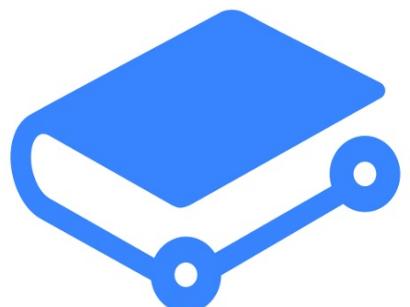


Python 2.7

для сетевых инженеров

Наташа Самойленко

Опубликовано
с помощью **GitBook**



Содержание

Введение	1.1
О курсе	1.2
Как учиться по этому курсу	1.3
Пример плана обучения	1.4
FAQ	1.5
Благодарности	1.6
1. Подготовка к работе	1.7
ОС и редактор	1.7.1
Система управления пакетами pip	1.7.2
virtualenv, virtualenvwrapper	1.7.3
Интерпретатор Python (проверка)	1.7.4
Задания	1.7.5
2. Начало работы с Python	1.8
Синтаксис Python	1.8.1
Интерпретатор Python. iPython	1.8.2
Магия iPython	1.8.2.1
Переменные	1.8.3
3. Типы данных в Python	1.9
Числа	1.9.1
Строки (Strings)	1.9.2
Полезные методы для работы со строками	1.9.2.1
Форматирование строк	1.9.2.2
Список (List)	1.9.3
Полезные методы для работы со списками	1.9.3.1
Варианты создания списка	1.9.3.2
Словарь (Dictionary)	1.9.4
Полезные методы для работы со словарями	1.9.4.1
Варианты создания словаря	1.9.4.2
Словарь из двух списков (advanced)	1.9.4.2.1
Кортеж (Tuple)	1.9.5

Множество (Set)	1.9.6
Полезные методы для работы с множествами	1.9.6.1
Операции с множествами	1.9.6.2
Варианты создания множества	1.9.6.3
Преобразование типов	1.9.7
Проверка типов	1.9.8
Задания	1.9.9
4. Создание базовых скриптов	1.10
Передача аргументов скрипту	1.10.1
Ввод информации пользователем	1.10.2
Задания	1.10.3
5. Контроль хода программы	1.11
if/elif/else	1.11.1
for	1.11.2
Вложенные for	1.11.2.1
Совмещение for и if	1.11.2.2
Итератор enumerate	1.11.2.3
while	1.11.3
break, continue, pass	1.11.4
for/else, while/else	1.11.5
Работа с исключениями try/except/else/finally	1.11.6
Задания	1.11.7
6. Работа с файлами	1.12
Открытие файлов	1.12.1
Чтение файлов	1.12.2
Запись файлов	1.12.3
Закрытие файлов	1.12.4
Конструкция with	1.12.5
Задания	1.12.6
7. Функции	1.13
Создание функций	1.13.1
Пространства имен. Области видимости	1.13.2
Параметры и аргументы функций	1.13.3
Типы параметров	1.13.3.1

Типы аргументов	1.13.3.2
Аргументы переменной длины	1.13.3.3
Распаковка аргументов	1.13.3.4
Пример использования	1.13.3.5
Задания	1.13.4
8. Модули	1.14
Импорт модуля	1.14.1
Создание своих модулей	1.14.2
Задания	1.14.3
9. Регулярные выражения	1.15
Модуль re	1.15.1
Специальные символы	1.15.2
Жадность регулярных выражений	1.15.3
Группировка выражений	1.15.4
Разбор вывода команды show ip dhcp snooping с помощью именованных групп	1.15.4.1
Задания	1.15.5
10. Сериализация данных	1.16
CSV	1.16.1
JSON	1.16.2
YAML	1.16.3
Задания	1.16.4
11. Работа с базами данных	1.17
SQL	1.17.1
SQLite	1.17.2
Основы SQL (в sqlite3 CLI)	1.17.3
CREATE	1.17.3.1
DROP	1.17.3.2
INSERT	1.17.3.3
SELECT	1.17.3.4
WHERE	1.17.3.5
ALTER	1.17.3.6
UPDATE	1.17.3.7
REPLACE	1.17.3.8

DELETE	1.17.3.9
ORDER BY	1.17.3.10
AND, OR, NOT, IN	1.17.3.11
Модуль sqlite3	1.17.4
Пример использования SQLite	1.17.4.1
Задания	1.17.5
12. Подключение к оборудованию	1.18
Ввод пароля	1.18.1
Реҳпест	1.18.2
Telnetlib	1.18.3
Paramiko	1.18.4
Netmiko	1.18.5
Возможности netmiko	1.18.5.1
Одновременное подключение к нескольким устройствам	1.18.6
Модуль threading	1.18.6.1
Модуль multiprocessing	1.18.6.2
Задания	1.18.7
13. Шаблоны конфигураций с Jinja	1.19
Пример использования Jinja	1.19.1
Пример использования Jinja с корректным использованием программного интерфейса	1.19.2
Синтаксис шаблонов Jinja2	1.19.3
Контроль символов whitespace	1.19.3.1
Переменные	1.19.3.2
for	1.19.3.3
if/elif/else	1.19.3.4
Фильтры	1.19.3.5
Тесты	1.19.3.6
Присваивание (set)	1.19.3.7
Include	1.19.3.8
Наследование шаблонов	1.19.4
Задания	1.19.5
14. TextFSM. Обработка вывода команд	1.20
Синтаксис шаблонов TextFSM	1.20.1

Примеры использования TextFSM	1.20.2
CLI Table	1.20.3
Задания	1.20.4
15. Ansible	1.21
Основы Ansible	1.21.1
Инвентарный файл	1.21.1.1
Ad-Hoc команды	1.21.1.2
Конфигурационный файл	1.21.1.3
Модули	1.21.1.4
Основы playbook	1.21.2
Переменные	1.21.2.1
Результат выполнения модуля	1.21.2.2
Сетевые модули	1.21.3
ios_command	1.21.3.1
ios_facts	1.21.3.2
ios_config	1.21.3.3
lines (commands)	1.21.3.3.1
parents	1.21.3.3.2
Отображение обновлений	1.21.3.3.3
save	1.21.3.3.4
backup	1.21.3.3.5
defaults	1.21.3.3.6
after	1.21.3.3.7
before	1.21.3.3.8
match	1.21.3.3.9
replace	1.21.3.3.10
src	1.21.3.3.11
ntc_ansible	1.21.3.4
Подробнее об Ansible	1.21.4
Задания	1.21.5
Дополнительная информация	1.22
Полезные модули	1.22.1
Модуль subprocess	1.22.1.1

Модуль os	1.22.1.2
Модуль argparse	1.22.1.3
Модуль ipaddress	1.22.1.4
Полезные функции	1.22.2
Функция sorted	1.22.2.1
Функция lambda	1.22.2.2
Функция zip	1.22.2.3
Функция map	1.22.2.4
Функция filter	1.22.2.5
Функции any и all	1.22.2.6
Соглашение об именах	1.22.3
Подчеркивание в именах	1.22.3.1
Полезные ссылки	1.23
Материалы по темам курса	1.23.1
Продолжение обучения	1.23.2
Отзывы	1.24
Словарь терминов	1.25

Python 2.7 для сетевых инженеров

Автор: Наташа Самойленко

В этой версии книги используется Python 2.7

[Перейти на версию для Python 3](#)

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 разы больше литературы, форумов, блогов
- практически для любого вопроса/проблемы есть решение, если вы ввели запрос в гугл на английском

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 сетевым инженером и без знания английского. Английский просто дает возможности, но он не является обязательным требованием. Аналогично и с программированием.

О курсе

Если в двух словах, то это такой “CCNA” по питону. С одной стороны, курс достаточно базовый, чтобы его мог одолеть любой желающий, с другой стороны, в курсе рассматриваются все основные темы, которые позволяют дальше расти самостоятельно.

Курс не ставит своей целью глубокое рассмотрение языка Python. Задача курса, объяснить понятным языком основы Python и дать необходимые инструменты для его практического использования.

Всё, что рассматривается в курсе, ориентировано на сетевое оборудование и работу с ним. Это дает возможность сразу использовать в работе то, что было изучено на курсе.

Все примеры показываются на примере оборудования Cisco. Но, конечно же, аналогичные принципы применимы для любого другого оборудования.

Для кого этот курс

Для сетевых инженеров с опытом программирования и без.

Все примеры и домашние задания будут построены с уклоном на сетевое оборудование.

Этот курс будет полезен для сетевиков, которые хотят автоматизировать задачи с которыми сталкиваются каждый день и/или хотели заняться программированием, но не знали с какой стороны подойти.

Ещё не решили нужен ли курс? Почитайте [отзывы](#).

Обсуждение

Для обсуждения книги и заданий используется [slack](#). Обсуждения на GitBook закрыты.

Все вопросы, предложения и замечания по книге также пишите в Slack



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

О курсе

Краткое содержание курса

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

Если вы скачали PDF и испугались, что в курсе почти 600 страниц, не переживайте. Половину курса занимают разделы 12-15. В них много выводов, которые показывают результаты выполнения скриптов, поэтому объем получился большой.

Курс всегда будет оставаться бесплатным. Поэтому вам не нужно переживать, что он будет удален.

ОС, Python

В курсе используются:

- Debian Linux
- Python 2.7

Позже курс будет переведен на Python 3.x.

Для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

Примеры

Все примеры, которые используются в курсе, можно скачать одним архивом в [zip](#) или [tar.gz](#).

Примеры, которые рассматриваются в разделах курса, являются обучающими. Это значит, что они не обязательно показывают лучший вариант решения задачи, так как они основаны только на той информации, которая рассматривалась в курсе.

Кроме того, довольно часто, примеры, которые давались в разделах, развиваются в заданиях. То есть, вам нужно будет сделать более хорошую, универсальную, правильную версию кода.

Если есть возможность, лучше набирать код, который используется в курсе, самостоятельно. Или, как минимум, скачать примеры и попробовать что-то в них изменить. Так информация будет лучше запоминаться.

Если такой возможности нет, например, потому что вы читаете курс в дороге, можно попробовать повторить примеры самостоятельно позже.

Но, в любом случае, обязательно нужно делать задания.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом в [zip](#) или [tar.gz](#).

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.

Однако, если получается, лучше делать задания по порядку.

В курсе специально не выложены ответы на задания. К сожалению, когда есть ответы, очень часто, вместо того, чтобы попытаться решить сложное задание самостоятельно, подглядывают в ответы.

Конечно, иногда возникает ситуация, когда никак не получается решить задание. Тогда попробуйте отложить его и сделать какое-то другое.

На [stackoverflow](#) есть ответы практически на любые вопросы. Так что, если гугл отправил вас туда, значит, с большой вероятностью, 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 как более правильно/короче выполнить его. Но, на этот счет, тоже не переживайте, так как, скорее всего, в следующих разделах встретится пример, в котором будет показано, как писать такой код.

Презентации

Для части тем курса есть презентации. По ним удобно быстро просматривать информацию и повторять. Они особенно полезны для базовых тем.

Скачать все презентации одним архивом.

Со временем, будут добавлены презентации для всех разделов.

Обновление курса

Курс был полностью завершен 07.02.2017.

Но, в раздел [дополнительная информация](#) ещё будут добавлены несколько подразделов. Также, скорее всего, будут добавлены новые задания.

Поэтому, если вы не будете читать курс в ближайшее время, то лучше сохраните ссылку на онлайн версию курса, а не PDF/mobi/epub. А, когда решите читать, скачаете, если нужно, свежую версию.

Дополнения будут делаться в ближайшие 2-6 месяцев и не будут влиять на изначальную структуру курса.

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

Форматы курса

Курс можно читать в нескольких форматах:

- онлайн
- PDF/ePub/Mobi

Они автоматически обновляются, поэтому всегда содержат одинаковую информацию.

Пожалуйста, не выкладывайте скачанные версии курса. Вместо этого, просто давайте ссылку на курс.

Комментарии

GitBook позволяет оставлять комментарии у соответствующей части курса. Для этого нужно зарегистрироваться на GitBook и зайти под своим аккаунтом.

В онлайн версии курса, если навести курсор на какой-то абзац текста, справа появится плюс:

Пример использования модуля `threading` для подключения по SSH с помощью `netmiko` (файл `netmiko_threading.py`):

```
from netmiko import ConnectHandler
import sys
import yaml
import threading
```

Если нажать на него, откроется окно для набора сообщения:

Пример использования модуля `threading` для подключения по SSH с помощью `netmiko` (файл `netmiko_threading.py`):

```
from netmiko import ConnectHandler
import sys
import yaml
import threading
```

+ Start a new discussion
Post

После публикации, сообщение находится около конкретного текста в курсе и другие читатели могут прочесть его. Если это был комментарий об опечатке, я удаляю его, после того, как правки сделаны.

Если же комментарий был о непонятном моменте в курсе, комментарий останется, вместе с моими пояснениями.

Комментарии, соответственно, доступны в онлайн версии курса, рядом с текстом:

Пример использования модуля `threading` для подключения по SSH с помощью `netmiko` (файл `netmiko_threading.py`):

```
from netmiko import ConnectHandler
import sys
import yaml
import threading
```

1

Наташа Самойленко
Тут точно должен быть аргумент
function?
th = threading.Thread(target =
function, args = (device, command))

[Comment](#) [Close](#) [New Thread](#)

А также, на главной странице курса:

The screenshot shows a discussion thread on the GitBook platform. At the top, there's a navigation bar with the GitBook logo, a 'WE ARE HIRING!' button, 'Pricing', 'Explore', 'About', 'Blog', 'Sign In', and a 'Sign Up' button. Below the navigation, it says 'natenka > PyNEng' and 'Updated 2 hours ago'. There are tabs for 'ABOUT' and '2 DISCUSSIONS'. The 'DISCUSSIONS' tab is active, showing a single discussion post by 'Наташа Самойленко' started '2 minutes ago'. The post content is: "Пример использования модуля threading для подключения по SSH с помощью netmiko (файл netmiko_threading.py):" followed by the code snippet. To the right of the post is a 'Notifications' sidebar with a message: "You're not receiving notifications from this thread." and a '1 participant' section with a user icon.

Как учиться по этому курсу

1 вариант:

Этот вариант подходит в таких случаях:

- Если вы не уверены, что хотите проходить весь курс
- Если вам нужно просто автоматизировать подключение к устройствам и выполнение команд
- Если вы не уверены, что хотите изучать Python

В этом случае, начните с [Ansible](#).

Для его установки и базового использования не требуется знать Python. Достаточно установить Ansible, взять пример использования и попробовать выполнить команды show на оборудовании.

После этого, можно использовать Ansible для отправки команд, которые не влияют на передачу трафика:

- подпись интерфейсов
- настройка общих параметров, таких как, например, alias
- генерация конфигурации по шаблону, для первичной настройки оборудования

Если какие-то задачи не получается решить с Ansible, или вы просто захотите изучить Python, переходите к варианту 2.

2 вариант: Если вы уже решили изучать Python

Если вы уже решили изучать Python и примерно представляете, что он может дать, то можно просто взять [расписание](#) курса и идти по нему.

И хотя многие вещи, особенно в начале курса, будут базовыми, не пропускайте их, если вы их не знаете. Эти базовые структуры данных, управляющие структуры, будут основой всего, что мы будем изучать дальше.

Обязательно практикуйтесь:

- **пробуйте воспроизвести все примеры в главах (набирайте примеры вручную или пробуйте повторять по памяти)**
- **пробуйте менять какие-то параметры в примерах, чтобы посмотреть "а можно ли так сделать"**
- **обязательно выполняйте все задания после глав**

План занятий можно изменить под себя, но не стоит делать большие перерывы между темами и не стоит заниматься слишком много:

- если делать большие перерывы, то всё, что изучено раньше, начнет забываться
- если торопиться, то скорее всего, всё сведется к чтению текста, без выполнения заданий. В таком случае, от курса будет очень мало пользы

Если следовать расписанию, пусть и с небольшими изменениями, то к концу курса наберется два месяца практики. За это время все базовые понятия хорошо усваются и, главное, после завершения курса, не бросать практиковаться и учиться.

По ходу курса, скорее всего, будут возникать идеи, что сделать по работе. Это отлично! Делайте 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 будут развиваться: будет добавляться функционал, будут улучшения, которые нужны именно вам.

Если же вы чувствуете, что читаете, но пока никаких идей нет, ничего страшного. В курсе достаточно упражнений, с которыми вы сможете потренироваться писать код. И которые, весьма вероятно, натолкнут на какие-то идеи.

После курса

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 на 2-4 недели, то большая часть из них существенно выветрится.

Если вам удалось напридумывать себе задач, по ходу курса - отлично. Реализуйте их. Напишите список и делайте задачи постепенно. Это отличный способ изучать дальше и повторять пройденное.

Идеи сами будут двигать вас дальше, вы будете изучать новые темы, новые возможности естественно, по ходу развития ваших программ.

Создайте репозиторий на [github](#) и выкладывайте туда свои скрипты. Дорабатывайте их, поделитесь с коллегами.

Отличный способ запомнить лучше какую-то тему - рассказать другому.

Если вы хотите и дальше развиваться в этой теме, продолжайте учиться, читать. В разделе [ресурсы](#) я подготовила различные ссылки на видео, книги, курсы, блоги и задачки по Python. Выберите тот ресурс, который вам нравится и изучайте Python

далее.

И обязательно практикуйтесь. Только читать книгу/статью или смотреть видео, не достаточно. Обязательно пишите что-то.

Пример плана обучения

Тут находится пример плана, по которому можно учиться по этому курсу.

В плане всё разбито по неделям. Можно идти быстрее или медленнее, в зависимости от уровня и наличия времени. По возможности, попробуйте придерживаться темпа, возможно, он вам подойдет.

Например, 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 Типы данных в Python Создание базовых скриптов	Типы данных Базовые скрипты
2	Контроль хода программы Работа с файлами	Контроль хода программы Файлы
3	Функции	Функции
4	Модули Регулярные выражения	Модули Регулярные выражения
5	Сериализация данных	Сериализация данных
6	Работа с базами данных	Базы данных
7	Подключение к оборудованию	Подключение к оборудованию
8	Шаблоны конфигураций с Jinja	Jinja2
9	Обработка неструктурированного вывода команд с TextFSM	TextFSM
10	Ansible	Ansible

Чем это отличается от обычного вводного курса по питону?

- тут основы даются достаточно коротко
- подразумевается определенная предметная область знаний (опыт работы с сетевым оборудованием)
- все примеры, по возможности, ориентированы на сетевое оборудование

Я сетевик. Для чего мне нужен этот курс?

В первую очередь, для автоматизации рутинных задач.

Автоматизация дает несколько преимуществ:

- дает возможность мыслить более высокоуровнево
 - когда вы свободны от рутинной работы, проще подняться над всем
 - у вас появляется время и возможность думать об улучшениях
- доверие. Вы не боитесь делать каждое изменение
 - каждое изменение как правило, сопряжено с риском, так как сеть это основа работы всех приложений и цена ошибки, как правило, высока
- вы получаете консистентную конфигурацию, от настроенных пользователей и подписей интерфейсов, до функционала безопасности
 - вы можете не переживать о том, не забыли ли вы что-то

Конечно, не будет такого, что после курса вы "всё автоматизируете и наступит счастье". Но это шаг в эту сторону.

И 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, даст сразу довольно много.

Зачем тогда учить python? Проблема в том, что тот же Ansible не решит все вопросы. И, возможно, вам понадобится добавить какой-то функционал самостоятельно.

И, кроме непосредственной настройки оборудования, есть ежедневные рутинные задачи, которые можно автоматизировать с помощью Python.

Скажем так, если вы не хотите разбираться с Python, но хотите автоматизировать процесс настройки/работы с оборудованием, посмотрите на Ansible. В любом случае, даже из коробки, он уже будет очень полезен.

Если же вы потом войдете во вкус и захотите добавить что-то свое, чего нет в Ansible, возвращайтесь :)

И еще, этот курс не только о том как использовать Python для подключения на оборудование и настройки его. 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 и подобные задачи.

Почему "для сетевых инженеров"?

Есть несколько причин:

- сетевые инженеры уже обладают опытом работы в ИТ и часть концепций им знакома.
 - скорее всего какие-то основы программирования большинству уже будут знакомы. А это значит, что будет гораздо проще разобраться и с Python.
- работа в командной строке и написание скриптов, вряд ли кого-то из них испугает
- у сетевых инженеров есть знакомая им предметная область, на которую можно опираться, при составлении примеров и заданий.
 - Если рассказывать на примерах о котиках и зайчиках, это одно. Но когда в примерах есть возможность использовать идеи из предметной области, все становится проще, рождаются конкретные идеи как улучшить какую-то программу, скрипт. А когда человек пытается ее улучшить, он начинает разбираться с чем-то новым. И это очень сильно помогает продвигаться вперед

Почему Python?

- В контексте работы с сетевым оборудованием часто используется именно python.
 - В некотором оборудовании python встроен или есть API, которые поддерживают Python
- Python достаточно прост для изучения.
 - Конечно, это всё относительно и кому-то более простым может быть другой язык, но скорее это будет из-за 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 скрипtingа, но и для разработки приложений. Конечно, для этого курса это не важно, но, по крайней мере, вы потратите время на язык, который позволит вам легко шагнуть дальше, чем простые скрипты

- из сетевой сферы, например, на Python написан [GNS3](#)

И еще один момент, в контексте курса, Python нужно рассматривать не как единственно правильный вариант, не как "правильный" язык. Нет, это просто инструмент.

Инструмент как отвертка, например. Мы учимся им пользоваться, для конкретных задач.

То есть, никакой идеологической подоплеки тут нет. Никакого "только Python" тоже. Никакого "поклонения" тем более.

Странно поклоняться отвертке :)

Тут всё просто. Есть хороший удобный инструмент, который подойдет к разным задачам. Он не лучший во всем и далеко не единственный язык в принципе.

Начните с него. Потом вы сможете самостоятельно выбрать что-то другое, если захотите. Эти знания всё равно не пропадут.

Почему python 2, а не python 3.x?

Опять про отвертку :)

Есть отвертка обычная (python 2) и новая (python 3), классная, с улучшенным дизайном, более безопасная, красивая и так далее.

Для наших задач подойдет и обычная :)

Если вы пишете приложение, а не просто пару скриптов, то конечно лучше писать на python 3.x.

Грубо говоря, нам рано о таком думать.

А если серьезно, то, к сожалению, на данный момент не все проекты поддерживают python 3. Поэтому, пока что, в курсе будет использоваться python 2.

В дальнейшем, скорее всего, курс будет переделан на python 3.x.

Я не знаю нужно ли мне это.

Я конечно же считаю, что нужно :) Иначе, я бы не писала этот курс.

Но да, совсем не факт, что вам захочется погружаться в это всё. Поэтому, для начала, попробуйте разобраться с [Ansible](#). Возможно, вам хватит его достаточно надолго.

Начните с простых команд show. Попробуйте подключиться сначала к тестовому оборудованию (виртуалкам).

Затем попробуйте выполнить команду show на реальной сети, на 2-3 устройствах. Потом на большем количестве.

Если вам этого достаточно, можно остановиться тут.

Следующим шагом я бы попробовала использовать 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 и подобным. А потому, что сетевой инженер постоянно сталкивается с:

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

А, на данный момент, большое количество оборудования, по-прежнему, предлагает нам только интерфейс командной строки и не структурированный вывод команд. Управляющий софт часто ограничен вендором, дорого стоит и имеет урезанные возможности.

И 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, уже не всегда просто сделать. Да, 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.

И да, я не считаю, что всем надо бежать учиться программировать. Но считаю, что для инженера это очень важный навык. Да, для **инженера**. Не для всех на свете.

Сейчас явно наблюдается тенденция "Все учимся программировать". И это хорошо, в целом. Но программирование это не что-то элементарное. Это сложно, в это нужно вкладывать много времени, особенно, если вы никогда не имели отношения к инженерному/техническому миру.

А то иногда складывается впечатление, что достаточно пройти "вот эти вот курсы" и через 3 месяца вы крутой программист с высокой зарплатой. Нет, этот курс не об этом :)

Мы не говорим в этом курсе о программировании как профессии и не ставим такую цель. Мы говорим о программировании как инструменте, таком как: знание CLI unix, например.

И дело не в том, что инженеры какие-то особенные. А просто, как правило:

- они уже имеют техническое образование
- многие работают, так или иначе в командной строке
- сталкивались, как минимум, с каким-то языком программирования
- у них, так сказать, "инженерный склад ума"

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

Курс будет когда-то платным?

Нет, курс, который опубликован тут, всегда будет бесплатным.

Я читаю платно [онлайн курс "Python для сетевых инженеров"](#). Но это не будет влиять на этот курс. Он всегда будет бесплатным.

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!

Подготовка к работе

До начала работы с Python, нужно убедиться, что Python установлен в операционной системе.

Если вы используете Linux/Unix/Mac OS, то, скорее всего, Python уже установлен. И нужно только проверить, что установлена версия 2.7 (которая используется в курсе).

Если вы используете Windows, то, скорее всего, Python нужно будет установить. Один из самых простых вариантов для Windows, установить окружение [Anaconda](#). В окружении также есть IDE Spyder, который можно использовать вместо редактора.

Для установки Ansible понадобится Linux.

Ещё один важный момент - выбор редактора. В следующем разделе приведены примеры редакторов для разных ОС.

Вместо редактора можно использовать IDE ([Integrated development environment](#)).

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. Как правило, для этого может потребоваться установить дополнительные модули.

В начале работы, может получиться так, что IDE будет только отвлекать обилием возможностей.

Список IDE для Python можно можно посмотреть [тут](#). Например, можно выбрать [PyCharm](#) или (для Windows) Spyder.

Идеальным вариантом для курса будет использование виртуалки под управлением Linux/Unix. Это не обязательно, но многие вещи будут намного удобней. Плюс, не надо будет переживать за свою ОС, за то, чтобы не сделать чего-то лишнего.

ОС и редактор

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

Можно выбрать любую ОС и любой редактор. Но, желательно использовать Python версии 2.7, так как в этом курсе будет использоваться эта версия.

Для курса желательно использовать не Windows, так как, например, Ansible можно установить только на Linux/Unix.

Так как почти для всех разделов, подойдет любая ОС, можно отложить вопрос до раздела по Ansible.

В курсе используются:

- Debian Linux
- vim (редактор не имеет принципиального значения, лучше выбрать наиболее удобный)
- Python 2.7

Установка Python 2.7, если его нет в ОС, выполняется самостоятельно.

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

- Linux:
 - gEdit
 - nano
 - Sublime Text
 - geany
- Mac OS
 - TextMate
 - TextWrangler
- Windows:
 - Notepad++

Для начала работы, можно взять первый из списка для соответствующей операционной системы.

Далее выводы команд, интерпретатора, скриптов, выполняются на Debian Linux. В других ОС вывод может незначительно отличаться.

Система управления пакетами pip

Для установки пакетов Python, которые понадобятся в курсе, будет использоваться **pip**.

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 pip:

```
$ pip --version  
pip 1.1 from /usr/lib/python2.7/dist-packages (python 2.7)
```

Если команда выдала ошибку, значит pip не установлен. Установить его можно так:

```
sudo apt-get install python-pip
```

virtualenv, virtualenvwrapper

virtualenv - это инструмент, который позволяет создавать виртуальные окружения.

Виртуальные окружения:

- позволяют изолировать различные проекты
- зависимости, которых требуют разные проекты, находятся в разных местах
 - Например, если в проекте 1 требуется пакет версии 1.0, а в проекте 2 требуется тот же пакет, но версии 3.1
- пакеты, которые установлены в виртуальных окружениях, не перебивают глобальные пакеты

В курсе используется **virtualenvwrapper**: он позволяет немного проще работать с virtualenv.

Установка virtualenvwrapper с помощью pip:

```
sudo pip install virtualenvwrapper
```

После установки, в `.bashrc` нужно добавить несколько строк

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

`WORKON_HOME` - указывает расположение виртуальных окружений. А вторая строка - где находится скрипт, установленный с пакетом `virtualenvwrapper`:

Для того чтобы скрипт `virtualenvwrapper.sh` выполнился и можно было работать с виртуальными окружениями, надо перезапустить bash. Например, таким образом:

```
exec bash
```

Такой вариант может быть не всегда правильным. Подробнее в ответе на [stackoverflow](#).

Работа с виртуальными окружениями

Создание нового виртуального окружения:

```
$ mkvirtualenv PyNEng
New python executable in PyNEng/bin/python
Installing distribute.....done.
Installing pip.....done.
(PyNEng)$
```

В скобках перед стандартным приглашением отображается имя проекта (виртуального окружения). Это означает, что мы находимся в этом виртуальном окружении.

В `virtualenvwrapper` по Tab работает автопродолжение имени виртуального окружения.

Это особенно удобно в тех случаях, когда виртуальных окружений много.

Теперь в том каталоге, который был указан в `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
```

Выйти 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, используется команда **rmvirtualenv**:

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

Посмотреть какие пакеты установлены в виртуальном окружении:

```
(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  
pexpect  
pexpect-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
```

Зависимости (requirements)

При работе над проектом, со временем, в виртуальном окружении находится всё больше установленных пакетов. И, при необходимости поделиться проектом или скопировать его на другой сервер, нужно будет заново установить все зависимости.

В Python есть возможность (и принято) записать все зависимости в отдельный файл.

Это делается таким образом:

```
(PyNEng)$ pip freeze > requirements.txt
```

Теперь в файле requirements.txt находятся все зависимости, с версиями пакетов (файл requirements.txt):

```
Jinja2==2.8
pexpect==4.0.1
tornado==4.3
...
```

После этого, при необходимости установить все зависимости, надо перейти в новое виртуальное окружение, которое вы создали на сервере или просто на новый сервер и дать команду:

```
$ pip install -r requirements.txt
```

Тут в приглашении нет имени виртуального окружения, так как устанавливать таким образом зависимости можно не только в виртуальном окружении, но и в системе в целом.

Установка пакетов

Например, установим в виртуальном окружении пакет simplejson.

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

Если перейти в ipython (рассматривается в разделе [ipython](#)) 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 виртуально 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 (проверка)

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

```
$ python
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Вывод показывает, что установлен Python 2.7. Приглашение **>>>** это стандартное приглашение интерпретатора Python.

Вызов интерпретатора выполняется по команде **python**, чтобы выйти нужно набрать **quit()** либо нажать **Ctrl+D**

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 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 курсе ориентированы на Linux (Debian), желательно использовать его
 - желательно использовать новую виртуальную машину, чтобы было спокойней экспериментировать
- установить Python 2.7
 - проверить, что Python и pip установлены
- создать виртуальное окружение, в котором вы будете работать весь курс
- определиться с редактором/IDE
- начиная с раздела 12, мы будем подключаться к оборудованию. Поэтому вам нужно подготовить виртуальное или реальное оборудование

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 "Тут можно выполнить ещё что-то"
else:
    print "B больше или равно A"
    print "Продолжаем тут же"

print "The End"
```

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 написать комментарий. Например, чтобы описать особенности работы кода.

Комментарии в Python могут быть однострочными:

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

Однострочные 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
```

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 (если виртуальная машина устанавливается с нуля):

```
sudo apt-get install python-pip python-dev
sudo pip install --upgrade pip
sudo pip install ipython
```

Если с установкой ipython возникают проблемы, после которых не работает pip, выполните такие действия:

Если ставили через apt pip, то надо его удалить:

```
sudo apt-get remove python-pip
```

После этого заново переустановить pip и установить ipython таким образом:

```
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
sudo apt-get install python-dev
sudo pip install ipython
```

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 подписыны:

- 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
```

print

Оператор `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
```

По умолчанию, в конце выражения будет перевод строки. Если необходимо, чтобы после вывода выражения не было перевода строки, то в конце команды ставится запятая (обратите внимание, что в интерпретаторе это незаметно. Поэтому лучше это попробовать в файле).

Допустим, есть файл `file4.py` (обратите внимание на запятую после первой строки):

```
print 'Hello!',
print 'Hello!'
```

Тогда при выполнении скрипта, вывод будет выглядеть так (после первого Hello нет перевода строки, но есть пробел):

```
Hello! Hello!
```

Как создавать скрипты рассматривается в разделе [Создание базовых скриптов](#).

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

Типы данных в Python

В Python есть несколько стандартных типов данных:

- Numbers (числа)
- Strings (строки)
- Lists (списки)
- Dictionary (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

Пример различных типов числовых значений:

- int (40, -80)
- float (1.5, -30.7)
- long (52934861L)

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
```

Обратите внимание на отличия деления int и float:

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

```
Out[5]: 3
```

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

```
Out[6]: 3.3333333333333335
```

```
In [7]: 10 / float(3)
```

```
Out[7]: 3.3333333333333335
```

```
In [8]: float(10) / 3
```

```
Out[8]: 3.3333333333333335
```

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 исчисления:

```
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]: 'o'
```

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

Так как конфигурационный файл - это просто текстовый файл, при автоматизации, очень часто надо будет работать со строками.

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

Чаще всего эти методы будут применяться в циклах, при обработке файла.

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]: 'fAScTeTHERNET'  
  
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
```

Так происходит из-за того, что строка это неизменяемый тип данных.

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'
```

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'
```

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 используется пробел. 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 [51]: string1 = 'switchport trunk allowed vlan 10,20,30,100-200'

In [52]: string1.split()
Out[52]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30,100-200']
```

Еще один пример:

```
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.

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 `%`.

Пример использования метода `format`:

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

Аналогичный пример с оператором `%`:

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

В форматировании строк специальные символы указывают какой тип данных будет подставляться:

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

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

In [5]: print "{:>15} {:>15} {:>15}".format(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

In [7]: print "{:15} {:15} {:15}".format(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

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

Конвертировать в двоичный формат, указать сколько цифр должно быть в отображении числа и дополнить недостающее нулями:

```
In [10]: '{:08b}'.format(10)
Out[10]: '00001010'
```

Аналогичный результат можно получить с помощью метода zfill:

```
In [11]: bin(10)[2:].zfill(8)
Out[11]: '00001010'
```

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

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

join()

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

```
In [16]: vlans = ['10', '20', '30', '100-200']

In [17]: ','.join(vlans[:-1])
Out[17]: '10,20,30'
```

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']
```

extend()

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

```
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']

In [25]: vlans + vlans2
Out[25]: ['10', '20', '30', '100-200', '300', '400', '500', '300', '400', '500']

In [26]: vlans
Out[26]: ['10', '20', '30', '100-200', '300', '400', '500']
```

Но при этом метод `extend()` расширяет список "на месте", а при операции сложения выводится итоговый суммарный список, но исходные списки не меняются.

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']
```

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']
```

Словарь (Dictionary)

Словари - это изменяемый, неупорядоченный тип данных (к слову, 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 неупорядоченны, 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 указывать другое значение, вместо `None`:

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

```
Ooops
```

keys(), values(), items()

Методы **keys()**, **values()**, **items()**:

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

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

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

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

del

Удалить ключ и значение:

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

```
In [29]: del(london['name'])
```

```
In [30]: london
Out[30]: {'location': 'London Str', 'vendor': 'Cisco'}
```

Варианты создания словаря

Словарь можно создать с помощью литерала:

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

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'}
```

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, None)

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

И последний метод создания словаря - **генераторы словарей**. Сгенерируем словарь с нулевыми значениями, как в предыдущем примере:

```
In [16]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

In [17]: d = {x: None for x in d_keys}

In [18]: d
Out[18]:
{'IOS': None,
 'IP': None,
 'hostname': None,
 'location': None,
 'model': None,
 'vendor': None}
```

Словарь из двух списков (advanced)

Слово 'advanced' в заголовке, означает, что то, что рассматривается в этом разделе ещё не было в темах курса. Но, в дальнейшем, такой прием может пригодится.

В разделе [варианты создания словаря](#) рассматривался такой вариант создания словаря:

```
In [4]: r1 = dict([('model', '4451'), ('IOS', '15.4'))]  
In [5]: r1  
Out[5]: {'IOS': '15.4', 'model': '4451'}
```

Как правило, такой способ создания используется не в том случае, когда словарь создается вручную в текстовом файле, а когда словарь создается из каких-то других промежуточных объектов.

Воспользуемся таким способом создания словаря, чтобы объединить два списка одинаковой длины в словарь.

Для этого воспользуемся функцией `zip()`, которая возвращает список кортежей:

```
In [1]: a = [1, 2, 3]  
In [2]: b = [100, 200, 300]  
In [3]: zip(a,b)  
Out[3]: [(1, 100), (2, 200), (3, 300)]
```

Теперь на более полезном примере:

```
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]: 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:

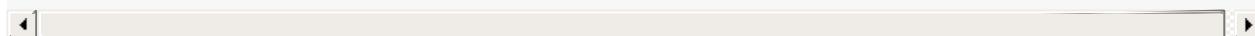
```
In [10]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']

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'}}
```



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  
set([40, 100, 10, 20, 30])
```

Полезные методы для работы с множествами

add()

Метод `add()` добавляет элемент в 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
set([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 шестнадцатиричный 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 пригодится 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
```

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

Обработать строку `ospf_route` и вывести информацию 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
```

В разделе [Форматирование строк](#) добавились примеры, которые помогут с отображением вывода столбцами.

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

Задание 3.2

Преобразовать строку MAC с формата XXXX:XXXX:XXXX в XXXX.XXXX.XXXX

```
MAC = "AAAA:BBBB:CCCC"
```

Задание 3.3

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

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

Задание 3.4

Из строк command1 и command2 получить список VLAN, которые есть и в команде command1 и в команде command2. Не использовать для решения задачи циклы, оператор if.

Для данного примера, результатом должен быть список: [1, 3, 100]

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

Задание 3.5

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.

Не использовать для решения задачи циклы, оператор if.

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

Задание 3.6

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

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

Задание 3.7

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

```
MAC = "AAAA:BBBB:CCCC"
```

Задание 3.8

Преобразовать IP-адрес (переменная IP) в двоичный формат и вывести вывод столбцами, 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
```

```
IP = '192.168.3.1'
```

Задание 3.9

Найти индекс последнего вхождения элемента с конца.

Например, для списка num_list, индекс последнего вхождения элемента 10 - 4; для списка word_list, индекс последнего вхождения элемента 'ruby' - 6.

Сделать решение общим (то есть, не привязываться к конкретному элементу) и проверить на разных списках и элементах.

Не использовать для решения циклы (for, while) и условия (if/else).

Подсказка: функция len() возвращает длину списка.

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

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 %d',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

print '\n'.join(access_template) % 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.

Кодировка

Теперь попробуем вывести текст, набранный кириллицей, на [стандартный поток вывода](#) (файл `access_template_encoding.py`):

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

print "Конфигурация интерфейса в режиме access:"
print '\n'.join(access_template) % 5
```

При попытке запустить скрипт получаем такую ошибку:

```
$ python access_template_encoding.py
  File "access_template_encoding.py", line 7
SyntaxError: Non-ASCII character '\xd0' in file access_template_encoding.py on line 7,
but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details
```

Для того чтобы не было такой ошибки, необходимо добавить в начале файла такую строку:

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

Тогда скрипт `access_template_encoding.py` будет выглядеть так:

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

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

print "Конфигурация интерфейса в режиме access:"
print '\n'.join(access_template) % 5
```

Теперь ошибки нет:

```
$ python access_template_encoding.py
Конфигурация интерфейса в режиме access:
switchport mode access
switchport access vlan 5
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

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 находится строка `#!/usr/bin/env python`

Пример файла `access_template_exec.py`:

```
#!/usr/bin/env python

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

print '\n'.join(access_template) % 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 Python, в модуле sys есть очень простой и удобный способ для работы с аргументами - argv.

В Python существует несколько модулей для обработки аргументов командной строки. В одной из задач раздела [Базы данных](#) будет использоваться [модуль argparse](#).

Посмотрим на пример (файл access_template_argv.py):

```
from sys import argv

interface, vlan = argv[1:]

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

print 'interface %s' % interface
print '\n'.join(access_template) % int(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
- 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'
```

Обратите внимание, что argv всегда возвращает аргументы в виде строки. А, так как в списке access_template в синтаксисе форматирования строк, указано число, переменную `vlan` нужно преобразовать в число (последняя строка).

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 `raw_input()` :

```
In [1]: print raw_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 = raw_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 оператора `print`:

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

In [5]: protocol = raw_input()
OSPF

In [6]: print protocol
OSPF
```

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

Запрашивание информации из скрипта (файл `access_template_raw_input.py`):

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

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

print '\n' + '-' * 30
print 'interface %s' % interface
print '\n'.join(access_template) % vlan
```

В первых двух строках, запрашивается информация у пользователя. Функция `raw_input()`, как и `argv` возвращает данные в виде строк. Поэтому, параметр `vlan` преобразуем в число.

Еще появилась строка `print '\n' + '-' * 30`. Она используется просто для того, чтобы отделить запрос информации от вывода.

Выполняем скрипт:

```
$ python access_template_raw_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
```

Разница между функциями `raw_input()` и `input()` хорошо описана на [stackoverflow](#)

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

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.

Задание 4.1a

Всё, как в задании 4.1. 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 4.1.

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

```
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.

Задание 4.1b

Преобразовать скрипт из задания 4.1а таким образом, чтобы сеть/маска не запрашивались у пользователя, а передавались как аргумент скрипту.

Задание 4.2

В этой задаче нельзя использовать условие if и нельзя изменять словарь london_co.

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:

```
$ python task_4_2.py  
Enter device name: r1  
{'ios': '15.4', 'model': '4451', 'vendor': 'Cisco', 'location': '21 New Globe Walk',  
'ip': '10.255.0.1'}
```

```
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 4.2а

В этой задаче нельзя использовать условие if и нельзя изменять словарь london_co.

Переделать скрипт из задания 4.2 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_4_2a.py
Enter device name: r1
Enter parameter name: ios
15.4
```

```
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 4.2b

В этой задаче нельзя использовать условие if и нельзя изменять словарь london_co.

Переделать скрипт из задания 4.2а таким образом, чтобы, при запросе параметра, отображался список возможных параметров.

Вывести информацию о соответствующем параметре, указанного устройства.

Пример выполнения скрипта:

```
$ python task_4_2b.py
Enter device name: r1
Enter parameter name (ios,model,vendor,location,ip): ip
10.255.0.1
```

```
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 4.2с

В этой задаче нельзя использовать условие if и нельзя изменять словарь london_co.

Переделать скрипт из задания 4.2b 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_4_2c.py
Enter device name: r1
Enter parameter name (ios,model,vendor,location,ip): io
Такого параметра нет
```

```
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 4.2d

В этой задаче нельзя использовать условие if и нельзя изменять словарь london_co.

Переделать скрипт из задания 4.2c таким образом, чтобы, при запросе параметра, пользователь мог вводить название параметра в любом регистре.

Пример выполнения скрипта:

```
$ python task_4_2d.py
Enter device name: r1
Enter parameter name (ios,model,vendor,location,ip): IOS
15.4
```

```

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 4.3

(Задача на основе примеров в разделе)

В этой задаче нельзя использовать условие if.

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 VLANа (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 VLANа (или список VLANов).

Ниже примеры выполнения скрипта, чтобы было проще понять задачу.

Пример выполнения скрипта, при выборе режима access:

```
$ python task_4_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_4_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 %s',
                   'switchport nonegotiate',
                   'spanning-tree portfast',
                   'spanning-tree bpduguard enable']

trunk_template = ['switchport trunk encapsulation dot1q',
                  'switchport mode trunk',
                  'switchport trunk allowed vlan %s']
```

Задание 4.3а

В этой задаче нельзя использовать условие if.

Дополнить скрипт из задания 4.3 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:'

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

trunk_template = ['switchport trunk encapsulation dot1q',
                  'switchport mode trunk',
                  'switchport trunk allowed vlan %s']
```

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 == 0:
...:     print 'a равно 0'
...: 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 не пустая строка
 - любой не пустой 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 [32]: def print_str(string):  
....:     print string  
....:  
  
In [33]: '' or print_str('test string')  
test string  
  
In [34]: '' or 'string1' or print_str('test string')  
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 = raw_input('Введите имя пользователя: ')
password = raw_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 %s установлен' % 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, по которым может проходиться цикл for:

- строка
- список
- словарь
- функция "range()" или итератор "xrange()"
- любой другой итератор (например, "sorted()", "enumerate()")

В курсе не рассматривается, что такое итератор, но, если в двух словах, это такой специальный объект, который дает следующее значение только тогда, когда оно нужно. Например, если при использовании выражения `range(10)`, функция вернет список чисел от 0 до 9 (включительно) сразу. Если же воспользоваться итератором `xrange()`, то он вернет специальный объект. И не будет возвращать числа, пока к нему не будут обращаться, например, в цикле for. Из-за таких особенностей, всегда, когда надо работать с большим количеством объектов, лучше использовать итератор. В общем случае, его лучше использовать везде, где он подходит по функциональности.

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 (итератором) `xrange()`:

for

```
In [2]: for i in xrange(10):
...:     print 'interface FastEthernet0/' + str(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 xrange(10). Этот итератор генерирует числа в диапазоне от нуля, до указанного числа (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 %d' % vlan
...:     print ' name VLAN_%d' % 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`, вместо словаря, возвращает список кортежей:

```
In [9]: r1.items()
Out[9]:
[('vendor', 'Cisco'),
 ('IP', '10.255.0.1'),
 ('hostname', 'london_r1'),
 ('IOS', '15.4'),
 ('location', '21 New Globe Walk'),
 ('model', '4451')]
```

Также есть метод `iteritems()`. Он полностью аналогичен методу `items`, но возвращает итератор, а не список кортежей:

```
In [10]: r1.iteritems()
Out[10]: <dictionary-itemiterator at 0x102daff18>
```

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 ' + intf
...:     for command in commands:
...:         print '%s' % 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 in fast_int['access']:
    print 'interface FastEthernet' + intf
    for command in access_template:
        if command.endswith('access vlan'):
            print ' %s %s' % (command, fast_int['access'][intf])
        else:
            print ' %s' % command
```

Комментарии к коду:

- В первом цикле for перебираются ключи во вложенном словаре fast_int['access']
- Текущий ключ, на данный момент цикла, хранится в переменной intf
- 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, обратившись к вложенному словарю по текущему ключу intf
 - 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
```

Итератор 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.

Но, гораздо удобнее это делать с помощью итератора `enumerate()`.

Базовый пример:

```
In [1]: list1 = ['str1', 'str2', 'str3']

In [2]: 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 [1]: list1 = ['str1', 'str2', 'str3']

In [2]: for position, string in enumerate(list1, 100):
...:     print position, string
...:
100 str1
101 str2
102 str3
```

Пример использования enumerate для EEM (advanced)

Слово 'advanced' в заголовке, означает, что то, что рассматривается в этом разделе ещё не было в темах курса. Но, в дальнейшем, такой прием может пригодится.

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 `eem.py`):

```
import sys

config = sys.argv[1]

with open(config, 'r') as file:
    for (i, command) in enumerate(file, 1):
        print 'action %04d cli command "%s"' % (i, command.rstrip())
```

В данном примере командычитываются из файла, а затем к каждой строке добавляется приставка, которая нужна для EEM.

Конструкция `with` и как работать с файлами рассматриваются в разделе [Работа с файлами](#)

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 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"
```

Упростим, и уберем считывание файла, чтобы было проще понять.

Теперь команды хранятся в списке:

```
In [1]: list1 = ['en\n',
...:   'conf t\n',
...:   'no int Gi0/0/0.300 \n',
...:   'no int Gi0/0/0.301 \n',
...:   'no int Gi0/0/0.302 \n',
...:   'int range gi0/0/0-2\n',
...:   ' channel-group 1 mode active\n',
...:   'interface Port-channel1.300\n',
...:   ' encapsulation dot1Q 300\n',
...:   ' vrf forwarding Management\n',
...:   ' ip address 10.16.19.35 255.255.255.248\n']
```

Повторяем цикл из файла:

```
In [2]: for (i, command) in enumerate(list1, 1):
...:   print 'action %04d cli command "%s"' % (i, command.rstrip())
...
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"
```


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 = raw_input('Введите имя пользователя: ')
password = raw_input('Введите пароль: ')

pass_OK = False

while not pass_OK:
    if len(password) < 8:
        print 'Пароль слишком короткий\n'
        password = raw_input('Введите пароль еще раз: ')
    elif username in password:
        print 'Пароль содержит имя пользователя\n'
        password = raw_input('Введите пароль еще раз: ')
    else:
        print 'Пароль для пользователя %s установлен' % username
        pass_OK = 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
```

Ещё пример:

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

username = raw_input('Введите имя пользователя: ')
password = raw_input('Введите пароль: ')

while True:
    if len(password) < 8:
        print 'Пароль слишком короткий\n'
        password = raw_input('Введите пароль еще раз: ')
    elif username in password:
        print 'Пароль содержит имя пользователя\n'
        password = raw_input('Введите пароль еще раз: ')
    else:
        print 'Пароль для пользователя %s установлен' % username
        break
```

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
```

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 высказывала 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: integer division or modulo by zero

In [2]: 'test' + 2
-----
TypeError: cannot concatenate 'str' and 'int' objects
```

В данном случае, возникло два исключения: **ZeroDivisionError** и **TypeError**.

Чаще всего, можно предсказать какого рода исключения возникнут во время исполнения программы.

Например, если программа на вход ожидает два числа, а на выходе выдает их сумму, а пользователь ввел вместо одного числа, строку, появится ошибка **TypeError**, как в примере выше.

Python позволяет работать с исключениями. Их можно перехватывать и выполнять 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 ошибки.

Например, скрипт divide.py делит два числа введенных пользователем:

```
# -*- coding: utf-8 -*-
try:
    a = raw_input("Введите первое число: ")
    b = raw_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  
Результат: На ноль делить нельзя  
  
$ python divide.py  
Введите первое число: qewr  
Введите второе число: 3  
Результат: 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 = raw_input("Введите первое число: ")  
    b = raw_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  
Результат: Что-то пошло не так...  
  
$ python divide_ver2.py  
Введите первое число: 5  
Введите второе число: 0  
Результат: 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 = raw_input("Введите первое число: ")
    b = raw_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 = raw_input("Введите первое число: ")
    b = raw_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.
```

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

1. Запросить у пользователя ввод IP-адреса в десятично-точечном формате.
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

Задание 5.1a

Сделать копию скрипта задания 5.1.

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'

Задание 5.1b

Сделать копию скрипта задания 5.1а.

Дополнить скрипт:

- Если адрес был введен неправильно, запросить адрес снова.

Задание 5.2

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.

Усложненный вариант: сделать преобразование в одной строке скрипта.

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

Задание 5.3

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 VLAN, 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

```
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 int in fast_int['access']:
    print 'interface FastEthernet' + int
    for command in access_template:
        if command.endswith('access vlan'):
            print ' %s %s' % (command, fast_int['access'][int])
        else:
            print ' %s' % 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

Обычно, возникает путаница с режимами `w+` и `r+`. Разница между ними в том, что `w+`, при открытии, удаляет содержимое файла, а `r+` - нет.

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\\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!\\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 нем 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 поставить запятую в конце выражения `print` или вызвать 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  
!
```

Конструкция `with` может использоваться не только с файлами.

Подробнее об этом можно почитать по ссылке:

<http://stackoverflow.com/questions/3012488/what-is-the-python-with-statement-designed-for>

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

Аналогично заданию 3.1 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

Так как это первое задание с открытием файла, заготовка для открытия файла уже сделана.

```
with open('ospf.txt', 'r') as f:  
    for line in f:  
        print line
```

Задание 6.2

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.

Задание 6.2a

Сделать копию скрипта задания 6.2.

Дополнить скрипт:

- Скрипт не должен выводить команды, в которых содержатся слова, которые указаны в списке ignore.

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

Задание 6.2b

Дополнить скрипт из задания 6.2a:

- 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.

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

Задание 6.2c

Переделать скрипт из задания 6.2b:

- передавать как аргументы:
 - 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.

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

Задание 6.3

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    aabb.cc80.7000    Gi0/1
200    aabb.cc80.7000    Gi0/2
300    aabb.cc80.7000    Gi0/3
100    aabb.cc80.7000    Gi0/4
500    aabb.cc80.7000    Gi0/5
200    aabb.cc80.7000    Gi0/6
300    aabb.cc80.7000    Gi0/7
```

Задание 6.3a

Сделать копию скрипта задания 6.3

Дополнить скрипт:

- Отсортировать вывод по номеру VLAN

Задание 6.3b

Сделать копию скрипта задания 6.3a

Дополнить скрипт:

- Запросить у пользователя ввод номера VLAN.
- Выводить информацию только по указанному VLAN.

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, в этом файле, или каком-то другом, эта функция просто будет использоваться.

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 разделе, 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 ожидает имя файла, в качестве аргумента, 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 нее, 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.

Код функций, которые используются в этом разделе, можно скопировать из файла func_params_args.py

Для того чтобы функция могла принимать входящие значения, ее нужно создать с параметрами:

```
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. Аналогично, возникнет ошибка, если передать три и больше аргументов:

```
In [5]: delete_exclamation_from_cfg('r1.txt')
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-7-66ae381f1c4f> in <module>()
----> 1 delete_exclamation_from_cfg('r1.txt')

TypeError: delete_exclamation_from_cfg() takes exactly 2 arguments (1 given)
```

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)

Код функций, которые используются в этом разделе, можно скопировать из файла func_params_types.py

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:

```
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:

```
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.

Код функций, которые используются в этом разделе, можно скопировать из файла func_args_types.py

Посмотрим на разные способы передачи аргументов, на примере функции:

```
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.

В случае с функцией `cfg_to_list`, получится такой результат:

```
In [3]: cfg_to_list(False, 'r1.txt')
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-18-e6da7e2657eb> in <module>()
----> 1 cfg_to_list(False, 'r1.txt')

<ipython-input-15-21a013e5e92c> in cfg_to_list(cfg_file, delete_exclamation)
      1 def cfg_to_list(cfg_file, delete_exclamation):
      2     result = []
----> 3     with open( cfg_file ) as f:
      4         for line in f:
      5             if delete_exclamation and line.startswith('!'):

TypeError: coercing to Unicode: need string or buffer, bool found
```

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-19-5efdee7ce6dd>", line 1
            cfg_to_list(delete_exclamation=False, 'r1.txt')
SyntaxError: non-keyword arg after keyword arg
```

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 функцию 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 arg
....:     return sum(arg)
....:

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 функцию 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-6-71c121dc2cf7>", line 1
    sum_arg(b=10,c=20,d=30,10)
SyntaxError: non-keyword arg after keyword arg
```

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

Для примера, используем функцию `config_interface` (файл `func_args_var_unpacking.py`):

```
def config_interface(intf_name, ip_address, cidr_mask):
    interface = 'interface %s'
    no_shut = 'no shutdown'
    ip_addr = 'ip address %s %s'
    result = []
    result.append(interface % intf_name)
    result.append(no_shut)

    mask_bits = int(cidr_mask.split('/')[-1])
    bin_mask = '1'*mask_bits + '0'*(32-mask_bits)
    dec_mask = '.'.join([str(int(bin_mask[i:i+8], 2)) for i in [0,8,16,24] ])

    result.append(ip_addr % (ip_address, dec_mask))
    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, например 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-32-fb83ecc1fbcf> in <module>()
      1 for info in interfaces_info:
----> 2     print config_interface(info)
      3

TypeError: config_interface() takes exactly 3 arguments (1 given)
```

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`:

```
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.

Весь код функции можно вставить в ipython с помощью команды %cpaste.

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-40-1affbd99c2f5> in <module>()
      1 for d in cfg:
----> 2     print config_to_list(d)
      3

<ipython-input-35-6337ba2bfe7a> 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: coercing to Unicode: need string or buffer, dict found
```

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', '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
```

Весь код функции можно вставить в ipython с помощью команды `%cpaste`.

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`.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

Создать функцию, которая генерирует конфигурацию для access-портов.

Параметр access 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:

```
[  

'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`.

```
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 7.1а

Сделать копию скрипта задания 7.1.

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.

```
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 7.1b

Сделать копию скрипта задания 7.1а.

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.

```
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
    if psecurity:
        - если значение True, то настройка выполняется с добавлением шаблона port_security
    else:
        - 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 7.2

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`.

```
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 7.2a

Сделать копию скрипта задания 7.2

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`.

```
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 7.3

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`

Задание 7.3а

Сделать копию скрипта задания 7.3.

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

Задание 7.4

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.

```

ignore = ['duplex', 'alias', 'Current configuration']

def ignore_command(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
    т
    """
    ignore_command = False

    for word in ignore:
        if word in command:
            return True
    return ignore_command


def config_to_dict(config):
    """
    config - имя конфигурационного файла коммутатора

    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
    """
    pass

```

Задание 7.4а

Задача такая же, как и задании 7.4. Проверить работу функции надо на примере файла config_r1.txt

Обратите внимание на конфигурационный файл. В нем 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.

Теперь:

- если уровня 2, то команды верхнего уровня будут ключами словаря, а команды подуровней - списками;
- если уровня 3, то самый вложенный 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']}}
```

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

def check_ignore(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
    """
    ignore_command = False

    for word in ignore:
        if word in command:
            ignore_command = True
    return ignore_command

def config_to_dict(config):
    """
    config - имя конфигурационного файла
    """
    pass
```

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 не понятно, что какая-то функция взята из модуля `os`, например. 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
from sw_data import sw1_fast_int

def generate_access_cfg(sw_dict):
    result = []
    for intf in sw_dict['access']:
        result.append('interface FastEthernet' + intf)
        for command in sw_int_templates.access_template:
            if command.endswith('access vlan'):
                result.append(' %s %s' % (command, sw_dict['access'][intf]))
            else:
                result.append(' %s' % 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__"
```

Иногда, скрипт, который вы создали, может выполняться и самостоятельно, и может быть импортирован как модуль, другим скриптом.

Добавим ещё один скрипт, к предыдущему примеру, который будет импортировать функцию из файла generate_sw_int_cfg.py.

Файл sw_cfg_templates.py с шаблонами конфигурации:

```
basic_cfg = """
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
!
"""

lines_cfg = """
!
line con 0
logging synchronous
history size 100
line vty 0 4
logging synchronous
history size 100
transport input ssh
!
"""


"""

```

В файле generate_sw_cfg.py импортируются шаблоны из sw_cfg_templates.py и функции из предыдущих файлов:

```
from sw_data import sw1_fast_int
from generate_sw_int_cfg import generate_access_cfg
from sw_cfg_templates import basic_cfg, lines_cfg


print basic_cfg
print '\n'.join(generate_access_cfg(sw1_fast_int))
print lines_cfg
```

В результате, должны отобразиться такие части конфигурации, по порядку: шаблон basic_cfg, настройка интерфейсов, шаблон lines_cfg.

Обратите внимание, что из файла можно импортировать несколько объектов:

```
from sw_cfg_templates import basic_cfg, lines_cfg
```

Результат выполнения:

```
$ python generate_sw_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

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
!

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

!
line con 0
logging synchronous
history size 100
line vty 0 4
logging synchronous
history size 100
transport input ssh
!
```

Полученный вывод не совсем правильный: перед строками шаблона basic_cfg, идет лишняя конфигурация интерфейсов.

Так получилось из-за строки print в файле generate_sw_int_cfg.py:

```
print '\n'.join(generate_access_cfg(sw1_fast_int))
```

Когда скрипт импортирует какой-то модуль, всё, что находится в модуле, выполняется. И, так как в данном случае, в файле generate_sw_int_cfg.py есть строка с print, на стандартный поток вывода попадает результат выполнения этого выражения, при запуске файла generate_sw_int_cfg.py.

В Python есть специальный прием, который позволяет указать, что какой-то код должен выполняться, только когда файл запускается напрямую.

Файл generate_sw_int_cfg2.py:

```
import sw_int_templates
from sw_data import sw1_fast_int

def generate_access_cfg(sw_dict):
    result = []
    for intf in sw_dict['access']:
        result.append('interface FastEthernet' + intf)
        for command in sw_int_templates.access_template:
            if command.endswith('access vlan'):
                result.append(' %s %s' % (command, sw_dict['access'][intf]))
            else:
                result.append(' %s' % command)
    return result

if __name__ == "__main__":
    print '\n'.join(generate_access_cfg(sw1_fast_int))
```

Обратите внимание на запись:

```
if __name__ == "__main__":
    print '\n'.join(generate_access_cfg(sw1_fast_int))
```

Переменная `__name__` это специальная переменная, которая выставляется равной `"__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.

Измените в файле generate_sw_cfg.py строку:

```
from generate_sw_int_cfg import generate_access_cfg
```

на строку:

```
from generate_sw_int_cfg2 import generate_access_cfg
```

И попробуйте запустить скрипт:

```
$ python generate_sw_cfg.py

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
!

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

!
line con 0
logging synchronous
history size 100
line vty 0 4
logging synchronous
history size 100
transport input ssh
!
```

Теперь print из файла generate_sw_int_cfg2.py не выводится.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 8.1

Создать отдельный файл `my_func.py`.

Перенести в него функции из заданий:

- 7.1 или 7.1a
- 7.2
- 7.3 или 7.3a

Создать файл `main.py`:

- По ходу задания импортировать нужные функции из файла `my_func`.
- Файл `main.py` должен ожидать как аргумент имя конфигурационного файла коммутатора.
- Имя конфигурационного файла передать как аргумент функции `get_int_vlan_map` (из задания 7.3-7.3a)
 - На выходе функции, мы должны получить кортеж двух словарей.
- Словари, соответственно, надо передать функциям:
 - `generate_access_config` (из задания 7.1-7.1a)
 - `generate_trunk_config` (из задания 7.2)
- Эти функции, в свою очередь, возвращают список со строками готовой конфигурации которую надо записать в файл `result.txt` в виде стандартного

конфига (то есть, строк)

Задание 8.2

Создать функцию `parse_cdp_neighbors`, которая обрабатывает вывод команды `show cdp neighbors`.

Функция ожидает, как аргумент, вывод команды одной строкой.

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')}
```

Проверить работу функции на содержимом файла `sw1_sh_cdp_neighbors.txt`

Задание 8.2a

Для выполнения этого задания, должен быть установлен `graphviz`:

```
apt-get install graphviz
```

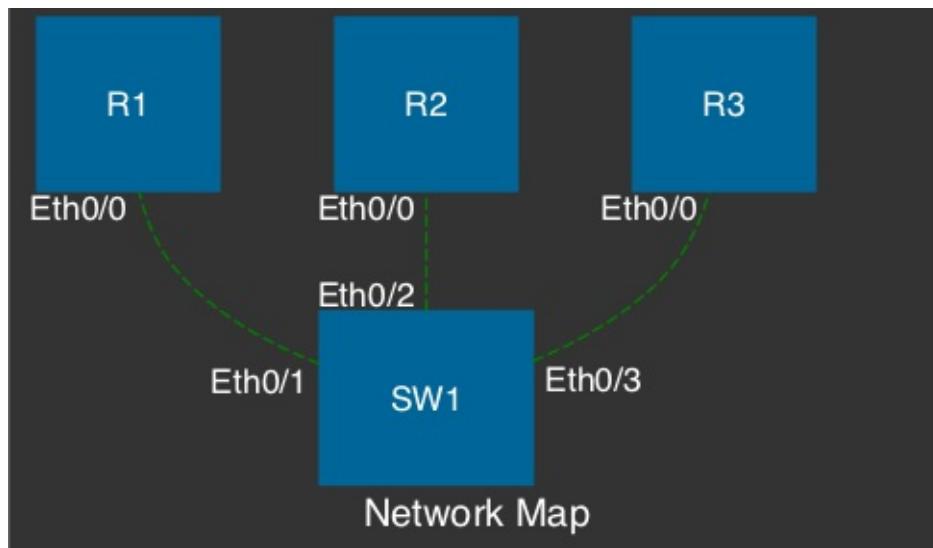
И модуль `python` для работы с `graphviz`:

```
pip install graphviz
```

С помощью функции `parse_cdp_neighbors` из задания 8.2 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_8_2a_topology.svg`



Задание 8.2b

Для выполнения этого задания, должен быть установлен graphviz:

```
apt-get install graphviz
```

И модуль python для работы с graphviz:

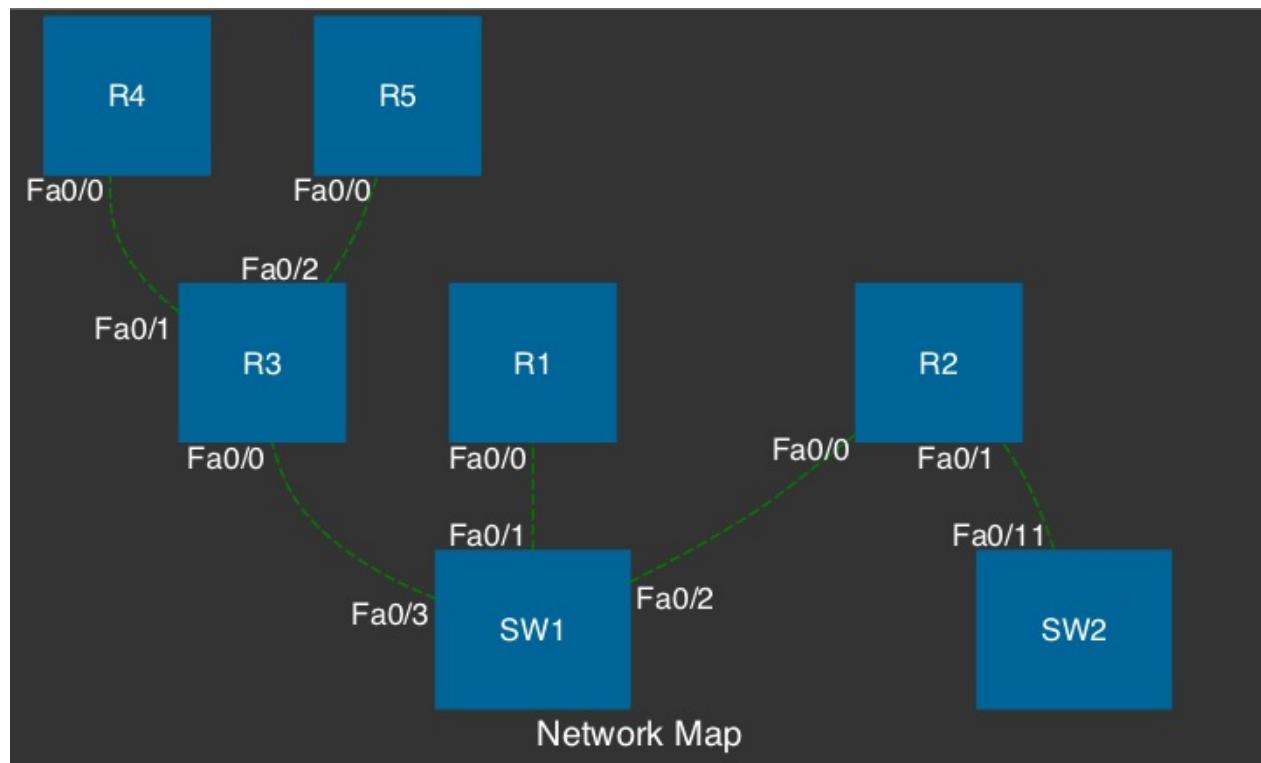
```
pip install graphviz
```

С помощью функции `parse_cdp_neighbors` из задания 8.2 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_8_2b_topology.svg`



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 обработки вывода команд show. Например, если со всего оборудования надо собрать информацию про версию ОС и uptime, можно получить эту информацию из вывода show version, обработав его с помощью регулярных выражений

В самом сетевом оборудовании, регулярные выражения можно использовать для фильтрации вывода любых команд show и more. Или, например, для фильтрации таблицы BGP.

Отличный сайт для тестирования регулярных выражений: regex101.com

Модуль re

В Python для работы с регулярными выражениями используется модуль `re`.

Соответственно, для начала работы с регулярными выражениями в Python, надо импортировать модуль `re`.

В этом разделе используется пример поиска подстроки в строке, без специальных символов, чтобы сосредоточиться на возможностях модуля `re`. В следующих разделах будут использоваться более сложные примеры.

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

search()

Функция `search()`:

- используется для поиска подстроки, которая соответствует шаблону
- возвращает объект `Match`, если подстрока найдена
- возвращает `None`, если подстрока не найдена

Поиск подстроки в строке:

```
In [1]: import re

In [2]: line = '00:09:BB:3D:D6:58    10.1.10.2      86250  dhcp-snooping  10  FastEth
ernet0/1'

In [3]: print re.search('dhcp', line)
<_sre.SRE_Match object at 0x10edeb9f0>

In [4]: print re.search('dhcpd', line)
None
```

Из объекта `Match` можно получить несколько вариантов полезной информации.

Например, с помощью метода `span()`, можно получить числа, указывающие начало и конец подстроки:

```
In [5]: match = re.search('dhcp', line)

In [6]: match.span()
Out[6]: (49, 53)

In [7]: line[49:53]
Out[7]: 'dhcp'
```

Метод `group()` позволяет получить подстроку, которая соответствует шаблону:

```
In [15]: match.group()
Out[15]: 'dhcp'
```

Важный момент в использовании функции `search()`, то что она ищет только первое совпадение в строке, которое соответствует шаблону:

```
In [16]: line2 = 'test dhcp, test2 dhcp2'

In [17]: match = re.search('dhcp', line2)

In [18]: match.group()
Out[18]: 'dhcp'

In [19]: match.span()
Out[19]: (5, 9)
```

.findall()

Для того чтобы найти все совпадения, можно использовать функцию `.findall()`:

```
In [20]: line2 = 'test dhcp, test2 dhcp2'

In [21]: match = re.findall('dhcp', line2)

In [22]: print match
['dhcp', 'dhcp']
```

Особенность функции `.findall()` в том, что она возвращает список подстрок, которые соответствуют шаблону, а не объект Match. Поэтому нельзя вызвать методы, которые использовались в функции `search()`.

finditer()

Для того чтобы получить все совпадения, но при этом, получить совпадения в виде объекта Match(), можно использовать функцию `finditer()`:

```
In [23]: line2 = 'test dhcp, test2 dhcp2'

In [24]: match = re.finditer('dhcp', line2)

In [25]: print match
<callable-iterator object at 0x10efd2cd0>

In [26]: for i in match:
....:     print i.span()
....:
(5, 9)
(17, 21)

In [27]: line2[5:9]
Out[27]: 'dhcp'

In [28]: line2[17:21]
Out[28]: 'dhcp'
```

Можно воспользоваться и методами `start()`, `end()` (так удобнее получить позиции подстрок):

```
In [29]: line2 = 'test dhcp, test2 dhcp2'

In [30]: match = re.finditer('dhcp', line2)

In [31]: for i in match:
....:     b = i.start()
....:     e = i.end()
....:     print line2[b:e]
....:
dhcp
dhcp
```

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.

Пример компиляции регулярного выражения и его использования:

```
In [32]: line2 = 'test dhcp, test2 dhcp2'

In [33]: regex = re.compile('dhcp')

In [34]: match = regex.finditer(line2)

In [35]: for i in match:
....:     b = i.start()
....:     e = i.end()
....:     print line2[b:e]
....:

dhcp
dhcp
```

Специальные символы

Полностью возможности регулярных выражений проявляются при использовании специальных символов.

Специальные символы:

- `.` - любой символ, кроме символа новой строки (опция `m` позволяет включить и символ 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. Текст, который совпал с выражением, запоминается

Повторение:

- `regex*` - ноль или более повторений предшествующего элемента
- `regex+` - один 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

Предопределенные наборы символов:

- `\d` - любая цифра
- `\D` - любое нечисловое значение
- `\s` - whitespace (`\t\n\r\f\v`)
- `\S` - все, кроме whitespace
- `\w` - любая буква или цифра
- `\W` - все, кроме букв и цифр

Посмотрим на примеры использования специальных символов:

```
In [1]: import re
In [2]: line = "FastEthernet0/1      10.0.12.1      YES manual up
          up"
```

Точка обозначает любой символ, поэтому в строке line найдено 3 совпадения с регулярным выражением `.0`:

```
In [3]: print re.findall('.0', line)
['t0', '10', '.0']
```

Символ `^` означает начало строки. Выражению `^F` соответствует только одна подстрока:

```
In [4]: print re.findall('^F', line)
['F']
```

Выражению `^.a` соответствует подстрока 'Fa':

```
In [5]: print re.findall('^.a', line)
['Fa']
```

Символ `$` обозначает конец строки:

```
In [6]: print re.findall('up$', line)
['up']

In [7]: print re.findall('up', line)
['up', 'up']
```

Символы, которые перечислены в квадратных скобках, означают, что любой из этих символов будет совпадением. Таким образом можно описывать разные регистры:

```
In [8]: print re.findall('[Ff]ast', line)
['Fast']

In [9]: print re.findall('[Ff]ast[Ee]thernet', line)
['FastEthernet']
```

Если после открывающейся квадратной скобки, указан символ `^`, совпадением будет любой символ, кроме указанных в скобках (в данном случае, всё, кроме букв и пробела):

```
In [10]: print re.findall('[^a-zA-Z ]', line)
['0', '/', '1', '1', '0', '.', '0', '.', '1', '2', '.', '1']
```

Вертикальная черта работает как 'или':

```
In [11]: print re.findall('up|down', line)
['up', 'up']
```

Жадность регулярных выражений

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>'
```

Обратите внимание, что жадность касается именно символов повторения

Группировка выражений

Нумерованные группы

С помощью определения групп элементов в шаблоне, можно изолировать части текста, которые соответствуют шаблону.

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, которые 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'
```

Для вывода всех подстрок, которые соответствуют указанным группам, используется метод `groups`:

```
In [13]: match.groups()
Out[13]: ('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 [14]: line = "FastEthernet0/1      10.0.12.1      YES manual up
           up"

In [15]: match = re.search('(?P<intf>\S+)\s+(?P<address>[\d\.]+)\s+', line)
```

Теперь к этим группам можно обращаться по имени:

```
In [15]: match.group('intf')
Out[15]: 'FastEthernet0/1'

In [16]: match.group('address')
Out[16]: '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 [17]: match.groupdict()
Out[17]: {'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 [18]: match = re.search('(?P<intf>\S+)\s+(?P<address>[\d\.]+)\s+[\w\s]+(?P<status>
up|down|administratively down)\s+(?P<protocol>up|down)', line)

In [19]: match.groupdict()
Out[19]:
{'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

regex = re.compile('(?P<mac>.+?) +(?P<ip>.*?) +(\d+) +([\w-]+) +(?P<vlan>\d+) +(?P<int>.*$)')
result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        if line[0].isdigit():
            result.append(regex.search(line).groupdict())

print "К коммутатору подключено %d устройства" % len(result)

for num, comp in enumerate(result, 1):
    print "Параметры устройства %s:" % num
    for key in comp:
        print "\t%s:\t%s" % (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
```

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 9.1

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.

В данном случае, скрипт будет работать как фильтр `include` в CLI Cisco. Вы можете попробовать использовать регулярные выражения для [фильтрации вывода команд show](#).

Пример: `sh ip int br | include up +up`

Задание 9.1а

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 9.1, и файл `sh_ip_int_br.txt`.

Задание 9.1b

Переделайте регулярное выражение из задания 9.1a 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 кавычек вокруг регулярного выражения).

Проверьте регулярное выражение, используя скрипт, который был создан в задании 9.1, и файл `sh_ip_int_br.txt`.

Задание 9.1c

Проверить работу скрипта из задания 9.1 и регулярного выражения из задания 9.1a или 9.1b 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.

Задание 9.2

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 9.3

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 9.3а

Переделать функцию `parse_cfg` из задания 9.3 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 9.3b

Проверить работу функции `parse_cfg` из задания 9.3а 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 9.3а 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.

Задание 9.4

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 9.4a

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 9.4

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']
```

Сериализация данных

Сериализация данных - это сохранение данных в каком-то формате. Чаще всего, это сохранение в каком-то структурированном формате.

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 о 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.

В этом разделе, рассматриваются только базовые операции чтения и записи, без дополнительных параметров. Подробнее, можно почитать в документации модулей.

Кроме того, на сайте [PyMOTW](#) очень хорошо расписываются все модули Python, которые входят в стандартную библиотеку (устанавливаются, вместе с самим Python):

- [CSV](#)
- [JSON](#)

- [YAML](#)

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 использования модуля 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.

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
```

Вывод будет таким:

```
$ python csv_read_dict.py
{'model': '3750', 'hostname': 'sw1', 'vendor': 'Cisco', 'location': 'London'}
{'model': '3850', 'hostname': 'sw2', 'vendor': 'Cisco', 'location': 'Liverpool'}
{'model': '3650', 'hostname': 'sw3', 'vendor': 'Cisco', 'location': 'Liverpool'}
{'model': '3650', 'hostname': 'sw4', 'vendor': 'Cisco', 'location': 'London'}
```

Обратите внимание, что сам reader это итератор. Поэтому, если просто вывести reader, то вывод будет таким:

```
In [1]: import csv

In [2]: with open('sw_data.csv') as f:
...:     reader = csv.reader(f)
...:     print reader
...:

<_csv.reader object at 0x10385b050>
```

Но, если нужно все объекты передать куда-то дальше, 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']]
```

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: последнее значение, взято в кавычки, а остальные строки - нет.

Так получилось из-за того, что в последней строке есть запятая. И кавычки указывают на то, что именно является целой строкой. Когда запятая находится в кавычках, модуль csv не воспринимает её как разделитель.

Иногда, лучше, чтобы все строки были в кавычках. Конечно, в этом 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_ver2.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_ver2.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.

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 словари в 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 и возвращает объекты Python
- `json.loads()` - метод считывает строку в формате JSON и возвращает объекты Python

json.load()

Чтение файла в объект Python:

```
In [1]: import json

In [2]: with open('sw_templates.json') as f:
...:     templates = json.load(f)
...:

In [3]: templates
Out[3]:
{u'access': [u'switchport mode access',
  u'switchport access vlan',
  u'switchport nonegotiate',
  u'spanning-tree portfast',
  u'spanning-tree bpduguard enable'],
 u'trunk': [u'switchport trunk encapsulation dot1q',
  u'switchport mode trunk',
  u'switchport trunk native vlan 999',
  u'switchport trunk allowed vlan']}

In [4]: for section, commands in templates.items():
...:     print section
...:     print '\n'.join(commands)
...:
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
```

Результат чтения - словарь. Обратите внимание, что при чтении из файла в формате JSON, строки будут в unicode.

Мы не будем рассматривать работу с Unicode. Хороший документ о работе с unicode: [Unicode HowTo](#).

json.loads()

Считывание строки в формате JSON в объект Python:

```
In [5]: with open('sw_templates.json') as f:  
...:     templates = json.loads(f.read())  
...:  
  
In [6]: templates  
Out[6]:  
{u'access': [u'switchport mode access',  
    u'switchport access vlan',  
    u'switchport nonegotiate',  
    u'spanning-tree portfast',  
    u'spanning-tree bpduguard enable'],  
 u'trunk': [u'switchport trunk encapsulation dot1q',  
    u'switchport mode trunk',  
    u'switchport trunk native vlan 999',  
    u'switchport trunk allowed vlan']}
```

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.dump()

Запись объекта Python в файл:

```
In [7]: 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}
...:

In [8]: with open('sw_templates.json', 'w') as f:
...:     json.dump(to_json, f)
...:

In [9]: cat 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"]}
```

Когда нужно записать информацию в формате JSON в файл, лучше использовать метод `dump`.

`json.dumps()`

Преобразование объекта в строку в формате JSON:

```
In [10]: with open('sw_templates.json', 'w') as f:
...:     f.write(json.dumps( to_json ))
...:

In [11]: cat 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"]}
```

Метод `json.dumps()` подходит для ситуаций, когда надо вернуть строку в формате JSON.

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_ver2.py`):

```
In [13]: with open('sw_templates.json', 'w') as f:  
...:     json.dump(to_json, f, sort_keys=True, indent=2)  
...:
```

Теперь содержимое файла `sw_templates.json` выглядит так:

```
In [14]: cat 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"  
    ]  
}
```

Изменение типа данных

Еще один важный аспект преобразования данных в формат `json`: данные не всегда будут того же типа, что исходные данные в Python.

Такая ситуация уже возникала со строками - они были в формате `unicode`. Кроме того, при записи в JSON кортежей, они превращаются в списки.

Например:

```
In [15]: import json

In [16]: trunk_template = ('switchport trunk encapsulation dot1q',
...:                      'switchport mode trunk',
...:                      'switchport trunk native vlan 999',
...:                      'switchport trunk allowed vlan')

In [17]: print type(trunk_template)
<type 'tuple'>

In [18]: with open('trunk_template.json', 'w') as f:
...:     json.dump(trunk_template, f, sort_keys=True, indent=2)
...:

In [19]: cat trunk_template.json
[
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
]
In [20]: templates = json.load(open('trunk_template.json'))

In [21]: type(templates)
Out[21]: list

In [22]: print templates
[u'switchport trunk encapsulation dot1q', u'switchport mode trunk', u'switchport trunk native vlan 999', u'switchport trunk allowed vlan']
```

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"]}
```

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.

Пройдемся по тому как в YAML записываются различные форматы данных.

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 выполняется очень просто:

```
In [1]: import yaml

In [2]: with open('info.yaml') as f:
...:     templates = yaml.load(f)
...:

In [3]: templates
Out[3]:
[{'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:
    f.write(yaml.dump(to_yaml))

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]
```

То есть, 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_ver2.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:
    f.write(yaml.dump(to_yaml, 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
```

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 10.1

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`
- обрабатывает вывод, с помощью регулярных выражений
- возвращает кортеж из трёх элементов:
 - `ios` - в формате "12.4(5)T"
 - `image` - в формате "flash:c2800-adviservicesk9-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 скрипте, с помощью модуля glob, создан список файлов, имя которых начинается на sh_vers. 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_vers*')
#print sh_version_files

headers = ['hostname', 'ios', 'image', 'uptime']
```

Задание 10.2

Создать функции:

- generate_access_config - генерирует конфигурацию для access-портов, на основе словарей access и psecurity из файла sw_templates.yaml
- generate_trunk_config - генерирует конфигурацию для trunk-портов, на основе словаря trunk из файла sw_templates.yaml
- generate_mngmt_config - генерирует конфигурацию менеджмент настроек, на основе словаря mngmt из файла templates.yaml
- generate_ospf_config - генерирует конфигурацию ospf, на основе словаря ospf из файла templates.yaml
- generate_alias_config - генерирует конфигурацию alias, на основе словаря alias из

файла templates.yaml

- `generate_switch_config` - генерирует конфигурацию коммутатора, в зависимости от переданных параметров, использует для этого остальные функции

```
import yaml

access_dict = { 'FastEthernet0/12':10,
                'FastEthernet0/14':11,
                'FastEthernet0/16':17,
                'FastEthernet0/17':150 }

trunk_dict = { 'FastEthernet0/1':[10,20,30],
                'FastEthernet0/2':[11,30],
                'FastEthernet0/4':[17] }

def generate_access_config(access, psecurity=False):
    """
    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
    rity
        - если значение False, то настройка не выполняется

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """
    pass

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 список всех команд, которые были сгенерированы на основе шаблона
    """
    pass

def generate_ospf_config(filename):
    """
    filename - имя файла в формате YAML, в котором находится шаблон ospf.

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """
    pass
```

```

templates = yaml.load(open(filename))

def generate_mngmt_config(filename):
    """
    filename - имя файла в формате YAML, в котором находится шаблон mngmt.

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """
    pass

def generate_alias_config(filename):
    """
    filename - имя файла в формате YAML, в котором находится шаблон alias.

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """
    pass

def generate_switch_config(access=True, psecurity=False, trunk=True,
                           ospf=True, mngmt=True, alias=False):
    """
    Аргументы контролируют какие настройки надо выполнить.
    По умолчанию, будет настроено все, кроме psecurity и alias.

    Возвращает список всех команд, которые были сгенерированы на основе шаблона
    """
    pass

# Сгенерировать конфигурации для разных коммутаторов:

sw1 = generate_switch_config()
sw2 = generate_switch_config(psecurity=True, alias=True)
sw3 = generate_switch_config(ospf=False)

```

Задание 10.3

Создать функцию `parse_sh_cdp_neighbors`, которая обрабатывает вывод команды `show cdp neighbors`.

Функция ожидает, как аргумент, вывод команды одной строкой.

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'}}}}
```

Проверить работу функции на содержимом файла sh_cdp_n_sw1.txt

Задание 10.3а

С помощью функции parse_sh_cdp_neighbors из задания 10.3, 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'}}}}
```

Не копировать код функции parse_sh_cdp_neighbors

Задание 10.3б

Переделать функциональность скрипта из задания 10.3а, 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 10.3а.

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 10.3с

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`.

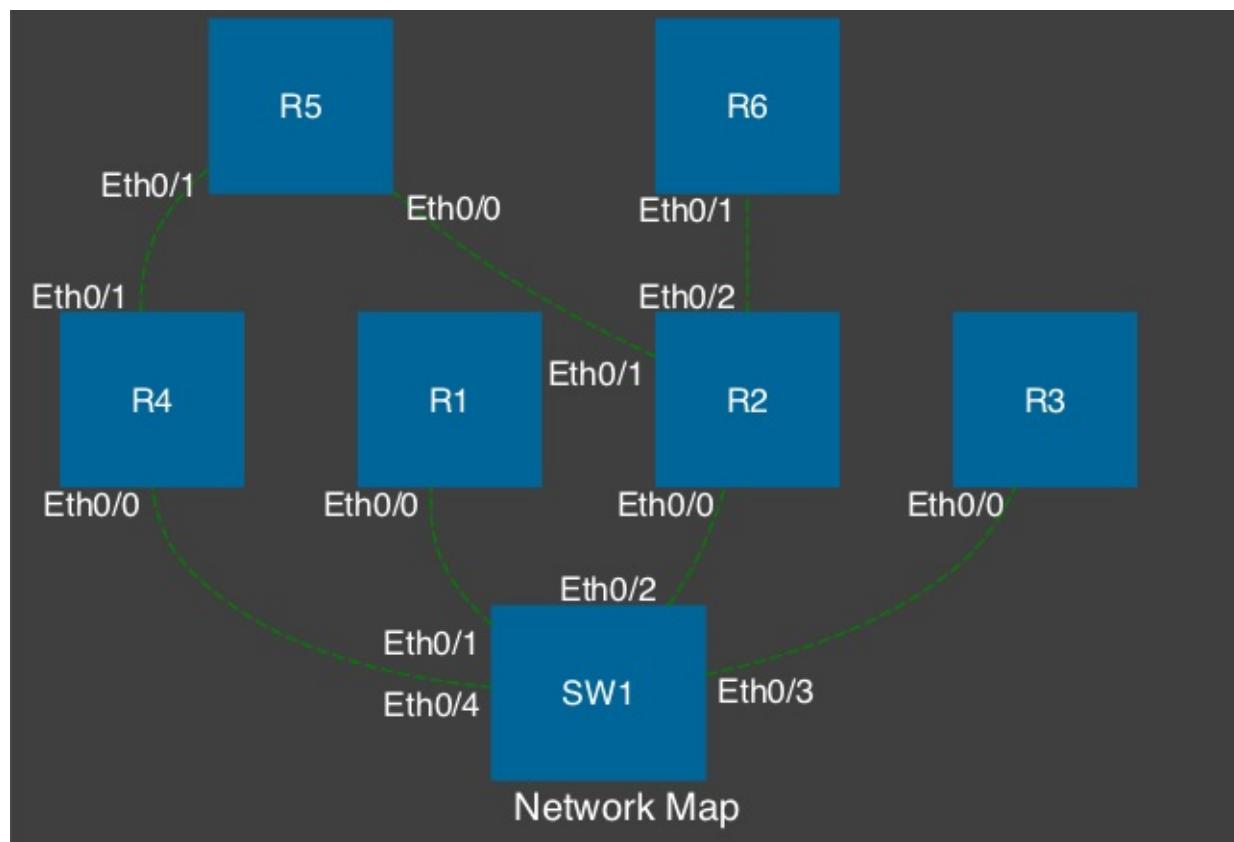
Описание топологии из файла `topology.yaml` нужно преобразовать соответствующим образом, чтобы использовать функцию `draw_topology`.

Для решения задания можно создать любые вспомогательные функции.

В итоге, должно быть сгенерировано изображение топологии. Результат должен выглядеть так же, как схема в файле `task_10_3c_topology.svg`

Не копировать код функции `draw_topology`.

Для выполнения этого задания, должен быть установлен `graphviz`: `pip install graphviz`



Работа с базами данных

Использование баз данных - это еще один способ хранения информации.

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.

В разговорной речи (но и в литературе также) часто можно встретить использование термина БД, вместо термина СУБД. Это не совсем корректно, но, тем не менее, встречается довольно часто.

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 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 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

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 нем 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 :

WHERE

```
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 sw:

```
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 отсортированными 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 primary key, hostname text, mode  
1 text, location text)")  
Out[4]: <sqlite3.Cursor at 0x1085be880>
```

Выражения SQL могут быть параметризованы - вместо данных можно подставлять специальные значения. Засчет 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 применились, 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 [14]: connection = sqlite3.connect('new_db.db')

In [15]: cursor = connection.cursor()

In [16]: cursor.executescript("""
...:     create table switches(
...:         hostname    text primary key,
...:         location   text
...:     );
...:
...:     create table dhcp(
...:         mac        text primary key,
...:         ip         text,
...:         vlan       text,
...:         interface  text,
...:         switch     text not null references switches(hostname)
...:     );
...: """)

Out[16]: <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 [1]: import sqlite3

In [2]: connection = sqlite3.connect('sw_inventory.db')

In [3]: cursor = connection.cursor()

In [4]: cursor.execute('select * from switch')
Out[4]: <sqlite3.Cursor at 0x104eda810>

In [5]: cursor.fetchone()
Out[5]: (u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
```

Обратите внимание, что хотя запрос SQL подразумевает, что запрашивалось всё содержимое таблицы, метод `fetchone` вернул только одну строку.

Если повторно вызвать метод, он вернет следующую строку:

```
In [6]: print cursor.fetchone()
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
```

Аналогичным образом метод будет возвращать следующие строки. После обработки всех строк, метод начинает возвращать `None`.

Засчет этого, метод можно использовать в цикле, например, так:

```
In [7]: cursor.execute('select * from switch')
Out[7]: <sqlite3.Cursor at 0x104eda810>

In [8]: while True:
...:     next_row = cursor.fetchone()
...:     if next_row:
...:         print next_row
...:     else:
...:         break
...:
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0001', u'sw5', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0002', u'sw6', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0003', u'sw7', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0004', u'sw8', u'Cisco 3750', u'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 [9]: print cursor.arraysize
1
```

Например, таким образом можно возвращать по три строки из запроса:

```
In [10]: cursor.execute('select * from switch')
Out[10]: <sqlite3.Cursor at 0x104eda810>

In [11]: while True:
...:     three_rows = cursor.fetchmany(3)
...:     if three_rows:
...:         print three_rows
...:         print
...:     else:
...:         break
...:
[(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str'),
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str'),
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')]

[(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0001', u'sw5', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0002', u'sw6', u'Cisco 3750', u'London, Green Str')]

[(u'0000.1111.0003', u'sw7', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0004', u'sw8', u'Cisco 3750', u'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 [12]: cursor.execute('select * from switch')
Out[12]: <sqlite3.Cursor at 0x104eda810>

In [13]: cursor.fetchall()
Out[13]:
[(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str'),
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str'),
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str'),
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0001', u'sw5', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0002', u'sw6', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0003', u'sw7', u'Cisco 3750', u'London, Green Str'),
(u'0000.1111.0004', u'sw8', u'Cisco 3750', u'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 [14]: cursor.execute('select * from switch')
Out[14]: <sqlite3.Cursor at 0x104eda810>

In [15]: cursor.fetchone()
Out[15]: (u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')

In [16]: cursor.fetchone()
Out[16]: (u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')

In [17]: cursor.fetchall()
Out[17]:
[(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str'),
 (u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str'),
 (u'0000.1111.0001', u'sw5', u'Cisco 3750', u'London, Green Str'),
 (u'0000.1111.0002', u'sw6', u'Cisco 3750', u'London, Green Str'),
 (u'0000.1111.0003', u'sw7', u'Cisco 3750', u'London, Green Str'),
 (u'0000.1111.0004', u'sw8', u'Cisco 3750', u'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 [18]: result = cursor.execute('select * from switch')

In [19]: for row in result:
...:     print row
...:
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0001', u'sw5', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0002', u'sw6', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0003', u'sw7', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0004', u'sw8', u'Cisco 3750', u'London, Green Str')
```

И, конечно же, аналогичный вариант отработает и без присваивания переменной:

```
In [20]: for row in cursor.execute('select * from switch'):
...:     print row
...:
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0001', u'sw5', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0002', u'sw6', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0003', u'sw7', u'Cisco 3750', u'London, Green Str')
(u'0000.1111.0004', u'sw8', u'Cisco 3750', u'London, Green Str')
```

Использование модуля sqlite3

Без явного создания курсора

Методы execute доступны и в объекте Connection. При их использовании курсор создается, но не явно. Однако, методы fetch в Connection недоступны.

Но, если использовать курсор, который возвращает методы execute, как итератор, методы fetch могут и не понадобиться.

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 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
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'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 [22]: con = sqlite3.connect('sw_inventory2.db')

In [23]: query = "INSERT into switch values ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str')"

In [24]: 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 [25]: try:
...:     con.execute(query)
...: except sqlite3.IntegrityError as e:
...:     print "Error occurred: ", e
...:
Error occurred:  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` и закрывать соединение. 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 primary key, hostname text, model text, location text)")

try:
    with con:
        query = "INSERT into switch values (?, ?, ?, ?)"
        con.executemany(query, data)
except sqlite3.IntegrityError as e:
    print "Error occurred: ", e

for row in con.execute("select * from switch"):
    print row
```

Обратите внимание, что хотя транзакция будет откатываться, при возникновении исключения, само исключение всё равно надо перехватывать.

Для проверки этого функционала, попробуем записать в таблицу данные, в которых MAC-адрес повторяется (файл create_sw_inventory_ver3.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 primary key, hostname text, model text, location text)")

try:
    with con:
        query = "INSERT into switch values (?, ?, ?, ?)"
        con.executemany(query, data)
except sqlite3.IntegrityError as e:
    print "Error occurred: ", e

for row in con.execute("select * from switch"):
    print row

print '-'*30

#MAC-адрес sw7 совпадает с MAC-адресом существующего коммутатора - sw3
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')]

try:
    with con:
        query = "INSERT into switch values (?, ?, ?, ?)"
        con.executemany(query, data2)
except sqlite3.IntegrityError as e:
    print "Error occurred: ", e

for row in con.execute("select * from switch"):
    print row
```

Первая часть скрипта осталась неизменной. После, аналогичным образом записываются данные, но в списке data2 у коммутатора sw7 MAC-адрес совпадает с уже существующим коммутатором sw3.

Результат выполнения скрипта:

```
$ python create_sw_inventory_ver3.py
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
-----
Error occurred: UNIQUE constraint failed: switch.mac
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
```

Обратите внимание, что содержимое таблицы `switch` до и после второго 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.

Файл `create_sw_inventory_ver4.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 primary key, hostname text, model text, location text)")

try:
    with con:
        query = "INSERT into switch values (?, ?, ?, ?)"
        con.executemany(query, data)
except sqlite3.IntegrityError as e:
    print "Error occurred: ", e

for row in con.execute("select * from switch"):
    print row

print '-'*30

#MAC-адрес sw7 совпадает с MAC-адресом существующего коммутатора - sw3
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')]

for row in data2:
    try:
        with con:
            query = "INSERT into switch values (?, ?, ?, ?)"
            con.execute(query, row)
    except sqlite3.IntegrityError as e:
        print "Error occurred: ", e

for row in con.execute("select * from switch"):
    print row

```

Теперь результат выполнения будет таким (пропущен только sw7):

```
$ python create_sw_inventory_ver4.py
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
-----
Error occurred: UNIQUE constraint failed: switch.mac
(u'0000.AAAA.CCCC', u'sw1', u'Cisco 3750', u'London, Green Str')
(u'0000.BBBB.CCCC', u'sw2', u'Cisco 3780', u'London, Green Str')
(u'0000.AAAA.DDDD', u'sw3', u'Cisco 2960', u'London, Green Str')
(u'0011.AAAA.CCCC', u'sw4', u'Cisco 3750', u'London, Green Str')
(u'0055.AAAA.CCCC', u'sw5', u'Cisco 3750', u'London, Green Str')
(u'0066.BBBB.CCCC', u'sw6', u'Cisco 3780', u'London, Green Str')
(u'0088.AAAA.CCCC', u'sw8', u'Cisco 3750', u'London, Green Str')
```

Блок с менеджером контекста повторяется, поэтому конечно же можно сделать функцию, в которой будет находиться подобный блок кода.

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 dhcp (
    mac      text primary key,
    ip       text,
    vlan     text,
    interface text
);
```

Для всех полей определен тип данных "текст". И MAC-адрес является первичным ключом нашей таблицы. Что вполне логично, так как, MAC-адрес должен быть уникальным.

Теперь надо создать файл БД, подключиться к базе данных и создать таблицу (файл `create_sqlite_ver1.py`):

```
import sqlite3

with sqlite3.connect('dhcp_snooping.db') as conn:
    print 'Creating schema...'
    with open('dhcp_snooping_schema.sql', 'r') as f:
        schema = f.read()
    conn.executescript(schema)
print "Done"
```

Комментарии к файлу:

- используется менеджер контекста `with`
- при выполнении строки `with sqlite3.connect('dhcp_snooping.db') as conn :`
 - создается файл dhcp_snooping.db, если его нет
 - создается объект Connection
- в БД создается таблица, на основании команд, которые указаны в файле dhcp_snooping_schema.sql:
 - открываем файл dhcp_snooping_schema.sql
 - `schema = f.read()` - считываем весь файл как одну строку
 - `conn.executescript(schema)` - метод executescript позволяет выполнять команды SQL, которые прописаны в файле

Выполняем скрипт:

```
$ 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.

Выведем список созданных таблиц (запрос такого вида позволяет проверить какие таблицы созданы в DB):

```
$ 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
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, добавляются записи в БД (файл create_sqlite3_ver2.py):

```
import sqlite3
import re

regex = re.compile('(.+?) +(.*?) +\d+ +[\w-]+ +(\d+) +(.*$)')
result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        if line[0].isdigit():
            result.append(regex.search(line).groups())

with sqlite3.connect('dhcp_snooping.db') as conn:
    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:
        query = """insert into dhcp (mac, ip, vlan, interface)
values (?, ?, ?, ?)"""
        conn.execute(query, row)
```

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"
00:09:BB:3D:D6:58|10.1.10.2|10|FastEthernet0/1
00:04:A3:3E:5B:69|10.1.5.2|5|FastEthernet0/10
00:05:B3:7E:9B:60|10.1.5.4|5|FastEthernet0/9
00:07: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'"
00:04:A3:3E:5B:69|10.1.5.2|5|FastEthernet0/10
```

То есть, теперь на основании одного параметра, можно получать остальные.

Переделаем наш 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('(.+?) +(.*?) +\d+ +[\w-]+ +(\d+) +(.*$)')

with open(data_filename) as data:
    result = [regex.search(line).groups() for line in data if line[0].isdigit()]

db_exists = os.path.exists(db_filename)

with sqlite3.connect(db_filename) as conn:
    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 val in result:
        query = """insert into dhcp (mac, ip, vlan, interface)
values (?, ?, ?, ?)"""
        conn.execute(query, val)

```

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.

Проверим. В случае если файл уже есть:

```
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
```

Если файла нет (предварительно его удалить):

```
$ rm dhcp_snooping.db
$ python create_sqlite_ver3.py
Creating schema...
Done
Inserting DHCP Snooping data
```

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)

with sqlite3.connect(db_filename) as conn:
    #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, считаются 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 информацию 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 `description`

Файл `get_data_ver2.py`:

```
# -*- coding: utf-8 -*-
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

query_dict = {'vlan': "select * from dhcp where vlan = ?",
              'mac': "select * from dhcp where mac = ?",
              'ip': "select * from dhcp where ip = ?",
              'interface': "select * 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:

    with sqlite3.connect(db_filename) as conn:
        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,))
        # метод description позволяет получить заголовки полученных столбцов.
        # В нем будут находиться только те столбцы, который соответствуют запросу.
        all_rows = [r[0] for r in result.description]

        for row in result:
            for row_name in all_rows:
                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 упражнениях к этому разделу.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

На основе файла `create_sqlite_ver3.py` из 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 11.1а

Добавить в файл add_data.py, из задания 11.1, 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 11.2

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

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 11.2а

Дополнить скрипт get_data.py из задания 11.2

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 11.3

В прошлых заданиях информация добавлялась в пустую БД. В этом задании, разбирается ситуация, когда в БД уже есть информация.

Попробуйте выполнить скрипт add_data.py повторно, на существующей БД. Должна возникнуть ошибка.

При создании схемы БД, было явно указано, что поле МАС-адрес, должно быть уникальным. Поэтому, при добавлении записи с таким же МАС-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 удаленные строки отображаются в таблице как неактивные.

Задание 11.4

Обновить файл `get_data.py` из задания 11.2 или 11.2a.

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
-----
```

Задание 11.5

Теперь в БД остается и старая информация. И, если какой-то MAC-адрес не появлялся в новых записях, запись с ним, может оставаться в БД очень долго.

И, хотя это может быть полезно, чтобы посмотреть, где MAC-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 нем 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 11.5а

После выполнения задания 11.5, 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 11.6

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 нем в разделе [Дополнительная информация](#).

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 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
Getting 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 add sw1_dhcp_snooping.txt sw2_dhcp_snooping.txt sw3_dh  
cp_snooping.txt  
Reading info from file(s)  
sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt
```

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.

Для работы с сетевым оборудованием, удобнее использовать netmiko.

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 `raw_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
total 368
  8 -rw-r--r--  1 natasha  staff      372 Sep 26 08:36 LICENSE.md
  8 -rw-r--r--  1 natasha  staff     3483 Sep 26 08:36 README.md
 16 -rw-r--r--  1 natasha  staff    7098 Oct  5 11:41 SUMMARY.md
  8 -rw-r--r--  1 natasha  staff      70 Sep 25 19:01 about.md
  0 drwxr-xr-x 17 natasha  staff      578 Sep 25 19:05 book
  8 -rw-r--r--  1 natasha  staff     239 Sep 27 06:32 book.json
 288 -rw-r--r--  1 natasha  staff  146490 Sep 27 09:11 cover.jpg
  0 drwxr-xr-x  6 natasha  staff     204 Sep 25 19:01 exercises
 24 -rw-r--r--  1 natasha  staff   10024 Sep 25 19:01 faq.md
  0 drwxr-xr-x  3 natasha  staff     102 Sep 27 16:25 resources
  8 -rw-r--r--  1 natasha  staff   3633 Sep 27 15:52 schedule.md
```

pexpect.spawn

Класс `spawn` поддерживает больше возможностей. Он позволяет взаимодействовать с вызванной программой, отправляя данные и ожидая ответ.

Простой пример использования `pexpect.spawn`:

```
t = pexpect.spawn('ssh user@10.1.1.1')

t.expect('Password:')
t.sendline("userpass")
t.expect('>')
```

Сначала выполняется подключение по SSH, затем `pexpect` ожидает строку `Password:`. Как только эта строка появилась, отправляется пароль. После отправки, `pexpect` ожидает строку `>`.

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 = raw_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 %s" % IP
    t = pexpect.spawn('ssh %s@%s' % (USER, IP))

    t.expect('Password:')
    t.sendline(PASSWORD)

    t.expect('>')
    t.sendline('enable')

    t.expect('Password:')
    t.sendline(ENABLE_PASS)

    t.expect('#')
    t.sendline("terminal length 0")

    t.expect('#')
    t.sendline(COMMAND)

    t.expect('#')
    print t.before

```

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 pexpect до предыдущей подстроки в expect

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.

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 SUMMARY"')

In [3]: p.expect(pexpect.EOF)
Out[3]: 0

In [4]: print p.before
16 -rw-r--r-- 1 natasha staff 7156 Oct 5 13:05 SUMMARY.md
```

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')

/Library/Python/2.7/site-packages/pexpect/spawnbase.pyc in expect(self, pattern, timeout, searchwindowsize, async)
    313         compiled_pattern_list = self.compile_pattern_list(pattern)
    314         return self.expect_list(compiled_pattern_list,
--> 315             timeout, searchwindowsize, async)
    316
    317     def expect_list(self, pattern_list, timeout=-1, searchwindowsize=-1,
```



```
/Library/Python/2.7/site-packages/pexpect/spawnbase.pyc in expect_list(self, pattern_list, timeout, searchwindowsize, async)
```

```

337             return expect_async(exp, timeout)
338         else:
--> 339             return exp.expect_loop(timeout)
340
341     def expect_exact(self, pattern_list, timeout=-1, searchwindowsize=-1,
342
/Lib/Python/2.7/site-packages/pexpect/expect.pyc in expect_loop(self, timeout)
100             timeout = end_time - time.time()
101         except EOF as e:
--> 102             return self.eof(e)
103         except TIMEOUT as e:
104             return self.timeout(e)
105
106     def eof(self, err):
107
108         if err is not None:
109             msg = str(err) + '\n' + msg
--> 110         raise EOF(msg)
111
112     def timeout(self, err=None):
113
EOF: End Of File (EOF). Empty string style platform.
<pexpect.pty_spawn.spawn object at 0x107100b10>
command: /bin/bash
args: ['/bin/bash', '-c', 'ls -ls | grep SUMMARY']
searcher: None
buffer (last 100 chars): ''
before (last 100 chars): ' 16 -rw-r--r--  1 natasha  staff    7156 Oct  5 13:05 SUMMARY
RY.md\r\n'
after: <class 'pexpect.exceptions.EOF'>
match: None
match_index: None
exitstatus: 0
flag_eof: True
pid: 85765
child_fd: 7
closed: False
timeout: 30
delimiter: <class 'pexpect.exceptions.EOF'>
logfile: None
logfile_read: None
logfile_send: None
maxread: 2000
ignorecase: False
searchwindowsize: None
delaybeforesend: 0.05
delayafterclose: 0.1
delayafterterminate: 0.1

```

Но, если передать в expect EOF, ошибки не будет.

Возможности pexpect.expect

`pexpect.expect` в качестве шаблона может принимать не только строку.

Что может использоваться как шаблон в `pexpect.expect`:

- строка
- 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 SUMMARY"')  
In [8]: p.expect(['nattaur', 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

Документация `pexpect`

Документация модуля: [pexpect](#).

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`, поэтому пример ниже должен быть понятен.

Файл `2_telnetlib.py`:

```
import telnetlib
import time
import getpass
import sys

COMMAND = sys.argv[1]
USER = raw_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 %s" % IP
    t = telnetlib.Telnet(IP)

    t.read_until("Username:")
    t.write(USER + '\n')

    t.read_until("Password:")
    t.write(PASSWORD + '\n')
    t.write("enable\n")

    t.read_until("Password:")
    t.write(ENABLE_PASS + '\n')
    t.write("terminal length 0\n")
    t.write(COMMAND + '\n')

    time.sleep(5)

    output = t.read_very_eager()
    print output
```

Первая особенность, которая бросается в глаза - в конце отправляемых команд, надо добавлять символ перевода строки.

В остальном, telnetlib очень похож на rexpert:

- `t = telnetlib.Telnet(ip)` - 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: nata
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#
```

Мы не будем подробнее останавливаться на telnetlib. Остальные методы, которые поддерживает модуль, можно посмотреть в документации.

Документация модуля telnetlib: [telnetlib](#)

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 = raw_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 %s" % 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)
    ssh = client.invoke_shell()

    ssh.send("enable\n")
    ssh.send(ENABLE_PASS + '\n')
    time.sleep(1)

    ssh.send("terminal length 0\n")
    time.sleep(1)
    print ssh.recv(1000)

    ssh.send(COMMAND + "\n")
    time.sleep(2)
    result = ssh.recv(5000)
    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 поставив `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.
 - `ssh = client.invoke_shell()`
 - 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)
print result
```

Документация Paramiko

- Всё, что касается клиента: [Paramiko Client](#)
- Всё, что касается соединения (в нашем примере, всё, что относится к переменной `ssh`): [Paramiko Channel](#)

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):

```
from netmiko import ConnectHandler
import getpass
import sys

COMMAND = sys.argv[1]
USER = raw_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 %s" % IP
    DEVICE_PARAMS = {'device_type': 'cisco_ios',
                     'ip': IP,
                     'username': USER,
                     'password': PASSWORD,
                     'secret': ENABLE_PASS }

    ssh = ConnectHandler(**DEVICE_PARAMS)
    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'

- `ssh = ConnectHandler(**DEVICE_PARAMS)` - 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
- `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 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 закомитить 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):

```
from netmiko import ConnectHandler
import getpass
import sys
import time

COMMAND = sys.argv[1]
USER = raw_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 %s" % IP
    DEVICE_PARAMS = {'device_type': 'cisco_ios_telnet',
                     'ip': IP,
                     'username': USER,
                     'password': PASSWORD,
                     'secret': ENABLE_PASS,
                     'verbose': True}
    ssh = ConnectHandler(**DEVICE_PARAMS)
    ssh.enable()

    result = ssh.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()

Параллельные сессии

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 курсе используются два модуля:

- `threading`
- `multiprocessing`

Измерение времени выполнения скрипта

Для оценки времени выполнения скрипта есть несколько вариантов. В курсе используются самые простые варианты:

- утилиту 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 с 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
```

Проблемы с потоками в Python (CPython)

Подробности, которые описаны ниже не обязательно знать. В общем случае, лучше использовать модуль `multiprocessing` и можно плюс-минус не задумываться обо всех этих особенностях.

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 выполняться. 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 выполняться 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.

В любом случае, у потоков в Python есть свои области применения. В следующих разделах рассматривается как использовать потоки для подключения по Telnet/SSH. И проверяется, какое суммарное время будет занимать выполнение скрипта, по сравнению с последовательным исполнением и с использованием процессов.

Процессы

Обычно, чтобы не погружаться во все эти тонкости с GIL и потоками в Python, проще просто использовать модуль multiprocessing и всё.

Если же вы столкнетесь с ситуацией где вам нужно будет использовать потоки, скорее всего, к тому времени вы уже разберетесь с этим вопросом. Как минимум достаточно в целом знать об описанных выше особенностях.

Дополнительная информация

Если вы хотите подробнее разобраться с этими вопросами, или захотите вернуться к ним позже, несколько ссылок:

- GIL:
 - [GIL \(на русском\)](#)
 - [Inside the Python GIL](#)
- Отличная статья [Python threads and the GIL](#)
- Коротко о GIL, threads, processes:

- <http://stackoverflow.com/questions/3044580/multiprocessing-vs-threading-python>
- <http://stackoverflow.com/questions/18114285/python-what-are-the-differences-between-the-threading-and-multiprocessing-modul>

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 с 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, засчет 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 перенесен в функцию
- параметры устройств перенесены в отдельный файл в формате YAML

Файл `netmiko_function.py`:

```
from netmiko import ConnectHandler
import sys
import yaml

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command):

    print "Connection to device %s" % device_dict['ip']

    ssh = ConnectHandler(**device_dict)
    ssh.enable()

    result = ssh.send_command(command)
    print result

for router in devices['routers']:
    connect_ssh(router, COMMAND)
```

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):

```
from netmiko import ConnectHandler
import sys
import yaml
import threading

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)

    print "Connection to device %s" % 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 -*-
from netmiko import ConnectHandler
import sys
import yaml
import threading
from Queue import Queue

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)
    print "Connection to device %s" % device_dict['ip']

    #Добавляем словарь в очередь
    queue.put({ device_dict['ip']: result })

def conn_threads(function, devices, command):
    threads = []
    #Создаем очередь
    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

print 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. Но, конкретно в данном примере, можно было бы использовать и список.

Пример со списком, скорее всего, будет проще понять. Поэтому ниже аналогичный код, но с использованием обычного списка, вместо очереди (файл `netmiko_threading_data_list.py`):

```
# -*- coding: utf-8 -*-
from netmiko import ConnectHandler
import sys
import yaml
import threading

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)
    print "Connection to device %s" % 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()

    # Эта часть нам не нужна, так как, при использовании списка,
    # мы просто можем вернуть его
    #results = []
    #for t in threads:
    #    results.append(q.get())

    return q

print conn_threads(connect_ssh, devices['routers'], COMMAND)
```


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
from netmiko import ConnectHandler
import sys
import yaml

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)

    print "Connection to device %s" % 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

print( 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`.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 `send_show_command`.

Функция подключается по SSH (с помощью `netmiko`) к устройствам из списка, и выполняет команду на основании переданных аргументов.

Параметры функции:

- `devices_list` - список словарей с параметрами подключения к устройствам, которым надо передать команды
- `command` - команда, которую надо выполнить

Функция возвращает словарь с результатами выполнения команды:

- ключ - IP устройства
- значение - результат выполнения команды

Проверить работу функции на примере:

- устройств из файла `devices.yaml` (для этого надо считать информацию из файла)
- и команды `command`

```
import netmiko

command = "sh ip int br"

def send_show_command(device_list, command):
    """
    Функция подключается по SSH к устройствам из списка, и выполняет команду
    на основании переданных аргументов.

    Параметры функции:
    - devices_list - список словарей с параметрами подключения к устройствам,
        которым надо передать команды
    - command - команда, которую надо выполнить

    Функция возвращает словарь с результатами выполнения команды:
    - ключ - IP устройства
    - значение - результат выполнения команды
    """

```

Задание 12.2

Создать функцию `send_config_commands`

Функция подключается по SSH (с помощью `netmiko`) к устройствам из списка, и выполняет перечень команд в конфигурационном режиме на основании переданных аргументов.

Параметры функции:

- `devices_list` - список словарей с параметрами подключения к устройствам, которым надо передать команды
- `config_commands` - список команд, которые надо выполнить

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 списка команд `commands`

```
import netmiko

commands = [ 'logging 10.255.255.1',
             'logging buffered 20010',
             'no logging console' ]

def send_config_commands(device_list, config_commands):
    """
    Функция подключается по SSH к устройствам из списка,
    и выполняет команды в конфигурационном режиме.

    Параметры функции:
    - devices_list - список словарей с параметрами подключения к устройствам, которым надо передать команды
      - config_commands - список команд, которые надо выполнить

    Функция возвращает словарь с результатами выполнения команды:
    - ключ - IP устройства
    - значение - вывод с выполнением команд
    """

```

Задание 12.2a

Дополнить функцию `send_config_commands` из задания 12.2

Добавить аргумент `output`, который контролирует будет ли результат выполнения команд выводиться на стандартный поток вывода. По умолчанию, результат должен выводиться.

Задание 12.2b

Дополнить функцию `send_config_commands` из задания 12.2a или 12.2

Добавить проверку на ошибки:

- При выполнении команд, 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.

Проверить функцию на команде с ошибкой.

Задание 12.3

Создать функцию `send_commands` (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 зависимости от того, какой аргумент был передан, функция вызывает разные функции внутри.

Далее комбинация из аргумента и соответствующей функции:

- `show` -- функция `send_show_command` из задания 12.1
- `config` -- функция `send_config_commands` из задания 12.2, 12.2a или 12.2b
- `filename` -- функция `send_commands_from_file` (ее также 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`

```

from netmiko import ConnectHandler

commands = [ 'logging 10.255.255.1',
             'logging buffered 20010',
             'no logging console' ]
command = "sh ip int br"

def send_show_command(device_list, show_command):
    pass

def send_config_commands(device_list, config_commands, output=True):
    pass

def send_commands_from_file(device_list, filename):
    pass

def send_commands(device_list, config=[], show='', filename=''):
    pass

```

Задание 12.3а

Изменить функцию `send_commands` таким образом, чтобы в списке словарей `device_list` не надо было указывать имя пользователя, пароль, и пароль на enable.

Функция должна запрашивать имя пользователя, пароль и пароль на enable при старте. Пароль не должен отображаться при наборе.

В файле `devices2.yaml` эти параметры уже удалены.

Задание 12.3б

Дополнить функцию `send_commands` таким образом, чтобы перед подключением к устройствам по SSH, выполнялась проверка доступности устройства `ping`ом (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.

Для удобства можно сделать отдельную функцию для проверки доступности и затем использовать ее в функции `send_commands`.

Задание 12.4

В задании используется пример из раздела про [модуль threading](#).

Переделать пример таким образом, чтобы:

- вместо функции connect_ssh, использовалась функция send_commands из задания 12.3
 - переделать функцию send_commands, чтобы использовалась очередь и функция conn_threads по-прежнему возвращала словарь с результатами.
 - Проверить работу со списком команд, с командами из файла, с командой show

Пример из раздела:

```

from netmiko import ConnectHandler
import sys
import yaml
import threading
from Queue import Queue

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):

    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)
    print "Connection to device %s" % device_dict['ip']

    queue.put({ device_dict['ip']: result })

def conn_threads(function, devices, command):
    threads = []
    q = Queue()

    for device in devices:
        th = threading.Thread(target = function, args = (device, command, q))
        th.start()
        threads.append(th)

    for th in threads:
        th.join()

    results = []
    for t in threads:
        results.append(q.get())

    return results

print conn_threads(connect_ssh, devices['routers'], COMMAND)

```

Задание 12.5

Использовать функции полученные в результате выполнения задания 12.4.

Переделать функцию `conn_threads` таким образом, чтобы с помощью аргумента `limit`, можно было указывать сколько подключений будут выполняться параллельно. По умолчанию, значение аргумента должно быть 2.

Изменить функцию соответственно, так, чтобы параллельных подключений выполнялось столько, сколько указано в аргументе `limit`.

Задание 12.6

В задании используется пример из раздела про [модуль multiprocessing](#).

Переделать пример таким образом, чтобы:

- вместо функции connect_ssh, использовалась функция send_commands из задания 12.3
 - переделать функцию send_commands, чтобы использовалась очередь и функция conn_processes по-прежнему возвращала словарь с результатами.
 - Проверить работу со списком команд, с командами из файла, с командой show

Пример из раздела:

```

import multiprocessing
from netmiko import ConnectHandler
import sys
import yaml

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)

    print "Connection to device %s" % 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

print( conn_processes(connect_ssh, devices['routers'], COMMAND) )

```

Задание 12.7

Использовать функции полученные в результате выполнения задания 12.6.

Переделать функцию `conn_processes` таким образом, чтобы с помощью аргумента `limit`, можно было указывать сколько подключений будут выполняться параллельно. По умолчанию, значение аргумента должно быть 2.

Изменить функцию соответственно, так, чтобы параллельных подключений выполнялось столько, сколько указано в аргументе `limit`.

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(u"""
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 строкарендерит 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 сгенерировать конфигурационный файл
- router_config_generator.py - в этом скрипте импортируется файл с шаблоном и считывается информация из файла в формате YAML, а затем генерируются конфигурационные файлы маршрутизаторов

Файл router_template.py

```
# -*- coding: utf-8 -*-
from jinja2 import Template
import sys
reload(sys)
sys.setdefaultencoding('utf-8')

template_r1 = Template(u"""
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
!
""")
```

Эти строки меняют кодировку по умолчанию ascii на utf-8:

```
reload(sys)
sys.setdefaultencoding('utf-8')
```

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 jinja2 import Template
from router_template import template_r1

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

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

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}}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 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
import sys
reload(sys)
sys.setdefaultencoding('utf-8')

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

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

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

Файл router_config_generator.py импортирует из модуля jinja2:

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

```
# -*- coding: utf-8 -*-
from jinja2 import Environment, FileSystemLoader
import yaml
import sys
reload(sys)
sys.setdefaultencoding('utf-8')

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

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

vars_dict = yaml.load( open( VARS_FILE ) )

print template.render( vars_dict )
```

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` описываются 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.iteritems() %}
    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 `{% %}`. И 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.iteritems() %}
  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.iteritems() %}
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.iteritems() 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 (`|`) 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, засчет 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 удалит 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 созданы 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. Засчет блоков, которые были определены в базовом шаблоне, дочерний шаблон может расширять родительский шаблон.

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

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. Засчет 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 нем 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 нем
 - 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.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 13.1

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
reload(sys)
sys.setdefaultencoding('utf-8')

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

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

vars_dict = yaml.load( open( VARS_FILE ) )

print template.render( vars_dict )
```

Задание 13.1а

Переделать функцию generate_cfg_from_template:

- добавить поддержку использования шаблона, который находится в текущем каталоге

Для проверки, скопируйте один из шаблонов из каталога templates, в текущий каталог скрипта.

Можно проверить на тех же шаблоне и данных, что и в прошлом задании:

- шаблоне templates/for.txt (но скопировать его в текущий каталог) и данных data_files/for.yml

Задание 13.1б

Дополнить функцию generate_cfg_from_template из задания 13.1 или 13.1а:

- добавить поддержку аргументов окружения (Environment)

Функция generate_cfg_from_template должна принимать любые аргументы, которые принимает класс Environment и просто передавать их ему.

Проверить функциональность на аргументах:

- trim_blocks
- lstrip_blocks

Задание 13.1с

Дополнить функцию generate_cfg_from_template из задания 13.1, 13.1a или 13.1b:

- 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 13.1d

Переделать функцию generate_cfg_from_template из задания 13.1, 13.1a, 13.1b или 13.1c:

- 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 13.2

На основе конфигурации `config_r1.txt`, создать шаблоны:

- `templates/cisco_base.txt` - в нем 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 13.1-13.1d. Не копируйте код функции.

В качестве данных, используйте словарь router_info

```
router_info = { 'hostname': 'R1' }
```

Задание 13.3

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 13.1-13.1d. Не копируйте код функции.

Задание 13.3a

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 13.1-13.1d. Не копируйте код функции.

Задание 13.3b

Измените шаблон templates/ospf.txt из задания 13.3a 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 13.1-13.1d. Не копируйте код функции.

Задание 13.4

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 13.1-13.1d. 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 gtextfsm
```

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.textfsm')
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.textfsm')` - 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

Шаблон TextFSM описывает каким образом данные должны обрабатываться.

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.

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 Action** - переход в другое состояние
- если нет указанных действий, то по умолчанию используется действие **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 значение, 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 состояния 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 нем 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 и только шаблон и данные 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 выводе `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. Засчет 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 нем. 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}\\\($\S+ +[\w-]+ +[\w ]+ +\\$+ +${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. Но, 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 ей 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.

Если при работе с `clitable` возникнет ошибка, что нет модуля `terminal`, посмотрите в каталоге, в котором находится установленный TextFSM (в зависимости от ОС), файл `terminal.py`. Если его нет, просто создайте его вручную и скопируйте содержимое этого файла из [репозитория TextFSM](#). Определить, где находится модуль, можно таким образом: `print textfsm.__file__`

Для начала, импортируем класс `clitable`:

```
In [1]: import textfsm.clitable as 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  
able.template_dir  
cli_table.KeyValue  
cli_table.LabelValueTable
```

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  
=====
```

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 = []

In [10]: for row in cli_table:
....:     current_row = []
....:     for value in row:
....:         current_row.append(value)
....:     data_rows.append(current_row)
....:

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]: cli_table.header.viewvalues()
Out[12]: dict_values([])

In [13]: header = []

In [13]: for name in cli_table.header:
....:     header.append(name)
....:

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 textfsm.clitable as 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 = []

for row in cli_table:
    current_row = []
    for value in row:
        current_row.append(value)
    data_rows.append(current_row)

header = []
for name in cli_table.header:
    header.append(name)

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.

Вывод 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.

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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 14.1

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]

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)
```

Задание 14.1а

Переделать функцию `parse_output` из задания 14.1 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 14.2

В этом задании нужно использовать функцию `parse_output` из задания 14.1. 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 14.3

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 14.1.

Задание 14.4

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 14.1а):

- ключи - названия столбцов
- значения - соответствующие значения в столбцах

Проверить работу функции на примере вывода команды `sh ip int br`.

Пример из раздела:

```
import textfsm.clitable as 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 = []

for row in cli_table:
    current_row = []
    for value in row:
        current_row.append(value)
    data_rows.append(current_row)

header = []
for name in cli_table.header:
    header.append(name)

print header
for row in data_rows:
    print row
```

Задание 14.4а

Переделать функцию из задания 14.4:

- 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 в виде, который возвращает метод `FormattedTable` (пример есть в разделе)

Задание 14.5

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 12.1
- функцией parse_command_dynamic из упражнения 14.4

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 14.6

Это задание похоже на задание 14.5, но в этом задании подключения надо выполнять параллельно.

Для этого надо использовать функции connect_ssh и conn_processes (пример из раздела multiprocessing) и функцию parse_command_dynamic из упражнения 14.4.

В этом упражнении нужно создать функцию send_and_parse_command_parallel:

- она должна использовать внутри себя функции connect_ssh, conn_processes и parse_command_dynamic
- какие аргументы должны быть у функции send_and_parse_command_parallel, нужно решить самостоятельно

- но надо иметь в виду, какие аргументы ожидают три готовые функции, которые используются
- функция send_and_parse_command_parallel должна возвращать словарь, в котором:
 - ключ - IP устройства
 - значение - список словарей (то есть, тот вывод, который был получен из функции parse_command_dynamic)

Для функции conn_processes создан файл devices.yaml, в котором находятся параметры подключения к устройствам.

Проверить работу функции send_and_parse_command_parallel на команде sh ip int br.

Пример из раздела multiprocessing:

```
import multiprocessing
from netmiko import ConnectHandler
import sys
import yaml

COMMAND = sys.argv[1]
devices = yaml.load(open('devices.yaml'))

def connect_ssh(device_dict, command, queue):
    ssh = ConnectHandler(**device_dict)
    ssh.enable()
    result = ssh.send_command(command)

    print "Connection to device %s" % 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

print( conn_processes(connect_ssh, devices['routers'], COMMAND) )
```

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 нем 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 его довольно легко начать использовать.

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 2.7 (или 2.6)
- Windows не может быть управляемым хостом

Ansible довольно часто обновляется, поэтому лучше установить его в виртуальном окружении.

Установить Ansible можно [по-разному](#).

Например, с помощью pip:

```
$ pip install ansible==2.2.0.0
```

Пока что, лучше поставить Ansible с явным указанием версии, так как в версии 2.2.1.0 были внесены изменения из-за которых часть функционала работает с ошибками. Скорее всего, это скоро исправят, но при моей последней проверке (05.02.2017), были проблемы.

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 нем 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`.

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.

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.

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
```

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 аутентификацию 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 приоритетности):

- 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] - это 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 выполнения 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 поддерживаются 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

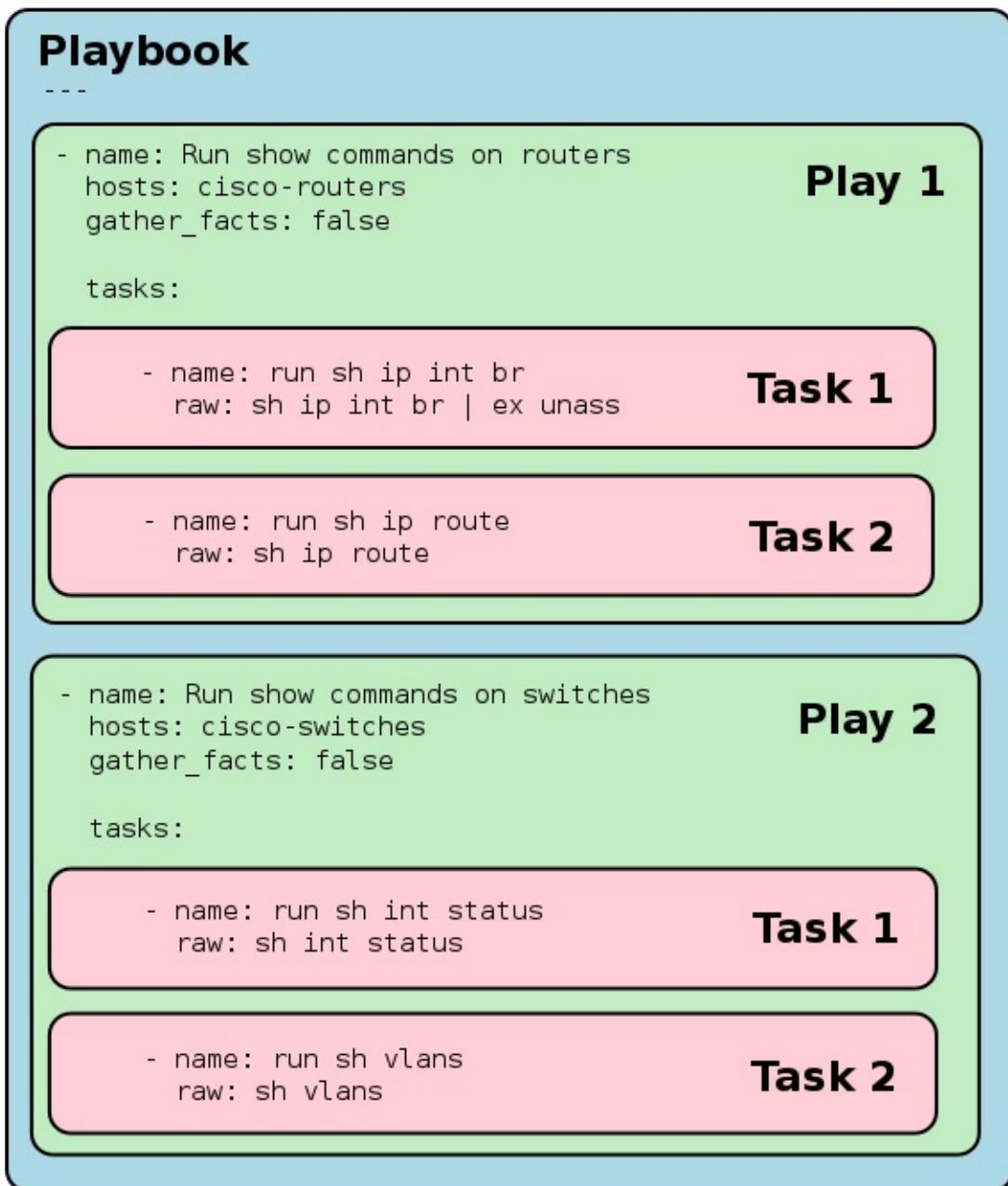
Пример playbook 1_show_commands_with_raw.yml:

```
---  
  
- 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 \r\nEthernet0/1
192.168.200.1 YES NVRAM up \r\nLoopback0
10.1.1.1 YES manual up \r\n
": [{"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
"}]}
```

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 нем 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
```

```
PLAY [Run show commands on routers] ****
TASK [run sh ip int br] ****
changed: [192.168.100.3]
changed: [192.168.100.2]
fatal: [192.168.100.1]: FAILED! => {"changed": true, "failed": true, "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 vlans] ****
changed: [192.168.100.100]
      to retry, use: --limit @/home/nata/pyneng_course/chapter15/1_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/nata/pyneng_course/chapter15/1_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/nata/pyneng_course/chapter15/1_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.retry
```

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 в 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 разпознает 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"
  transport: cli
  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-адрес 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
```

Приоритетность 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 to 192.168.100.1 closed.\r\n", "stdout": "\r\nInterface IP-Address OK? Method Status Protocol\r\nEthernet0/0 192.168.100.1 YES NVRAM up \r\nEthernet0/1 192.168.200.1 YES NVRAM up \r\nLoopback0 10.1.1.1 YES manual up \r\nIP-Address OK? Method Status Protocol", "Ethernet0/0": {"IP-Address": "192.168.100.1", "OK?": "YES", "Method": "NVRAM", "Status": "up", "Protocol": "Ethernet0/0"}, "Ethernet0/1": {"IP-Address": "192.168.200.1", "OK?": "YES", "Method": "NVRAM", "Status": "up", "Protocol": "Ethernet0/1"}, "Loopback0": {"IP-Address": "10.1.1.1", "OK?": "YES", "Method": "manual", "Status": "up", "Protocol": "Loopback0"}}
```

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 будут 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 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 курса, уже есть следующая стабильная версия Ansible (версия в курсе 2.2), используйте её и посмотрите в документации, какие новые возможности и модули появились.

В курсе, 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, в курсе рассматривается модуль ntc-ansible, который не входит в core модули Ansible.

Варианты подключения

Ansible поддерживает такие типы подключений:

- **paramiko**
- **SSH** - OpenSSH. Используется по умолчанию
- **local** - действия выполняются локально, на управляемом хосте

При подключении по SSH, по умолчанию используются SSH ключи, но можно переключиться на использование паролей.

По умолчанию, Ansible загружает модуль Python на устройство, для того, чтобы выполнить действия. Если же оборудование не поддерживает Python, как в случае с доступом к сетевому оборудованию через CLI, нужно указать, что модуль должен запускаться локально, на управляемом хосте Ansible.

Особенности подключения к сетевому оборудованию

При работе с сетевым оборудованием, есть несколько параметров в playbook, которые нужно менять:

- `gather_facts` - надо отключить, так как для сетевого оборудования используются свои модули сбора фактов
- `connection` - управляет тем, как именно будет происходить подключение. Для сетевого оборудования необходимо установить в `local`

То есть, для каждого сценария (`play`), нужно указывать:

- `gather_facts: false`
- `connection: local`

Пример:

```
---
```

```
- name: Run show commands on routers
  hosts: cisco-routers
  gather_facts: false
  connection: local
```

В Ansible переменные можно указывать в разных местах, поэтому те же настройки можно указать по-другому.

Например, в разделе о [конфигурационном файле](#) рассматривалось как отключить сбор фактов по умолчанию (файл `ansible.cfg`):

```
[defaults]
gathering = explicit
```

Такой вариант подходит в том случае, когда Ansible используется больше для подключения к сетевым устройствам (или, локальные playbook используются для подключения к сетевому оборудованию).

В таком случае, нужно будет наоборот явно включать сбор фактов, если он нужен.

Указать, что нужно использовать локальное подключение, также можно по-разному.

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=local
```

Или в файлах переменных, например, в `group_vars/all.yml`:

```
---
ansible_connection: local
```

В следующих разделах будет использоваться такой вариант:

```
---
- name: Run show commands on routers
  hosts: cisco-routers
  gather_facts: false
  connection: local
```

В реальной жизни нужно выбрать тот вариант, который наиболее удобен для работы.

Аргумент provider

Модули, которые используются для работы с сетевым оборудованием, требуют задания нескольких аргументов.

Для каждой задачи должны быть указаны такие аргументы:

- **host** - имя или IP-адрес удаленного устройства
- **port** - к какому порту подключаться
- **username** - имя пользователя
- **password** - пароль
- **transport** - тип подключения: CLI или API. По умолчанию - cli
- **authorize** - нужно ли переходить в привилегированный режим (enable, для Cisco)
- **auth_pass** - пароль для привилегированного режима

Если для подключения Ansible создан отдельный пользователь с privilege 15, можно не использовать параметры `authorize` и `auth_pass`.

Но, Ansible также позволяет собрать их в один аргумент - **provider**.

Пример задания всех аргументов в задаче (task):

```
tasks:
  - name: run show version
    ios_command:
      commands: show version
      host: "{{ inventory_hostname }}"
      username: cisco
      password: cisco
      transport: cli
```

Аргументы созданы как переменная `cli` в playbook, а затем передаются как переменная аргументу provider:

```
vars:
  cli:
    host: "{{ inventory_hostname }}"
    username: cisco
    password: cisco
    transport: cli

tasks:
  - name: run show version
    ios_command:
      commands: show version
      provider: "{{ cli }}"
```

И, самый удобный вариант, задавать аргументы в каталоге group_vars.

Например, если у всех устройств одинаковые значения аргументов, можно задать их в файле group_vars/all.yml:

```
---
cli:
  host: "{{ inventory_hostname }}"
  username: cisco
  password: cisco
  transport: cli
  authorize: yes
  auth_pass: cisco
```

Затем переменная используется в playbook так же, как и в случае указания переменных в playbook:

```

tasks:
  - name: run show version
    ios_command:
      commands: show version
      provider: "{{ cli }}"

```

Кроме того, Ansible поддерживает задание параметров в переменных окружения:

- ANSIBLE_NET_USERNAME - для переменной username
- ANSIBLE_NET_PASSWORD - password
- ANSIBLE_NET_SSH_KEYFILE - ssh_keyfile
- ANSIBLE_NET_AUTHORIZE - authorize
- ANSIBLE_NET_AUTH_PASS - auth_pass

Приоритетность значений в порядке возрастания приоритетности:

- значения по умолчанию
- значения переменных окружения
- параметр provider
- аргументы задачи (task)

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

remote_user = cisco
ask_pass = True

```

В файле group_vars/all.yml надо создать переменную cli, чтобы не указывать каждый раз все параметры, которые нужно передать аргументу provider:

```
---
```

```
cli:
  host: "{{ inventory_hostname }}"
  username: "cisco"
  password: "cisco"
  transport: cli
  authorize: yes
  auth_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`.

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.

Пример использования модуля `ios_command` (playbook `1_ios_command.yml`):

```
---
- name: Run show commands on routers
  hosts: cisco-routers
  gather_facts: false
  connection: local

  tasks:
    - name: run sh ip int br
      ios_command:
        commands: show ip int br
        provider: "{{ cli }}"
        register: sh_ip_int_br_result

    - name: Debug registered var
      debug: var=sh_ip_int_br_result.stdout_lines
```

Модуль `ios_command` ожидает параметры:

- `commands` - список команд, которые нужно отправить на устройство
- `provider` - словарь с параметрами подключения
 - в нашем случае, он указан в файле `group_vars/all.yml`

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
  gather_facts: false
  connection: local

  tasks:

    - name: run show commands
      ios_command:
        commands:
          - show ip int br
          - sh ip route
        provider: "{{ cli }}"
      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.

Засчет 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
```

SSH password:

```
PLAY [Run show commands on routers] *****

TASK [run sh ip int br] *****
fatal: [192.168.100.1]: FAILED! => {"changed": false, "failed": true, "msg": "matched error in response: shw ip int br\r\n      ^\r\n% Invalid input detected at '^' marker.\r\n\r\n\r\nR1#"}
fatal: [192.168.100.2]: FAILED! => {"changed": false, "failed": true, "msg": "matched error in response: shw ip int br\r\n      ^\r\n% Invalid input detected at '^' marker.\r\n\r\n\r\nR2#"}
fatal: [192.168.100.3]: FAILED! => {"changed": false, "failed": true, "msg": "matched error in response: shw ip int br\r\n      ^\r\n% Invalid input detected at '^' marker.\r\n\r\n\r\nR3#"}
      to retry, use: --limit @/home/nata/pyneng_course/chapter15/2_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

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`

Для того, 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 ipv6 interface`
 - `show lldp`

- show lldp neighbors detail

Собрать все факты:

```
- ios_facts:  
  gather_subset: all  
  provider: "{{ cli }}"
```

Собрать только подмножество interfaces:

```
- ios_facts:  
  gather_subset:  
    - interfaces  
  provider: "{{ cli }}"
```

Собрать всё, кроме hardware:

```
- ios_facts:  
  gather_subset:  
    - "!hardware"  
  provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Facts
      ios_facts:
        gather_subset: all
        provider: "{{ cli }}"

```

```
$ 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
  gather_facts: false
  connection: local

  tasks:
    - name: Facts
      ios_facts:
        gather_subset: all
        provider: "{{ cli }}"

    - 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, 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
  gather_facts: false
  connection: local

  tasks:

    - name: Facts
      ios_facts:
        gather_subset: all
        provider: "{{ cli }}"
      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
```

Изменения с опцией --diff

В Ansible можно не только увидеть, что изменения произошли, но и увидеть какие именно изменения были сделаны. Например, в ситуации с сохранением фактов об устройстве это может быть очень полезно.

Пример запуска playbook с опцией --diff и с внесенными изменениями на одном из устройств:

```
$ ansible-playbook 3_ios_facts.yml --diff --limit=192.168.100.1
```

```
SSH password:

PLAY [Collect IOS facts] *****

TASK [Facts] *****
ok: [192.168.100.1]

TASK [Copy facts to files] *****
--- before: all_facts/192.168.100.1_facts.json
+++ after: /tmp/tmp09sKQx
@@ -3,7 +3,7 @@
    "ansible_net_all_ipv4_addresses": [
        "192.168.200.1",
        "192.168.100.1",
        "10.1.1.1"
+       "10.10.1.1"
    ],
    "ansible_net_all_ipv6_addresses": [],
    "ansible_net_filesystems": []
@@ -76,11 +76,11 @@
        "description": null,
        "duplex": null,
        "ipv4": {
-           "address": "10.1.1.1",
+           "address": "10.10.1.1",
            "masklen": 32
        },
        "lineprotocol": "up",
-       "macaddress": "10.1.1.1/32",
+       "macaddress": "10.10.1.1/32",
        "mediatype": null,
        "mtu": 1514,
        "operstatus": "up",
changed: [192.168.100.1]

PLAY RECAP *****
192.168.100.1 : ok=2     changed=1     unreachable=0    failed=0
```

В этом выводе видно не только то, что были внесены изменения, но и то, на каком устройстве и какие именно изменения.

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, и 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 должны быть в глобальном режиме конфигурации
- **replace** - параметр указывает как выполнять настройку устройства
- **save** - 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.

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config password encryption
      ios_config:
        lines:
          - service password-encryption
        provider: "{{ cli }}"

```

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 выполниться 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  
gather_facts: false  
connection: local  
  
tasks:  
  
- name: Send config commands  
ios_config:  
  lines:  
    - service password-encryption  
    - no ip http server  
    - no ip http secure-server  
    - no ip domain lookup  
  provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
        provider: "{{ cli }}"
```

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 100000000 1000000
```

Тогда playbook 2_ios_config_parents_mult.yml будет выглядеть так:

```
---
- name: Run cfg commands on routers
  hosts: cisco-routers
  gather_facts: false
  connection: local

  tasks:
    - name: Config QoS policy
      ios_config:
        parents:
          - policy-map OUT_QOS
          - class class-default
        lines:
          - shape average 100000000 1000000
        provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
        provider: "{{ cli }}"
```

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/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on routers] ****
TASK [Config line vty] ****
ok: [192.168.100.3] => {"changed": false, "warnings": []}
changed: [192.168.100.1] => {"changed": true, "updates": ["line vty 0 4", "transport input ssh"], "warnings": []}
ok: [192.168.100.2] => {"changed": false, "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
```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
        provider: "{{ cli }}"
      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 ssh"  
    ]  
}  
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

Параметр **save** позволяет указать нужно ли сохранять текущую конфигурацию в стартовую. По умолчанию, значение параметра - **no**.

Доступные варианты значений:

- no (или false)
- yes (или true)

К сожалению, на данный момент (версия ansible 2.2), 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 выставленным в yes, появляется такая ошибка:

```
$ ansible-playbook 4_ios_config_save.yml
```

```
SSH password:  
  
PLAY [Run cfg commands on routers] *****  
  
TASK [Config line vty] *****  
fatal: [192.168.100.3]: FAILED! => {"changed": false, "failed": true, "msg": "timeout  
trying to send command: copy running-config startup-config\\r"}  
fatal: [192.168.100.2]: FAILED! => {"changed": false, "failed": true, "msg": "timeout  
trying to send command: copy running-config startup-config\\r"}  
fatal: [192.168.100.1]: FAILED! => {"changed": false, "failed": true, "msg": "timeout  
trying to send command: copy running-config startup-config\\r"}  
      to retry, use: --limit @/home/nata/pyneng_course/chapter15/6c_ios_config_save  
.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
```

Но, можно самостоятельно сделать сохранение, используя модуль `ios_command`.

Playbook `4_ios_config_save.yml`:

```
---  
- name: Run cfg commands on routers  
  hosts: cisco-routers  
  gather_facts: false  
  connection: local  
  
  tasks:  
  
    - name: Config line vty  
      ios_config:  
        parents:  
          - line vty 0 4  
        lines:  
          - login local  
          - transport input ssh  
        #save: yes - в версии 2.2 не работает корректно  
        provider: "{{ cli }}"  
      register: cfg  
  
    - name: Save config  
      ios_command:  
        commands:  
          - write  
      provider: "{{ cli }}"  
      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.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
  gather_facts: false
  connection: local

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
        backup: yes
        provider: "{{ cli }}"
```

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
```

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.

Присходит 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
  gather_facts: false
  connection: local

  tasks:

    - name: Config interface
      ios_config:
        parents:
          - interface Ethernet0/2
        lines:
          - ip address 192.168.200.1 255.255.255.0
          - ip mtu 1500
      provider: "{{ cli }}
```

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  
  gather_facts: false  
  connection: local  
  
  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  
        provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config interface
      ios_config:
        parents:
          - interface Ethernet0/3
        lines:
          - ip address 192.168.230.1 255.255.255.0
        after:
          - no shutdown
      provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config line vty
      ios_config:
        parents:
          - line vty 0 4
        lines:
          - login local
          - transport input ssh
        after:
          - end
          - write
      provider: "{{ cli }}"

```

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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"

```

В playbook 8_ios_config_before.yml ACL IN_to_OUT сначала удалятся, 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 их положение относительно друг друга
- **exact** - команды должны в точности сопадать 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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"

```

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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"

```

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 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

```

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.

В версии Ansible 2.1 `match: exact` работал по-другому и такой результат достигался комбинацией параметров `match: exact` и `replace: block`. В версии 2.2 достаточно `match: exact`.

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
  gather_facts: false
  connection: local

  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
        provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  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
        provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"
```

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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"
```

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 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
  gather_facts: false
  connection: local

  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
      provider: "{{ cli }}"

```

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
  gather_facts: false
  connection: local

  tasks:

    - name: Config ACL
      ios_config:
        src: templates/acl_cfg.txt
        provider: "{{ cli }}"
```

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
```

```
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, "warnings": []}

PLAY RECAP ****
192.168.100.1 : ok=1    changed=1    unreachable=0    failed=0
```

Неприятная особенность параметра src в том, что не видно какие изменения были внесены. Но, возможно, в следующих версиях Ansible это будет исправлено.

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, но 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
  gather_facts: false
  connection: local

  tasks:

    - name: Config OSPF
      ios_config:
        src: templates/ospf.j2
        provider: "{{ cli }}"
```

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
```

```
Using /home/nata/pyneng_course/chapter15/ansible.cfg as config file
SSH password:

PLAY [Run cfg commands on router] ****
TASK [Config OSPF] ****
changed: [192.168.100.2] => {"changed": true, "warnings": []}
changed: [192.168.100.3] => {"changed": true, "warnings": []}
changed: [192.168.100.1] => {"changed": true, "warnings": []}

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, но 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
```

Если при установке возникнут 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 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](#).

Задания

Все задания и вспомогательные файлы можно скачать одним архивом [zip](#) или [tar.gz](#).

Также для курса подготовлены две виртуальные машины на выбор: [Vagrant](#) или [VMware](#). В них установлены все пакеты, которые используются в курсе.

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

Создайте playbook, 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 15.1a

Создайте playbook, 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 15.1b

Создайте playbook, 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 15.1с

Создайте playbook, 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 15.2

Создайте playbook, 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 15.2а

Создайте playbook, 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 15.2b

Создайте playbook, 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 15.3

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 настроен неправильный режим логирования в консоль.

Задание 15.4

Создайте playbook, который выполняет такую задачу:

- создает ACL INET-to-LAN и применяет его к интерфейсу `Ethernet0/1`

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 15.4a

Проверьте работу playbook из задания 15.4, 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 15.4b

Добавьте в playbook из задания 15.4a 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 15.4c

Измените playbook из задания 15.4b 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.

Полезные модули

В этом разделе описаны такие модули:

- subprocess
- os
- argparse
- ipaddress

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.

Функция subprocess.call()

Функция `call()` :

- позволяет выполнить команду
 - при этом, она ожидает завершения команды.
- функция возвращает код возврата

Пример выполнения команды `ls` :

```
In [1]: import subprocess

In [2]: result = subprocess.call('ls')
LICENSE.md      course_presentations      faq.md
README.md       course_presentations.zip   howto.md
SUMMARY.md      cover.jpg                 images
ToDo.md         examples                  resources
about.md        examples.zip              schedule.md
book            exercises
book.json       exercises.zip
```

В переменной `result` теперь содержится код возврата (код 0 означает, что программа выполнилась успешно):

```
In [3]: print result
0
```

Обратите внимание, что, если необходимо вызвать команду с аргументами, её нужно передавать таким образом (как список):

```
In [4]: result = subprocess.call(['ls', '-ls'])
total 3624
  8 -rw-r--r--  1 nata  nata      372 Dec 10 21:34 LICENSE.md
 16 -rw-r--r--  1 nata  nata     4528 Jan 12 09:16 README.md
 32 -rw-r--r--  1 nata  nata    12480 Jan 23 11:15 SUMMARY.md
   8 -rw-r--r--  1 nata  nata     2196 Jan 23 09:16 ToDo.md
   8 -rw-r--r--  1 nata  nata       70 Dec 10 21:34 about.md
   0 drwxr-xr-x 19 nata  nata     646 Jan 23 11:05 book
   8 -rw-r--r--  1 nata  nata     355 Jan 12 09:16 book.json
   0 drwxr-xr-x 16 nata  nata     544 Dec 10 21:34 course_presentations
2176 -rw-r--r--  1 nata  nata 1111234 Dec 10 21:34 course_presentations.zip
 528 -rw-r--r--@ 1 nata  nata 267824 Dec 11 08:25 cover.jpg
   0 drwxr-xr-x 20 nata  nata     680 Jan 23 13:05 examples
 360 -rw-r--r--  1 nata  nata 181075 Jan 21 14:10 examples.zip
   0 drwxr-xr-x 19 nata  nata     646 Jan 17 10:24 exercises
 416 -rw-r--r--  1 nata  nata 210621 Jan 21 14:10 exercises.zip
  32 -rw-r--r--  1 nata  nata    14684 Jan 18 05:33 faq.md
 16 -rw-r--r--  1 nata  nata    7043 Jan 17 10:28 howto.md
   0 drwxr-xr-x  4 nata  nata     136 Jan 14 11:01 images
   0 drwxr-xr-x 10 nata  nata    340 Jan 17 08:44 resources
 16 -rw-r--r--@  1 nata  nata   6219 Jan 17 11:37 schedule.md
```

Все файлы, с расширением md:

```
In [5]: result = subprocess.call(['ls', '-ls', '*md'])
ls: *md: No such file or directory
```

Возникла ошибка.

Чтобы вызывать команды, в которых используются регулярные выражения, нужно добавлять параметр shell:

```
In [6]: result = subprocess.call(['ls', '-ls', '*md'], shell=True)
LICENSE.md           course_presentations      faq.md
README.md           course_presentations.zip  howto.md
SUMMARY.md          cover.jpg                  images
ToDo.md             examples                  resources
about.md            examples.zip              schedule.md
book                exercises
book.json           exercises.zip
```

Если установлен аргумент `shell=True`, указанная команда выполняется через shell. В таком случае, команду можно указывать так:

```
In [7]: result = subprocess.call('ls -ls *md', shell=True)
  8 -rw-r--r-- 1 nata  nata   372 Dec 10 21:34 LICENSE.md
16 -rw-r--r-- 1 nata  nata  4528 Jan 12 09:16 README.md
32 -rw-r--r-- 1 nata  nata 12480 Jan 23 11:15 SUMMARY.md
  8 -rw-r--r-- 1 nata  nata  2196 Jan 23 09:16 ToDo.md
  8 -rw-r--r-- 1 nata  nata    70 Dec 10 21:34 about.md
32 -rw-r--r-- 1 nata  nata 14684 Jan 18 05:33 faq.md
16 -rw-r--r-- 1 nata  nata  7043 Jan 17 10:28 howto.md
16 -rw-r--r--@ 1 nata  nata  6219 Jan 17 11:37 schedule.md
```

Ещё одна особенность функции `call()` - она ожидает завершения выполнения команды. Если попробовать, например, запустить команду `ping`, то этот аспект будет заметен:

```
In [8]: reply = subprocess.call(['ping', '-c', '3', '-n', '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=49.868 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.243 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=50.029 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 49.243/49.713/50.029/0.339 ms
```

Особенно, если попробовать пингануть какой-то недоступный IP-адрес.

Функция `call()` подходит, если нужно:

- подождать выполнения программы, прежде чем выполнять следующие шаги
- нужно получить только код выполнения и не нужен вывод

Ещё один аспект работы функции `call()`, она выводит результат выполнения команды, на стандартный поток вывода.

Файл `subprocess_call.py`:

```
import subprocess

reply = subprocess.call(['ping', '-c', '3', '-n', '8.8.8.8'])

if reply == 0:
    print "Alive"
else:
    print "Unreachable"
```

Результат выполнения будет таким:

```
$ python subprocess_call.py
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=49.930 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=48.981 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=48.360 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.360/49.090/49.930/0.646 ms
Alive
```

То есть, результат выполнения команды, выводится на стандартный поток вывода.

Если нужно это отключить и не выводить результат выполнения, надо перенаправить stdout в devnull (файл subprocess_call_devnull.py):

```
import subprocess
import os

DNULL = open(os.devnull, 'w')

reply = subprocess.call(['ping', '-c', '3', '-n', '8.8.8.8'], stdout=DNULL)

if reply == 0:
    print "Alive"
else:
    print "Unreachable"
```

Теперь результат выполнения будет таким:

```
$ python subprocess_call_devnull.py
Alive
```

Функция `subprocess.check_output()`

Функция `check_output()`:

- позволяет выполнить команду
 - при этом, она ожидает завершения команды.
- если команда отработала корректно (код возврата 0), функция возвращает результат выполнения команды
- если возникла ошибка, при выполнении команды, функция генерирует исключение

Пример использования функции `check_output()` (файл subprocess_check_output.py):

```
import subprocess

reply = subprocess.check_output(['ping', '-c', '3', '-n', '8.8.8.8'])

print "Result:"
print reply
```

Результат выполнения (если убрать строку `print reply`, на стандартный поток вывода ничего не будет выведено):

```
$ python subprocess_check_output.py
Result:
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=49.785 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=57.231 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=51.071 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 49.785/52.696/57.231/3.250 ms
```

Если выполнить команду, которая вызовет ошибку и, соответственно, код возврата будет не 0 (файл `subprocess_check_output_catch_exception.py`):

```
$ python subprocess_check_output_catch_exception.py
ping: cannot resolve a: Unknown host
Traceback (most recent call last):
  File "subprocess_check_output_catch_exception.py", line 3, in <module>
    reply = subprocess.check_output(['ping', '-c', '3', '-n', 'a'])
  File "/usr/local/Cellar/python/2.7.11/Frameworks/Python.framework/Versions/2.7/lib/python2.7/subprocess.py", line 573, in check_output
    raise CalledProcessError(retcode, cmd, output=output)
subprocess.CalledProcessError: Command '['ping', '-c', '3', '-n', 'a']' returned non-zero exit status 68
```

Возникло исключение `CalledProcessError` и соответствующее сообщение об ошибке.

Функция `check_output()` всегда будет возвращать это исключение, когда код возврата не равен 0.

Это значит, что в скрипте можно написать выражение `try/except`, с помощью которого будет выполняться проверка корректно ли отработала команда (дополняем файл `subprocess_check_output_catch_exception.py`):

```
import subprocess

try:
    reply = subprocess.check_output(['ping', '-c', '3', '-n', 'a'])
except subprocess.CalledProcessError as e:
    print "Error occurred"
    print "Return code:", e.returncode
```

Результат выполнения:

```
$ python subprocess_check_output_catch_exception.py
ping: cannot resolve a: Unknown host
Error occurred
Return code: 68
```

Теперь программа завершилась корректно и вывела сообщение об ошибке и код возврата. И, хотя сообщение об ошибке, не выводилось, оно попало на стандартный поток вывода.

Попробуем собрать всё в финальную функцию и добавим перехват сообщения об ошибке:

```
import subprocess
from tempfile import TemporaryFile

def ping_ip(ip_address):
    """
    Ping IP address and return tuple:
    On success:
        * return code = 0
        * command output
    On failure:
        * return code
        * error output (stderr)
    """
    with TemporaryFile() as temp:
        try:
            output = subprocess.check_output(['ping', '-c', '3', '-n', ip_address],
                                            stderr=temp)
            return 0, output
        except subprocess.CalledProcessError as e:
            temp.seek(0)
            return e.returncode, temp.read()

print ping_ip('8.8.8.8')
print ping_ip('a')
```

Результат выполнения будет таким:

```
$ python subprocess_ping_function.py
(0, '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=46.106 ms\n64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=46.114 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=47.390 ms\n--- 8.8.8.8 ping statistics ---\n3 packets transmitted, 3 packets received, 0.0% packet loss\nround-trip min/avg/max/stddev = 46.106/46.537/47.390/0.603 ms\n')
(68, 'ping: cannot resolve a: Unknown host\n')
```

В примере использованы идеи из [ответа на stackoverflow](#)

Модуль `tempfile` входит в стандартную библиотеку Python и используется тут для того, чтобы сохранить сообщение об ошибке. Функция `TemporaryFile` создает временный файл и удаляет его автоматически, после того, как файл закрывается.

Подробнее о модуле `tempfile` можно почитать на сайте [PyMOTW](#).

На основе этой функции, можно сделать функцию, которая будет проверять список IP-адресов и возвращать, в результате выполнения, два списка: доступные и недоступные адреса.

Если количество IP-адресов, которые нужно проверить, большое, можно использовать модуль `multiprocessing`, чтобы ускорить проверку.

Это вынесено в задания к разделу.

Это далеко не все возможности модуля `subprocess`. Подробнее о нём можно почитать в [документации](#) или в статье [PyMOTW](#)

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')  
-----  
OSSError Traceback (most recent call last)  
<ipython-input-4-cbf3b897c095> in <module>()  
----> 1 os.mkdir('test')  
  
OSSError: [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/16_additional_info'  
  
In [14]: os.path.split(file)  
Out[14]: ('Programming/PyNEng/book/16_additional_info', 'README.md')
```

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
from tempfile import TemporaryFile
import argparse

def ping_ip(ip_address, count=3):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    with TemporaryFile() as temp:
        try:
            output = subprocess.check_output(['ping', '-c', str(count), '-n', ip_address],
                                            stderr=temp)
            return 0, output
        except subprocess.CalledProcessError as e:
            temp.seek(0)
            return e.returncode, temp.read()

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/Cellar/python/2.7.11/Frameworks/Python.framework/Versions/2.7/lib/python2.7/subprocess.py", line 566, in check_output
    process = Popen(stdout=PIPE, *popenargs, **kwargs)
  File "/usr/local/Cellar/python/2.7.11/Frameworks/Python.framework/Versions/2.7/lib/python2.7/subprocess.py", line 710, in __init__
    errread, errwrite)
  File "/usr/local/Cellar/python/2.7.11/Frameworks/Python.framework/Versions/2.7/lib/python2.7/subprocess.py", line 1335, in _execute_child
    raise child_exception
TypeError: execv() arg 2 must contain only strings
```

Если бы функция была вызвана без аргументов, когда не используется argparse, возникла 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: argument -a is required
```

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 опцию, 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=3):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    with TemporaryFile() as temp:
        try:
            output = subprocess.check_output(['ping', '-c', str(count), '-n', ip_address],
                                             stderr=temp)
            return 0, output
        except subprocess.CalledProcessError as e:
            temp.seek(0)
            return e.returncode, temp.read()

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 %s with DB schema %s" % (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%s" % ', '.join( args.filename )
        print "\nAdding data to db %s" % args.db_file

def get(args):
    if args.key and args.value:
        print "Getting data from DB: %s" % args.db_file
        print "Request data for host(s) with %s %s" % (args.key, args.value)
    elif args.key or args.value:
        print "Please give two or zero args\n"
    else:
        print "Showing %s content..." % 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()
    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 %s with DB schema %s" % (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`, и 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 возможно 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

И, 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 `ipaddress`

Модуль `ipaddress` может пригодится для работы с IP-адресами.

Он не входит в стандартную библиотеку Python, поэтому его нужно установить:

```
$ pip install ipaddress
```

Для работы с модулем `ipaddress`, нужно, чтобы строки, в которых описывается IP-адрес, были в формате unicode.

Для этого, надо либо конвертировать строки в unicode (если адрес задается вручную):

```
ipaddress.ip_address(u'1.2.3.4')
```

или так:

```
ipaddress.ip_address(unicode('1.2.3.4'))
```

либо импортировать `unicode_literals` таким образом:

```
from __future__ import unicode_literals
```

[Подробнее](#) о нюансах использования `unicode_literals` в Python2

В последнем случае, все строки будут unicode.

Такое особенности связаны с тем, что модуль `ipaddress` создавался для Python3.

А в нём все строки unicode.

`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(u'10.0.1.1')  
  
In [3]: ipv4  
Out[3]: IPv4Address(u'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(u'10.0.1.1')

In [11]: ip2 = ipaddress.ip_address(u'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(u'10.0.1.6')

In [19]: ip1 - 5
Out[19]: IPv4Address(u'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(u'80.0.1.0/28')
```

Как и у адреса, у сети есть различные атрибуты и методы:

```
In [21]: subnet1.broadcast_address
Out[21]: IPv4Address(u'80.0.1.15')

In [22]: subnet1.with_netmask
Out[22]: u'80.0.1.0/255.255.255.240'

In [23]: subnet1.with_hostmask
Out[23]: u'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(u'80.0.1.1'),
 IPv4Address(u'80.0.1.2'),
 IPv4Address(u'80.0.1.3'),
 IPv4Address(u'80.0.1.4'),
 IPv4Address(u'80.0.1.5'),
 IPv4Address(u'80.0.1.6'),
 IPv4Address(u'80.0.1.7'),
 IPv4Address(u'80.0.1.8'),
 IPv4Address(u'80.0.1.9'),
 IPv4Address(u'80.0.1.10'),
 IPv4Address(u'80.0.1.11'),
 IPv4Address(u'80.0.1.12'),
 IPv4Address(u'80.0.1.13'),
 IPv4Address(u'80.0.1.14')]
```

Метод `subnets` позволяет разбивать на подсети. По умолчанию, он разбивает на две подсети:

```
In [27]: list(subnet1.subnets())
Out[27]: [IPv4Network(u'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(u'80.0.1.0/30'),
 IPv4Network(u'80.0.1.4/30'),
 IPv4Network(u'80.0.1.8/30'),
 IPv4Network(u'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(u'80.0.1.0/30'),
 IPv4Network(u'80.0.1.4/30'),
 IPv4Network(u'80.0.1.8/30'),
 IPv4Network(u'80.0.1.12/30')]

In [30]: list(subnet1.subnets(new_prefix=29))
Out[30]: [IPv4Network(u'80.0.1.0/29'), IPv4Network(u'80.0.1.8/29')]
```

По IP-адресам в сети можно проходить 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(u'80.0.1.0')

In [33]: subnet1[5]
Out[33]: IPv4Address(u'80.0.1.5')
```

Таким образом можно проверять находится ли IP-адрес в сети:

```
In [34]: ip1 = ipaddress.ip_address(u'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(u'10.0.1.1/24')
```

Используя методы объекта `IPv4Interface`, можно получать адрес, маску или сеть интерфейса:

```
In [37]: int1.ip
Out[37]: IPv4Address(u'10.0.1.1')

In [38]: int1.network
Out[38]: IPv4Network(u'10.0.1.0/24')

In [39]: int1.netmask
Out[39]: IPv4Address(u'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 = u'10.0.1.1/24'

In [41]: IP2 = u'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
```

Полезные функции

В этом разделе описаны такие функции:

- `format`
- `sorted`
- `lambda`
- `zip`
- `map`
- `filter`
- `all, any`

Функция sorted

Функция `sorted()` - возвращает новый отсортированный список, который получен из последовательности, которая была передана как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.

Первый аспект на который важно обратить внимание - `sorted` возвращает список.

Соответственно, 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']
```

Сортировка кортежа:

```
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']
```

Кроме встроенных функций, можно также использовать свои функции. Также тут удобно использовать анонимную функцию lambda.

Также, 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)]
```

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 zip

Функция zip():

- на вход функции передаются последовательности
- zip() возвращает список кортежей, 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

Пример использования zip:

```
In [1]: a = [1, 2, 3]
In [2]: b = [100, 200, 300]
In [3]: 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]: zip(a,b,c)
Out[7]: [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
```

Использование zip для создания словаря:

```
In [8]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
In [8]: d_values = ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255
.0.1']

In [9]: zip(d_keys,d_values)
Out[9]:
[('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('vendor', 'Cisco'),
 ('model', '4451'),
 ('IOS', '15.4'),
 ('IP', '10.255.0.1')]

In [10]: dict(zip(d_keys,d_values))
Out[10]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
In [11]: r1 = dict(zip(d_keys,d_values))
```

Функция map

Функция map применяет функцию к каждому элементу последовательности и возвращает список результатов.

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]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

Конвертация в числа:

```
In [3]: list_of_str = ['1', '2', '5', '10']

In [4]: 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]: 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]: map(lambda x, y: x*y, nums, nums2)
Out[9]: [100, 400, 900, 1600, 2500]
```

Функция filter

Функция filter() применяет функцию ко всем элементам последовательности, и возвращает те объекты, 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]: ['100', '1', '50']
```

Например, из списка чисел оставить только нечетные:

```
In [3]: filter(lambda x: x%2, [10, 111, 102, 213, 314, 515])
Out[3]: [111, 213, 515]
```

Аналогично, только четные:

```
In [4]: 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]: filter(lambda x: len(x) > 2, list_of_words)
Out[6]: ['one', 'two', 'list', 'dict']
```

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 (или объект пустой).

```
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
```


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 дальше 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 с 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 равен имени модуля, когда импортируется
- `__file__` - эта переменная равна имени скрипта, который был запущен напрямую. И равна полному 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  
/Users/natasha/Desktop/current/pyneng_things/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](#) - хотя тут примеры для Python3, большинство методов работают аналогично. В этой книге отлично расписаны эти методы.

Дополнительные ресурсы

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 курс. Но, скорее всего, в каком-то виде, надо будет читать о той же информации несколько раз.

В этом разделе собраны ссылки на различные ресурсы по Python, связанные с сетями и нет. Эти ресурсы позволяют вам повторить информацию, почитать (посмотреть, послушать) то же самое, но другими словами. Кроме того, даны несколько ресурсов для продолжения обучения.

Подкасты об автоматизации с Python

Хорошо подходят для вдохновления темой автоматизации

- [Packet Pushers Show 176 – Intro To Python & Automation For Network Engineers](#)
- [Packet Pushers Show 198 – Kirk Byers On Network Automation With Python & Ansible](#)
- [Packet Pushers Show 270: Design & Build 9: Automation With Python And Netmiko](#)
- [PQ Show 99: Netmiko & NAPALM For Network Automation](#)
- [Network automation tools with Jason Edelman on Software Gone Wild](#)
- [Packet Pushers Show 332: Don't Believe The Programming Hype](#) - отличный подкаст об автоматизации, программировании, скрипtingе и как всё это относится к сетевым инженерам.
- [Packet Pushers Show 333: Automation & Orchestration In Networking](#)

Блоги (python + network)

- [Kirk Byers](#)
- [Jason Edelman](#)
- [Matt Oswalt](#)
- [Michael Kashin](#)
- [Henry Ölsner](#)
- [Mat Wood](#)

Платные и бесплатные курсы (python + network)

Я эти курсы не проходила, поэтому они собраны в произвольном порядке, просто чтобы было проще оценить, что есть на сегодняшний день и выбрать то, что больше нравится.

Бесплатные курсы:

- Бесплатный email курс от Kirk Byers
- Network Programmability and Network Automation using GNS3 and Python
- Курсы на Cisco DevNet

Платные курсы

- Python for Network Engineers (курс от Kirk Byers)
- Курсы от Network to Code
- Python Programming for Network Engineers (INE)
- Network Automation with Ansible (INE)
- Python Programming For Network Engineers (Udemy)
- Python Network Programming - Part 1: Build 7 Python Apps (GNS3 Academy)
- Network Automation using Python for Engineers (ehacking academy)
- Network automation webinars (Ivan Pepelnjak)

Материалы по темам курса

Тут собраны ссылки по темам курса. Большая часть из них пересекаются с темами курса, но есть и дополнения. В любом случае, даже те, которые пересекаются полезно почитать.

Ссылки на статьи и другие ресурсы

Основы Python

Книги по Python:

- [A Byte of Python](#)
 - [эта же книга на русском](#)
- [Python 101](#)
- [Learn Python the Hard Way](#)

Курсы по основам Python:

- [Программирование на Python](#)

Регулярные выражения

Статьи:

- [Parse Cisco IOS configurations using RegEx](#)
- [Using Python to generate Cisco configs](#)

Сайт для проверки регулярных выражений:

- [regex101](#)

Базы данных

Более подробное описание возможностей SQLite:

- [SQLite tutorial](#)

Telnet, SSH

Статьи:

- [Netmiko Library](#)
- [Automate SSH connections with netmiko](#)
- [Network Automation Using Python: BGP Configuration](#)

Jinja2

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](#)

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](#)

Ansbile

У Ansible очень хорошая документация:

- <http://docs.ansible.com/ansible/>

Отличные видео от Ansible:

- [AUTOMATING YOUR NETWORK](#)
 - [Репозиторий с примерами из вебинара](#)

Ansible без привязки к сетевому оборудованию

Очень хорошая серия видео, с транскриптом и хорошими ссылками:

- <https://sysadmincasts.com/episodes/43-19-minutes-with-ansible-part-1-4>

Примеры использования Ansible:

- <https://github.com/ansible/ansible-examples>

Примеры Playbook с демонстрацией различных возможностей

- https://github.com/ansible/ansible-examples/tree/master/language_features

Ansible for network devices

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>

Статьи:

- <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/>

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/>

Ссылки на документацию

Документация модулей, которые использовались в курсе:

- [re](#)
- [YAML](#)
- [CSV](#)
- [JSON](#)
- [sqlite3](#)
- [Telnetlib](#)
- [Pexpect](#)
- [Paramiko](#)
- [Netmiko](#)
- [threading](#)
- [multiprocessing](#)
- [Jinja2](#)
- [TextFSM](#)
- [Ansible](#)

Продолжение обучения

Python без привязки к сетевому оборудованию

Если вам интересно развивать свои знания по Python в целом, ниже несколько полезных ссылок.

- [The Hitchhiker's Guide to Python!](#)
- [Best Python Resources](#) - подборка ссылок на хорошие ресурсы по Python
- [Automate the Boring Stuff with Python](#)
- [Problem Solving with Algorithms and Data Structures using Python](#) - отличная книга по структурам данных и алгоритмам. Много примеров и домашних заданий.
 - [на русском](#)

Онлайн курсы:

- [MITx - 6.00.1x Introduction to Computer Science and Programming Using Python](#)

Видео курсы:

- [Python от Computer Science Center](#)

Сайты с задачами:

- [CodeEval](#)
- [HackerRank](#)

Python + network

Лучше всего после курса попытаться применить полученные знания в работе. Этот вариант позволит лучше запомнить то, что вы уже выучили. Кроме того, таким образом знания расширить будет проще, так как у вас будет практическая задача, которую надо решить.

Если говорить о развитии дальше применимо к сетям и сетевому оборудованию, то тут вариантов много. Это, в том числе, и развитие знаний по Python в целом, но также и более специфические вещи, такие как доступ к сетевому оборудованию через API, и др. Ниже ссылки на некоторые проекты, которые вас могут заинтересовать.

Проекты:

- [Scapy](#)
- [CiscoConfParse](#)

- NAPALM
- NOC Project
- Requests
- NetworkX
- Twisted
- AutoNetKit

Отзывы читателей и слушателей курса

На xgu.ru находилась информация из первых одинадцати разделов. Сейчас с xgu.ru всё удалено и перенесено на GitBook.

Илья про версию курса на xgu.ru

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.

аргумент

Аргумент - это фактическое значение (данные), которое передается функции (или методу), при вызове.

атрибут

итератор

итерируемый объект

метод

Метод - это функция, которая относится к конкретному объекту. И соответственно вызывается применимо к объекту.

Например, print - это функция:

```
In [11]: print('test')
test
```

А append - это метод списка. Соответственно его можно вызывать только применимо к объекту который является списком:

```
In [12]: list1 = [1, 2, 3]
In [13]: list1.append(4)
```

объект

В Python все является объектом. Официальное определение - это сущность у которой есть какое-то состояние и определенное поведение.

Примеры объектов: список, строка, файл и так далее.

Например, таким образом можно создать объект файл:

```
In [1]: f = open('output.py')

In [2]: f
Out[2]: <_io.TextIOWrapper name='output.py' mode='r' encoding='UTF-8'>
```

У этого объекта есть такие методы и атрибуты:

```
In [3]: print([m for m in dir(f) if not m.startswith('_')])
['buffer', 'close', 'closed', 'detach', 'encoding', 'errors', 'fileno', 'flush', 'isatty', 'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline', 'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write', 'writelines']
```

Объект `f` в данном случае, представляет реальный файл `output.py`. И содержит методы и атрибуты, которые поддерживает Python по отношению к файлам.

параметр

Параметр - это переменная, которая используется, при создании функции.

последовательность

Последовательность (sequence) -

функция

Функция - блок кода, который возвращает какое-то значение. Функция также может принимать аргументы, которые влияют на выполнение кода в теле функции.

Пример функции:

```
In [14]: def f(a, b):
    ...
    return a+b
    ...:
```

У функции `f` два параметра - `a` и `b`. Она возвращает сумму этих параметров.

При вызове функции с аргументами 5 и 10, она возвращает результат 15, который присваивается в переменную `result`:

```
In [15]: result = f(5, 10)
```

```
In [16]: result
```

```
Out[16]: 15
```