или уже позже, когда я сам или с помощью других пытался освоить азы программирования вплоть до того момента как Наташа предложила пройти мне её курс.

Наташа особенный человек, человек - маяк, человек - наставник, который берёт за руку и ведёт легко и не принужденно сквозь все дебри ужасного для меня программирования. Здесь можно прыгнуть, здесь обойти, а здесь и вовсе отбросить. И что особенно меня поражает в Наташе, так это то, что она никогда не оставит без внимания ни один маломальский вопрос, всегда ответит, да ещё и с комментариями и различными вариантами ответов. И курс у неё выстроен также, как выложенная прямая красавая тропинка, но с препятствиями в виде интересных практических заданий, чтобы было весело и не скучно осваивать азы программирования. Кроме всего прочего в курсе даже учтены практически все инструменты для работы именно для начинающих. Первым занятием идёт подробное объяснение, как все устанавливать и начать работать. Для меня это оказалось чуть ли не самым важным, потому что до этого я никогда не сталкивался ни с гитом, ни с виртуальными окружением, ни со средой программирования.

Скажу честно, этот курс заставил мои мозги напрячься, но я давно не получал такого удовлетворения от процесса и самое главное избавился от фобии программирования. Рекомендую пройти этот курс и если есть какие-то сомнения, отбросить их, Вы не

только сможете, но и получите массу удовольствия.

Илья про книгу

Благодаря курсу Python для сетевых инженеров от Наташи Самойленко, я захотел сменить квалификацию на девелопера, уже успешно решил ряд рутинных рабочих задач, постоянно надоедавших своим однообразием. Все начиналось с простой статьи на xgu.ru, но потом это стало чем то большим.

Простота и грациозность описания автоматизации процессов Наташи Самойленко позволила мне открыть дверь в ранее недоступный модный "DevOps". В связи с этим, помимо развития своих профессиональных навыков, я так же получил значительный бонус на рынке труда в виде дополнительных знаний. Мне как человеку который изучал немного Delphi в университете, да и то не достаточно глубоко, было довольно интересно и увлекательно разбираться с новой для меня стязей. Подача материала крайне "легка" для восприятия, и наглядна. Хорошие и полезные в ежедневной работе примеры.

Спасибо Наташе за отличный курс

Алексей Кириллов про онлайн курс

Об этом курсе я узнал совершенно случайно. Наташа предложила моему непосредственному начальнику прочитать данный курс для подчиненных инженеров. Перед нашим отделом как раз стояла актуальная задача тестирования оборудования. После непродолжительного согласования мы приступили к обучению.

Для большинства из нас это было первое знакомство с python. Но благодаря отличной подаче материала, а так же заданиям с разным уровнем сложности, обучение проходило весьма интересно и продуктивно. К сожалению, не все темы нашли применение в нашей работе, но главная цель была достигнута - мы начали создавать систему автоматизированного тестирования. Причем эти знания пригодились не только для одной конкретной задачи, но также позволили решить множество рутинных задач. А из некоторых скриптов выросли отдельные проекты.

Дело за малым - интересом. Подход, предлагаемый Наташой помогает не лезть в дебри программирования, а дает инструмент для автоматизации (а кто не хочет иметь больше свободного времени:), который легок в понимании человеку, который до этого работал только с сетями. До этого курса я пытался изучать python по популярным книгам в интернете, но каждый раз это быстро заканчивалось из-за скучности и

непонимания как я могу это применить. В курсе же практически на каждую тему есть задачи, по которым Вы видите практическое применение того или иного объекта языка.

Слава Скороход про онлайн курс

О курсе я узнал из группы linkteur в соцсети. Движимый желанием попрактиковаться в программировании под руководством такого уважаемого в сетевых (и не только) кругах человека, как Наташа, записался на него, и впоследствии не пожалел. Имел за плечами небольшой опыт программирования на других языках, но с Python столкнулся впервые, что, впрочем, не составило трудностей в его освоении, по большей части из-за качественной подачи материала. Полагаю, что курс найдет интерес как у начинающих, так и у сетевиков/разработчиков с определенным багажом знаний, так как темы рассматриваются не только по верхам, но и с погружением в детали реализации – например, про GIL я точно не ожидал услышать

Считаю удачным подход с демонстрацией конкретных, «живых» сетевых, а не абстрактных примеров. Хоть мне и кажется, что важно в первую очередь понимание концепции, а область ее применения со временем подскажут рабочие задачи, но, когда видишь реальные приложения того или иного модуля – это может дополнительно направить мысль в продуктивное русло.

После курса в сознании очень четко выстраивается картина того, куда еще расти и на какие темы стоит обратить внимание. Область применения средств автоматизации не просто обширна, а необъятна, и всегда есть, что еще изучить, но если Вы хотели познакомиться с Python или автоматизацией в целом, но не знали, откуда подступиться – настоятельно рекомендую

В заключение хочу поблагодарить Наташу за проделанную работу. Это очень качественный слой информации на просторах рунета, который совершенно определенно может помочь как минимум избавиться от рутинных задач на текущем месте работы, а как максимум – может даже стать первым шагом на пути к смене квалификации.

Марат Сибгатулин про онлайн курс

Не могу сказать, что Python ворвался в мои трудовые будни и окрасил их в жёлтый и синий. Всё-таки прямого применения скрипtingу в моей сфере я не вижу. Нет, это было, скорее, очередное необходимое расширение зоны незнания.

Что действительно изменило мир - это регулярные выражения. Теперь, открывая 50 мегабайт текстовых логов в Notepad++, я надеюсь, что придётся придумать регулярку похитрее, чтобы извлечь максимум релевантных строк. Да и фильтрация вывода в консоли стала более гибкой и функциональной.

За это спасибо лабораторкам в курсе и тестам после лекции, где задач на регулярные выражения было с избытком. Вообще практика построена очень удачным образом - она опирается на то, что было в лекциях, но добавляет существенные детали. Получается, что выполняя задания, не только практикуешь известную теорию, но и с неожиданной стороны видишь то, что казалось понятным. Лабораторки с изюминкой.

Лекции тоже важны. Несмотря на то, что они чётко идут по галавам книги, в них огромное количество отступлений, комментариев и фирменного наташиного юмора.

Один совет: не стесняйтесь задавать вопросы. Лектору очень важно понимать, что аудитория, тем более онлайн, следит за ним.

Кирилл Плетнёв про книгу

Последние годы ни один настоящий сетевой инженер не мог не заметить звучащих тут и там таких "страшных" слов, как Network Automation, Network Programmability и Software Defined Networking. Для себя я давно решил, что нужно вливаться в струю не потому что, как часто говорят: "Вы устареете и не будете востребованы"; я думаю, "классический сетевой инженер" никуда не денется; но потому что, если Вы любите свою профессию, работать в современных реалиях и с современными инструментами - всегда очень интересно и приятно.

Не имея ни малейшего опыта программирования и послушав много мнений, я выбрал Python за то, что это язык общего назначения (т.е. применяется для решения самых разнообразных задач), он зрелый, хорошо поддерживается, имеет огромное комьюнити и как следствие очень много учебных материалов. Код Python легко пишется и читается, а подключение внешних библиотек обеспечивает представление результатов в любом желаемом формате. Также Python предпочтуется многими вендорами сетевого оборудования, которых включают Python API в некоторые версии своих операционных систем. Узнав, что Наташа Самойленко написала книгу и сделала онлайн-курс "Python для сетевых инженеров", я решил, что это будет отправной точкой в моём путешествии в мир Network Programmability.

Книга замечательно написана и любое предложение или абзац закреплены практическими примерами, что, как мне кажется, является наилучшим форматом обучающего материала для технаря. И примеров этих много, очень. Материал закрепляется особенно хорошо благодаря заданиям к каждой главе, которые всегда

заставляют, подперев лицо кулаком, серьёзно подумать над стратегией решения. Большое разнообразие заданий довольно быстро учит стараться писать код красиво (насколько это возможно для новичка), а в Python это означает гибко и эффективно.

Уже по прошествии 6-и глав (а это буквально пару недель занятий) , мне подвернулись две рабочих задачи, которые были решены в кратчайшие сроки:

1. Перенос конфигурации NAT для более 1500 трансляций с Cisco IOS на FortiGate (т.е. абсолютно другой формат конфига).
2. И проверка работы системы фильтрации web-запросов.

Не знаю, сколько бы я колупался в Notepad++ в первом случае или выборочно открывал странички из присланного заказчиком списка, но благодаря Python и конкретно курсу Наташи, каждая из задач была решена менее, чем за день. Перед тем, как взяться за курс, я и представить не мог, насколько Python удобен и прост в освоении.

В комплекте с курсом идёт тёплый приём в команде в Slack, где всегда можно спросить совета или поделиться своими наработками.

В 1001-й раз хочу поблагодарить Наташу за её замечательный и нужный труд, ежедневную помощь и ангельское терпение к моим "грязным" решениям =) Ни один мой вопрос не остался без ответа!

Разумеется один курс и 2 месяца вашей жизни не сделают из Вас Киану Ривза в Python, но его более чем достаточно, чтобы понять, какие возможности он может Вам дать. И как по мне, это безумно интересно =)

Алексей про книгу

Я считаю, что это отличная книга, для людей которые направлены на результат. Написанная инженером, для инженеров. Всё чётко и лаконично. Прочёл главу и приступил к работе.

Очень много практики и это замечательно, если всю её делать, то Вы обязательно сможете закрепить весь пройденный материал. Что не мало важно, практика может быть использована в реальной жизни и то, что Вы напишите, скорее всего можно будет использовать в реальной жизни для реальных проектов.

Да, наверное, можно было бы или ещё меньше написать или наоборот "воды налить", но я считаю, что человеку которому это нужно для конкретных задач и который уже с чем-то подобным был знаком ранее это хороший вариант и отличный старт познания Python'a. Для меня эта книга сейчас как шпаргалка. Всего в голове не удержишь, а так я точно знаю, где и что быстро найти.

Я не скажу, что я сразу кинулся всё автоматизировать, но определённое количество задач я уже реализовал.

Спасибо большое Наташе за её труд и доступное изложение информации.