Python для сетевых инженеров

Natasha Samoylenko

июл. 31, 2025

Оглавление

1 Введение **3**

О книге . . . 3

Для кого эта книга . . . 3

Зачем Вам учиться программировать? . . . 4

Требуемые версии ОС и Python . . . 4

Примеры . . . 4

Задания . . . 5

Вопросы . . . 5

Презентации . . . 6

Форматы файлов книги . . . 6

Обсуждение . . . 6

Часто задаваемые вопросы (FAQ) . . . 6

Будет ли печатная версия книги? . . . 6

Почему в книге нет темы X? . . . 6

Чем это отличается от обычного вводного курса по Python? . . . 7

Почему книга именно для сетевых инженеров? . . . 7

Почему именно Python? . . . 7

Книга будет когда-то платной? . . . 8

Благодарности . . . 8

2 I. Основы Python **9**

1. Подготовка к работе . . . 10

Подготовка рабочего окружения . . . 11

ОС и редактор . . . 13

Система управления пакетами pip . . . 14

Виртуальные окружения . . . 16

Интерпретатор Python . . . 17

Дополнительные материалы . . . 18

Задания . . . 19

2. Использование Git и GitHub . . . 20

Основы Git . . . 20
Отображение статуса репозитория в приглашении . . . 21
Работа с Git . . . 22
Дополнительные возможности . . . 26
Аутентификация на GitHub . . . 30
Работа со своим репозиторием заданий . . . 31
Работа с репозиторием заданий и примеров . . . 35
Дополнительные материалы . . . 37
Задания . . . 39
3. Начало работы с Python . . . 40
Синтаксис Python . . . 40
Интерпретатор Python. IPython . . . 42
Специальные команды ipython . . . 47
Переменные . . . 49
Задания . . . 52
4. Типы данных в Python . . . 53
Числа . . . 53
Строки (Strings) . . . 56
Список (List) . . . 69
Словарь (Dictionary) . . . 74
Кортеж (Tuple) . . . 83
Множество (Set) . . . 84
Булевы значения . . . 87
Преобразование типов . . . 88
Проверка типов . . . 90
Вызов методов цепочкой . . . 92
Основы сортировки данных . . . 94
Дополнительные материалы . . . 94
Задания . . . 96
5. Создание базовых скриптов . . . 100
Исполняемый файл . . . 100
Передача аргументов скрипту (argv) . . . 101
Ввод информации пользователем . . . 102
Задания . . . 104
6. Контроль хода программы . . . 113
if/elif/else . . . 113
for . . . 119
while . . . 125
break, continue, pass . . . 127
for/else, while/else . . . 131
Работа с исключениями try/except/else/finally . . . 133
Дополнительные материалы . . . 139
Задания . . . 141
7. Работа с файлами . . . 144
Открытие файлов . . . 145

Чтение файлов . . . 146
Запись файлов . . . 150
Закрытие файлов . . . 153
Конструкция with . . . 155
Примеры работы с файлами . . . 157
Дополнительные материалы . . . 163
Задания . . . 164
8. Полезные возможности и инструменты . . . 168
Форматирование строк с помощью f-строк . . . 168
Распаковка переменных . . . 173
List, dict, set comprehensions . . . 179
Отладка кода . . . 187
Дополнительные материалы . . . 199

3 II. Повторное использование кода 201
9. Функции . . . 202
Создание функций . . . 203
Пространства имен. Области видимости . . . 207
Параметры и аргументы функций . . . 210
Аргументы, которые можно передавать только как ключевые . . . 226
Распространенные проблемы/нюансы работы с функциями . . . 226
Дополнительные материалы . . . 231
Задания . . . 232
10. Полезные функции . . . 240
Функция print . . . 240
Функция range . . . 243
Функция sorted . . . 246
enumerate . . . 250
Функция zip . . . 252
Функция all . . . 255
Функция any . . . 255
Анонимная функция (лямбда-выражение) . . . 256
Функция map . . . 257
Функция filter . . . 259
11. Модули . . . 261
Импорт модуля . . . 261
Создание своих модулей . . . 264
`if __name__ == "__main__"` . . . 266
Пути поиска модулей . . . 268
Рекомендации по поводу расположения функций в коде . . . 269
Задания . . . 271
12. Полезные модули . . . 276
Модуль subprocess . . . 276
Модуль os . . . 281
Модуль ipaddress . . . 286

Модуль tabulate . . . 291
Модуль pprint . . . 295
Дополнительные материалы . . . 299
Задания . . . 300
13. Итераторы, итерируемые объекты и генераторы . . . 302
Итерируемый объект . . . 302
Итераторы . . . 303
Генератор (generator) . . . 305
Дополнительные материалы . . . 307

4 III. Регулярные выражения 309
14. Синтаксис регулярных выражений . . . 310
Синтаксис регулярных выражений . . . 310
Наборы символов . . . 312
Символы повторения . . . 313
Специальные символы . . . 318
Жадность символов повторения . . . 323
Группировка выражений . . . 324
Разбор вывода команды show ip dhcp snooping с помощью именованных групп 326
Группа без захвата . . . 329
Повторение захваченного результата . . . 330
Дополнительные материалы . . . 331
15. Модуль re . . . 332
Объект Match . . . 332
Функция search . . . 339
Функция match . . . 344
Функция finditer . . . 345
Функция findall . . . 350
Функция compile . . . 352
Флаги . . . 355
Функция re.split . . . 358
Функция re.sub . . . 360
Дополнительные материалы . . . 361
Задания . . . 363

5 IV. Запись и передача данных 369
16. Unicode . . . 371
Стандарт Юникод . . . 371
Юникод в Python 3 . . . 372
Конвертация между байтами и строками . . . 375
Примеры конвертации между байтами и строками . . . 376
Ошибки при конвертации . . . 380
Дополнительные материалы . . . 383
17. Работа с файлами в формате CSV, JSON, YAML . . . 384
Работа с файлами в формате CSV . . . 384

Работа с файлами в формате JSON . . . 390
Работа с файлами в формате YAML . . . 397
Дополнительные материалы . . . 402
Задания . . . 403

6 V. Работа с сетевым оборудованием 411
18. Подключение к оборудованию . . . 412
Ввод пароля . . . 413
Модуль pexpect . . . 414
Модуль telnetlib . . . 422
Модуль paramiko . . . 431
Модуль netmiko . . . 436
Модуль scrapli . . . 443
Дополнительные материалы . . . 456
Задания . . . 457
19. Одновременное подключение к нескольким устройствам . . . 465
Измерение времени выполнения скрипта . . . 465
Процессы и потоки в Python (CPython) . . . 466
Количество потоков . . . 468
Потоковая безопасность . . . 469
Модуль logging . . . 471
Модуль concurrent.futures . . . 473
Дополнительные материалы . . . 486
Задания . . . 488
20. Шаблоны конфигураций с Jinja2 . . . 495
Начало работы с Jinja2 . . . 495
Пример использования Jinja . . . 497
Синтаксис шаблонов Jinja2 . . . 499
Наследование шаблонов . . . 523
Дополнительные материалы . . . 528
Задания . . . 529
21. Обработка вывода команд TextFSM . . . 534
Начало работы с TextFSM . . . 534
Синтаксис шаблонов TextFSM . . . 536
Правила состояний . . . 539
Примеры использования TextFSM . . . 541
TextFSM CLI Table . . . 557
Дополнительные материалы . . . 562
Задания . . . 564

7 VI. Основы объектно-ориентированного программирования 569
22. Основы ООП . . . 570
Основы ООП . . . 570
Создание класса . . . 572
Создание метода . . . 573

Параметр self . . . 575
Метод __init__ . . . 577
Пример класса . . . 578
Область видимости . . . 579
Переменные класса . . . 580
Задания . . . 582
23. Специальные методы . . . 591
Подчеркивание в именах . . . 591
Методы __str__, __repr__ . . . 595
Поддержка арифметических операторов . . . 597
Протоколы . . . 600
Дополнительные материалы . . . 612
Задания . . . 613
24. Наследование . . . 618
Основы наследования . . . 618
Задания . . . 623

8 VII. Работа с базами данных 629
25. Работа с базами данных . . . 630
SQL . . . 630
SQLite . . . 631
Основы SQL (в sqlite3 CLI) . . . 633
Модуль sqlite3 . . . 648
Дополнительные материалы . . . 673
Задания . . . 674

9 VIII. Дополнительная информация 687
Модуль argparse . . . 687
Вложенные парсеры . . . 692
Форматирование строк с оператором % . . . 699
Соглашение об именах . . . 700
Имена переменных . . . 700
Имена модулей и пакетов . . . 701
Имена функций . . . 701
Имена классов . . . 701
Подчеркивание в именах . . . 701
Подчеркивание как имя . . . 702
Два подчеркивания . . . 704
Два подчеркивания перед именем . . . 704
Два подчеркивания перед и после имени . . . 705
Отличия Python 2.7 и Python 3.6 . . . 706
Unicode . . . 706
Функция print . . . 707
input вместо raw_input . . . 707
range вместо xrange . . . 708

Методы словарей . . . 708
Распаковка переменных . . . 709
Итератор вместо списка . . . 710
subprocess.run . . . 710
Jinja2 . . . 711
Модули pexpect, telnetlib, paramiko . . . 711
Мелочи . . . 711
Дополнительная информация . . . 711
Проверка заданий с помощью утилиты pyneng . . . 712
Где решать задания . . . 712
Установка скрипта pyneng . . . 712
Скрипт pyneng . . . 713
Проверка заданий тестами . . . 713
Получение ответов . . . 714
Вывод pyneng . . . 714
Проверка заданий с помощью pytest . . . 716
Основы pytest . . . 716
Особенности использования pytest для проверки заданий . . . 722
pytest-clarity . . . 726

10 Продолжение обучения **729**
Написание скриптов для автоматизации рабочих процессов . . . 729
Python для автоматизации работы с сетевым оборудованием . . . 730
Python без привязки к сетевому оборудованию . . . 731
Книги . . . 731
Курсы . . . 732
Сайты с задачами . . . 732
Подкасты . . . 732
Документация . . . 732

11 Скачать PDF/Epub **733**

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

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

В большинстве примеров в книге используется Python 3.7. При этом Python 3.7 это минимальная версия для работы с книгой, для версий >= 3.7 практически все что рассматривается в книге (изменился вывод некоторых сообщений), будет работать аналогично, для версий Python < 3.7 возможны нюансы.

1 Введение

О книге

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

Для кого эта книга

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

Ещё не решили, нужно ли читать книгу? Почитайте отзывы.

Зачем Вам учиться программировать?

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

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

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

Требуемые версии ОС и Python

Все примеры и выводы терминала в книге показываются на Debian Linux. В книге используется Python 3.7, но для большинства примеров подойдет и Python 3.x. Только в некоторых примерах требуется версия 3.6 или выше чем 3.5. Это всегда явно указано и, как правило, касается дополнительных возможностей.

Примеры

Все примеры, которые используются в книге, располагаются в репозитории. Примеры, которые рассматриваются в разделах книги, являются обучающими. Это значит, что они не обязательно показывают лучший вариант решения задачи, так как они основаны только на той информации, которая рассматривалась в предыдущих главах книги. Кроме того, довольно часто примеры, которые давались в разделах, развиваются в заданиях. То есть, в заданиях вам нужно будет сделать лучшую, более универсальную, и, в целом, более правильную версию кода. Если есть возможность, лучше набирать код, который используется в книге, самостоятельно, или, как минимум, скачать примеры и попробовать что-то в них изменить – так информация будет лучше запоминаться. Если такой возможности нет, например, когда вы читаете книгу в дороге, лучше повторить примеры самостоятельно позже. В любом случае, обязательно нужно делать задания вручную.

Задания

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

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

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

Вопросы

Для части тем книги созданы вопросы:

- Типы данных. Часть 1
- Типы данных. Часть 2
- Контроль хода программы. Часть 1
- Контроль хода программы. Часть 2
- Функции и модули. Часть 1
- Функции и модули. Часть 2
- Регулярные выражения. Часть 1
- Регулярные выражения. Часть 2
- Базы данных

Эти вопросы можно использовать как для проверки знаний, так и в роли заданий. Очень полезно поотвечать на вопросы после прочтения соответствующей темы. Они позволят вам вспомнить материал темы, а также увидеть на практике разные аспекты работы с Python. Постарайтесь сначала ответить самостоятельно, а затем подсмотреть ответы в IPython по тем вопросам, в которых вы сомневаетесь.

Презентации

Для всех тем книги есть презентации в репозитории. По ним удобно быстро просматривать информацию и повторять. Если вы знаете основы Python, то стоит их пролистать.

Форматы файлов книги

Книгу можно скачать в двух форматах: PDF, Epub. Они автоматически обновляются, поэтому всегда содержат одинаковую информацию.

Обсуждение

Для обсуждения книги, заданий, а также связанных вопросов используется Slack. Все вопросы, предложения и замечания по книге также пишите в Slack.

Часто задаваемые вопросы (FAQ)

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

Будет ли печатная версия книги?

Нет. Книга существует в каком-то виде с 2015 года. Все это время книга менялась. Мне нравится эта возможность менять книгу, писать что-то по-другому.

Почему в книге нет темы X?

Полезных тем еще огромное количество и просто невозможно все их вместить в одну книгу. Конечно, у каждого читателя есть приоритеты и кажется именно этот модуль очень нужен всем, но таких тем/модулей очень много. Глобально в книге уже ничего не будет менятся, новые темы добавляться не будут.

Чем это отличается от обычного вводного курса по Python?

Основных отличий три:

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

Почему книга именно для сетевых инженеров?

Есть несколько причин:

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

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

Почему именно Python?

Причины следующие:

- в контексте работы с сетевым оборудованием, сейчас часто используется именно Python;
- на некотором оборудовании Python встроен или есть API, который поддерживает Python;
- Python достаточно прост для изучения (конечно, это относительно, и более простым может казаться другой язык, но, скорее, это будет из-за имеющегося опыта работы с языком, а не потому, что Python сложный);
- с Python вы вряд ли быстро дойдете до границ возможностей языка;
- Python может использоваться не только для написания скриптов, но и для разработки приложений. Разумеется, это не является задачей этой книги, но, по крайней мере, вы потратите время на язык, который позволит вам легко шагнуть дальше, чем написание простых скриптов;
- из программ, связанных с сетями, на Python написан, например, GNS3.

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

Книга будет когда-то платной?

Эта книга всегда будет бесплатной. Я читаю платно онлайн курс «Python для сетевых инженеров», но это не будет влиять на эту книгу - она всегда будет бесплатной.

Благодарности

Спасибо всем, кто проявил интерес к первому анонсу курса – ваш интерес подтвердил, что это будет кому-то нужно.

Павел Пасынок, спасибо 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, спасибо тебе. Я всегда могла обсудить с тобой любой вопрос по курсу. Спасибо за то, что помогал мне поддерживать мотивацию и не уходить в дебри. Спасибо за вдохновение, положительные эмоции и поддержку!

Спасибо всем, кто писал комментарии к книге – благодаря вам в книге не только стало меньше опечаток и ошибок, но и содержание книги стало лучше.

Спасибо всем слушателям онлайн-курса – благодаря вашим вопросам книга стала намного лучше.

Слава Скороход, спасибо тебе огромное, что вызвался быть редактором – количество ошибок уменьшилось.

2

I. Основы Python

Первая часть книги посвящена основам Python. В ней рассматриваются:

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

1. Подготовка к работе

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

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

В книге используется Debian Linux (в других ОС вывод может незначительно отличаться) и Python 3.7.

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

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

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

Подготовка рабочего окружения

Для выполнения заданий книги можно использовать несколько вариантов:

- взять подготовленную виртуалку vmware или vagrant (virtualbox)
- подготовить виртуалку самостоятельно
- использовать вм или какой-то сервис в облаке
- работать без создания виртуальной машины

Подготовленные VM

Подготовлены виртуальные машины, в которых установлены:

- Debian 9.9
- Python 3.7 и 3.8 в виртуальном окружении
- IPython и другие модули, которые потребуются для выполнения заданий
- текстовые редакторы vim, Geany, Mu
- GNS3 для работы с сетевым оборудованием

Есть два варианта подготовленных виртуальных машин (по ссылкам инструкции для каждого варианта, в которых есть ссылки на образ и инструкция как работать с GNS3):

- vagrant
- vmware

Подготовка виртуальной машины/хоста самостоятельно

- Инструкция для подготовки Linux
- Нюансы подготовки и выполнения заданий на Windows

Список модулей, которые нужно установить:

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

Также надо установить graphviz принятым способом в ОС (пример для debian):

```
apt-get install graphviz
```

Облачные сервисы

Ещё один вариант – использовать один из следующих сервисов:

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

Сетевое оборудование

К 18 разделу книги, нужно подготовить виртуальное или реальное сетевое оборудование.

Все примеры и задания, в которых встречается сетевое оборудование, используют одинаковое количество устройств: три маршрутизатора с такими базовыми настройками:

- пользователь: cisco
- пароль: cisco
- пароль на режим enable: cisco
- SSH версии 2 (обязательно именно версия 2), Telnet
- IP-адреса маршрутизаторов: 192.168.100.1, 192.168.100.2, 192.168.100.3
- IP-адреса должны быть доступны из виртуалки на которой вы выполняете задания и могут быть назначены на физических/логических/loopback интерфейсах

Топология может быть произвольной. Пример топологии:

Базовый конфиг:

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

(continues on next page)

(продолжение с предыдущей страницы)

```
 login local
 transport input telnet ssh
```

На каком-то интерфейсе надо настроить IP-адрес

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

Алиасы (по желанию)

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

При желании можно настроить EEM applet для отображения команд, которые вводит пользователь:

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

ОС и редактор

Можно выбрать любую ОС и любой редактор, но желательно использовать Python версии 3.8 - 3.10, есть некоторые мелочи, которые требуют хотя бы Python 3.8.

Все примеры в книге выполнялись на Debian, на других ОС вывод может незначительно отличаться. Для выполнения заданий из книги можно использовать Linux, macOS или Windows.

Для работы с Python можно выбрать любой текстовый редактор или IDE, который поддерживает Python. Как правило, для работы с Python требуется минимум настройки редактора и часто редактор по умолчанию распознает Python.

Редактор Thonny

Thonny хороший редактор для начинающих:

- поддерживает Python 3.10 и может установить сразу и себя и Python 3.10
- удобно сделана работа с разными версиями Python и виртуальными окружениями, очень явно можно выбирать версию и это не прячется в глубине настроек
- несколько вариантов отладчика
- отладчик nicer просто незаменим для начинающих изучать Python, показывает пошагово как вычисляется каждое выражение в Python
- отладчик faster работает в целом как стандартный отладчик
- есть все стандартные плюшки с подсказками, подсветками и так далее (часть возможно надо будет включить в настройках)
- удобно подсвечивает незакрытые кавычки/скобки
- поддерживает Windows, Mac, Linux
- удобный интерфейс и есть возможность добавлять/удалять секции интерфейса по желанию

Для знакомства с Thonny можно посмотреть видео. Там рассматриваются основы и отладка кода в Thonny.

IDE PyCharm

PyCharm — интегрированная среда разработки для Python. Для начинающих изучать язык может оказаться сложным вариантом из-за обилия настроек, но это зависит от личных предпочтений. В PyCharm поддерживается огромное количество возможностей, даже в бесплатной версии.

PyCharm прекрасный IDE, но, на мой взгляд, он сложноват для начинающих. Я бы не советовала использовать его, если вы с ним не знакомы и только начинаете учить Python. Вы всегда сможете перейти на него после книги, но пока что лучше попробовать что-то другое.

Система управления пакетами pip

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

Проверка версии pip:

```
$ pip --version
pip 19.1.1 from /home/vagrant/venv/pyneng-py3-7/lib/python3.7/site-packages/pip (python 3.
↪7)
```

Если команда выдала ошибку, значит, pip не установлен. Установка pip описана в документации

Установка модулей

Для установки модулей используется команда `pip install`:

```
$ pip install tabulate
```

Удаление пакета выполняется таким образом:

```
$ pip uninstall tabulate
```

Кроме того, иногда необходимо обновить пакет:

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

pip или pip3

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

Вариант, когда pip соответствует Python 2.7:

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

Вариант, когда pip3 соответствует Python 3.7:

```
$ pip3 --version
pip 19.1.1 from /home/vagrant/venv/pyneng-py3-7/lib/python3.7/site-packages/pip (python 3.
↪7)
```

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

Также можно использовать альтернативный вариант вызова pip:

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

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

Виртуальные окружения

Виртуальные окружения:

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

Встроенный модуль venv

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

```
$ python3.11 -m venv ~/venv/pyneng
```

Вместо python3.11 может использоваться python или python3, в зависимости от того, как установлен Python 3.11. Эта команда создаёт указанный каталог и все необходимые каталоги внутри него, если они не были созданы.

Команда создаёт следующую структуру каталогов:

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

Для перехода в виртуальное окружение надо выполнить команду:

```
$ source ~/venv/pyneng/bin/activate
```

Для выхода из виртуального окружения используется команда deactivate:

```
$ deactivate
```

Подробнее о модуле venv в документации.

Установка пакетов

Например, установим в виртуальном окружении пакет simplejson.

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

Если перейти в интерпретатор Python и импортировать simplejson, то он доступен и никаких ошибок нет:

```
(pyneng)$ python
>>> import simplejson
>>> simplejson
<module 'simplejson' from '/home/vagrant/venv/pyneng/lib/python3.11/site-packages/
↪simplejson/__init__.py'>
>>>
```

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

```
(pyneng)$ deactivate

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

Интерпретатор Python

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

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

Вывод показывает, что установлен Python 3.7. Приглашение >>>, это стандартное приглашение интерпретатора Python. Вызов интерпретатора выполняется командой python, а чтобы выйти, нужно набрать `quit()`, либо нажать Ctrl+D.

Примечание: В книге, вместо стандартного интерпретатора Python, будет использоваться ipython

Дополнительные материалы

Документация:

- Python Setup and Usage
- pip
- venv
- virtualenvwrapper

Редакторы и IDE:

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

Thonny:

- Сайт проекта Thonny
- Документация Thonny
- Thonny debug

Задания

Задание 1.1

Единственное задание в этом разделе: подготовка к работе.

Для этого нужно:

- определиться с ОС, которую вы будете использовать
  - так как все примеры в курсе ориентированы на Linux (Debian), желательно использовать его
  - желательно использовать новую виртуальную машину, чтобы было спокойней экспериментировать
- установить Python 3.7, 3.8 или 3.9 (3.7, если будете использовать редактор Mu). Проверить, что Python и pip установлены
- создать виртуальное окружение, в котором вы будете работать весь курс
- определиться с редактором/IDE
- начиная с раздела 18, в заданиях надо будет подключаться к оборудованию. Поэтому нужно подготовить виртуальное или реальное оборудование

2. Использование Git и GitHub

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

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

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

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

GitHub позволяет централизованно хранить все перечисленные выше вещи, но следует учитывать, что эти ресурсы будут доступны и другим. У GitHub есть и приватные репозитории, но даже в них не стоит выкладывать такую информацию, как пароли.

Основы Git

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

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

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

Установка Git:

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

Первичная настройка Git

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

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

Посмотреть настройки Git можно таким образом:

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

Инициализация репозитория

Создание и переход в каталог first_repo

```
mkdir first_repo
cd first_repo
```

Инициализация репозитория выполняется с помощью команды git init:

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

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

Отображение статуса репозитория в приглашении

Примечание: Пропускаем эту часть на Windows.

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

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

А затем добавить в конец файла .bashrc такие строки:

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

Для того, чтобы изменения применились, перезапустить bash:

```
exec bash
```

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

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

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

Если же перейти в репозиторий Git:

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

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

Работа с Git

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

git status

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

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

Initial commit

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

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

Создание файла README и добавление в него строки «test»

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

После этого приглашение выглядит таким образом

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

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

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

Initial commit

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

        .README.un~
        README

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

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

Файл .gitignore

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

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

```
*.un~
```

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

После этого, git status показывает

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

Initial commit

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

        .gitignore
        README

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

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

git add

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

Можно указать что надо следить за конкретным файлом

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

Или за всеми файлами

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

Вывод git status

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

Initial commit

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

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

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

git commit

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

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

После этого git status отображает

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

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

Дополнительные возможности

git diff

Команда git diff позволяет посмотреть разницу между различными состояниями. Например, на данный момент, в репозитории внесены изменения в файл README и .gitignore.

Команда git status показывает, что оба файла изменены

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

        modified:   .gitignore
        modified:   README

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

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

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

Если добавить изменения, внесённые в файлы, в staging командой `git add` и ещё раз выполнить команду `git diff`, то она ничего не покажет

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

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

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

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

Закоммитим изменения

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

git log

Команда git log показывает, когда были выполнены последние изменения

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

    Update .gitignore and README

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

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

По умолчанию команда показывает все коммиты, начиная с ближайшего по времени. С помощью дополнительных параметров можно не только посмотреть информацию о коммитах, но и то, какие именно изменения были внесены.

Флаг `-p` позволяет отобразить отличия, которые были внесены каждым коммитом

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

    Update .gitignore and README

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

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

    First commit. Add .gitignore and README files

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

Более короткий вариант вывода можно вывести с флагом `--stat`

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

    Update .gitignore and README

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

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

    First commit. Add .gitignore and README files

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

Аутентификация на GitHub

Для того, чтобы начать работать с GitHub, надо на нём зарегистрироваться. Для безопасной работы с GitHub лучше использовать аутентификацию по ключам SSH.

Генерация нового SSH-ключа (используйте e-mail, который привязан к GitHub):

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

На всех вопросах достаточно нажать Enter (более безопасно использовать ключ с passphrase, но можно и без, если нажать Enter при вопросе, тогда passphrase не будет запрашиваться у вас постоянно при операциях с репозиторием).

SSH-агент используется для хранения ключей в памяти и удобен тем, что нет необходимости вводить пароль passphrase каждый раз при взаимодействии с удаленным хостом (в данном случае - github.com).

Запуск SSH-агента (пропускаем на Windows):

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

Добавить ключ в SSH-агент (пропускаем на Windows):

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

Добавление SSH-ключа на GitHub

Для добавления ключа надо его скопировать.

Например, таким образом можно отобразить ключ для копирования:

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

После копирования надо перейти на GitHub. Находясь на любой странице GitHub, в правом верхнем углу нажмите на картинку вашего профиля и в выпадающем списке выберите «Settings». В списке слева надо выбрать поле «SSH and GPG keys». После этого надо нажать «New SSH key» и в поле «Title» написать название ключа (например «Home»), а в поле «Key» вставить содержимое, которое было скопировано из файла ~/.ssh/id_rsa.pub.

Примечание: Если GitHub запросит пароль, введите пароль своего аккаунта на GitHub.

Чтобы проверить, всё ли прошло успешно, попробуйте выполнить команду `ssh -T git@github.com`.

Вывод должен быть таким:

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

Теперь вы готовы работать с Git и GitHub.

Работа со своим репозиторием заданий

В данной главе описывается, как создать свой репозиторий с копией файлов заданий.

Предупреждение: Если вы учитесь на курсе «Python для сетевых инженеров», эту часть НЕ надо выполнять. На курсе инструктор создает отдельный приватный репозиторий каждому слушателю. Все инструкции для курса по работе с git/github находятся на сайте курса.

Создание репозитория на GitHub

Для создания своего репозитория на основе шаблона нужно:

- залогиниться на GitHub
- открыть репозиторий с заданиями
- нажать «Use this template» и создать новый репозиторий на основе этого шаблона
- в открывшемся окне надо ввести название репозитория
- после этого готов новый репозиторий с копией всех файлов из исходного репозитория с заданиями

Клонирование репозитория с GitHub

Для локальной работы с репозиторием его нужно клонировать.

Для этого используется команда git clone:

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

По сравнению с приведённой в этом листинге командой, вам нужно изменить:

- имя пользователя natenka на имя своего пользователя на GitHub;
- имя репозитория my_pyneng_tasks на имя своего репозитория на GitHub.

В итоге, в текущем каталоге, в котором была выполнена команда git clone, появится каталог с именем репозитория, в моём случае – «my_pyneng_tasks». В этом каталоге теперь находится содержимое репозитория на GitHub.

Работа с репозиторием

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

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

Теперь готов полноценный локальный репозиторий Git, в котором вы можете работать. Обычно последовательность работы будет такой:

- перед началом работы, синхронизация локального содержимого с GitHub командой git pull
- изменение файлов репозитория
- добавление изменённых файлов в staging командой git add
- фиксация изменений через коммит командой git commit
- передача локальных изменений в репозитории на GitHub командой git push

При работе с заданиями на работе и дома, надо обратить особое внимание на первый и последний шаг:

- первый шаг – обновление локального репозитория
- последний шаг – загрузка изменений на GitHub

Синхронизация локального репозитория с удалённым

Все команды выполняются внутри каталога репозитория (в примере выше - my_pyneng_tasks).

Если содержимое локального репозитория одинаково с удалённым, вывод будет таким:

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

Если были изменения, вывод будет примерно таким:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 5 (delta 4), reused 5 (delta 4), pack-reused 0
Unpacking objects: 100% (5/5), done.
From ssh://github.com/natenka/my_pyneng_tasks
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   89c04b6..fc4c721  master     -> origin/master
Updating 89c04b6..fc4c721
Fast-forward
 exercises/03_data_structures/task_3_3.py | 2 ++
 1 file changed, 2 insertions(+)
```

Добавление новых файлов или изменений в существующих

Если необходимо добавить конкретный файл (в данном случае – README.md), нужно дать команду `git add README.md`. Добавление всех файлов текущей директории производится командой `git add .`.

Коммит

При выполнении коммита обязательно надо указать сообщение. Лучше, если сообщение будет со смыслом, а не просто «update» или подобное. Коммит делается командой, подобной `git commit -m "Сделаны задания 4.1-4.3"`.

Push на GitHub

Для загрузки всех локальных изменений на GitHub используется команда git push:

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

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

Работа с репозиторием заданий и примеров

Все примеры и задания книги выложены в отдельном репозитории.

Копирование репозитория с GitHub

Примеры и задания иногда обновляются, поэтому будет удобнее клонировать этот репозиторий на свою машину и периодически обновлять его.

Для копирования репозитория с GitHub выполните команду git clone:

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

Обновление локальной копии репозитория

При необходимости обновить локальный репозиторий, чтобы синхронизировать его с версией на GitHub, надо выполнить git pull, находясь внутри созданного каталога pyneng-examples-exercises.

Если обновлений не было, вывод будет таким:

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

Если обновления были, вывод будет примерно таким:

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

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

Просмотр изменений

Если вы хотите посмотреть, какие именно изменения были внесены, можно воспользоваться командой git log:

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

    Update task_24_4.md

diff --git a/exercises/24_ansible_for_network/task_24_4.md b/exercises/24_ansible_for_
↪network/task_24_4.md
index c4307fa..137a221 100644
--- a/exercises/24_ansible_for_network/task_24_4.md
+++ b/exercises/24_ansible_for_network/task_24_4.md
@@ -13,11 +13,12 @@
 * применить ACL к интерфейсу

 ACL должен быть таким:
+
 ip access-list extended INET-to-LAN
  permit tcp 10.0.1.0 0.0.0.255 any eq www
  permit tcp 10.0.1.0 0.0.0.255 any eq 22
  permit icmp any any
-
+

 Проверьте работу playbook на маршрутизаторе R1.
```

В этой команде флаг `-p` указывает, что надо отобразить вывод утилиты Linux diff для внесённых изменений, а не только сообщение коммита. В свою очередь, `-1` указывает, что надо показать только один самый свежий коммит.

Просмотр изменений, которые будут синхронизированы

Прошлый вариант git log опирается на количество коммитов, но это не всегда удобно. До выполнения команды git pull можно посмотреть, какие изменения были выполнены с момента последней синхронизации.

Для этого используется следующая команда:

```
$ git log -p ..origin/master
commit 4c1821030d20b3682b67caf362fd777d098d9126
Author: Наташа Самойленко <nataliya.samoylenko@gmail.com>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Date:   Mon May 29 07:53:45 2017 +0300

Update README.md

diff --git a/tools/README.md b/tools/README.md
index 2b6f380..4f8d4af 100644
--- a/tools/README.md
+++ b/tools/README.md
@@ -1 +1,4 @@
+
+Тут находятся PDF версии руководств по настройке инструментов, которые используются на␣
↪курсе.
```

В данном случае изменения были только в одном файле. Эта команда будет очень полезна для того, чтобы посмотреть, какие изменения были внесены в формулировку заданий и каких именно заданий. Так будет легче ориентироваться, и понимать, касается ли это заданий, которые вы уже сделали, и если касается, то надо ли их изменять.

Примечание: «..origin/master» в команде `git log -p ..origin/master` означает показать все коммиты, которые есть в origin/master (в данном случае, это GitHub), но которых нет в локальной копии репозитория

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

Дополнительные материалы

Документация:

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

Про Git/GitHub:

- GitHowTo - интерактивный howto на русском;
- git/github guide. a minimal tutorial - минимально необходимые знания для работы с Git и GitHub;
- Pro Git book. Эта же книга на русском;
- Системы контроля версий (GIT) (курс на Hexlet).

- CRLF vs. LF: Normalizing Line Endings in Git

Задания

Задание 2.1

В этом задании необходимо:

- создать свой репозиторий для выполнения заданий на GitHub
- клонировать его на свою виртуалку/хост

Создать свой репозиторий на основе шаблона репозиторий с заданиями и примерами.

Примечание: Как создать репозиторий на основе шаблона.

---

3. 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 в Python, это то, что отступы имеют значение:

- они определяют, какой код попадает в блок;
- когда блок кода начинается и заканчивается.

Пример кода Python:

```
a = 10
b = 5

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

print("Конец")

def open_file(filename):
    print("Чтение файла", filename)
    with open(filename) as f:
        return f.read()
        print("Готово")
```

Примечание: Этот код показан для демонстрации синтаксиса. И, несмотря на то, что ещё не рассматривалась конструкция if/else, скорее всего, суть кода будет понятной.

Python понимает, какие строки относятся к if на основе отступов. Выполнение блока `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` и после `else`).

Несколько правил и рекомендаций по отступам:

- В качестве отступов могут использоваться табы или пробелы (лучше использовать пробелы, а точнее, настроить редактор так, чтобы таб был равен 4 пробелам – тогда при использовании клавиши табуляции будут ставиться 4 пробела, вместо 1 знака табуляции).
- Количество пробелов должно быть одинаковым в одном блоке (лучше, чтобы количество пробелов было одинаковым во всём коде – популярный вариант, это использовать 2-4 пробела, так, например, в этой книге используются 4 пробела).

Ещё одна особенность приведённого кода, это пустые строки. С их помощью код форматируется, чтобы его было проще читать. Остальные особенности синтаксиса будут показаны в процессе знакомства со структурами данных в Python.

Примечание: В Python есть специальный документ, в котором описано как лучше писать код Python PEP 8 - the Style Guide for Python Code.

Комментарии

При написании кода часто нужно оставить 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 решётки. Обратите внимание, что комментарий может быть как в строке, где находится сам код, так и в отдельной строке.

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

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

Для многострочного комментария можно использовать три двойные или три одинарные кавычки. Комментарии могут использоваться как для того, чтобы комментировать, что происходит в коде, так и для того, чтобы исключить выполнение определённой строки или блока кода (то есть закомментировать их).

Интерпретатор Python. IPython

Интерпретатор позволяет получать моментальный отклик на выполненные действия. Можно сказать, что интерпретатор работает как CLI (Command Line Interface) сетевых устройств: каждая команда будет выполняться сразу же после нажатия Enter. Однако есть исключение – более сложные объекты (например циклы или функции) выполняются только после двухкратного нажатия Enter.

В предыдущем разделе, для проверки установки Python вызывался стандартный интерпретатор. Кроме него, есть и усовершенствованный интерпретатор 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;
- можно передвигаться по истории выполнения команд, или же посмотреть её «волшебной» командой %history.

Установить IPython можно с помощью pip (установка будет производиться в виртуальном окружении, если оно настроено):

```
pip install ipython
```

После этого, перейти в IPython можно следующим образом:

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

In [1]:
```

Для выхода используется команда quit. Далее описывается, как будет использоваться IPython.

Для знакомства с интерпретатором можно попробовать использовать его 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 помечены:

- In – входные данные пользователя
- Out – результат, который возвращает команда (если он есть)
- числа после In или Out – это порядковые номера выполненных команд в текущей сессии IPython

Пример вывода строки функцией print():

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

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

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

help()

В IPython есть возможность посмотреть справку по произвольному объекту, функции или методу с помощью help():

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

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

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

Второй вариант:

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

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

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

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

print()

Функция `print()` позволяет вывести информацию на стандартный поток вывода (текущий экран терминала). Если необходимо вывести строку, то её 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)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
25
```

Если нужно вывести подряд несколько значений через пробел, то нужно 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 каждого выражения, переданного в print(), будет перевод строки. Если необходимо, чтобы после вывода каждого выражения не было бы перевода строки, надо в качестве последнего выражения в print() указать дополнительный аргумент end.

См.также:

Дополнительные параметры функции print *Функция print*

dir()

Функция `dir()` может использоваться для того, чтобы посмотреть, какие имеются атрибуты (переменные, привязанные к объекту) и методы (функции, привязанные к объекту).

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']
```

Аналогично для строки:

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

(continues on next page)

(продолжение с предыдущей страницы)

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

Если выполнить dir() без передачи значения, то она показывает существующие методы, атрибуты и переменные, определённые в текущей сессии интерпретатора:

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

Например, после создания переменной a и test():

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

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

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

Специальные команды ipython

В IPython есть специальные команды, которые упрощают работу с интерпретатором. Все они начинаются со знака процента.

%history

Например, команда `%history` позволяет просмотреть историю введённых пользователем команд в текущей сессии IPython:

```
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
```

С помощью %history можно скопировать нужный блок кода.

%time

Команда `%time` показывает сколько секунд выполнялось выражение:

```
In [5]: import subprocess

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

(continues on next page)

(продолжение с предыдущей страницы)

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

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

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

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

Подробнее об IPython можно почитать в документации 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:
```

(continues on next page)

(продолжение с предыдущей страницы)

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

History:

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

Переменные

Переменные в Python не требуют объявления типа переменной (так как Python – язык с динамической типизацией) и являются ссылками на область памяти. Правила именования переменных:

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

Пример создания переменных в Python:

```
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 можно продемонстрировать с помощью 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 одна особенность, которая может немного сбить с понимания: числа от -5 до 256 заранее созданы и хранятся в массиве (списке). Поэтому при создании числа из этого диапазона фактически создаётся ссылка на число в созданном массиве.

Примечание: Эта особенность характерна именно для реализации CPython, которая рассматривается в книге

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 a, b и число 3 имеют одинаковые идентификаторы. Все они являются ссылками на существующее число в списке.

Если сделать то же самое с числом больше 256, идентификаторы у всех будут разные:

```
In [14]: a = 500

In [15]: b = 500
```

(continues on next page)

(продолжение с предыдущей страницы)

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

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

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

При этом, если сделать присваивание переменных друг другу, то идентификаторы будут у всех одинаковые (в таком варианте a, b и c ссылаются на один и тот же объект):

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

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

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

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

Имена переменных

Имена переменных не должны пересекаться с названиями операторов и модулей или же других зарезервированных слов. В Python есть рекомендации по именованию функций, классов и переменных:

- имена переменных обычно пишутся или полностью большими или полностью маленькими буквами (например DB_NAME, db_name);
- имена функций задаются маленькими буквами, с подчёркиваниями между словами (например, get_names);
- имена классов задаются словами с заглавными буквами без пробелов, это так называемый CamelCase (например, CiscoSwitch).

Задания

Задание 3.1

Выполните установку IPython в виртуальном окружении или глобально в системе, если виртуальные окружения не используются. После установки, по команде ipython должен открываться интерпретатор IPython (вывод может незначительно отличаться):

```
$ ipython
Python 3.8.0 (default, Nov  9 2019, 12:40:50)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.18.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

4. Типы данных в Python

В Python есть несколько стандартных типов данных:

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

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

- изменяемые (списки, словари и множества)
- неизменяемые (числа, строки и кортежи)
- упорядоченные (списки, кортежи, строки и словари)
- неупорядоченные (множества)

Содержание раздела:

Числа

С числами можно выполнять различные математические операции.

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

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

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

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

Деление int и float:

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

(continues on next page)

(продолжение с предыдущей страницы)

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

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

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

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

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

Конвертация в int типа float:

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

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

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

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

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

Аналогично, функция hex() позволяет получить шестнадцатеричное значение:

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

И, конечно же, можно делать несколько преобразований одновременно:

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

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

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

```
In [28]: import  math

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

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

Строки (Strings)

Строка в Python это:

- последовательность символов, заключенная в кавычки
- 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 profile␣
↪DMVPN\n'

In [13]: print(tunnel)

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

Строки можно суммировать. Тогда они объединяются в одну строку:

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

(continues on next page)

(продолжение с предыдущей страницы)

```
In [15]: tun = 'Tunnel0'

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

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

Строку 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 единицы).

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

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

Кроме обращения к конкретному символу, можно делать срезы строк, 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'
```

Также в срезе можно указывать шаг. Так можно получить нечетные числа:

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

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

А таким образом можно получить все четные числа строки a:

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

Срезы также можно использовать для получения строки в обратном порядке:

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

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

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

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

Функция `len` позволяет получить количество символов в строке:

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

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

Примечание: Функция и метод отличаются тем, что метод привязан к объекту конкретного типа, а функция, как правило, более универсальная и может применяться к объектам разного типа. Например, функция `len` может применяться к строкам, спискам, словарям и так далее,

а метод startswith относится только к строкам.

Полезные методы для работы со строками

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

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

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

Метод join

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

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

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

Методы upper, lower, swapcase, capitalize

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

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

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

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

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

In [29]: string2 = 'tunnel 0'

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

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

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

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

Метод count

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

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

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

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

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

Метод find

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

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

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

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

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

Методы startswith, endswith

Проверка на то, начинается или заканчивается ли строка на определенные символы (методы `startswith()`, `endswith()`):

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

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

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

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

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

Методам `startswith()` и `endswith()` можно передавать несколько значений (обязательно как кортеж):

```
In [1]: "test".startswith(("r", "t"))
Out[1]: True

In [2]: "test".startswith(("r", "a"))
Out[2]: False

In [3]: "rtest".startswith(("r", "a"))
Out[3]: True

In [4]: "rtest".endswith(("r", "a"))
Out[4]: False

In [5]: "rtest".endswith(("r", "t"))
Out[5]: True
```

Метод replace

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

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

In [46]: string1.replace('Fast', 'Gigabit')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[46]: 'GigabitEthernet0/1'
```

Метод strip

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

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

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

In [48]: print(string1)

    interface FastEthernet0/1


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

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

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

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

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

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

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

Метод split

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

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

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

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

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

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

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

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

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

Пример разделения адреса на октеты:

```
In [10]: ip = "192.168.100.1"

In [11]: ip.split(".")
Out[11]: ['192', '168', '100', '1']
```

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

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

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

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

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

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

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

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

Форматирование строк

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

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

Форматирование строк может помочь, например, в таких ситуациях:

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

Существует несколько вариантов форматирования строк:

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

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

Предупреждение: Так как для полноценного объяснения f-строк, надо показывать примеры с циклами и работой с объектами, которые еще не рассматривались, это тема рассматривается в разделе *Форматирование строк с помощью f-строк* с дополнительными примерами и пояснениями.

Форматирование строк с методом format

Пример использования метода format:

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

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

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

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

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

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

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

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

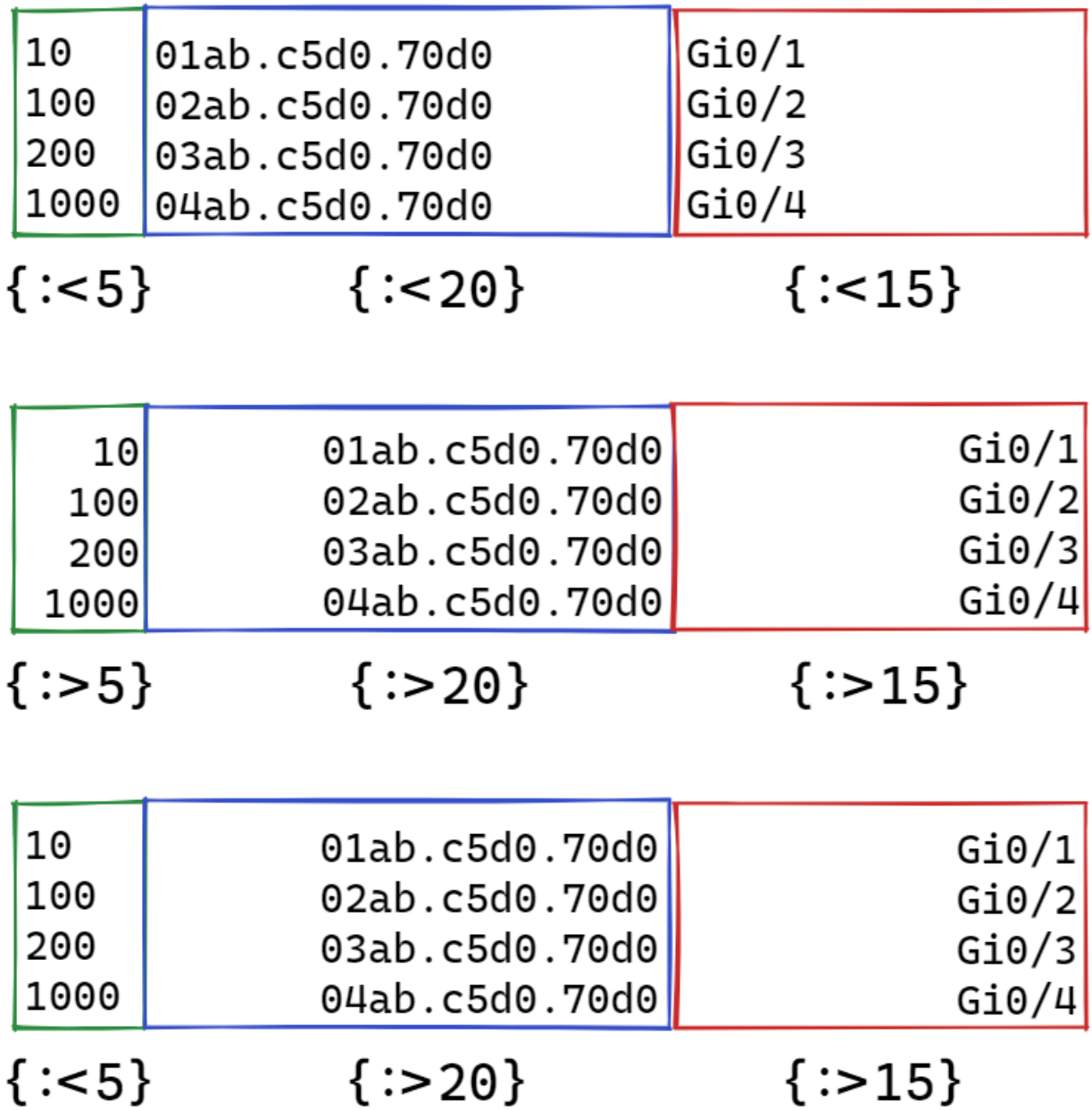
```
In [3]: vlan, mac, intf = ['100', 'aabb.cc80.7000', 'Gi0/1']

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

Выравнивание по левой стороне:

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

Примеры выравнивания



Шаблон для вывода может быть и многострочным:

```
ip_template = '''
IP address:
{}
'''

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

IP address:
10.1.1.1
```

С помощью форматирования строк можно также влиять на отображение чисел.

Например, можно указать, сколько цифр после запятой выводить:

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

С помощью форматирования строк можно конвертировать числа в двоичный формат:

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

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

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

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

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

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

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

Еще одна полезная возможность форматирования строк - указание номера аргумента:

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

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

```
ip_template = '''
IP address:
{:<8} {:<8} {:<8} {:<8}
{:08b} {:08b} {:08b} {:08b}
'''

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

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

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

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

```
ip_template = '''
IP address:
{0:<8} {1:<8} {2:<8} {3:<8}
{0:08b} {1:08b} {2:08b} {3:08b}
'''

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

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

Объединение литералов строк

В Python есть очень удобная функциональность - объединение литералов строк. Она дает возможность разбивать строки на части при написании кода и даже переносить эти части на разные строки кода. Это нужно как для разделения длинного текста на части из-за рекомендаций по максимальной длине строки в Python, так и для удобства восприятия.

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

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

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

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

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

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

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

Этим очень удобно пользоваться в регулярных выражениях:

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

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

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

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

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

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

Список (List)

Список в Python это:

- последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки
- изменяемый упорядоченный тип данных

Примеры списков:

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

Создание списка с помощью литерала:

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

Примечание: 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']
```

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'],
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ....: ['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'
```

Функция len возвращает количество элементов в списке:

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

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

А функция sorted сортирует элементы списка по возрастанию и возвращает новый список с отсортированными элементами:

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

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

Полезные методы для работы со списками

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

append

Метод append добавляет в конец списка указанный элемент:

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

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

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

Метод append меняет список на месте и ничего не возвращает. Если в скрипте надо добавить элемент в список, а потом вывести список print, надо делать это на разных строках кода.

extend

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

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

Метод extend:

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

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

In [23]: vlans.extend(vlans2)

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

Суммирование списков:

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

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

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

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

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

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

pop

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

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

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

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

Без указания номера удаляется последний элемент списка.

remove

Метод remove удаляет указанный элемент.

remove() не возвращает удаленный элемент:

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

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

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

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

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

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

index

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

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

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

insert

Метод insert позволяет вставить элемент на определенное место в списке:

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

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

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

sort

Метод sort сортирует список на месте:

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

In [41]: vlans.sort()

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

Словарь (Dictionary)

Словари - это изменяемый упорядоченный тип данных:

- данные в словаре - это пары ключ: значение
- доступ к значениям осуществляется по ключу, а не по номеру, как в списках
- данные в словаре упорядочены по порядку добавления элементов
- 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

Примечание: В других языках программирования тип данных подобный словарю может называться ассоциативный массив, хеш или хеш-таблица.

Пример словаря:

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

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',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        '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'
```

Функция sorted сортирует ключи словаря по возрастанию и возвращает новый список с отсортированными ключами:

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

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

Полезные методы для работы со словарями

clear

Метод clear позволяет очистить словарь:

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

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
```

(continues on next page)

(продолжение с предыдущей страницы)

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

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

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

get

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

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

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

KeyError: 'ios'
```

Метод get запрашивает ключ, и если его нет, вместо ошибки возвращает None.

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

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

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

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

setdefault

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

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

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

In [23]: print(ios)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
None

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

Если ключ есть, setdefault возвращает значение, которое ему соответствует:

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

Второй аргумент позволяет указать, какое значение должно соответствовать ключу:

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

In [27]: print(model)
Cisco3580

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

Метод setdefault заменяет такую конструкцию:

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

keys, values, items

Методы keys, values, items:

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

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

In [26]: london.values()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[26]: dict_values(['London1', 'London Str', 'Cisco'])

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

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

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

На примере метода keys:

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

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

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

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

Если добавить в словарь еще одну пару ключ-значение, объект keys тоже поменяется:

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

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

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

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

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

del

Удалить ключ и значение:

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

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

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

update

Метод update позволяет добавлять в словарь содержимое другого словаря:

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

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

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

Аналогичным образом можно обновить значения:

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

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

Варианты создания словаря

Литерал

Словарь можно создать с помощью литерала:

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

dict

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

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

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

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

Второй вариант создания словаря с помощью dict:

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

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

dict.fromkeys

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Примечание: Для такой задачи лучше подходит генератор словаря. Смотри раздел *List, dict, set comprehensions*

Кортеж (Tuple)

Кортеж в Python это:

- последовательность элементов, которые разделены между собой запятой и заключены в скобки
- неизменяемый упорядоченный тип данных

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
```

Функция sorted сортирует элементы кортежа по возрастанию и возвращает новый список с отсортированными элементами:

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

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

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]
```

(continues on next page)

(продолжение с предыдущей страницы)

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

In [3]: set1 = set(vlans)

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

Полезные методы для работы с множествами

add()

Метод add() добавляет элемент во множество:

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

In [2]: set1.add(50)

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

discard()

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

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

In [4]: set1.discard(55)

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

In [6]: set1.discard(50)

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

clear()

Метод clear() очищает множество:

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

In [9]: set1.clear()

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

Операции с множествами

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

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

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

Булевы значения

Булевы значения в Python это две константы `True` и `False`.

В Python истинными и ложными значениями считаются не только True и False.

- истинное значение:
  - любое ненулевое число
  - любая непустая строка
  - любой непустой объект
- ложное значение:
  - 0
  - None
  - пустая строка
  - пустой объект

Остальные истинные и ложные значения, как правило, логически следуют из условия.

Для проверки булевого значения объекта, можно воспользоваться `bool`:

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

In [3]: empty_list = []

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

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

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

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

Преобразование типов

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

int

`int` преобразует строку в int:

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

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

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

bin

Преобразовать десятичное число в двоичный формат можно с помощью bin:

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

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

hex

Аналогичная функция есть и для преобразования в шестнадцатеричный формат:

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

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

list

Функция list преобразует аргумент в список:

```
In [7]: list("string")
Out[7]: ['s', 't', 'r', 'i', 'n', 'g']

In [8]: list({1, 2, 3})
Out[8]: [1, 2, 3]

In [9]: list((1, 2, 3, 4))
Out[9]: [1, 2, 3, 4]
```

set

Функция set преобразует аргумент в множество:

```
In [10]: set([1, 2, 3, 3, 4, 4, 4, 4])
Out[10]: {1, 2, 3, 4}

In [11]: set((1, 2, 3, 3, 4, 4, 4, 4))
Out[11]: {1, 2, 3, 4}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [12]: set("string string")
Out[12]: {' ', 'g', 'i', 'n', 'r', 's', 't'}
```

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

tuple

Функция `tuple` преобразует аргумент в кортеж:

```
In [13]: tuple([1, 2, 3, 4])
Out[13]: (1, 2, 3, 4)

In [14]: tuple({1, 2, 3, 4})
Out[14]: (1, 2, 3, 4)

In [15]: tuple("string")
Out[15]: ('s', 't', 'r', 'i', 'n', 'g')
```

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

str

Функция `str` преобразует аргумент в строку:

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

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
```

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 или цифр:

```
In [11]: "a".isalnum()
Out[1]: True
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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") == 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)) == tuple
Out[16]: True

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

Вызов методов цепочкой

Часто с данными надо выполнить несколько операций, например:

```
In [1]: line = "switchport trunk allowed vlan 10,20,30"

In [2]: words = line.split()

In [3]: words
Out[3]: ['switchport', 'trunk', 'allowed', 'vlan', '10,20,30']

In [4]: vlans_str = words[-1]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [5]: vlans_str
Out[5]: '10,20,30'

In [6]: vlans = vlans_str.split(",")

In [7]: vlans
Out[7]: ['10', '20', '30']
```

Или в скрипте:

```
line = "switchport trunk allowed vlan 10,20,30"
words = line.split()
vlans_str = words[-1]
vlans = vlans_str.split(",")
print(vlans)
```

В этом случае переменные используются для хранения промежуточного результата и последующие методы/действия выполняются уже с переменной. Это совершенно нормальный вариант кода, особенно поначалу, когда тяжело воспринимать более сложные выражения.

Однако в Python часто встречаются выражения, в которых действия или методы применяются один за другим в одном выражении. Например, предыдущий код можно записать так:

```
line = "switchport trunk allowed vlan 10,20,30"
vlans = line.split()[-1].split(",")
print(vlans)
```

Так как тут нет выражений в скобках, которые бы указывали приоритет выполнения, все выполняется слева направо. Сначала выполняется `line.split()` - получаем список, затем к полученному списку применяется `[-1]` - получаем последний элемент списка, строку `10,20,30`. К этой строке применяется метод `split(",")` и в итоге получаем список `['10', '20', '30']`.

Главный нюанс при написании таких цепочек предыдущий метод/действие должен возвращать то, что ждет следующий метод/дествие. И обязательно чтобы что-то возвращалось, иначе будет ошибка.

Основы сортировки данных

При сортировке данных типа списка списков или списка кортежей, `sorted` сортирует по первому элементу вложенных списков (кортежей), а если первый элемент одинаковый, по второму:

```
In [1]: data = [[1, 100, 1000], [2, 2, 2], [1, 2, 3], [4, 100, 3]]

In [2]: sorted(data)
Out[2]: [[1, 2, 3], [1, 100, 1000], [2, 2, 2], [4, 100, 3]]
```

Если сортировка делается для списка чисел, которые записаны как строки, сортировка будет лексикографической, не натуральной и порядок будет таким:

```
In [7]: vlans = ['1', '30', '11', '3', '10', '20', '30', '100']

In [8]: sorted(vlans)
Out[8]: ['1', '10', '100', '11', '20', '3', '30', '30']
```

Чтобы сортировка была «правильной» надо преобразовать вланы в числа.

Эта же проблема проявляется, например, с IP-адресами:

```
In [2]: ip_list = ["10.1.1.1", "10.1.10.1", "10.1.2.1", "10.1.11.1"]

In [3]: sorted(ip_list)
Out[3]: ['10.1.1.1', '10.1.10.1', '10.1.11.1', '10.1.2.1']
```

Как решить проблему с сортировкой IP-адресов рассматривается в разделе 10. Полезные функции.

Дополнительные материалы

Документация:

- Strings. String Methods
- Lists basics. More on lists
- Tuples. More on tuples
- Sets basics. More on sets
- Dict basics. More on dicts
- Common Sequence Operations

Форматирование строк:

- Примеры использования форматирования строк

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

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Примечание: В разделе 4 тесты можно легко «обмануть» сделав нужный вывод, без получения результатов из исходных данных с помощью Python. Это не значит, что задание сделано правильно, просто на данном этапе сложно иначе проверять результат.

Задание 4.1

Используя подготовленную строку nat, получить новую строку, в которой в имени интерфейса вместо FastEthernet написано GigabitEthernet. Полученную новую строку вывести на стандартный поток вывода (stdout) с помощью print.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.2

Преобразовать строку в переменной mac из формата XXXX:XXXX:XXXX в формат XXXX.XXXX.XXXX Полученную новую строку вывести на стандартный поток вывода (stdout) с помощью print.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.3

Получить из строки config такой список VLANов:

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

Записать итоговый список в переменную result. (именно эта переменная будет проверяться в тесте)

Полученный список result вывести на стандартный поток вывода (stdout) с помощью print. Тут очень важный момент, что надо получить именно список (тип данных), а не, например, строку, которая похожа на показанный список.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.4

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

Из списка vlans нужно получить новый список уникальных номеров VLANов, отсортированный по возрастанию номеров. Для получения итогового списка нельзя удалять конкретные vlanы вручную.

Записать итоговый список уникальных номеров VLANов в переменную result. (именно эта переменная будет проверяться в тесте)

Полученный список result вывести на стандартный поток вывода (stdout) с помощью print.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.5

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

В данном случае, результатом должен быть такой список: `['1', '3', '8']`

Записать итоговый список в переменную result. (именно эта переменная будет проверяться в тесте)

Полученный список result вывести на стандартный поток вывода (stdout) с помощью print.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.6

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

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

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.7

Преобразовать MAC-адрес в строке mac в двоичную строку такого вида: „101010101010101010111011101110111100110011001100“

Полученную новую строку вывести на стандартный поток вывода (stdout) с помощью print.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 4.8

Преобразовать IP-адрес в переменной ip в двоичный формат и вывести на стандартный поток вывода 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 (в двоичном формате надо добавить два пробела между столбцами для разделения октетов между собой)

Пример вывода для адреса 10.1.1.1:

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

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
ip = "192.168.3.1"
```

5. Создание базовых скриптов

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

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

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

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

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

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

После этого надо сохранить файл и перейти в командную строку.

Так выглядит выполнение скрипта:

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

Ставить расширение `.py` у файла не обязательно, но, если вы используете Windows, то это желательно делать, так как Windows использует расширение файла для определения того, как обрабатывать файл.

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

Исполняемый файл

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

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

Пример файла access_template_exec.py:

```
#!/usr/bin/env python3

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

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

После этого:

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

Теперь можно вызывать файл таким образом:

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

Передача аргументов скрипту (argv)

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

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

Модуль sys позволяет работать с аргументами скрипта с помощью argv.

Пример access_template_argv.py:

```
from sys import argv

interface = argv[1]
vlan = argv[2]

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

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

Проверка работы скрипта:

```
$ python access_template_argv.py Gi0/7 4
interface Gi0/7
switchport mode access
switchport access vlan 4
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

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

Тут надо пояснить несколько моментов:

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

В данном случае в списке argv находятся такие элементы:

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

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

Ввод информации пользователем

Иногда необходимо получить информацию от пользователя, например, запросить пароль.

Для получения информации от пользователя используется функция `input`:

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

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

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

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

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

Запрос информации из скрипта (файл access_template_input.py):

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

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

print('\n' + '-' * 30)
print('interface {}'.format(interface))
print('\n'.join(access_template).format(vlan))
```

В первых двух строках запрашивается информация у пользователя.

Строка `print('\n' + '-' * 30)` используется для того, чтобы визуально отделить запрос информации от вывода.

Выполнение скрипта:

```
$ python access_template_input.py
Enter interface type and number: Gi0/3
Enter VLAN number: 55

------------------------------
interface Gi0/3
switchport mode access
switchport access vlan 55
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 5.1

В задании создан словарь, с информацией о разных устройствах.

Необходимо 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_5_1.py
Введите имя устройства: r1
{'location': '21 New Globe Walk', 'vendor': 'Cisco', 'model': '4451', 'ios': '15.4', 'ip
↪': '10.255.0.1'}
```

Ограничение: нельзя изменять словарь london_co.

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Задание 5.1a

Переделать скрипт из задания 5.1 таким образом, чтобы, кроме имени устройства, запрашивался также параметр устройства, который нужно отобразить.

Вывести информацию о соответствующем параметре, указанного устройства.

Пример выполнения скрипта:

```
$ python task_5_1a.py
Введите имя устройства: r1
Введите имя параметра: ios
15.4
```

Ограничение: нельзя изменять словарь london_co.

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Задание 5.1b

Переделать скрипт из задания 5.1a таким образом, чтобы, при запросе параметра, отображался список возможных параметров. Список параметров надо получить из словаря, а не прописывать вручную.

Вывести информацию о соответствующем параметре, указанного устройства.

Пример выполнения скрипта:

```
$ python task_5_1b.py
Введите имя устройства: r1
Введите имя параметра (location, vendor, model, ios, ip): ip
10.255.0.1

$ python task_5_1b.py
Введите имя устройства: sw1
Введите имя параметра (location, vendor, model, ios, ip, vlans, routing): ip
10.255.0.101
```

Ограничение: нельзя изменять словарь london_co.

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Задание 5.1c

Переделать скрипт из задания 5.1b таким образом, чтобы, при запросе параметра, которого нет в словаре устройства, отображалось сообщение „Такого параметра нет“. Задание относится только к параметрам устройств, не к самим устройствам.

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

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

Пример выполнения скрипта:

```
$ python task_5_1c.py
Введите имя устройства: r1
Введите имя параметра (ios, model, vendor, location, ip): ips
Такого параметра нет
```

Ограничение: нельзя изменять словарь london_co.

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

```
london_co = {
    "r1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Задание 5.1d

Переделать скрипт из задания 5.1c таким образом, чтобы, при запросе параметра, пользователь мог вводить название параметра в любом регистре.

Пример выполнения скрипта:

```
$ python task_5_1d.py
Введите имя устройства: r1
Введите имя параметра (ios, model, vendor, location, ip): IOS
15.4
```

Ограничение: нельзя изменять словарь london_co.

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.1"
    },
    "r2": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "4451",
        "ios": "15.4",
        "ip": "10.255.0.2"
    },
    "sw1": {
        "location": "21 New Globe Walk",
        "vendor": "Cisco",
        "model": "3850",
        "ios": "3.6.XE",
        "ip": "10.255.0.101",
        "vlans": "10,20,30",
        "routing": True
    }
}
```

Задание 5.2

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.

Подсказка: Получить маску в двоичном формате можно так:

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

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 5.2a

Всё, как в задании 5.2, но, если пользователь ввел адрес хоста, а не адрес сети, надо преобразовать адрес хоста в адрес сети и вывести адрес сети и маску, как в задании 5.2.

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
- 10.0.5.195/28 - хост из сети 10.0.5.192/28

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

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

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

Проверить работу скрипта на разных комбинациях хост/маска, например: 10.0.5.195/28, 10.0.1.1/24

Подсказка:

Есть адрес хоста в двоичном формате и маска сети 28. Адрес сети это первые 28 бит адреса хоста + 4 ноля. То есть, например, адрес хоста 10.1.1.195/28 в двоичном формате будет `bin_ip = "00001010000000010000000111000011"`.

А адрес сети будет первых 28 символов из bin_ip + 0000 (4 потому что всего в адресе может быть 32 бита, а 32 - 28 = 4): `00001010000000010000000111000000`

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 5.3

Скрипт должен запрашивать у пользователя:

- информацию о режиме интерфейса (access/trunk)
- номере интерфейса (тип и номер, вида Gi0/3)
- номер VLANа (для режима trunk будет вводиться список VLANов)

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).

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

Подсказка: Подводящим к этому заданию было задание 5.1. Чтобы было легче решить это задание, можно посмотреть на задание 5.1 и разобраться как там получилось вывести разную информацию в зависимости от ввода пользователя.

Ниже примеры выполнения скрипта, чтобы было проще понять задачу.

Пример выполнения скрипта, при выборе режима access:

```
$ python task_5_3.py
Введите режим работы интерфейса (access/trunk): access
Введите тип и номер интерфейса: Fa0/6
Введите номер влан(ов): 3

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

Пример выполнения скрипта, при выборе режима trunk:

```
$ python task_5_3.py
Введите режим работы интерфейса (access/trunk): trunk
Введите тип и номер интерфейса: Fa0/7
Введите номер влан(ов): 2,3,4,5

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
trunk_template = [
    "switchport trunk encapsulation dot1q", "switchport mode trunk",
    "switchport trunk allowed vlan {}"
]
```

Задание 5.3a

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

- для access: «Введите номер VLAN:»
- для trunk: «Введите разрешенные VLANы:»

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

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

trunk_template = [
    "switchport trunk encapsulation dot1q", "switchport mode trunk",
    "switchport trunk allowed vlan {}"
]
```

6. Контроль хода программы

До сих пор, весь код выполнялся последовательно - все строки скрипта выполнялись в том порядке, в котором они записаны в файле. В этом разделе рассматриваются возможности Python в управлении ходом программы:

- ответвления в ходе программы с помощью конструкции `if/elif/else`
- повторение действий в цикле с помощью конструкций `for` и `while`
- обработка ошибок с помощью конструкции `try/except`

if/elif/else

Конструкция `if/elif/else` позволяет делать ответвления в ходе программы. Программа уходит в ветку при выполнении определенного условия.

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

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

Пример конструкции:

```
In [1]: a = 9

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

Условия

Конструкция if построена на условиях: после if и elif всегда пишется условие. Блоки if/elif выполняются только когда условие возвращает True, поэтому первое с чем надо разобраться - это что является истинным, а что ложным в Python.

True и False

В Python, кроме очевидных значений True и False, всем остальным объектам также соответствует ложное или истинное значение:

- истинное значение:
  - любое ненулевое число
  - любая непустая строка
  - любой непустой объект
- ложное значение:
  - 0
  - None
  - пустая строка
  - пустой объект

Например, так как пустой список это ложное значение, 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 несколько иначе:

```
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
```

Примечание: Обратите внимание, что равенство проверяется двойным ==.

Пример использования операторов сравнения:

```
In [1]: a = 9

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

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',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ....:  '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 операндов.

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 операндов.

При оценке операндов возвращается первый истинный операнд:

```
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 ложными, возвращается последнее значение:

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

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

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

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

Пример использования конструкции if/elif/else

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

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

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

if len(password) < 8:
    print('Пароль слишком короткий')
elif username in password:
    print('Пароль содержит имя пользователя')
else:
    print('Пароль для пользователя {} установлен'.format(username))
```

Проверка скрипта:

```
$ python check_password.py
Введите имя пользователя: nata
Введите пароль: nata1234
Пароль содержит имя пользователя

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 123nata123
Пароль содержит имя пользователя

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 1234
Пароль слишком короткий

$ python check_password.py
Введите имя пользователя: nata
Введите пароль: 123456789
Пароль для пользователя nata установлен
```

Тернарное выражение (Ternary expression)

Иногда удобнее использовать тернарный оператор, нежели развернутую форму:

```
s = [1, 2, 3, 4]
result = True if len(s) > 5 else False
```

Этим лучше не злоупотреблять, но в простых выражениях такая запись может быть полезной.

for

Очень часто одно и то же действие надо выполнить для набора однотипных данных. Например, преобразовать все строки в списке в верхний регистр. Для выполнения таких действий в Python используется цикл `for`.

Цикл for перебирает по одному элементы указанной последовательности и выполняет действия, которые указаны в блоке for, для каждого элемента.

Примеры последовательностей элементов, по которым может проходиться цикл for:

- строка
- список
- словарь
- *Функция range*

- любой *Итерируемый объект*

Пример преобразования строк в списке в верхний регистр без цикла for:

```
In [1]: words = ['list', 'dict', 'tuple']

In [2]: upper_words = []

In [3]: words[0]
Out[3]: 'list'

In [4]: words[0].upper() # преобразование слова в верхний регистр
Out[4]: 'LIST'

In [5]: upper_words.append(words[0].upper()) # преобразование и добавление в новый список

In [6]: upper_words
Out[6]: ['LIST']

In [7]: upper_words.append(words[1].upper())

In [8]: upper_words.append(words[2].upper())

In [9]: upper_words
Out[9]: ['LIST', 'DICT', 'TUPLE']
```

У данного решения есть несколько нюансов:

- одно и то же действие надо повторять несколько раз
- код привязан к определенному количеству элементов в списке words

Те же действия с циклом for:

```
In [10]: words = ['list', 'dict', 'tuple']

In [11]: upper_words = []

In [12]: for word in words:
    ...:     upper_words.append(word.upper())
    ...:

In [13]: upper_words
Out[13]: ['LIST', 'DICT', 'TUPLE']
```

Выражение `for word in words: upper_words.append(word.upper())` означает «для каждого слова в списке words выполнить действия в блоке for». При этом word это имя переменной, которое каждую итерацию цикла ссылается на разные значения.

Примечание: Проект pythontutor может очень помочь в понимании циклов. Там есть специальная визуализация кода, которая позволяет увидеть, что происходит на каждом этапе выполнения кода, что особенно полезно на первых порах с циклами. На сайте pythontutor можно загружать свой код, но для примера, по этой ссылке можно посмотреть пример выше.

Цикл for может работать с любой последовательностью элементов. Например, выше использовался список и цикл перебирал элементы списка. Аналогичным образом for работает с кортежами.

При работе со строками, цикл for перебирает символы строки, например:

```
In [1]: for letter in 'Test string':
   ...:     print(letter)
   ...:
T
e
s
t

s
t
r
i
n
g
```

Примечание: В цикле используется переменная с именем **letter**. Хотя имя может быть любое, удобно, когда имя подсказывает, через какие объекты проходит цикл.

Иногда в цикле необходимо использовать последовательность чисел. В этом случае, лучше всего использовать функцию *Функция range*

Пример цикла for с функцией range():

```
In [2]: for i in range(10):
   ...:     print(f'interface FastEthernet0/{i}')
   ...:
interface FastEthernet0/0
interface FastEthernet0/1
interface FastEthernet0/2
interface FastEthernet0/3
interface FastEthernet0/4
interface FastEthernet0/5
interface FastEthernet0/6
```

(continues on next page)

(продолжение с предыдущей страницы)

```
interface FastEthernet0/7
interface FastEthernet0/8
interface FastEthernet0/9
```

В этом цикле используется range(10). Функция range генерирует числа в диапазоне от нуля до указанного числа (в данном примере - до 10), не включая его.

В этом примере цикл проходит по списку VLANов, поэтому переменную можно назвать vlan:

```
In [3]: vlans = [10, 20, 30, 40, 100]
In [4]: for vlan in vlans:
   ...:     print(f'vlan {vlan}')
   ...:     print(f' name VLAN_{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:

```
In [34]: r1 = {
    ...:      'ios': '15.4',
    ...:      'ip': '10.255.0.1',
    ...:      'hostname': 'london_r1',
    ...:      'location': '21 New Globe Walk',
    ...:      'model': '4451',
    ...:      'vendor': 'Cisco'}
    ...:

In [35]: for k in r1:
    ...:     print(k)
    ...:
ios
ip
hostname
location
model
vendor
```

Если необходимо выводить пары ключ-значение в цикле, можно делать так:

```
In [36]: for key in r1:
    ...:     print(key + ' => ' + r1[key])
    ...:
ios => 15.4
ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe Walk
model => 4451
vendor => Cisco
```

Или воспользоваться методом items, который позволяет проходиться в цикле сразу по паре ключ-значение:

```
In [37]: for key, value in r1.items():
    ...:     print(key + ' => ' + value)
    ...:
ios => 15.4
ip => 10.255.0.1
hostname => london_r1
location => 21 New Globe Walk
model => 4451
vendor => Cisco
```

Метод items возвращает специальный объект view, который отображает пары ключ-значение:

```
In [38]: r1.items()
Out[38]: dict_items([('ios', '15.4'), ('ip', '10.255.0.1'), ('hostname', 'london_r1'), (
↪'location', '21 New Globe Walk'), ('model', '4451'), ('vendor', 'Cisco')])
```

Вложенные for

Циклы for можно вкладывать друг в друга.

В этом примере в списке commands хранятся команды, которые надо выполнить для каждого из интерфейсов в списке fast_int:

```
In [7]: commands = ['switchport mode access', 'spanning-tree portfast', 'spanning-tree␣
↪bpduguard enable']
In [8]: fast_int = ['0/1','0/3','0/4','0/7','0/9','0/10','0/11']

In [9]: for intf in fast_int:
   ...:     print('interface FastEthernet {}'.format(intf))
   ...:     for command in commands:
   ...:         print(' {}'.format(command))
   ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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:

```
1  access_template = ['switchport mode access',
2                     'switchport access vlan',
3                     'spanning-tree portfast',
4                     'spanning-tree bpduguard enable']
5
6  access = {'0/12': 10, '0/14': 11, '0/16': 17, '0/17': 150}
7
8  for intf, vlan in access.items():
9      print('interface FastEthernet' + intf)
10     for command in access_template:
11         if command.endswith('access vlan'):
12             print(' {} {}'.format(command, vlan))
13         else:
14             print(' {}'.format(command))
```

Комментарии к коду:

- В первом цикле for перебираются ключи и значения во вложенном словаре access
- Текущий ключ, на данный момент цикла, хранится в переменной intf
- Текущее значение, на данный момент цикла, хранится в переменной vlan
- Выводится строка interface FastEthernet с добавлением к ней номера интерфейса

- Во втором цикле for перебираются команды из списка access_template
- Так как к команде switchport access vlan надо добавить номер VLAN:
  - внутри второго цикла for проверяются команды
  - если команда заканчивается на access vlan
    - выводится команда, и к ней добавляется номер VLAN
  - во всех остальных случаях просто выводится команда

Результат выполнения скрипта:

```
$ python generate_access_port_config.py
interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/17
 switchport mode access
 switchport access vlan 150
 spanning-tree portfast
 spanning-tree bpduguard enable
```

while

Цикл while - это еще одна разновидность цикла в Python.

В цикле while, как и в выражении if, надо писать условие. Если условие истинно, выполняются действия внутри блока while. При этом, в отличие от if, после выполнения кода в блоке, 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.

Примечание: 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 *Пример использования конструкции if/elif/else*. В том примере приходилось заново запускать скрипт, если пароль не соответствовал требованиям.

С помощью цикла while можно сделать так, что скрипт сам будет запрашивать пароль заново, если он не соответствует требованиям.

Файл check_password_with_while.py:

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

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

password_correct = False

while not password_correct:
    if len(password) < 8:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        print('Пароль слишком короткий\n')
        password = input('Введите пароль еще раз: ')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
        password = input('Введите пароль еще раз: ')
    else:
        print(f'Пароль для пользователя {username} установлен')
        password_correct = True
```

В этом случае цикл while полезен, так как он возвращает скрипт снова в начало проверок, позволяет снова набрать пароль, но при этом не требует перезапуска самого скрипта.

Теперь скрипт отрабатывает так:

```
$ python check_password_with_while.py
Введите имя пользователя: nata
Введите пароль: nata
Пароль слишком короткий

Введите пароль еще раз: natanata
Пароль содержит имя пользователя

Введите пароль еще раз: 123345345345
Пароль для пользователя nata установлен
```

break, continue, pass

В Python есть несколько операторов, которые позволяют менять поведение циклов по умолчанию.

Оператор break

Оператор break позволяет досрочно прервать цикл:

- break прерывает текущий цикл и продолжает выполнение следующих выражений
- если используется несколько вложенных циклов, break прерывает внутренний цикл и продолжает выполнять выражения, следующие за блоком * break может использоваться в циклах for и while

Пример с циклом for:

```
In [1]: for num in range(10):
   ...:     if num < 7:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:         print(num)
   ...:     else:
   ...:         break
   ...:
0
1
2
3
4
5
6
```

Пример с циклом while:

```
In [2]: i = 0
In [3]: while i < 10:
   ...:     if i == 5:
   ...:         break
   ...:     else:
   ...:         print(i)
   ...:         i += 1
   ...:
0
1
2
3
4
```

Использование break в примере с запросом пароля (файл check_password_with_while_break.py):

```
username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

while True:
    if len(password) < 8:
        print('Пароль слишком короткий\n')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
    else:
        print('Пароль для пользователя {} установлен'.format(username))
        # завершает цикл while
        break
    password = input('Введите пароль еще раз: ')
```

Теперь можно не повторять строку `password = input('Введите пароль еще раз: ')` в каждом ответвлении, достаточно перенести ее в конец цикла.

И, как только будет введен правильный пароль, break выведет программу из цикла while.

Оператор continue

Оператор continue возвращает управление в начало цикла. То есть, continue позволяет «перепрыгнуть» оставшиеся выражения в цикле и перейти к следующей итерации.

Пример с циклом for:

```
In [4]: for num in range(5):
   ...:     if num == 3:
   ...:         continue
   ...:     else:
   ...:         print(num)
   ...:
0
1
2
4
```

Пример с циклом while:

```
In [5]: i = 0
In [6]: while i < 6:
   ....:     i += 1
   ....:     if i == 3:
   ....:         print("Пропускаем 3")
   ....:         continue
   ....:         print("Это никто не увидит")
   ....:     else:
   ....:         print("Текущее значение: ", i)
   ....:
Текущее значение:  1
Текущее значение:  2
Пропускаем 3
Текущее значение:  4
Текущее значение:  5
Текущее значение:  6
```

Использование continue в примере с запросом пароля (файл check_password_with_while_continue.py):

```
username = input('Введите имя пользователя: ')
password = input('Введите пароль: ')

password_correct = False
```

(continues on next page)

(продолжение с предыдущей страницы)

```
while not password_correct:
    if len(password) < 8:
        print('Пароль слишком короткий\n')
    elif username in password:
        print('Пароль содержит имя пользователя\n')
    else:
        print('Пароль для пользователя {} установлен'.format(username))
        password_correct = True
        continue
    password = input('Введите пароль еще раз: ')
```

Тут выход из цикла выполнятся с помощью проверки флага password_correct. Когда был введен правильный пароль, флаг выставляется равным True, и с помощью continue выполняется переход в начало цикла, перескочив последнюю строку с запросом пароля.

Результат выполнения будет таким:

```
$ python check_password_with_while_continue.py
Введите имя пользователя: nata
Введите пароль: nata12
Пароль слишком короткий

Введите пароль еще раз: natalksdjflsdjf
Пароль содержит имя пользователя

Введите пароль еще раз: asdfsujljhdflaskjdfh
Пароль для пользователя nata установлен
```

Оператор pass

Оператор pass ничего не делает. Фактически, это такая заглушка для объектов.

Например, pass может помочь в ситуации, когда нужно прописать структуру скрипта. Его можно ставить в циклах, функциях, классах. И это не будет влиять на исполнение кода.

Пример использования pass:

```
In [6]: for num in range(5):
   ....:     if num < 3:
   ....:         pass
   ....:     else:
   ....:         print(num)
   ....:
3
4
```

for/else, while/else

В циклах for и while опционально может использоваться блок else.

for/else

В цикле for:

- блок else выполняется в том случае, если цикл завершил итерацию списка
- но else **не выполняется**, если в цикле был выполнен break

Пример цикла for с else (блок else выполняется после завершения цикла for):

```
In [1]: for num in range(5):
   ....:     print(num)
   ....: else:
   ....:     print("Числа закончились")
   ....:
0
1
2
3
4
Числа закончились
```

Пример цикла for с else и break в цикле (из-за break блок else не выполняется):

```
In [2]: for num in range(5):
   ....:     if num == 3:
   ....:         break
   ....:     else:
   ....:         print(num)
   ....: else:
   ....:     print("Числа закончились")
   ....:
0
1
2
```

Пример цикла for с else и continue в цикле (continue не влияет на блок else):

```
In [3]: for num in range(5):
   ....:     if num == 3:
   ....:         continue
   ....:     else:
   ....:         print(num)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ....: else:
   ....:     print("Числа закончились")
   ....:
0
1
2
4
Числа закончились
```

while/else

В цикле while:

- блок else выполняется в том случае, если условие в while ложно
- 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")
  ....:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
0
1
2
```

Работа с исключениями try/except/else/finally

try/except

Если вы повторяли примеры, которые использовались ранее, то наверняка были ситуации, когда выскакивала 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.

Тем не менее, даже если код синтаксически написан правильно, могут возникать ошибки. В Python эти ошибки называются **исключения (exceptions)**.

Примеры исключений:

```
In [1]: 2/0
---------------------------------------------------------
ZeroDivisionError: division by zero

In [2]: 'test' + 2
---------------------------------------------------------
TypeError: must be str, not int
```

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

Чаще всего можно предсказать, какого рода 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 из чисел строку, появится ошибка TypeError, как в примере выше.

Python позволяет работать с исключениями. Их можно перехватывать и выполнять определенные действия в том случае, если возникло исключение.

Примечание: 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 выполнения блока try не возникло никаких исключений, блок except пропускается, и выполняется дальнейший код
- если во время выполнения блока try в каком-то месте возникло исключение, оставшаяся часть блока try пропускается
  - если в блоке except указано исключение, которое возникло, выполняется код в блоке except
  - если исключение, которое возникло, не указано в блоке except, выполнение программы прерывается и выдается ошибка

Обратите внимание, что строка `Cool!` в блоке try не выводится:

```
In [4]: try:
   ...:     print("Let's divide some numbers")
   ...:     2/0
   ...:     print('Cool!')
   ...: except ZeroDivisionError:
   ...:     print("You can't divide by zero")
   ...:
Let's divide some numbers
You can't divide by zero
```

В конструкции try/except может быть много except, если нужны разные действия в зависимости от типа ошибки.

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

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

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    print("Результат: ", int(a)/int(b))
except ValueError:
    print("Пожалуйста, вводите только числа")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
except ZeroDivisionError:
    print("На ноль делить нельзя")
```

Примеры выполнения скрипта:

```
$ python divide.py
Введите первое число: 3
Введите второе число: 1
Результат:  3

$ python divide.py
Введите первое число: 5
Введите второе число: 0
На ноль делить нельзя

$ python divide.py
Введите первое число: qewr
Введите второе число: 3
Пожалуйста, вводите только числа
```

В данном случае исключение **ValueError** возникает, когда пользователь ввел строку вместо числа, во время перевода строки в число.

Исключение ZeroDivisionError возникает в случае, если второе число было равным 0.

Если нет необходимости выводить различные сообщения на ошибки ValueError и ZeroDivisionError, можно сделать так (файл divide_ver2.py):

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

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    print("Результат: ", int(a)/int(b))
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
```

Проверка:

```
$ python divide_ver2.py
Введите первое число: wer
Введите второе число: 4
Что-то пошло не так...

$ python divide_ver2.py
Введите первое число: 5
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Введите второе число: 0
Что-то пошло не так...
```

Примечание: 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 исключения.

Например, если необходимо выполнять в дальнейшем какие-то операции с данными, которые ввел пользователь, можно записать их в блоке else (файл divide_ver3.py):

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

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
else:
    print("Результат в квадрате: ", result**2)
```

Пример выполнения:

```
$ python divide_ver3.py
Введите первое число: 10
Введите второе число: 2
Результат в квадрате:  25

$ python divide_ver3.py
Введите первое число: werq
Введите второе число: 3
Что-то пошло не так...
```

try/except/finally

Блок finally - это еще один опциональный блок в конструкции try. Он выполняется **всегда**, независимо от того, было ли исключение или нет.

Сюда ставятся действия, которые надо выполнить в любом случае. Например, это может быть закрытие файла.

Файл divide_ver4.py с блоком finally:

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

try:
    a = input("Введите первое число: ")
    b = input("Введите второе число: ")
    result = int(a)/int(b)
except (ValueError, ZeroDivisionError):
    print("Что-то пошло не так...")
else:
    print("Результат в квадрате: ", result**2)
finally:
    print("Вот и сказочке конец, а кто слушал - молодец.")
```

Проверка:

```
$ python divide_ver4.py
Введите первое число: 10
Введите второе число: 2
Результат в квадрате:  25
Вот и сказочке конец, а кто слушал - молодец.

$ python divide_ver4.py
Введите первое число: qwerewr
Введите второе число: 3
Что-то пошло не так...
Вот и сказочке конец, а кто слушал - молодец.

$ python divide_ver4.py
Введите первое число: 4
Введите второе число: 0
Что-то пошло не так...
Вот и сказочке конец, а кто слушал - молодец.
```

Когда использовать исключения

Как правило, один и тот же код можно написать и с использованием исключений, и без них.

Например, этот вариант кода:

```
while True:
    a = input("Введите число: ")
    b = input("Введите второе число: ")
    try:
        result = int(a)/int(b)
    except ValueError:
        print("Поддерживаются только числа")
    except ZeroDivisionError:
        print("На ноль делить нельзя")
    else:
        print(result)
        break
```

Можно переписать таким образом без try/except (файл try_except_divide.py):

```
while True:
    a = input("Введите число: ")
    b = input("Введите второе число: ")
    if a.isdigit() and b.isdigit():
        if int(b) == 0:
            print("На ноль делить нельзя")
        else:
            print(int(a)/int(b))
            break
    else:
        print("Поддерживаются только числа")
```

Далеко не всегда аналогичный вариант без использования исключений будет простым и понятным.

Важно в каждой конкретной ситуации оценивать, какой вариант кода более понятный, компактный и универсальный - с исключениями или без.

Если вы раньше использовали какой-то другой язык программирования, есть вероятность, что в нём использование исключений считалось плохим тоном. В Python это не так. Чтобы немного больше разобраться с этим вопросом, посмотрите ссылки на дополнительные материалы в конце этого раздела.

raise

Иногда в коде надо сгенерировать исключение, это можно сделать так:

```
raise ValueError("При выполнении команды возникла ошибка")
```

Встроенные исключения

В Python есть много встроенных исключений, каждое из которых генерируется в определенной ситуации.

Например, TypeError обычно генерируется когда ожидался один тип данных, а передали другой

```
In [1]: "a" + 3
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-5aa8a24e3e06> in <module>
----> 1 "a" + 3
TypeError: can only concatenate str (not "int") to str
```

ValueError когда значение не соответствует ожидаемому:

```
In [2]: int("a")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-d9136db7b558> in <module>
----> 1 int("a")
ValueError: invalid literal for int() with base 10: 'a'
```

Дополнительные материалы

Документация:

- Compound statements (if, while, for, try)
- break, continue
- Errors and Exceptions
- Built-in Exceptions
- Operator precedence

Статьи:

- Write Cleaner Python: Use Exceptions

- Robust exception handling
- Python Exception Handling Techniques

Stack Overflow:

- Why does python use „else“ after for and while loops?
- Is it a good practice to use try-except-else in Python?

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 6.1

Список mac содержит MAC-адреса в формате XXXX:XXXX:XXXX. Однако, в оборудовании cisco MAC-адреса используются в формате XXXX.XXXX.XXXX.

Написать код, который преобразует MAC-адреса в формат cisco и добавляет их в новый список result. Полученный список result вывести на стандартный поток вывода (stdout) с помощью print.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 6.2

1. Запросить у пользователя ввод IP-адреса в формате 10.0.1.1
2. В зависимости от типа адреса (описаны ниже), вывести на стандартный поток вывода:
    - «unicast» - если первый байт в диапазоне 1-223
    - «multicast» - если первый байт в диапазоне 224-239
    - «local broadcast» - если IP-адрес равен 255.255.255.255
    - «unassigned» - если IP-адрес равен 0.0.0.0
    - «unused» - во всех остальных случаях

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.2a

Сделать копию скрипта задания 6.2.

Добавить проверку введенного IP-адреса. Адрес считается корректно заданным, если он:

- состоит из 4 чисел (а не букв или других символов)
- числа разделенны точкой
- каждое число в диапазоне от 0 до 255

Если адрес задан неправильно, выводить сообщение: «Неправильный IP-адрес». Сообщение «Неправильный IP-адрес» должно выводиться только один раз, даже если несколько пунктов выше не выполнены.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.2b

Сделать копию скрипта задания 6.2a.

Дополнить скрипт: Если адрес был введен неправильно, запросить адрес снова.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 6.3

В скрипте сделан генератор конфигурации для access-портов. Сделать аналогичный генератор конфигурации для портов trunk.

В транках ситуация усложняется тем, что VLANов может быть много, и надо понимать, что с ним делать. Поэтому в соответствии каждому порту стоит список и первый (нулевой) элемент списка указывает как воспринимать номера VLAN, которые идут дальше.

Пример значения и соответствующей команды:

- [«add», «10», «20»] - команда switchport trunk allowed vlan add 10,20
- [«del», «17»] - команда switchport trunk allowed vlan remove 17
- [«only», «11», «30»] - 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

Код не должен привязываться к конкретным номерам портов. То есть, если в словаре trunk будут другие номера интерфейсов, код должен работать.

Для данных в словаре trunk_template вывод на стандартный поток вывода должен быть таким:

```
interface FastEthernet 0/1
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan add 10,20
interface FastEthernet 0/2
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan 11,30
interface FastEthernet 0/4
 switchport trunk encapsulation dot1q
 switchport mode trunk
 switchport trunk allowed vlan remove 17
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
access_template = [
    "switchport mode access",
    "switchport access vlan",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable",
]

trunk_template = [
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk allowed vlan",
]

access = {"0/12": "10", "0/14": "11", "0/16": "17", "0/17": "150"}
trunk = {"0/1": ["add", "10", "20"], "0/2": ["only", "11", "30"], "0/4": ["del", "17"]}

for intf, vlan in access.items():
    print("interface FastEthernet" + intf)
    for command in access_template:
        if command.endswith("access vlan"):
            print(f" {command} {vlan}")
        else:
            print(f" {command}")
```

7. 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

Примечание: r - read; a - append; w - write

Чтение файлов

В Python есть несколько методов чтения файла:

- read - считывает содержимое файла в строку
- readline - считывает файл построчно
- readlines - считывает строки файла и создает список из строк

Посмотрим как считывать содержимое файлов, на примере файла r1.txt:

```
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

read

Метод read - считывает весь файл в одну строку.

Пример использования метода read:

```
In [1]: f = open('r1.txt')

In [2]: f.read()
Out[2]: '!\nservice timestamps debug datetime msec localtime show-timezone year\nservice␣
↪timestamps log datetime msec localtime show-timezone year\nservice password-encryption\
↪nservice sequence-numbers\n!\nno ip domain lookup\n!\nip ssh version 2\n!\n'

In [3]: f.read()
Out[3]: ''
```

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]:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
['!',
 '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\nservice␣
↪timestamps log datetime msec localtime show-timezone year\nservice password-encryption\
↪nservice sequence-numbers\n!\nno ip domain lookup\n!\nip ssh version 2\n!'
```

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 yearservice timestamps␣
↪log datetime msec localtime show-timezone yearservice password-encryptionservice␣
↪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',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 'no ip domain lookup\n',
 '!\n',
 'ip ssh version 2\n',
```

Если итоговый список записать заново в файл, то в нём уже будут переводы строк:

```
In [18]: f = open('r2.txt', 'w')

In [19]: f.writelines(cfg_lines2)

In [20]: f.close()

In [21]: cat r2.txt
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Закрытие файлов

Примечание: 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
```

Конструкция with

Конструкция with называется менеджер 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, как считываются строки файла:

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

Когда с файлом нужно работать построчно, лучше использовать такой вариант.

В предыдущем выводе, между строками файла были лишние пустые строки, так как print добавляет ещё один перевод строки.

Чтобы избавиться от этого, можно использовать метод `rstrip`:

```
In [2]: with open('r1.txt', 'r') as f:
  ....:     for line in f:
  ....:         print(line.rstrip())
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  ....:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!

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

И конечно же, с конструкцией `with` можно использовать не только такой построчный вариант считывания, все методы, которые рассматривались до этого, также работают:

```
In [4]: with open('r1.txt', 'r') as f:
  ....:     print(f.read())
  ....:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Открытие двух файлов

Иногда нужно работать одновременно с двумя файлами. Например, надо записать некоторые строки из одного файла, в другой.

В таком случае, в блоке with можно открывать два файла таким образом:

```
In [5]: with open('r1.txt') as src, open('result.txt', 'w') as dest:
   ...:     for line in src:
   ...:         if line.startswith('service'):
   ...:             dest.write(line)
   ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [6]: cat result.txt
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
```

Это равнозначно таким двум блокам with:

```
In [7]: with open('r1.txt') as src:
   ...:     with open('result.txt', 'w') as dest:
   ...:         for line in src:
   ...:             if line.startswith('service'):
   ...:                 dest.write(line)
   ...:
```

Примеры работы с файлами

В этом подразделе рассматривается работа с файлами и объединяются темы: файлы, циклы и условия.

При обработке вывода команд или конфигурации часто надо будет записать итоговые данные в словарь. Не всегда очевидно, как обрабатывать вывод команд и каким образом в целом подходить к разбору вывода на части. В этом подразделе рассматриваются несколько примеров, с возрастающим уровнем сложности.

Разбор вывода столбцами

В этом примере будет разбираться вывод команды sh ip int br. Из вывода команды нам надо получить соответствия имя интерфейса - IP-адрес. То есть имя интерфейса - это ключ словаря, а IP-адрес - значение. При этом, соответствие надо делать только для тех интерфейсов, у которых назначен IP-адрес.

Пример вывода команды sh ip int br (файл sh_ip_int_br.txt):

```
R1#show ip interface brief
Interface                  IP-Address      OK? Method Status                Protocol
FastEthernet0/0            15.0.15.1       YES manual up                    up
FastEthernet0/1            10.0.12.1       YES manual up                    up
FastEthernet0/2            10.0.13.1       YES manual up                    up
FastEthernet0/3            unassigned      YES unset  up                    down
Loopback0                  10.1.1.1        YES manual up                    up
Loopback100                100.0.0.1       YES manual up                    up
```

Файл working_with_dict_example_1.py:

```
result = {}

with open('sh_ip_int_br.txt') as f:
    for line in f:
        line_list = line.split()
        if line_list and line_list[1][0].isdigit():
            interface = line_list[0]
            address = line_list[1]
            result[interface] = address

print(result)
```

Команда sh ip int br отображает вывод столбцами. Значит нужные поля находятся в одной строке. Скрипт обрабатывает вывод построчно и каждую строку разбивает с помощью метода split.

Полученный в итоге список содержит столбцы вывода. Так как из всего вывода нужны только интерфейсы на которых настроен IP-адрес, выполняется проверка первого символа второго столбца: если первый символ число, значит на интерфейсе назначен адрес и эту строку надо обрабатывать.

Так как для каждой строки есть пара ключ и значение, они присваиваются в словарь: `result[interface] = address`.

Результатом выполнения скрипта будет такой словарь (тут он разбит на пары ключ-значение для удобства, в реальном выводе скрипта словарь будет отображаться в одну строку):

```
{'FastEthernet0/0': '15.0.15.1',
 'FastEthernet0/1': '10.0.12.1',
 'FastEthernet0/2': '10.0.13.1',
 'Loopback0': '10.1.1.1',
 'Loopback100': '100.0.0.1'}
```

Получение ключа и значения из разных строк вывода

Очень часто вывод команд выглядит таким образом, что ключ и значение находятся в разных строках. И надо придумать каким образом обрабатывать вывод, чтобы получить нужное соответствие.

Например, из вывода команды `sh ip interface` надо получить соответствие имя интерфейса - MTU (файл sh_ip_interface.txt):

```
Ethernet0/0 is up, line protocol is up
  Internet address is 192.168.100.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/1 is up, line protocol is up
  Internet address is 192.168.200.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/2 is up, line protocol is up
  Internet address is 19.1.1.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
```

Имя интерфейса находится в строке вида `Ethernet0/0 is up, line protocol is up`, а MTU в строке вида `MTU is 1500 bytes`.

Например, попробуем запоминать каждый раз интерфейс и выводить его значение, когда встречается MTU, вместе со значением MTU:

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

Вывод организован таким образом, что всегда сначала идет строка с интерфейсом, а затем через несколько строк - строка с MTU. Если запоминать имя интерфейса каждый раз, когда оно встречается, то на момент когда встретится строка с MTU, последний запомненный интерфейс - это тот к которому относится MTU.

Теперь, если необходимо создать словарь с соответствием интерфейс - MTU, достаточно записать значения на момент, когда был найден MTU.

Файл working_with_dict_example_2.py:

```
result = {}

with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'MTU is' in line:
            mtu = line.split()[-2]
            result[interface] = mtu

print(result)
```

Результатом выполнения скрипта будет такой словарь (тут он разбит на пары ключ-значение для удобства, в реальном выводе скрипта словарь будет отображаться в одну строку):

```
{'Ethernet0/0': '1500',
 'Ethernet0/1': '1500',
 'Ethernet0/2': '1500',
 'Ethernet0/3': '1500',
 'Loopback0': '1514'}
```

Этот прием будет достаточно часто полезен, так как вывод команд, в целом, организован очень похожим образом.

Вложенный словарь

Если из вывода команды надо получить несколько параметров, очень удобно использовать словарь с вложенным словарем.

Например, из вывода `sh ip interface` надо получить два параметра: IP-адрес и MTU. Для начала, вывод информации:

```
Ethernet0/0 is up, line protocol is up
  Internet address is 192.168.100.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
Ethernet0/1 is up, line protocol is up
  Internet address is 192.168.200.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  ...
Ethernet0/2 is up, line protocol is up
  Internet address is 19.1.1.1/24
  Broadcast address is 255.255.255.255
  Address determined by non-volatile memory
  MTU is 1500 bytes
  Helper address is not set
  ...
```

На первом этапе каждое значение запоминается в переменную, а затем, выводятся все три значения. Значения выводятся, когда встретилась строка с MTU, потому что она идет последней:

```
In [2]: with open('sh_ip_interface.txt') as f:
   ...:     for line in f:
   ...:         if 'line protocol' in line:
   ...:             interface = line.split()[0]
   ...:         elif 'Internet address' in line:
   ...:             ip_address = line.split()[-1]
   ...:         elif 'MTU' in line:
   ...:             mtu = line.split()[-2]
   ...:             print('{:15}{:17}{}'.format(interface, ip_address, mtu))
   ...:
Ethernet0/0    192.168.100.1/24 1500
Ethernet0/1    192.168.200.1/24 1500
Ethernet0/2    19.1.1.1/24      1500
Ethernet0/3    192.168.230.1/24 1500
Loopback0      4.4.4.4/32       1514
```

Тут используется такой же прием, как в предыдущем примере, но добавляется еще одна вложенность словаря:

```
result = {}

with open('sh_ip_interface.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
            result[interface] = {}
        elif 'Internet address' in line:
            ip_address = line.split()[-1]
            result[interface]['ip'] = ip_address
        elif 'MTU' in line:
            mtu = line.split()[-2]
            result[interface]['mtu'] = mtu
```

(continues on next page)

(продолжение с предыдущей страницы)

```
print(result)
```

Каждый раз, когда встречается интерфейс, в словаре `result` создается ключ с именем интерфейса, которому соответствует пустой словарь. Эта заготовка нужна для того, чтобы на момент когда встретится IP-адрес или MTU можно было записать параметр во вложенный словарь соответствующего интерфейса.

Результатом выполнения скрипта будет такой словарь (тут он разбит на пары ключ-значение для удобства, в реальном выводе скрипта словарь будет отображаться в одну строку):

```
{'Ethernet0/0': {'ip': '192.168.100.1/24', 'mtu': '1500'},
 'Ethernet0/1': {'ip': '192.168.200.1/24', 'mtu': '1500'},
 'Ethernet0/2': {'ip': '19.1.1.1/24', 'mtu': '1500'},
 'Ethernet0/3': {'ip': '192.168.230.1/24', 'mtu': '1500'},
 'Loopback0': {'ip': '4.4.4.4/32', 'mtu': '1514'}}
```

Вывод с пустыми значениями

Иногда, в выводе будут попадаться секции с пустыми значениями. Например, в случае с выводом `sh ip interface`, могут попадаться интерфейсы, которые выглядят так:

```
Ethernet0/1 is up, line protocol is up
  Internet protocol processing disabled
Ethernet0/2 is administratively down, line protocol is down
  Internet protocol processing disabled
Ethernet0/3 is administratively down, line protocol is down
  Internet protocol processing disabled
```

Соответственно тут нет MTU или IP-адреса.

И если выполнить предыдущий скрипт для файла с такими интерфейсами, результат будет таким (вывод для файла sh_ip_interface2.txt):

```
{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Ethernet0/1': {},
 'Ethernet0/2': {},
 'Ethernet0/3': {},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}
```

Если необходимо добавлять интерфейсы в словарь только, когда на интерфейсе назначен IP-адрес, надо перенести создание ключа с именем интерфейса на момент, когда встречается строка с IP-адресом (файл working_with_dict_example_4.py):

```
result = {}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
with open('sh_ip_interface2.txt') as f:
    for line in f:
        if 'line protocol' in line:
            interface = line.split()[0]
        elif 'Internet address' in line:
            ip_address = line.split()[-1]
            result[interface] = {}
            result[interface]['ip'] = ip_address
        elif 'MTU' in line:
            mtu = line.split()[-2]
            result[interface]['mtu'] = mtu

print(result)
```

В этом случае результатом будет такой словарь:

```
{'Ethernet0/0': {'ip': '192.168.100.2/24', 'mtu': '1500'},
 'Loopback0': {'ip': '2.2.2.2/32', 'mtu': '1514'}}
```

Дополнительные материалы

Документация:

- Reading and Writing Files
- The with statement

Статьи:

- The Python «with» Statement by Example

Stack Overflow:

- What is the python “with” statement designed for?

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

> **Предупреждение:** Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 7.1

Обработать строки из файла ospf.txt и вывести информацию по каждой строке в таком виде на стандартный поток вывода:

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

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.2

Создать скрипт, который будет обрабатывать конфигурационный файл config_sw1.txt. Имя файла передается как аргумент скрипту.

Скрипт должен возвращать на стандартный поток вывода команды из переданного конфигурационного файла, исключая строки, которые начинаются с `!`.

Вывод должен быть без пустых строк.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Пример вывода:

```
$ python task_7_2.py config_sw1.txt
Current configuration : 2033 bytes
version 15.0
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
```

(continues on next page)

(продолжение с предыдущей страницы)

```
hostname sw1
interface Ethernet0/0
 duplex auto
interface Ethernet0/1
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan 100
 switchport mode trunk
 duplex auto
 spanning-tree portfast edge trunk
interface Ethernet0/2
 duplex auto
interface Ethernet0/3
 switchport trunk encapsulation dot1q
 switchport trunk allowed vlan 100
 duplex auto
 switchport mode trunk
 spanning-tree portfast edge trunk
...
```

Задание 7.2a

Сделать копию скрипта задания 7.2.

Дополнить скрипт: Скрипт не должен выводить команды, в которых содержатся слова, которые указаны в списке ignore.

При этом скрипт также не должен выводить строки, которые начинаются на !.

Проверить работу скрипта на конфигурационном файле config_sw1.txt. Имя файла передается как аргумент скрипту.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 7.2b

Переделать скрипт из задания 7.2a: вместо вывода на стандартный поток вывода, скрипт должен записать полученные строки в файл

Имена файлов нужно передавать как аргументы скрипту:

- имя исходного файла конфигурации
- имя итогового файла конфигурации

При этом, должны быть отфильтрованы строки, которые содержатся в списке ignore и строки, которые начинаются на „!“.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

Задание 7.3

Скрипт должен обрабатывать записи в файле CAM_table.txt. Каждая строка, где есть MAC-адрес, должна быть обработана таким образом, чтобы на стандартный поток вывода была выведена таблица вида:

```
100      01bb.c580.7000      Gi0/1
200      0a4b.c380.7c00      Gi0/2
300      a2ab.c5a0.700e      Gi0/3
10       0a1b.1c80.7000      Gi0/4
500      02b1.3c80.7b00      Gi0/5
200      1a4b.c580.7000      Gi0/6
300      0a1b.5c80.70f0      Gi0/7
10       01ab.c5d0.70d0      Gi0/8
1000     0a4b.c380.7d00      Gi0/9
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.3a

Сделать копию скрипта задания 7.3.

Переделать скрипт: Отсортировать вывод по номеру VLAN В результате должен получиться такой вывод:

```
10       01ab.c5d0.70d0      Gi0/8
10       0a1b.1c80.7000      Gi0/4
100      01bb.c580.7000      Gi0/1
200      0a4b.c380.7c00      Gi0/2
200      1a4b.c580.7000      Gi0/6
300      0a1b.5c80.70f0      Gi0/7
300      a2ab.c5a0.700e      Gi0/3
500      02b1.3c80.7b00      Gi0/5
1000     0a4b.c380.7d00      Gi0/9
```

Обратите внимание на vlan 1000 - он должен выводиться последним. Правильной сортировки можно добиться, если vlan будет числом, а не строкой.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 7.3b

Сделать копию скрипта задания 7.3a.

Переделать скрипт:

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

Пример работы скрипта:

```
Enter VLAN number: 10
10       0a1b.1c80.7000      Gi0/4
10       01ab.c5d0.70d0      Gi0/8
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

8. Полезные возможности и инструменты

В этом разделе собраны те темы, которые не вошли в предыдущие разделы.

Форматирование строк с помощью f-строк

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

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

Синтаксис

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

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

In [2]: mask = 24

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

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

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

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

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

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

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

```
In [1]: octets = ['10', '1', '1', '1']

In [2]: mask = 24

In [3]: f"IP: {'.'.join(octets)}, mask: {mask}"
Out[3]: 'IP: 10.1.1.1, mask: 24'
```

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

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

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

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

Особенности использования f-строк

При использовании f-строк нельзя сначала создать шаблон, а затем его использовать, как при использовании format.

F-строка сразу выполняется и в нее подставляются значения переменных, которые должны быть определены ранее:

```
In [7]: ip = '10.1.1.1'

In [8]: mask = 24

In [9]: print(f"IP: {ip}, mask: {mask}")
IP: 10.1.1.1, mask: 24
```

Если необходимо подставить другие значения, надо создать новые переменные (с теми же именами) и снова написать f-строку:

```
In [11]: ip = '10.2.2.2'

In [12]: mask = 24

In [13]: print(f"IP: {ip}, mask: {mask}")
IP: 10.2.2.2, mask: 24
```

При использовании f-строк в циклах, f-строку надо писать в теле цикла, чтобы она «подхватывала» новые значения переменных на каждой итерации:

```
In [1]: ip_list = ['10.1.1.1/24', '10.2.2.2/24', '10.3.3.3/24']

In [2]: for ip_address in ip_list:
   ...:     ip, mask = ip_address.split('/')
   ...:     print(f"IP: {ip}, mask: {mask}")
   ...:
IP: 10.1.1.1, mask: 24
IP: 10.2.2.2, mask: 24
IP: 10.3.3.3, mask: 24
```

Примеры использования f-строк

Базовая подстановка переменных:

```
In [1]: intf_type = 'Gi'

In [2]: intf_name = '0/3'

In [3]: f'interface {intf_type}/{intf_name}'
Out[3]: 'interface Gi0/3'
```

Выравнивание столбцами:

```
In [6]: topology = [['sw1', 'Gi0/1', 'r1', 'Gi0/2'],
   ...:             ['sw1', 'Gi0/2', 'r2', 'Gi0/1'],
   ...:             ['sw1', 'Gi0/3', 'r3', 'Gi0/0'],
   ...:             ['sw1', 'Gi0/5', 'sw4', 'Gi0/2']]
   ...:

In [7]: for connection in topology:
   ...:     l_device, l_port, r_device, r_port = connection
   ...:     print(f'{l_device:10} {l_port:7} {r_device:10} {r_port:7}')
   ...:
sw1        Gi0/1   r1         Gi0/2
sw1        Gi0/2   r2         Gi0/1
sw1        Gi0/3   r3         Gi0/0
sw1        Gi0/5   sw4        Gi0/2
```

Ширина столбцов может быть указана через переменную:

```
In [6]: topology = [['sw1', 'Gi0/1', 'r1', 'Gi0/2'],
   ...:             ['sw1', 'Gi0/2', 'r2', 'Gi0/1'],
   ...:             ['sw1', 'Gi0/3', 'r3', 'Gi0/0'],
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:             ['sw1', 'Gi0/5', 'sw4', 'Gi0/2']]
   ...:

In [7]: width = 10

In [8]: for connection in topology:
   ...:     l_device, l_port, r_device, r_port = connection
   ...:     print(f'{l_device:{width}} {l_port:{width}} {r_device:{width}} {r_port:{width}
↪}')
   ...:
sw1        Gi0/1      r1         Gi0/2
sw1        Gi0/2      r2         Gi0/1
sw1        Gi0/3      r3         Gi0/0
sw1        Gi0/5      sw4        Gi0/2
```

Работа со словарями

```
In [1]: session_stats = {'done': 10, 'todo': 5}

In [2]: if session_stats['todo']:
   ...:     print(f"Pomodoros done: {session_stats['done']}, TODO: {session_stats['todo']}
↪")
   ...: else:
   ...:     print(f"Good job! All {session_stats['done']} pomodoros done!")
   ...:
Pomodoros done: 10, TODO: 5
```

Вызов функции len внутри f-строки:

```
In [2]: topology = [['sw1', 'Gi0/1', 'r1', 'Gi0/2'],
   ...:             ['sw1', 'Gi0/2', 'r2', 'Gi0/1'],
   ...:             ['sw1', 'Gi0/3', 'r3', 'Gi0/0'],
   ...:             ['sw1', 'Gi0/5', 'sw4', 'Gi0/2']]
   ...:

In [3]: print(f'Количество подключений в топологии: {len(topology)}')
Количество подключений в топологии: 4
```

Вызов метода upper внутри f-строки:

```
In [1]: name = 'python'

In [2]: print(f'Zen of {name.upper()}')
Zen of PYTHON
```

Конвертация чисел в двоичный формат:

```
In [7]: ip = '10.1.1.1'

In [8]: oct1, oct2, oct3, oct4 = ip.split('.')

In [9]: print(f'{int(oct1):08b} {int(oct2):08b} {int(oct3):08b} {int(oct4):08b}')
00001010 00000001 00000001 00000001
```

Что использовать format или f-строки

Во многих случаях f-строки удобней использовать, так как шаблон выглядит понятней и компактней. Однако бывают случаи, когда метод format удобней. Например:

```
In [6]: ip = [10, 1, 1, 1]

In [7]: oct1, oct2, oct3, oct4 = ip
   ...: print(f'{oct1:08b} {oct2:08b} {oct3:08b} {oct4:08b}')
   ...:
00001010 00000001 00000001 00000001

In [8]: template = "{:08b} "*4

In [9]: template.format(oct1, oct2, oct3, oct4)
Out[9]: '00001010 00000001 00000001 00000001 '
```

Еще одна ситуация, когда format, как правило, удобней использовать: необходимость использовать в скрипте один и тот же шаблон много раз. F-строка выполнится первый раз и подставит текущие значения переменных и для использования шаблона еще раз, его надо заново писать. Это значит, что в скрипте будут находится копии одной и то же строки. В то же время format позволяет создать шаблон в одном месте и потом использовать его повторно, подставляя переменные по мере необходимости.

Это можно обойти создав функцию, но создавать функцию для вывода строки по шаблону далеко не всегда оправдано. Пример создания функции:

```
In [1]: def show_me_ip(ip, mask):
   ...:     return f"IP: {ip}, mask: {mask}"
   ...:

In [2]: show_me_ip('10.1.1.1', 24)
Out[2]: 'IP: 10.1.1.1, mask: 24'

In [3]: show_me_ip('192.16.10.192', 28)
Out[3]: 'IP: 192.16.10.192, mask: 28'
```

Распаковка переменных

Распаковка переменных - это специальный синтаксис, который позволяет присваивать переменным элементы итерируемого объекта.

Примечание: Достаточно часто этот функционал встречается под именем tuple unpacking, но распаковка работает на любом итерируемом объекте, не только с кортежами

Пример распаковки переменных:

```
In [1]: interface = ['FastEthernet0/1', '10.1.1.1', 'up', 'up']

In [2]: intf, ip, status, protocol = interface

In [3]: intf
Out[3]: 'FastEthernet0/1'

In [4]: ip
Out[4]: '10.1.1.1'
```

Такой вариант намного удобней использовать, чем использование индексов:

```
In [5]: intf, ip, status, protocol = interface[0], interface[1], interface[2],␣
↪interface[3]
```

При распаковке переменных каждый элемент списка попадает в соответствующую переменную. Важно учитывать, что переменных слева должно быть ровно столько, сколько элементов в списке.

Если переменных больше или меньше, возникнет исключение:

```
In [6]: intf, ip, status = interface
---------------------------------------------------------------
ValueError                 Traceback (most recent call last)
<ipython-input-11-a304c4372b1a> in <module>()
----> 1 intf, ip, status = interface

ValueError: too many values to unpack (expected 3)

In [7]: intf, ip, status, protocol, other = interface
---------------------------------------------------------------
ValueError                 Traceback (most recent call last)
<ipython-input-12-ac93e78b978c> in <module>()
----> 1 intf, ip, status, protocol, other = interface

ValueError: not enough values to unpack (expected 5, got 4)
```

Замена ненужных элементов _

Часто из всех элементов итерируемого объекта нужны только некоторые. При этом синтаксис распаковки требует указать ровно столько переменных, сколько элементов в итерируемом объекте.

Если, например, из строки line надо получить только VLAN, MAC и интерфейс, надо все равно указать переменную для типа записи:

```
In [8]: line = '100    01bb.c580.7000    DYNAMIC     Gi0/1'

In [9]: vlan, mac, item_type, intf = line.split()

In [10]: vlan
Out[10]: '100'

In [11]: intf
Out[11]: 'Gi0/1'
```

Если тип записи не нужен в дальнейшем, можно заменить переменную item_type нижним подчеркиванием:

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

Таким образом явно указывается то, что этот элемент не нужен.

Нижнее подчеркивание можно использовать и несколько раз:

```
In [13]: dhcp = '00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10
↪FastEthernet0/1'

In [14]: mac, ip, _, _, vlan, intf = dhcp.split()

In [15]: mac
Out[15]: '00:09:BB:3D:D6:58'

In [16]: vlan
Out[16]: '10'
```

Использование *

Распаковка переменных поддерживает специальный синтаксис, который позволяет распаковывать несколько элементов в один. Если поставить * перед именем переменной, в нее запишутся все элементы, кроме тех, что присвоены явно.

Например, так можно получить первый элемент в переменную first, а остальные в rest:

```
In [18]: vlans = [10, 11, 13, 30]

In [19]: first, *rest = vlans

In [20]: first
Out[20]: 10

In [21]: rest
Out[21]: [11, 13, 30]
```

При этом переменная со звездочкой всегда будет содержать список:

```
In [22]: vlans = (10, 11, 13, 30)

In [22]: first, *rest = vlans

In [23]: first
Out[23]: 10

In [24]: rest
Out[24]: [11, 13, 30]
```

Если элемент всего один, распаковка все равно отработает:

```
In [25]: first, *rest = vlans

In [26]: first
Out[26]: 55

In [27]: rest
Out[27]: []
```

Такая переменная со звездочкой в выражении распаковки может быть только одна.

```
In [28]: vlans = (10, 11, 13, 30)

In [29]: first, *rest, *others = vlans
  File "<ipython-input-37-dedf7a08933a>", line 1
    first, *rest, *others = vlans
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                                    ^
SyntaxError: two starred expressions in assignment
```

Такая переменная может находиться не только в конце выражения:

```
In [30]: vlans = (10, 11, 13, 30)

In [31]: *rest, last = vlans

In [32]: rest
Out[32]: [10, 11, 13]

In [33]: last
Out[33]: 30
```

Таким образом можно указать, что нужен первый, второй и последний элемент:

```
In [34]: cdp = 'SW1     Eth 0/0    140   S I   WS-C3750-  Eth 0/1'

In [35]: name, l_intf, *other, r_intf = cdp.split()

In [36]: name
Out[36]: 'SW1'

In [37]: l_intf
Out[37]: 'Eth'

In [38]: r_intf
Out[38]: '0/1'
```

Примеры распаковки

Распаковка итерируемых объектов

Эти примеры показывают, что распаковывать можно не только списки, кортежи и строки, но и любой другой итерируемый объект.

Распаковка range:

```
In [39]: first, *rest = range(1, 6)

In [40]: first
Out[40]: 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [41]: rest
Out[41]: [2, 3, 4, 5]
```

Распаковка zip:

```
In [42]: a = [1, 2, 3, 4, 5]

In [43]: b = [100, 200, 300, 400, 500]

In [44]: zip(a, b)
Out[44]: <zip at 0xb4df4fac>

In [45]: list(zip(a, b))
Out[45]: [(1, 100), (2, 200), (3, 300), (4, 400), (5, 500)]

In [46]: first, *rest, last = zip(a, b)

In [47]: first
Out[47]: (1, 100)

In [48]: rest
Out[48]: [(2, 200), (3, 300), (4, 400)]

In [49]: last
Out[49]: (5, 500)
```

Пример распаковки в цикле for

Пример цикла, который проходится по ключам:

```
In [50]: access_template = ['switchport mode access',
    ...:                    'switchport access vlan',
    ...:                    'spanning-tree portfast',
    ...:                    'spanning-tree bpduguard enable']
    ...:

In [51]: access = {'0/12': 10, '0/14': 11, '0/16': 17}

In [52]: for intf in access:
    ...:     print(f'interface FastEthernet {intf}')
    ...:     for command in access_template:
    ...:         if command.endswith('access vlan'):
    ...:             print(' {} {}'.format(command, access[intf]))
    ...:         else:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:             print(' {}'.format(command))
    ...:
interface FastEthernet0/12
 switchport mode access
 switchport access vlan 10
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/14
 switchport mode access
 switchport access vlan 11
 spanning-tree portfast
 spanning-tree bpduguard enable
interface FastEthernet0/16
 switchport mode access
 switchport access vlan 17
 spanning-tree portfast
 spanning-tree bpduguard enable
```

Вместо этого можно проходиться по парам ключ-значение и сразу же распаковывать их в разные переменные:

```
In [53]: for intf, vlan in access.items():
    ...:     print(f'interface FastEthernet {intf}')
    ...:     for command in access_template:
    ...:         if command.endswith('access vlan'):
    ...:             print(' {} {}'.format(command, vlan))
    ...:         else:
    ...:             print(' {}'.format(command))
    ...:
```

Пример распаковки элементов списка в цикле:

```
In [54]: table
Out[54]:
[['100', 'a1b2.ac10.7000', 'DYNAMIC', 'Gi0/1'],
 ['200', 'a0d4.cb20.7000', 'DYNAMIC', 'Gi0/2'],
 ['300', 'acb4.cd30.7000', 'DYNAMIC', 'Gi0/3'],
 ['100', 'a2bb.ec40.7000', 'DYNAMIC', 'Gi0/4'],
 ['500', 'aa4b.c550.7000', 'DYNAMIC', 'Gi0/5'],
 ['200', 'a1bb.1c60.7000', 'DYNAMIC', 'Gi0/6'],
 ['300', 'aa0b.cc70.7000', 'DYNAMIC', 'Gi0/7']]


In [55]: for line in table:
    ...:     vlan, mac, _, intf = line
    ...:     print(vlan, mac, intf)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:
100 a1b2.ac10.7000 Gi0/1
200 a0d4.cb20.7000 Gi0/2
300 acb4.cd30.7000 Gi0/3
100 a2bb.ec40.7000 Gi0/4
500 aa4b.c550.7000 Gi0/5
200 a1bb.1c60.7000 Gi0/6
300 aa0b.cc70.7000 Gi0/7
```

Но еще лучше сделать так:

```
In [56]: for vlan, mac, _, intf in table:
    ...:     print(vlan, mac, intf)
    ...:
100 a1b2.ac10.7000 Gi0/1
200 a0d4.cb20.7000 Gi0/2
300 acb4.cd30.7000 Gi0/3
100 a2bb.ec40.7000 Gi0/4
500 aa4b.c550.7000 Gi0/5
200 a1bb.1c60.7000 Gi0/6
300 aa0b.cc70.7000 Gi0/7
```

List, dict, set comprehensions

Python поддерживает специальные выражения, которые позволяют компактно создавать списки, словари и множества.

На английском эти выражения называются, соответственно:

- List comprehensions
- Dict comprehensions
- Set comprehensions

К сожалению, официальный перевод на русский звучит как абстракция списков или списковое включение, что не особо помогает понять суть объекта.

В книге использовался перевод «генератор списка», что, к сожалению, тоже не самый удачный вариант, так как в Python есть отдельное понятие генератор и генераторные выражения, но он лучше отображает суть выражения.

Эти выражения не только позволяют более компактно создавать соответствующие объекты, но и создают их быстрее. И хотя поначалу они требуют определенной привычки использования и понимания, они очень часто используются.

List comprehensions (генераторы списков)

Генератор списка (list comprehensions или list comp) - это выражение вида:

```
vlans = [int(vl) for vl in items]
```

Список items:

```
items = ["10", "20", "30", "1", "11", "100"]
```

В общем случае, list comprehension это выражение, которое преобразует итерируемый объект в список. То есть, последовательность элементов преобразуется и добавляется в новый список.

List comp выше аналогичен такой цикл:

```
items = ["10", "20", "30", "1", "11", "100"]

vlans = []
for vl in items:
    vlans.append(int(vl))

print(vlans)
# [10, 20, 30, 1, 11, 100]
```

Соответствие между обычным циклом и генератором списка:

```
items = ["10", "20", "30", "1", "11", "100"]

vlans = []
for vl in items:
    vlans.append(int(vl))

vlans = [int(vl) for vl in items]
```

В list comprehensions можно использовать выражение if. Таким образом можно добавлять в список только некоторые объекты.

Например, такой цикл отбирает те элементы, которые являются числами, конвертирует их и добавляет в итоговый список only_digits:

```
items = ['10', '20', 'a', '30', 'b', '40']

only_digits = []
```

(continues on next page)

(продолжение с предыдущей страницы)

```
for item in items:
    if item.isdigit():
        only_digits.append(int(item))

In [9]: print(only_digits)
[10, 20, 30, 40]
```

Аналогичный вариант в виде list comprehensions:

```
items = ['10', '20', 'a', '30', 'b', '40']
only_digits = [int(item) for item in items if item.isdigit()]

In [12]: print(only_digits)
[10, 20, 30, 40]
```

Соответствие между циклом с условием и генератором списка с условием:

```
items = ["10", "20", "30", "aa", "1", "11", "bb"]

vlans = []
for vl in items:
    if vl.isdigit():
        vlans.append(int(vl))


vlans = [int(vl) for vl in items if vl.isdigit()]
```

Конечно, далеко не все циклы можно переписать как генератор списка, но когда это можно сделать, и при этом выражение не усложняется, лучше использовать генераторы списка.

Примечание: В Python генераторы списка могут также заменить функции filter и map и считаются более понятными вариантами решения.

С помощью генератора списка также удобно получать элементы из вложенных словарей:

```
london_co = {
    'r1' : {
    'hostname': 'london_r1',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    'model': '4451',
    'ios': '15.4',
    'ip': '10.255.0.1'
    },
    'r2' : {
    'hostname': 'london_r2',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '4451',
    'ios': '15.4',
    'ip': '10.255.0.2'
    },
    'sw1' : {
    'hostname': 'london_sw1',
    'location': '21 New Globe Walk',
    'vendor': 'Cisco',
    'model': '3850',
    'ios': '3.6.XE',
    'ip': '10.255.0.101'
    }
}

In [14]: [london_co[device]['ios'] for device in london_co]
Out[14]: ['15.4', '15.4', '3.6.XE']

In [15]: [london_co[device]['ip'] for device in london_co]
Out[15]: ['10.255.0.1', '10.255.0.2', '10.255.0.101']
```

Полный синтаксис генератора списка выглядит так:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN ]
```

Это значит, можно использовать несколько for в выражении.

Например, в списке vlans находятся несколько вложенных списков с VLAN'ами:

```
vlans = [[10, 21, 35], [101, 115, 150], [111, 40, 50]]
```

Из этого списка надо сформировать один плоский список с номерами VLAN. Первый вариант — с помощью циклов for:

```
result = []
```

(continues on next page)

(продолжение с предыдущей страницы)

```
for vlan_list in vlans:
    for vlan in vlan_list:
        result.append(vlan)


In [19]: print(result)
[10, 21, 35, 101, 115, 150, 111, 40, 50]
```

Аналогичный вариант с генератором списков:

```
vlans = [[10, 21, 35], [101, 115, 150], [111, 40, 50]]
result = [vlan for vlan_list in vlans for vlan in vlan_list]

In [22]: print(result)
[10, 21, 35, 101, 115, 150, 111, 40, 50]
```

Соответствие между двумя вложенными циклами и генератором списка с двумя циклами:

```
all_vlans = [[10, 21, 35], [101, 115, 150], [111, 40, 50]]

result = []
for vl_list in all_vlans:
    for vl in vlan_list:
        result.append(vl)

result = [vl for vl_list in all_vlans for vl in vl_list]
```

Можно одновременно проходиться по двум последовательностям, используя zip:

```
vlans = [100, 110, 150, 200]
names = ['mngmt', 'voice', 'video', 'dmz']

result = ['vlan {}\n name {}'.format(vlan, name) for vlan, name in zip(vlans, names)]

In [26]: print('\n'.join(result))
vlan 100
 name mngmt
vlan 110
 name voice
vlan 150
 name video
vlan 200
 name dmz
```

Dict comprehensions (генераторы словарей)

Генераторы словарей аналогичны генераторам списков, но они используются для создания словарей.

Например, такое выражение:

```
d = {}

for num in range(1, 11):
    d[num] = num**2

In [29]: print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Можно заменить генератором словаря:

```
d = {num: num**2 for num in range(1, 11)}

In [31]: print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Еще один пример, в котором надо преобразовать существующий словарь и перевести все ключи в нижний регистр. Для начала, вариант решения без генератора словаря:

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

lower_r1 = {}

for key, value in r1.items():
    lower_r1[key.lower()] = value

In [35]: lower_r1
Out[35]:
{'hostname': 'london_r1',
 'ios': '15.4',
 'ip': '10.255.0.1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

Аналогичный вариант с помощью генератора словаря:

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

lower_r1 = {key.lower(): value for key, value in r1.items()}

In [38]: lower_r1
Out[38]:
{'hostname': 'london_r1',
 'ios': '15.4',
 'ip': '10.255.0.1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
```

Как и list comprehensions, dict comprehensions можно делать вложенными. Попробуем аналогичным образом преобразовать ключи во вложенных словарях:

```
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'
    }
```

(continues on next page)

(продолжение с предыдущей страницы)

```
}

lower_london_co = {}

for device, params in london_co.items():
    lower_london_co[device] = {}
    for key, value in params.items():
        lower_london_co[device][key.lower()] = value

In [42]: lower_london_co
Out[42]:
{'r1': {'hostname': 'london_r1',
  'ios': '15.4',
  'ip': '10.255.0.1',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'r2': {'hostname': 'london_r2',
  'ios': '15.4',
  'ip': '10.255.0.2',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'sw1': {'hostname': 'london_sw1',
  'ios': '3.6.XE',
  'ip': '10.255.0.101',
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

Аналогичное преобразование с dict comprehensions:

```
result = {device: {key.lower(): value for key, value in params.items()}
          for device, params in london_co.items()}

In [44]: result
Out[44]:
{'r1': {'hostname': 'london_r1',
  'ios': '15.4',
  'ip': '10.255.0.1',
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'r2': {'hostname': 'london_r2',
  'ios': '15.4',
  'ip': '10.255.0.2',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  'location': '21 New Globe Walk',
  'model': '4451',
  'vendor': 'Cisco'},
 'sw1': {'hostname': 'london_sw1',
  'ios': '3.6.XE',
  'ip': '10.255.0.101',
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

Set comprehensions (генераторы множеств)

Генераторы множеств в целом аналогичны генераторам списков.

Например, надо получить множество с уникальными номерами VLAN'ов:

```
vlans = [10, '30', 30, 10, '56']

unique_vlans = {int(vlan) for vlan in vlans}

In [47]: unique_vlans
Out[47]: {10, 30, 56}
```

Аналогичное решение, без использования set comprehensions:

```
vlans = [10, '30', 30, 10, '56']

unique_vlans = set()

for vlan in vlans:
    unique_vlans.add(int(vlan))

In [51]: unique_vlans
Out[51]: {10, 30, 56}
```

Отладка кода

Отладка кода это сложный процесс, который даже с опытом, остается сложным. Упростить его можно изучив доступные инструменты для отладки.

Глобально, можно сказать, есть два варианта отладки:

- добавляем print в каком-то виде
- используем отладчик

Оба варианта можно использовать и конечно при использовании отладчика намного больше возможностей, но при этом надо потратить время на изучение самого отладчика.

Также хочется выделить отдельно отладку для изучения Python. То есть когда вы используете какой-то отладчик и/или print не потому что код не работает, а потому что вы пытаетесь понять, что происходит в коде в тот или иной момент.

Для изучения Python подходят:

- сайт pythontutor
- отладчики в редакторах для начинающих: Mu, Thonny
- технически также подходит print
- если опыт в программировании уже есть, скорее всего, также подойдут отладчики в IDE

Отладчик

Отладчик (debugger) это отдельный модуль, софт или часть редактора/IDE, которая позволяет делать отладку кода.

Примеры отладчиков и редакторов с отладчиками:

- сайт pythontutor
- pdb - модуль стандартной библиотеки Python (есть много разновидностей pdbr, ipdb, pdbpp)
- Mu
- Thonny
- PyCharm
- VS Code

Отладка с помощью print и разновидностей

Отладка с помощью print обычно заключается в том, что в код, в непонятных местах или перед строкой с ошибкой, добавляется print в каком-то виде.

print vs pprint vs print(f"{value=}")

Обычный print с выводом значений переменных не всегда лучший выбор, так как, например, print одинаково выводит строку "100" и число 100.

```
In [1]: print("100")
100

In [2]: print(100)
100
```

Тут может быть удобнее pprint, который выводит строки с кавычками, а числа нет:

```
In [3]: from pprint import pprint

In [5]: pprint("100")
'100'

In [6]: pprint(100)
100
```

Также pprint удобен для вывода более сложных строк, так как он показывает специальные символы:

```
In [7]: line = "\nline1\t\nline2\r\n"

In [8]: print(line)

line1
line2


In [9]: pprint(line)
'\nline1\t\nline2\r\n'
```

Примечание: Технически можно вывести такую же строку с print, если использовать repr вместе с print:

```
In [11]: print(repr(line))
'\n\nline1\t\nline2\r\n'
```

Минус pprint в том, что он может выводить только одно значение, то есть нельзя сделать как с print, вывод сразу нескольких переменных. Плюс в том, что pprint также умеет выводит более сложные структуры данных с понятным форматированием, а также дает возможность указывать «глубину» данных, которую надо показывать. Подробнее о *Модуль pprint*.

Также, начиная с версии 3.8, в Python появилась специальная разновидность вывода в f-строках, именно для отладки - `f"{var=}"`:

```
In [13]: line = "\nline1\t\nline2"

In [14]: item = "100"

In [15]: print(f"{line=} {item=}")
line='\nline1\t\nline2' item='100'

In [16]: for i in range(5):
    ...:     print(f"{i=}")
    ...:
i=0
i=1
i=2
i=3
i=4
```

Этот вариант с одной стороны, позволяет отображать сколько угодно значений, плюс автоматически добавляет имя переменной к выводу, с другой, выводит как и pprint, строки со специальными символами и кавычками.

locals

Функция locals показывает все локальные переменные. Если в коде нет функций, это будут все глобальные переменные, если сделать вывод внутри функции, только переменные этой функции.

Пример кода:

```
from pprint import pprint

item = "100"
line = "\nline1\n\tline2"

pprint(locals())
```

Вывод locals

```
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': '/home/user/repos/pyneng-14/pyneng-course-tasks/exercises/15_module_re/code.
↪py',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0xb73f30d0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'item': '100',
 'line': '\nline1\n\tline2',
 'pprint': <function pprint at 0xb7253808>}
```

Пример кода с функцией:

```
from pprint import pprint

def num_sum(x, y):
    result = x + y
    pprint(locals())
    return result
```

Вывод locals при вызове функции:

```
In [4]: num_sum(10, 5)
{'result': 15, 'x': 10, 'y': 5}
Out[4]: 15
```

rich.inspect

Rich это сторонний модуль, в котором есть много возможностей для вывода информации терминале: красивые таблицы, вывод в цвете, progress bar и другие возможности. Одна из удобных возможностей rich - функция inspect. Она выводит информацию про указанный объект, его методы и атрибуты.

Установка rich

```
pip install rich
```

Пример использования rich.inspect со списком:

```
In [10]: from rich import inspect

In [11]: items = [10, 20, 30, 40]

In [12]: inspect(items)
╭──────────────────────── <class 'list'> ────────────────────────╮
│ Built-in mutable sequence.                                     │
│                                                                │
│ ╭────────────────────────────────────────────────────────────╮ │
```

(continues on next page)

(продолжение с предыдущей страницы)

```
│ │ [10, 20, 30, 40]                                                │ │
│ └─────────────────────────────────────────────────────────────────┘ │
│                                                                     │
│ 36 attribute(s) not shown. Run inspect(inspect) for options.        │
└─────────────────────────────────────────────────────────────────────┘

In [13]: inspect(items, methods=True)
┌──────────────────────────────────────── <class 'list'> ────────────────────────────────────────┐
│ Built-in mutable sequence.                                                                     │
│                                                                                                │
│ ┌──────────────────────────────────────────────────────────────────────────────────────────┐ │
│ │ [10, 20, 30, 40]                                                                         │ │
│ └──────────────────────────────────────────────────────────────────────────────────────────┘ │
│                                                                                                │
│  append = def append(object, /): Append object to the end of the list.                         │
│   clear = def clear(): Remove all items from list.                                             │
│    copy = def copy(): Return a shallow copy of the list.                                       │
│   count = def count(value, /): Return number of occurrences of value.                          │
│  extend = def extend(iterable, /): Extend list by appending elements from the                  │
│           iterable.                                                                            │
│   index = def index(value, start=0, stop=2147483647, /): Return first index of value.          │
│  insert = def insert(index, object, /): Insert object before index.                            │
│     pop = def pop(index=-1, /): Remove and return item at index (default last).                │
│  remove = def remove(value, /): Remove first occurrence of value.                              │
│ reverse = def reverse(): Reverse *IN PLACE*.                                                   │
│    sort = def sort(*, key=None, reverse=False): Sort the list in ascending order and           │
│           return None.                                                                         │
└────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Пример использования rich.inspect с файлом:

```
In [19]: f = open("code.py")

In [20]: inspect(f)
┌──────────────────────── <class '_io.TextIOWrapper'> ────────────────────────┐
│ Character and line based layer over a BufferedIOBase object, buffer.        │
│                                                                             │
│ ┌─────────────────────────────────────────────────────────────────────────┐ │
│ │ <_io.TextIOWrapper name='code.py' mode='r' encoding='UTF-8'>            │ │
│ └─────────────────────────────────────────────────────────────────────────┘ │
│                                                                             │
│         buffer = <_io.BufferedReader name='code.py'>                        │
│         closed = False                                                      │
│       encoding = 'UTF-8'                                                    │
│         errors = 'strict'                                                   │
│ line_buffering = False                                                      │
```

(continues on next page)

(продолжение с предыдущей страницы)

```
│           mode = 'r'                                                  │
│           name = 'code.py'                                            │
│       newlines = None                                                 │
│  write_through = False                                                │
└───────────────────────────────────────────────────────────────────────┘


In [21]: inspect(f, methods=True)
┌──────────────────────────────── <class '_io.TextIOWrapper'> ─────────────────────────────┐
│ Character and line based layer over a BufferedIOBase object, buffer.                      │
│                                                                                           │
│ ┌─────────────────────────────────────────────────────────────────────────────────────┐   │
│ │ <_io.TextIOWrapper name='code.py' mode='r' encoding='UTF-8'>                        │   │
│ └─────────────────────────────────────────────────────────────────────────────────────┘   │
│                                                                                           │
│         buffer = <_io.BufferedReader name='code.py'>                                      │
│         closed = False                                                                    │
│       encoding = 'UTF-8'                                                                  │
│         errors = 'strict'                                                                 │
│ line_buffering = False                                                                    │
│           mode = 'r'                                                                      │
│           name = 'code.py'                                                                │
│       newlines = None                                                                     │
│  write_through = False                                                                    │
│          close = def close(): Flush and close the IO object.                              │
│         detach = def detach(): Separate the underlying buffer from the TextIOBase and     │
│                  return it.                                                               │
│         fileno = def fileno(): Returns underlying file descriptor if one exists.          │
│          flush = def flush(): Flush write buffers, if applicable.                         │
│         isatty = def isatty(): Return whether this is an 'interactive' stream.            │
│           read = def read(size=-1, /): Read at most n characters from stream.             │
│       readable = def readable(): Return whether object was opened for reading.            │
│       readline = def readline(size=-1, /): Read until newline or EOF.                     │
│      readlines = def readlines(hint=-1, /): Return a list of lines from the stream.       │
│    reconfigure = def reconfigure(*, encoding=None, errors=None, newline=None,             │
│                  line_buffering=None, write_through=None): Reconfigure the text stream    │
│                  with new parameters.                                                     │
│           seek = def seek(cookie, whence=0, /): Change stream position.                   │
│       seekable = def seekable(): Return whether object supports random access.            │
│           tell = def tell(): Return current stream position.                              │
│       truncate = def truncate(pos=None, /): Truncate file to size bytes.                  │
│       writable = def writable(): Return whether object was opened for writing.            │
│          write = def write(text, /):                                                      │
│                  Write string to stream.                                                  │
│                  Returns the number of characters written (which is always equal to       │
│                  the length of the string).                                               │
```

(continues on next page)

(продолжение с предыдущей страницы)

```
│     writelines = def writelines(lines, /): Write a list of lines to stream.            │
└──────────────────────────────────────────────────────────────────────────────────────────┘
```

rich.traceback

Еще одна полезная возможность rich - красивый traceback.

Код с ошибкой:

```
vlans = ["1", "2", "3", "test", "4", "5", "switchport allowed vlans add"]

vlans_list = []
for vl in vlans:
    new_vl = int(vl)
    vlans_list.append(new_vl)

print(vlans_list)
```

Стандартный traceback для кода:

```
$ python basics_debug_05_rich_traceback.py
Traceback (most recent call last):
  File "/examples/basics_debug_05_rich_traceback.py", line 11, in <module>
    new_vl = int(vl)
ValueError: invalid literal for int() with base 10: 'test'
```

С использованием rich (часть вывода locals сокращена):

```
$ python basics_debug_05_rich_traceback.py
╭──────────────────────── Traceback (most recent call last) ────────────────────────╮
│ /examples/basics_debug_05_rich_traceback.py:11 in <module>                        │
│                                                                                   │
│    7                                                                              │
│    8                                                                              │
│    9 vlans_list = []                                                              │
│   10 for vl in vlans:                                                             │
│ ❯ 11 │   new_vl = int(vl)                                                         │
│   12 │   vlans_list.append(new_vl)                                                │
│   13                                                                              │
│   14 print(vlans_list)                                                            │
│   15                                                                              │
│                                                                                   │
│ ╭──────────────────────────────── locals ─────────────────────────────────────╮ │
│ │          new_vl = 3                                                          │ │
│ │              vl = 'test'                                                     │ │
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| |           vlans = [                                                  | |
| |                   |   '1',                                           | |
| |                   |   '2',                                           | |
| |                   |   '3',                                           | |
| |                   |   'test',                                        | |
| |                   |   '4',                                           | |
| |                   |   '5',                                           | |
| |                   |   'switchport allowed vlans add'                 | |
| |                   ]                                                  | |
| |      vlans_list = [1, 2, 3]                                          | |
| ╰──────────────────────────────────────────────────────────────────────╯ |
╰──────────────────────────────────────────────────────────────────────────╯
ValueError: invalid literal for int() with base 10: 'test'
```

Получить такой вывод можно добавив в файл с кодом такие строки:

```
from rich.traceback import install
install(show_locals=True, extra_lines=5)
```

Полный код

```
from rich.traceback import install
install(show_locals=True, extra_lines=5)

vlans = ["1", "2", "3", "test", "4", "5", "switchport allowed vlans add"]

vlans_list = []
for vl in vlans:
    new_vl = int(vl)
    vlans_list.append(new_vl)

print(vlans_list)
```

И, если такой traceback понравится, можно сделать так, чтобы он использовался по умолчанию. Для этого надо создать файл sitecustomize.py в каталоге site-packages с таким содержимым:

```
from rich.traceback import install
install(show_locals=True, extra_lines=5)
```

Как понять какой каталог site-packages использовать:

```
$ python -m site
sys.path = [
    '/home/user/repos/examples/',
    '/usr/local/lib/python310.zip',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    '/usr/local/lib/python3.10',
    '/usr/local/lib/python3.10/lib-dynload',
    '/home/user/venv/pyneng-py3-10-0/lib/python3.10/site-packages',
]
USER_BASE: '/home/user/.local' (exists)
USER_SITE: '/home/user/.local/lib/python3.10/site-packages' (doesn't exist)
```

Полный путь к site-packages показан в sys.path, в данном случае это путь:

```
'/home/user/venv/pyneng-py3-10-0/lib/python3.10/site-packages',
```

Встроенный отладчик pdb

Как запустить pdb

```
python -m pdb script.py
```

Для выхода из pdb используется команда q.

В любой момент можно перезапустить скрипт, без потери breakpoint, с помощью команды `run`.

Базовые команды передвижения по программе

- n (next) - выполнить все до следующей строки. Эта команда не заходит в функции, которые вызываются в строке
- s (step) - выполнить текущую строку, остановиться как можно раньше. Эта команда заходит в функции, которые вызываются в строке
- c (continue) - выполнить все до breakpoint. Также полезна, когда скрипт отрабатывает с исключением, позволяет дойти до строки, где возникло исключение

Контекст в коде, переменные

- l (list) - показывает следующую строку, которая будет выполняться и 5 строк до и после нее. При добавлении диапазона показывает указанные строки, например, `list 1, 20` покажет код с 1 по 20 строку
- ll (longlist) - показывает весь метод или функцию в котором мы находимся
- a (args) - показывает аргументы функции (или метода) и их значения. Работает только внутри функции

- p - показывает значение переменной, работает как print. Синтаксис `p vara`, где vara имя переменной
- pp - показывает значение переменной, работает как pprint. Синтаксис `pp vara`, где vara имя переменной

Выполнение Python команд в pdb

Любую команду можно выполнить указав `!` перед ней:

```
!vara = 55
!result.append(vara)
```

Таким образом можно пробовать выполнить какие-то действия в текущем контексте программы, изменить значения переменных.

Также можно перейти в интерпретатор python из текущего контекста. Для этого используется команда interact:

```
(Pdb) interact
*interactive*
>>> print(cfg)
<_io.TextIOWrapper name='sh_cdp_n_sw1.txt' mode='r' encoding='UTF-8'>
>>> cfg.closed
False
>>>
>>> data = ['1','2','3']
>>> print(','.join(data))
1,2,3
>>>
now exiting InteractiveConsole...
(Pdb)
```

Для выхода из интерпретатор используется команда `Ctrl-d`.

Дополнительные команды по передвижению

- until - выполнить все до указанной строки. Синтаксис `until 15`, где 15 номер строки
- return - выполняется внутри функции и выполняет все до return
- u (up) - передвинутся на один уровень выше в стеке вызовов. Например, если мы по цепоцке переходили в один вызов функции, затем в друго, чтобы вернуться назад надо использовать up
- d (down) - передвинутся на один уровень ниже в стеке вызовов

Breakpoints

- b (break) - команда для установки breakpoint

Если команда указывается с аргументом, например, `break 12` или `break check_ip`, устанавливается breakpoint. Без аргументов, команда показывает все установленные breakpoint.

Удаление breakpoint под номером 1:

```
clear 1
```

Удалить все breakpoint можно clear без аргументов.

Базовые варианты установки breakpoint

Установить breakpoint в строке 12:

```
break 12
```

Установить breakpoint в первой строке функции check_ip:

```
break check_ip
```

Breakpoint с условием

Сделать breakpoint в строке 12, если значение переменной num будет больше 10:

```
break 12, num > 10
```

Привязка команд к breakpoint

Создаем breakpoint (предполагаем, что он первый, поэтому его номер будет 1):

```
break 12
```

Добавляем команды, которые будут выполняться каждый раз, когда попадаем на breakpoint (var1, var2, result_dict должны быть заменены на ваши переменные)

```
commands 1
 pp var1
 pp var2
 pp result_dict
end
```

ipdb

Модуль ipdb это одна из разновидностей pdb, которая добавляет подсветку синтаксиса, вызов ipython вместо встроенного интерпретатора, автопродолжение команд.

Установка ipdb:

```
pip install ipdb
```

Как запустить ipdb

```
python -m ipdb script.py
```

В остальном, команды те же, что в и pdb, только по команде interact откроется ipython, а не встроенный интепретатор python.

Дополнительные материалы

Документация:

- PEP 3132 – Extended Iterable Unpacking

Статьи:

- List, Dict And Set Comprehensions By Example - хорошая статья. И в конце статьи есть несколько упражнений (с ответами)
- Python List Comprehensions: Explained Visually - отличное объяснение list comprehensions, плюс видео

Stack Overflow:

- Ответ со множеством вариантов распаковки

pdb:

- The Python Debugger (pdb) - основы работы с pdb
- Python 3 Module of the Week. pdb — Interactive Debugger
- Python Debugging With Pdb
- Nathan Yergler: In Depth PDB - PyCon 2014

Отладчики на основе pdb:

- Web-PDB
- PuDB
- ipdb
- pdb++

3

II. Повторное использование кода

При написании кода достаточно часто часть действий повторяется. Это может быть небольшой блок на 3-5 строк, а может быть и достаточно большая последовательность действий.

Копировать код — плохая затея. Так как, если потом понадобится обновить одну из копий, надо будет обновлять и другие.

Вместо этого, надо создать специальный блок кода с именем - функцию. И каждый раз, когда код надо повторить, достаточно вызвать функцию. Функция позволяет не только назвать какой-то блок кода, но и сделать его более абстрактным за счет параметров. Параметры дают возможность передавать разные исходные данные для выполнения функции. И, соответственно, получать разный результат, в зависимости от входящих параметров.

Созданию функций посвящён раздел *9. Функции*. Кроме того, в разделе *10. Полезные функции* рассматриваются полезные встроенные функции.

После разделения кода на функции, достаточно быстро наступает момент, когда необходимо использовать функцию в другом скрипте. Конечно же, копирование функции так же неудобно, как и копирование обычного кода. Для повторного использования кода из другого скрипта Python используются модули.

Одиннадцатый раздел *11. Модули* посвящён созданию собственных модулей, а в разделе *12. Полезные модули* рассматриваются полезные модули из стандартной библиотеки Python.

Последний раздел *13. Итераторы, итерируемые объекты и генераторы* этой части посвящён итерируемым объектам, итераторам и генераторам.

9. 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, начинающиеся со знака восклицания (для Cisco)
- удаление (или пропуск) пустых строк
- удаление символов перевода строки в конце строк
- преобразование полученного результата в список

Дальше действия могут отличаться в зависимости от того, что нужно делать.

Часто получается, что есть кусок кода, который повторяется. Конечно, его можно копировать из одного скрипта в другой. Но это очень неудобно, так как при внесении изменений в код нужно будет обновить его во всех файлах, в которые он скопирован.

Гораздо проще и правильней вынести этот код в функцию (это может быть и несколько функций).

И тогда будет производиться вызов этой функции - в этом файле или каком-то другом.

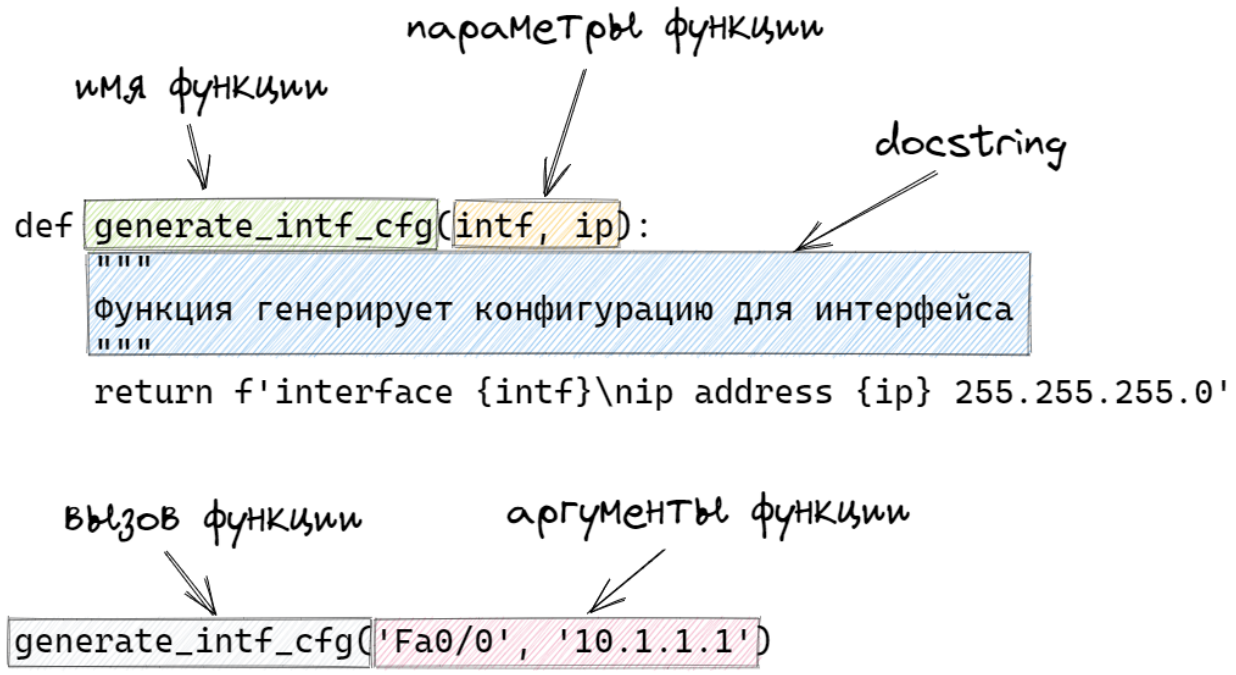
В этом разделе рассматривается ситуация, когда функция находится в том же файле.

А в разделе *11. 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, опционально, 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



Пример функции:

```
def configure_intf(intf_name, ip, mask):
    print('interface', intf_name)
    print('ip address', ip, mask)
```

Функция configure_intf создает конфигурацию интерфейса с указанным именем и IP-адресом. У функции есть три параметра: intf_name, ip, mask. При вызове функции в эти параметры попадут реальные данные.

Примечание: 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.

Примечание: Параметры - это переменные, которые используются при создании функции. Аргументы - это фактические значения (данные), которые передаются функции при вызове.

Функция configure_intf ожидает при вызове три значения, потому что она была создана с тремя параметрами:

```
In [2]: configure_intf('F0/0', '10.1.1.1', '255.255.255.0')
interface F0/0
ip address 10.1.1.1 255.255.255.0

In [3]: configure_intf('Fa0/1', '94.150.197.1', '255.255.255.248')
interface Fa0/1
ip address 94.150.197.1 255.255.255.248
```

Текущий вариант функции configure_intf выводит команды на стандартный поток вывода, команды можно увидеть, но при этом результат функции нельзя сохранить в переменную.

Например, функция sorted не просто выводит результат сортировки на стандартный поток вывода, а **возвращает** его, поэтому его можно сохранить в переменную таким образом:

```
In [4]: items = [40, 2, 0, 22]

In [5]: sorted(items)
Out[5]: [0, 2, 22, 40]

In [6]: sorted_items = sorted(items)

In [7]: sorted_items
Out[7]: [0, 2, 22, 40]
```

Примечание: Обратите внимание на строку `Out[5]` в ipython: таким образом ipython показывает, что функция/метод что-то возвращает и показывает, что именно возвращает.

Если же попытаться записать в переменную результат функции configure_intf, в переменной окажется значение None:

```
In [8]: result = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')
interface Fa0/0
ip address 10.1.1.1 255.255.255.0

In [9]: print(result)
None
```

Чтобы функция могла возвращать какое-то значение, надо использовать оператор `return`.

Оператор return

Оператор **return** используется для возврата какого-то значения, и в то же время он завершает работу функции. Функция может возвращать любой объект Python. По умолчанию, функция всегда возвращает None.

Для того, чтобы функция configure_intf возвращала значение, которое потом можно, например, присвоить переменной, надо использовать оператор `return`:

```
In [10]: def configure_intf(intf_name, ip, mask):
    ...:     config = f'interface {intf_name}\nip address {ip} {mask}'
    ...:     return config
    ...:

In [11]: result = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')

In [12]: print(result)
interface Fa0/0
ip address 10.1.1.1 255.255.255.0

In [13]: result
Out[13]: 'interface Fa0/0\nip address 10.1.1.1 255.255.255.0'
```

Теперь в переменой result находится строка с командами для настройки интерфейса.

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: после return, функция завершает работу, а значит выражения, которые идут после return, не выполняются.

Например, в функции ниже, строка «Конфигурация готова» не будет выводиться, так как она стоит после return:

```
In [14]: def configure_intf(intf_name, ip, mask):
    ...:     config = f'interface {intf_name}\nip address {ip} {mask}'
    ...:     return config
    ...:     print('Конфигурация готова')
    ...:

In [15]: configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')
Out[15]: 'interface Fa0/0\nip address 10.1.1.1 255.255.255.0'
```

Функция может возвращать несколько значений. В этом случае, они пишутся через запятую после оператора return. При этом фактически функция возвращает кортеж:

```
In [16]: def configure_intf(intf_name, ip, mask):
    ...:     config_intf = f'interface {intf_name}\n'
    ...:     config_ip = f'ip address {ip} {mask}'
    ...:     return config_intf, config_ip
    ...:

In [17]: result = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')

In [18]: result
Out[18]: ('interface Fa0/0\n', 'ip address 10.1.1.1 255.255.255.0')

In [19]: type(result)
Out[19]: tuple

In [20]: intf, ip_addr = configure_intf('Fa0/0', '10.1.1.1', '255.255.255.0')

In [21]: intf
Out[21]: 'interface Fa0/0\n'

In [22]: ip_addr
Out[22]: 'ip address 10.1.1.1 255.255.255.0'
```

Документация (docstring)

Первая строка в определении функции - это docstring, строка документации. Это комментарий, который используется как описание функции:

```
In [23]: def configure_intf(intf_name, ip, mask):
    ...:     '''
    ...:     Функция генерирует конфигурацию интерфейса
    ...:     '''
    ...:     config_intf = f'interface {intf_name}\n'
    ...:     config_ip = f'ip address {ip} {mask}'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:     return config_intf, config_ip
    ...:

In [24]: configure_intf?
Signature: configure_intf(intf_name, ip, mask)
Docstring: Функция генерирует конфигурацию интерфейса
File:      ~/repos/pyneng-examples-exercises/examples/06_control_structures/<ipython-
↪input-23-2b2bd970db8f>
Type:      function
```

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.

Пространства имен. Области видимости

Область видимости определяет где переменная доступна. Область видимость переменной зависит от того, где переменная создана.

Чаще всего, речь будет о двух областях видимости:

- глобальной - переменные, которые определены вне функции
- локальной - переменные, которые определены внутри функции

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.

Поиск переменных

При поиске переменных, Python использует правило LEGB. 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) - во встроенной (зарезервированные значения Python)

```
a = 10
b = 5

def sum(x, y):
    result = x + y

    def print_line():
        print("-" * 40)
        print(result)
        print("-" * 40)
    print_line()
    return result
```

Global

Enclosing

Local

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, чтобы они были доступны в другом модуле, их надо импортировать

```
a = 10
b = 5                                          Global

def sum(x, y):
    log_line = f"calling sum({x}, {y})"
    print(log_line)
    return x + y                               Local

def draw_line()
    sym = "-"
    repeat = 40
    return sym * repeat                        Local

result = sum(a, b)
line = draw_line()
```

Пример локальной intf_config:

```
In [1]: def configure_intf(intf_name, ip, mask):
   ...:     intf_config = f'interface {intf_name}\nip address {ip} {mask}'
   ...:     return intf_config
   ...:

In [2]: intf_config
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-5983e972fb1c> in <module>
----> 1 intf_config

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

Обратите внимание, что переменная intf_config недоступна за пределами функции. Для того чтобы получить результат функции, надо вызвать функцию и присвоить результат в переменную:

```
In [3]: result = configure_intf('F0/0', '10.1.1.1', '255.255.255.0')

In [4]: result
Out[4]: 'interface F0/0\nip address 10.1.1.1 255.255.255.0'
```

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.

Параметры бывают обязательные и необязательные.

Обязательные:

```
def f(a, b):
    pass
```

Необязательные (со значением по умолчанию):

```
def f(a=None):
    pass
```

В этом случае a - передавать необязательно.

Аргументы бывают позиционные и ключевые.

```
def summ(a, b):
    return a + b
```

Позиционные:

```
summ(1, 2)
```

Ключевые:

```
summ(a=1, b=2)
```

Независимо от того как параметры созданы, при вызове функции им можно передавать значения и как ключевые и как позиционные аргументы. При этом обязательные параметры надо передать в любом случае, любым способом (позиционными или ключевыми), а необязательные можно передавать, можно нет. Если передавать, то тоже любым способом.

Подробнее типы параметров и аргументов будут рассматриваться позже.

Для того, чтобы функция могла принимать входящие значения, ее нужно создать с параметрами (файл func_check_passwd.py):

```
In [1]: def check_passwd(username, password):
   ...:     if len(password) < 8:
   ...:         print('Пароль слишком короткий')
   ...:         return False
   ...:     elif username in password:
   ...:         print('Пароль содержит имя пользователя')
   ...:         return False
   ...:     else:
   ...:         print(f'Пароль для пользователя {username} прошел все проверки')
   ...:         return True
   ...:
```

В данном случае, у функции два параметра: username и password.

Функция проверяет пароль и возвращает False, если проверки не прошли и True если пароль прошел проверки:

```
In [2]: check_passwd('nata', '12345')
Пароль слишком короткий
Out[2]: False

In [3]: check_passwd('nata', '12345lsdkjflskfdjsnata')
Пароль содержит имя пользователя
Out[3]: False

In [4]: check_passwd('nata', '12345lsdkjflskfdjs')
Пароль для пользователя nata прошел все проверки
Out[4]: True
```

При таком определении функции надо обязательно передать оба аргумента. Если передать только один аргумент, возникнет ошибка:

```
In [5]: check_passwd('nata')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-e07773bb4cc8> in <module>
----> 1 check_passwd('nata')
```

(continues on next page)

(продолжение с предыдущей страницы)

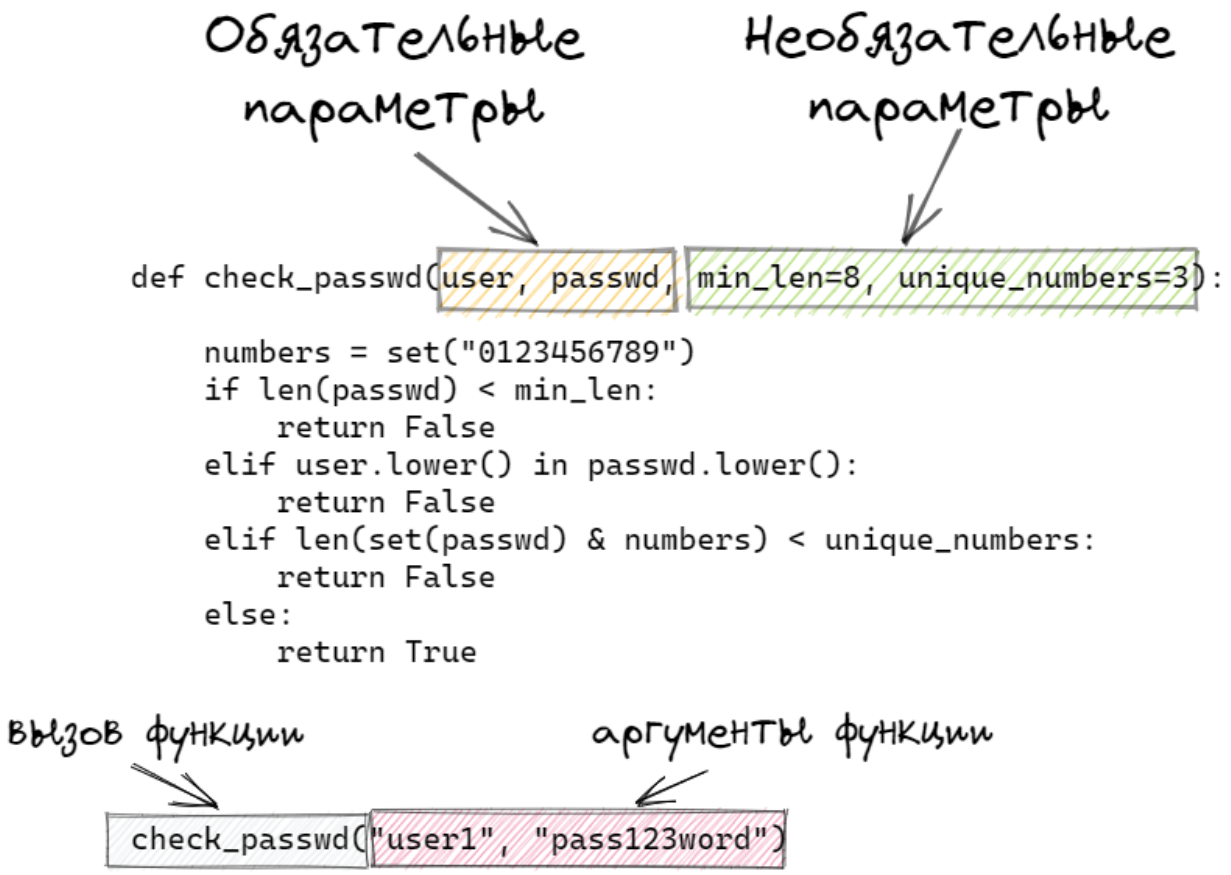
```
TypeError: check_passwd() missing 1 required positional argument: 'password'
```

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

Типы параметров функции

При создании функции можно указать, какие аргументы нужно передавать обязательно, а какие нет. Соответственно, функция может быть создана с:

- **обязательными параметрами**
- **необязательными параметрами** (опциональными, параметрами со значением по умолчанию)



Обязательные параметры

Обязательные параметры - определяют, какие аргументы нужно передать функции обязательно. При этом, их нужно передать ровно столько, сколько указано параметров функции (нельзя указать большее или меньшее количество)

Функция с обязательными параметрами (файл func_params_types.py):

```
def check_passwd(username, password):
    if len(password) < 8:
        print('Пароль слишком короткий')
        return False
    elif username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

Функция check_passwd ожидает два аргумента: username и password.

Функция проверяет пароль и возвращает False, если проверки не прошли и True, если пароль прошел проверки:

```
In [2]: check_passwd('nata', '12345')
Пароль слишком короткий
Out[2]: False

In [3]: check_passwd('nata', '12345lsdkjflskfdjsnata')
Пароль содержит имя пользователя
Out[3]: False

In [4]: check_passwd('nata', '12345lsdkjflskfdjs')
Пароль для пользователя nata прошел все проверки
Out[4]: True
```

Необязательные параметры (параметры со значением по умолчанию)

При создании функции можно указывать значение по умолчанию для параметра таким образом: `def check_passwd(username, password, min_length=8)`. В этом случае, параметр min_length указан со значением по умолчанию и может не передаваться при вызове функции.

Пример функции check_passwd с параметром со значением по умолчанию (файл func_check_passwd_optional_param.py):

```
def check_passwd(username, password, min_length=8):
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

Так как у параметра min_length есть значение по умолчанию, соответствующий аргумент можно не указывать при вызове функции, если значение по умолчанию подходит:

```
In [7]: check_passwd('nata', '12345')
Пароль слишком короткий
Out[7]: False
```

Если нужно поменять значение по умолчанию:

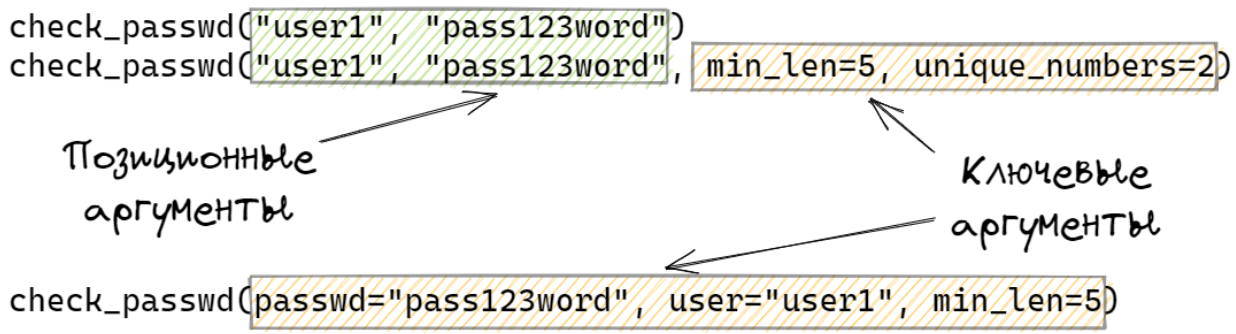
```
In [8]: check_passwd('nata', '12345', 3)
Пароль для пользователя nata прошел все проверки
Out[8]: True
```

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 аргумент
- 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.

```
def check_passwd(user, passwd, min_len=8, unique_numbers=3):
    numbers = set("0123456789")
    if len(passwd) < min_len:
        return False
    elif user.lower() in passwd.lower():
        return False
    elif len(set(passwd) & numbers) < unique_numbers:
        return False
    else:
        return True
```



Позиционные и ключевые аргументы могут быть использоваться одновременно при вызове функции. При этом сначала должны идти позиционные аргументы, а только потом - ключевые.

Посмотрим на разные способы передачи аргументов на примере функции check_passwd (файл func_check_passwd_optional_param.py):

```
def check_passwd(username, password):
    if len(password) < 8:
        print('Пароль слишком короткий')
        return False
    elif username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

Позиционные аргументы

Позиционные аргументы при вызове функции надо передать в правильном порядке (поэтому они и называются позиционные).

```
In [2]: check_passwd('nata', '12345')
Пароль слишком короткий
Out[2]: False
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [3]: check_passwd('nata', '12345lsdkjflskfdjsnata')
Пароль содержит имя пользователя
Out[3]: False

In [4]: check_passwd('nata', '12345lsdkjflskfdjs')
Пароль для пользователя nata прошел все проверки
Out[4]: True
```

Если при вызове функции поменять аргументы местами, скорее всего, возникнет ошибка, в зависимости от конкретной функции.

Ключевые аргументы

Ключевые аргументы:

- передаются с указанием имени аргумента
- за счет этого они могут передаваться в любом порядке

Если передать оба аргумента как ключевые, можно передавать их в любом порядке:

```
In [9]: check_passwd(password='12345', username='nata', min_length=4)
Пароль для пользователя nata прошел все проверки
Out[9]: True
```

> **Предупреждение:** Обратите внимание, что всегда сначала должны идти позиционные аргументы, а затем ключевые.

Если сделать наоборот, возникнет ошибка:

```
In [10]: check_passwd(password='12345', username='nata', 4)
  File "<ipython-input-10-7e8246b6b402>", line 1
    check_passwd(password='12345', username='nata', 4)
                                                    ^
SyntaxError: positional argument follows keyword argument
```

Но в такой комбинации можно:

```
In [11]: check_passwd('nata', '12345', min_length=3)
Пароль для пользователя nata прошел все проверки
Out[11]: True
```

В реальной жизни зачастую намного понятней и удобней указывать флаги (параметры со значениями True/False) или числовые значения как ключевой аргумент. Если задать хорошее название параметра, то по его имени сразу будет понятно, что именно он делает.

Например, можно добавить флаг, который будет контролировать, выполнять проверку наличия имени пользователя в пароле или нет:

```
def check_passwd(username, password, min_length=8, check_username=True):
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif check_username and username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

По умолчанию флаг равен True, а значит проверку выполнять надо:

```
In [14]: check_passwd('nata', '12345nata', min_length=3)
Пароль содержит имя пользователя
Out[14]: False

In [15]: check_passwd('nata', '12345nata', min_length=3, check_username=True)
Пароль содержит имя пользователя
Out[15]: False
```

Если указать значение равным False, проверка не будет выполняться:

```
In [16]: check_passwd('nata', '12345nata', min_length=3, check_username=False)
Пароль для пользователя nata прошел все проверки
Out[16]: True
```

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.

Примечание: Даже если вы не будете использовать этот прием в своих скриптах, есть большая вероятность, что вы встретите его в чужом коде.

Позиционные аргументы переменной длины

Параметр, который принимает позиционные аргументы переменной длины, создается добавлением перед именем параметра звездочки. Имя параметра может быть любым, но по договоренности чаще всего используют имя *args

Пример функции:

```
In [1]: def sum_arg(a, *args):
  ....:     print(a, args)
  ....:     return a + sum(args)
  ....:
```

Функция sum_arg создана с двумя параметрами:

- параметр a
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен
- параметр *args - ожидает аргументы переменной длины
  - сюда попадут все остальные аргументы в виде кортежа
  - эти аргументы могут отсутствовать

Вызов функции с разным количеством аргументов:

```
In [2]: sum_arg(1, 10, 20, 30)
1 (10, 20, 30)
Out[2]: 61

In [3]: sum_arg(1, 10)
1 (10,)
Out[3]: 11

In [4]: sum_arg(1)
1 ()
Out[4]: 1
```

Можно создать и такую функцию:

```
In [5]: def sum_arg(*args):
  ....:     print(args)
  ....:     return sum(args)
  ....:

In [6]: sum_arg(1, 10, 20, 30)
(1, 10, 20, 30)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[6]: 61

In [7]: sum_arg()
()
Out[7]: 0
```

Ключевые аргументы переменной длины

Параметр, который принимает ключевые аргументы переменной длины, создается добавлением перед именем параметра двух звездочек. Имя параметра может быть любым, но, по договоренности, чаще всего, используют имя **kwargs (от keyword arguments).

Пример функции:

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

Функция sum_arg создана с двумя параметрами:

- параметр a
  - если передается как позиционный аргумент, должен идти первым
  - если передается как ключевой аргумент, то порядок не важен
- параметр **kwargs - ожидает ключевые аргументы переменной длины
  - сюда попадут все остальные ключевые аргументы в виде словаря
  - эти аргументы могут отсутствовать

Вызов функции с разным количеством ключевых аргументов:

```
In [9]: sum_arg(a=10, b=10, c=20, d=30)
10 {'c': 20, 'b': 10, 'd': 30}
Out[9]: 70

In [10]: sum_arg(b=10, c=20, d=30, a=10)
10 {'c': 20, 'b': 10, 'd': 30}
Out[10]: 70
```

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.

В реальности, как правило, данные необходимо передавать в функцию программно. И часто данные идут в виде какого-то объекта Python.

Распаковка позиционных аргументов

Например, при форматировании строк часто надо передать методу format несколько аргументов. И часто эти аргументы уже находятся в списке или кортеже. Чтобы их передать методу format, приходится использовать индексы таким образом:

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

In [2]: print('One: {}, Two: {}, Three: {}'.format(items[0], items[1], items[2]))
One: 1, Two: 2, Three: 3
```

Вместо этого, можно воспользоваться распаковкой аргументов и сделать так:

```
In [4]: items = [1,2,3]

In [5]: print('One: {}, Two: {}, Three: {}'.format(*items))
One: 1, Two: 2, Three: 3
```

Еще один пример - функция config_interface (файл func_config_interface_unpacking.py):

```
In [8]: def config_interface(intf_name, ip_address, mask):
    ..:     interface = f'interface {intf_name}'
    ..:     no_shut = 'no shutdown'
    ..:     ip_addr = f'ip address {ip_address} {mask}'
    ..:     result = [interface, no_shut, ip_addr]
    ..:     return result
    ..:
```

Функция ожидает такие аргументы:

- intf_name - имя интерфейса
- ip_address - IP-адрес
- mask - маска

Функция возвращает список строк для настройки интерфейса:

```
In [9]: config_interface('Fa0/1', '10.0.1.1', '255.255.255.0')
Out[9]: ['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0']

In [11]: config_interface('Fa0/10', '10.255.4.1', '255.255.255.0')
Out[11]: ['interface Fa0/10', 'no shutdown', 'ip address 10.255.4.1 255.255.255.0']
```

Допустим, нужно вызвать функцию и передать ей информацию, которая была получена из другого источника, к примеру, из БД.

Например, список interfaces_info, в котором находятся параметры для настройки интерфейсов:

```
In [14]: interfaces_info = [['Fa0/1', '10.0.1.1', '255.255.255.0'],
    ...:                    ['Fa0/2', '10.0.2.1', '255.255.255.0'],
    ...:                    ['Fa0/3', '10.0.3.1', '255.255.255.0'],
    ...:                    ['Fa0/4', '10.0.4.1', '255.255.255.0'],
    ...:                    ['Lo0', '10.0.0.1', '255.255.255.255']]
    ...:
```

Если пройтись по списку в цикле и передавать вложенный список как аргумент функции, возникнет ошибка:

```
In [15]: for info in interfaces_info:
    ...:     print(config_interface(info))
    ...:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-d34ced60c796> in <module>
      1 for info in interfaces_info:
----> 2     print(config_interface(info))
      3

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

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 [16]: for info in interfaces_info:
    ...:     print(config_interface(*info))
    ...:
['interface Fa0/1', 'no shutdown', 'ip address 10.0.1.1 255.255.255.0']
['interface Fa0/2', 'no shutdown', 'ip address 10.0.2.1 255.255.255.0']
['interface Fa0/3', 'no shutdown', 'ip address 10.0.3.1 255.255.255.0']
['interface Fa0/4', 'no shutdown', 'ip address 10.0.4.1 255.255.255.0']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
['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.

Примечание: Таким же образом можно распаковывать и кортеж.

Распаковка ключевых аргументов

Аналогичным образом можно распаковывать словарь, чтобы передать его как ключевые аргументы.

Функция check_passwd (файл func_check_passwd_optional_param_2.py):

```
In [19]: def check_passwd(username, password, min_length=8, check_username=True):
    ...:     if len(password) < min_length:
    ...:         print('Пароль слишком короткий')
    ...:         return False
    ...:     elif check_username and username in password:
    ...:         print('Пароль содержит имя пользователя')
    ...:         return False
    ...:     else:
    ...:         print(f'Пароль для пользователя {username} прошел все проверки')
    ...:         return True
    ...:
```

Список словарей username_passwd, в которых указано имя пользователя и пароль:

```
In [20]: username_passwd = [{'username': 'cisco', 'password': 'cisco'},
    ...:                    {'username': 'nata', 'password': 'natapass'},
    ...:                    {'username': 'user', 'password': '123456789'}]
```

Если передать словарь функции check_passwd, возникнет ошибка:

```
In [21]: for data in username_passwd:
    ...:     check_passwd(data)
    ...:
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-21-ad848f85c77f> in <module>
      1 for data in username_passwd:
----> 2     check_passwd(data)
      3

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

Ошибка такая, так как функция восприняла словарь как один аргумент и считает что ей не хватает только аргумента password.

Если добавить ** перед передачей словаря функции, функция нормально отработает:

```
In [22]: for data in username_passwd:
    ...:     check_passwd(**data)
    ...:
Пароль слишком короткий
Пароль содержит имя пользователя
Пароль для пользователя user прошел все проверки

In [23]: for data in username_passwd:
    ...:     print(data)
    ...:     check_passwd(**data)
    ...:
{'username': 'cisco', 'password': 'cisco'}
Пароль слишком короткий
{'username': 'nata', 'password': 'natapass'}
Пароль содержит имя пользователя
{'username': 'user', 'password': '123456789'}
Пароль для пользователя user прошел все проверки
```

Python распаковывает словарь и передает его в функцию как ключевые аргументы. Запись `check_passwd(**data)` превращается в вызов вида `check_passwd(username='cisco', password='cisco')`.

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 check_passwd (файл func_add_user_kwargs_example.py):

```
In [1]: def check_passwd(username, password, min_length=8, check_username=True):
   ...:     if len(password) < min_length:
   ...:         print('Пароль слишком короткий')
   ...:         return False
   ...:     elif check_username and username in password:
   ...:         print('Пароль содержит имя пользователя')
   ...:         return False
   ...:     else:
   ...:         print(f'Пароль для пользователя {username} прошел все проверки')
   ...:         return True
   ...:
```

Функция проверяет пароль и возвращает True, если пароль прошел проверки и False - если нет.

Вызов функции в ipython:

```
In [3]: check_passwd('nata', '12345', min_length=3)
Пароль для пользователя nata прошел все проверки
Out[3]: True

In [4]: check_passwd('nata', '12345nata', min_length=3)
Пароль содержит имя пользователя
Out[4]: False

In [5]: check_passwd('nata', '12345nata', min_length=3, check_username=False)
Пароль для пользователя nata прошел все проверки
Out[5]: True

In [6]: check_passwd('nata', '12345nata', min_length=3, check_username=True)
Пароль содержит имя пользователя
Out[6]: False
```

Сделаем функцию add_user_to_users_file, которая запрашивает пароль для указанного пользователя, проверяет его и запрашивает заново, если пароль не прошел проверки или записывает пользователя и пароль в файл, если пароль прошел проверки.

```
In [7]: def add_user_to_users_file(user, users_filename='users.txt'):
   ...:     while True:
   ...:         passwd = input(f'Введите пароль для пользователя {user}: ')
   ...:         if check_passwd(user, passwd):
   ...:             break
   ...:     with open(users_filename, 'a') as f:
   ...:         f.write(f'{user},{passwd}\n')
   ...:

In [8]: add_user_to_users_file('nata')
Введите пароль для пользователя nata: natasda
Пароль слишком короткий
Введите пароль для пользователя nata: natasdlajsl;fjd
Пароль содержит имя пользователя
Введите пароль для пользователя nata: salkfdjsalkdjfsal;dfj
Пароль для пользователя nata прошел все проверки

In [9]: cat users.txt
nata,salkfdjsalkdjfsal;dfj
```

В таком варианте функции add_user_to_users_file нет возможности регулировать минимальную длину пароля и то надо ли проверять наличие имени пользователя в пароле. В следующем варианте функции add_user_to_users_file эти возможности добавлены:

```
In [5]: def add_user_to_users_file(user, users_filename='users.txt', min_length=8, check_
↪username=True):
   ...:     while True:
   ...:         passwd = input(f'Введите пароль для пользователя {user}: ')
   ...:         if check_passwd(user, passwd, min_length, check_username):
   ...:             break
   ...:     with open(users_filename, 'a') as f:
   ...:         f.write(f'{user},{passwd}\n')
   ...:

In [6]: add_user_to_users_file('nata', min_length=5)
Введите пароль для пользователя nata: natas2342
Пароль содержит имя пользователя
Введите пароль для пользователя nata: dlfjgkd
Пароль для пользователя nata прошел все проверки
```

Теперь при вызове функции можно указать параметр min_length или check_username. Однако, пришлось повторить параметры функции check_passwd в определении функции add_user_to_users_file. Это не очень хорошо и, когда параметров много, просто неудобно, особенно если учитывать, что у функции check_passwd могут добавиться другие параметры.

Такая ситуация случается довольно часто и в Python есть распространенное решение этой задачи: все аргументы для внутренней функции (в этом случае это check_passwd) будут приниматься в **kwargs. Затем, при вызове функции check_passwd они будут распаковываться в ключевые аргументы тем же синтаксисом **kwargs.

```
In [7]: def add_user_to_users_file(user, users_filename='users.txt', **kwargs):
   ...:     while True:
   ...:         passwd = input(f'Введите пароль для пользователя {user}: ')
   ...:         if check_passwd(user, passwd, **kwargs):
   ...:             break
   ...:     with open(users_filename, 'a') as f:
   ...:         f.write(f'{user},{passwd}\n')
   ...:

In [8]: add_user_to_users_file('nata', min_length=5)
Введите пароль для пользователя nata: alskfdjlksadjf
Пароль для пользователя nata прошел все проверки

In [9]: add_user_to_users_file('nata', min_length=5)
Введите пароль для пользователя nata: 345
Пароль слишком короткий
Введите пароль для пользователя nata: 309487538
Пароль для пользователя nata прошел все проверки
```

В таком варианте в функцию check_passwd можно добавлять аргументы без необходимости дублировать их в функции add_user_to_users_file.

Аргументы, которые можно передавать только как ключевые

Аргументы, которые указаны после * можно передавать только как ключевые при вызове функции.

Примечание: Этот функционал доступен в любой версии Python3.

Например, в этой функции аргументы min_length и check_username можно передавать только как ключевые.

```
def check_passwd(username, password, *, min_length=8, check_username=True):
    if len(password) < min_length:
        print('Пароль слишком короткий')
        return False
    elif check_username and username in password:
        print('Пароль содержит имя пользователя')
        return False
    else:
        print(f'Пароль для пользователя {username} прошел все проверки')
        return True
```

При передаче их как позиционных, возникнет исключение:

```
In [2]: check_passwd('nata', '12345', min_length=3)
Пароль для пользователя nata прошел все проверки
Out[2]: True

In [3]: check_passwd('nata', '12345', 3)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-4f346c9cf914> in <module>
----> 1 check_passwd('nata', '12345', 3)

TypeError: check_passwd() takes 2 positional arguments but 3 were given
```

Распространенные проблемы/нюансы работы с функциями

Список/словарь в который собираются данные в функции, создан за пределами функции

Очень часто в решении заданий встречается такой нюанс: функция должна собрать какие-то данные в список/словарь и список создан вне функции. Тогда вроде как функция работает правильно, но при этом тест не проходит. Это происходит потому что в таком варианте функция работает неправильно и каждый вызов добавляет элементы в тот же список:

```
In [1]: result = []

In [2]: def func(items):
   ...:     for i in items:
   ...:         result.append(i*100)
   ...:     return result
   ...:

In [3]: func([1, 2, 3])
Out[3]: [100, 200, 300]

In [4]: func([7, 8])
Out[4]: [100, 200, 300, 700, 800]
```

Исправить это можно переносом строки создания списка в функцию:

```
In [20]: def func(items):
    ...:     result = []
    ...:     for i in items:
    ...:         result.append(i*100)
    ...:     return result
    ...:

In [21]: func([1, 2, 3])
Out[21]: [100, 200, 300]

In [22]: func([7, 8])
Out[22]: [700, 800]
```

Всё, что относится к функции лучше всегда писать внутри функции. Тест тут не проходит потому что внутри файла задания функция вызывается первый раз - всё правильно, а потом тест вызывает её второй раз и там вдруг в два раза больше данных чем нужно.

Значения по умолчанию в параметрах создаются во время создания функции

Пример функции, которая должна выводить текущую дату и время при каждом вызове:

```
In [1]: from datetime import datetime

In [2]: import time

In [3]: def print_current_datetime(ptime=datetime.now()):
   ...:     print(f">>> {ptime}")
   ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [4]: for i in range(3):
   ...:     print("Имитируем долгое выполнение...")
   ...:     time.sleep(1)
   ...:     print_current_datetime()
   ...:
Имитируем долгое выполнение...
>>> 2021-02-23 09:01:49.845425
Имитируем долгое выполнение...
>>> 2021-02-23 09:01:49.845425
Имитируем долгое выполнение...
>>> 2021-02-23 09:01:49.845425
```

Так как `datetime.now()` указано в значении по умолчанию, это значение создается во время создания функции и в итоге при каждом вызове время одно и то же. Для корректного вывода, надо вызывать `datetime.now()` в теле функции:

```
In [5]: def print_current_datetime():
   ...:     print(f">>> {datetime.now()}")
   ...:
```

Второй пример где этот нюанс может привести к неожиданным результатам, если о нем не знать - изменяемые типы данных в значении по умолчанию.

Например, использование списка в значении по умолчанию:

```
In [15]: def add_item(item, data=[]):
    ...:     data.append(item)
    ...:     return data
    ...:
```

В этом случае список data создается один раз - при создании функции и при вызове функции, данные добавляются в один и тот же список. В итоге все повторные вызовы будут добавлять элементы:

```
In [16]: add_item(1)
Out[16]: [1]

In [17]: add_item(2)
Out[17]: [1, 2]

In [18]: add_item(4)
Out[18]: [1, 2, 4]
```

Если нужно сделать так, чтобы параметр data был необязательным и по умолчанию создавался пустой список, надо сделать так:

```
In [22]: def add_item(item, data=None):
    ...:     if data is None:
    ...:         data = []
    ...:     data.append(item)
    ...:     return data
    ...:

In [23]: add_item(1)
Out[23]: [1]

In [24]: add_item(2)
Out[24]: [2]
```

Ошибка UnboundLocalError: local variable referenced before assignment

Ошибка может возникнуть в таких случаях:

- обращение к переменной в функции идет до ее создания - это может быть случайность (ошибка) или следствие того, что какое-то условие не выполнилось
- обращение внутри функции к глобальной переменной, но при этом внутри функции создана такая же переменная позже

Первый случай - обратились к переменной до ее создания:

```
def f():
    print(b)
    b = 55

In [6]: f()
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [6], in <cell line: 1>()
----> 1 f()

Input In [5], in f()
      1 def f():
----> 2     print(b)
      3     b = 55

UnboundLocalError: local variable 'b' referenced before assignment
```

Переменная создается в условии, а условие не выполнилось:

```
def f():
    if 5 > 8:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        b = 55
    print(b)

In [8]: f()
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [8], in <cell line: 1>()
----> 1 f()

Input In [7], in f()
      2 if 5 > 8:
      3     b = 55
----> 4 print(b)

UnboundLocalError: local variable 'b' referenced before assignment
```

Имя глобальной и локальной переменной одинаковое и внутри функции сначала идет попытка обращения к глобальной, потом создание локальной:

```
a = 10

def f():
    print(a)
    a = 55
    print(a)


In [4]: f()
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 f()

Input In [3], in f()
      1 def f():
----> 2     print(a)
      3     a = 55
      4     print(a)

UnboundLocalError: local variable 'a' referenced before assignment
```

Python должен определить область видимости переменной. И в случае функций, Python считает, что переменная локальная, если она создана внутри функции. На момент когда мы доходим до вызова функции, Python уже решил, что a это локальная переменная и именно из-за этого первая строка `print(a)` дает ошибку.

Дополнительные материалы

Документация:

- Defining Functions
- Built-in Functions
- Sorting HOW TO
- Functional Programming HOWTO
- Функция range
- Подробнее о функциях в Python

Статьи:

- Asterisks in Python: what they are and how to use them

Docstring

- What are the most common Python docstring formats?
- Docstring Formats for Python
- Google style guide. Docstring
- PEP 8 docstring
- PEP 257 – Docstring Conventions
- Scrapli functions docstring example

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

> **Предупреждение:** Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 9.1

Создать функцию, которая генерирует конфигурацию для access-портов.

Функция ожидает такие аргументы:

1. словарь с соответствием интерфейс-VLAN такого вида:

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

2. шаблон конфигурации access-портов в виде списка команд (список access_mode_template)

Функция должна возвращать список всех портов в режиме access с конфигурацией на основе шаблона access_mode_template. В конце строк в списке не должно быть символа перевода строки.

В этом задании заготовка для функции уже сделана и надо только продолжить писать само тело функции.

Пример итогового списка (перевод строки после каждого элемента сделан для удобства чтения):

```
[
"interface FastEthernet0/12",
"switchport mode access",
"switchport access vlan 10",
"switchport nonegotiate",
"spanning-tree portfast",
"spanning-tree bpduguard enable",
"interface FastEthernet0/17",
"switchport mode access",
"switchport access vlan 150",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
"switchport nonegotiate",
"spanning-tree portfast",
"spanning-tree bpduguard enable",
...]
```

Проверить работу функции на примере словаря access_config и списка команд access_mode_template. Если предыдущая проверка прошла успешно, проверить работу функции еще раз на словаре access_config_2 и убедиться, что в итоговом списке правильные номера интерфейсов и вланов.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
access_mode_template = [
    "switchport mode access", "switchport access vlan",
    "switchport nonegotiate", "spanning-tree portfast",
    "spanning-tree bpduguard enable"
]

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

access_config_2 = {
    "FastEthernet0/03": 100,
    "FastEthernet0/07": 101,
    "FastEthernet0/09": 107,
}

def generate_access_config(intf_vlan_mapping, access_template):
    """
    intf_vlan_mapping - словарь с соответствием интерфейс-VLAN такого вида:
        {"FastEthernet0/12": 10,
         "FastEthernet0/14": 11,
         "FastEthernet0/16": 17}
    access_template - список команд для порта в режиме access

    Возвращает список всех портов в режиме access с конфигурацией на основе шаблона
    """
```

Задание 9.1a

Сделать копию функции generate_access_config из задания 9.1.

Дополнить скрипт: ввести дополнительный параметр, который контролирует будет ли настроен port-security:

- имя параметра «psecurity»
- по умолчанию значение None
- для настройки port-security, как значение надо передать список команд port-security (находятся в списке port_security_template)

Функция должна возвращать список всех портов в режиме access с конфигурацией на основе шаблона access_mode_template и шаблона port_security_template, если он был передан. В конце строк в списке не должно быть символа перевода строки.

Проверить работу функции на примере словаря access_config, с генерацией конфигурации port-security и без.

Пример вызова функции:

```
print(generate_access_config(access_config, access_mode_template))
print(generate_access_config(access_config, access_mode_template, port_security_template))
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
access_mode_template = [
    "switchport mode access", "switchport access vlan",
    "switchport nonegotiate", "spanning-tree portfast",
    "spanning-tree bpduguard enable"
]

port_security_template = [
    "switchport port-security maximum 2",
    "switchport port-security violation restrict",
    "switchport port-security"
]

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

Задание 9.2

Создать функцию generate_trunk_config, которая генерирует конфигурацию для trunk-портов.

У функции должны быть такие параметры:

1. intf_vlan_mapping: ожидает как аргумент словарь с соответствием интерфейс-VLANы такого вида:

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

2. trunk_template: ожидает как аргумент шаблон конфигурации trunk-портов в виде списка команд (список trunk_mode_template)

Функция должна возвращать список команд с конфигурацией на основе указанных портов и шаблона trunk_mode_template. В конце строк в списке не должно быть символа перевода строки.

Проверить работу функции на примере словаря trunk_config и списка команд trunk_mode_template. Если эта проверка прошла успешно, проверить работу функции еще раз на словаре trunk_config_2 и убедится, что в итоговом списке правильные номера интерфейсов и вланов.

Пример итогового списка (перевод строки после каждого элемента сделан для удобства чтения):

```
[
"interface FastEthernet0/1",
"switchport mode trunk",
"switchport trunk native vlan 999",
"switchport trunk allowed vlan 10,20,30",
"interface FastEthernet0/2",
"switchport mode trunk",
"switchport trunk native vlan 999",
"switchport trunk allowed vlan 11,30",
...]
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

trunk_config = {
    "FastEthernet0/1": [10, 20, 30],
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Задание 9.2a

Сделать копию функции generate_trunk_config из задания 9.2

Изменить функцию таким образом, чтобы она 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_config и шаблона trunk_mode_template.

Ограничение: Все задания надо выполнять используя только пройденные темы.

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

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

Задание 9.3

Создать функцию get_int_vlan_map, которая обрабатывает конфигурационный файл коммутатора и возвращает кортеж из двух словарей:

1. словарь портов в режиме access, где ключи номера портов, а значения access VLAN (числа):

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

2. словарь портов в режиме trunk, где ключи номера портов, а значения список разрешенных VLAN (список чисел):

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

У функции должен быть один параметр config_filename, который ожидает как аргумент имя конфигурационного файла.

Проверить работу функции на примере файла config_sw1.txt

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 9.3a

Сделать копию функции get_int_vlan_map из задания 9.3.

Дополнить функцию: 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

```
{"FastEthernet0/12": 10,
 "FastEthernet0/14": 11,
 "FastEthernet0/20": 1}
```

У функции должен быть один параметр config_filename, который ожидает как аргумент имя конфигурационного файла.

Проверить работу функции на примере файла config_sw2.txt

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 9.4

Создать функцию convert_config_to_dict, 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 в начале строки надо удалить).
- Если у команды верхнего уровня нет подкоманд, то значение будет пустым списком

У функции должен быть один параметр config_filename, который ожидает как аргумент имя конфигурационного файла.

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.

Проверить работу функции на примере файла config_sw1.txt

Часть словаря, который должна возвращать функция (полный вывод можно посмотреть в тесте test_task_9_4.py):

```
{
    "version 15.0": [],
    "service timestamps debug datetime msec": [],
    "service timestamps log datetime msec": [],
    "no service password-encryption": [],
    "hostname sw1": [],
    "interface FastEthernet0/0": [
        "switchport mode access",
        "switchport access vlan 10",
    ],
    "interface FastEthernet0/1": [
        "switchport trunk encapsulation dot1q",
        "switchport trunk allowed vlan 100,200",
        "switchport mode trunk",
    ],
    "interface FastEthernet0/2": [
        "switchport mode access",
        "switchport access vlan 20",
    ],
}
```

Ограничение: Все задания надо выполнять используя только пройденные темы.

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


def ignore_command(command, ignore):
    """
    Функция проверяет содержится ли в команде слово из списка ignore.

    command - строка. Команда, которую надо проверить
    ignore - список. Список слов

    Возвращает
    * True, если в команде содержится слово из списка ignore
    * False - если нет
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    """
    ignore_status = False
    for word in ignore:
        if word in command:
            ignore_status = True
    return ignore_status
```

10. Полезные функции

В этом разделе рассматриваются такие функции:

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

Функция print

Функция `print` уже не раз использовалась в книге, но до сих пор не рассматривался ее полный синтаксис:

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

Функция print выводит все элементы, разделяя их значением sep, и завершает вывод значением end.

Все элементы, которые передаются как аргументы, конвертируются в строки:

```
In [4]: def f(a):
   ...:     return a
   ...:

In [5]: print(1, 2, f, range(10))
1 2 <function f at 0xb4de926c> range(0, 10)
```

Для функций f и range результат равнозначен применению str():

```
In [6]: str(f)
Out[6]: '<function f at 0xb4de926c>'

In [7]: str(range(10))
Out[7]: 'range(0, 10)'
```

sep

Параметр sep контролирует то, какой разделитель будет использоваться между элементами.

По умолчанию используется пробел:

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

Можно изменить значение sep на любую другую строку:

```
In [9]: print(1, 2, 3, sep='|')
1|2|3

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

In [11]: print(1, 2, 3, sep=f"\n{'-' * 10}\n")
1
----------
2
----------
3
```

Примечание: Обратите внимание на то, что все аргументы, которые управляют поведением функции print, надо передавать как ключевые, а не позиционные.

В некоторых ситуациях функция print может заменить метод join:

```
In [12]: items = [1, 2, 3, 4, 5]

In [13]: print(*items, sep=', ')
1, 2, 3, 4, 5
```

end

Параметр end контролирует то, какое значение выведется после вывода всех элементов. По умолчанию используется перевод строки:

```
In [19]: print(1, 2, 3)
1 2 3
```

Можно изменить значение end на любую другую строку:

```
In [20]: print(1, 2, 3, end='\n'+'-'*10)
1 2 3
----------
```

file

Параметр file контролирует то, куда выводятся значения функции print. По умолчанию все выводится на стандартный поток вывода - sys.stdout.

Python позволяет передавать file как аргумент любой объект с методом write(string). За счет этого с помощью print можно записывать строки в файл:

```
In [1]: f = open('result.txt', 'w')

In [2]: for num in range(10):
   ...:     print(f'Item {num}', file=f)
   ...:

In [3]: f.close()

In [4]: cat result.txt
Item 0
Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7
Item 8
Item 9
```

flush

По умолчанию при записи в файл или выводе на стандартный поток вывода вывод буферизируется. Параметр flush позволяет отключать буферизацию.

Пример скрипта, который выводит число от 0 до 10 каждую секунду (файл print_nums.py):

```
import time

for num in range(10):
    print(num)
    time.sleep(1)
```

Попробуйте запустить скрипт и убедиться, что числа выводятся раз в секунду.

Теперь, аналогичный скрипт, но числа будут выводиться в одной строке (файл print_nums_oneline.py):

```
import time

for num in range(10):
    print(num, end=' ')
    time.sleep(1)
```

Попробуйте запустить функцию. Числа не выводятся по одному в секунду, а выводятся все через 10 секунд.

Это связано с тем, что при выводе на стандартный поток вывода flush выполняется после перевода строки.

Чтобы скрипт отрабатывал как нужно, необходимо установить flush равным True (файл print_nums_oneline_fixed.py):

```
import time

for num in range(10):
    print(num, end=' ', flush=True)
    time.sleep(1)
```

Функция range

Функция range возвращает неизменяемую последовательность чисел в виде объекта range.

Синтаксис функции:

```
range(stop)
range(start, stop[, step])
```

Параметры функции:

- **start** - с какого числа начинается последовательность. По умолчанию - 0
- **stop** - до какого числа продолжается последовательность чисел. Указанное число не включается в диапазон
- **step** - с каким шагом растут числа. По умолчанию 1

Функция range хранит только информацию о значениях start, stop и step и вычисляет значения по мере необходимости. Это значит, что независимо от размера диапазона, который описывает функция range, она всегда будет занимать фиксированный объем памяти.

Самый простой вариант range - передать только значение stop:

```
In [1]: range(5)
Out[1]: range(0, 5)

In [2]: list(range(5))
Out[2]: [0, 1, 2, 3, 4]
```

Если передаются два аргумента, то первый используется как start, а второй - как stop:

```
In [3]: list(range(1, 5))
Out[3]: [1, 2, 3, 4]
```

И чтобы указать шаг последовательности надо передать три аргумента:

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

In [5]: list(range(0, 10, 3))
Out[5]: [0, 3, 6, 9]
```

С помощью range можно генерировать и убывающие последовательности чисел:

```
In [6]: list(range(10, 0, -1))
Out[6]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

In [7]: list(range(5, -1, -1))
Out[7]: [5, 4, 3, 2, 1, 0]
```

Для получения убывающей последовательности надо использовать отрицательный шаг и соответственно указать start - большим числом, а stop - меньшим.

В убывающей последовательности шаг тоже может быть разным:

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

Функция поддерживает отрицательные значения start и stop:

```
In [9]: list(range(-10, 0, 1))
Out[9]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]

In [10]: list(range(0, -10, -1))
Out[10]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Объект range поддерживает все операции, которые поддерживают последовательности в Python, кроме сложения и умножения.

Проверка, входит ли число в диапазон, который описывает range:

```
In [11]: nums = range(5)

In [12]: nums
Out[12]: range(0, 5)

In [13]: 3 in nums
Out[13]: True

In [14]: 7 in nums
Out[14]: False
```

Примечание: Начиная с версии Python 3.2, эта проверка выполняется за постоянное время (O(1)).

Можно получить конкретный элемент диапазона:

```
In [15]: nums = range(5)

In [16]: nums[0]
Out[16]: 0

In [17]: nums[-1]
Out[17]: 4
```

Range поддерживает срезы:

```
In [18]: nums = range(5)

In [19]: nums[1:]
Out[19]: range(1, 5)

In [20]: nums[:3]
Out[20]: range(0, 3)
```

Можно получить длину диапазона:

```
In [21]: nums = range(5)

In [22]: len(nums)
Out[22]: 5
```

А также минимальный и максимальный элемент:

```
In [23]: nums = range(5)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [24]: min(nums)
Out[24]: 0

In [25]: max(nums)
Out[25]: 4
```

Кроме того, объект range поддерживает метод index:

```
In [26]: nums = range(1, 7)

In [27]: nums.index(3)
Out[27]: 2
```

Функция sorted

Функция `sorted` возвращает новый отсортированный список, который получен из итерируемого объекта, который был передан как аргумент. Функция также поддерживает дополнительные параметры, которые позволяют управлять сортировкой.

Первый аспект, на который важно обратить внимание - sorted всегда возвращает список.

Если сортировать список элементов, то возвращается новый список:

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [2]: sorted(list_of_words)
Out[2]: ['', 'dict', 'list', 'one', 'two']
```

При сортировке кортежа также возвращается список:

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

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

Сортировка множества:

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

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

Сортировка строки:

```
In [7]: string_to_sort = 'long string'

In [8]: sorted(string_to_sort)
Out[8]: [' ', 'g', 'g', 'i', 'l', 'n', 'n', 'o', 'r', 's', 't']
```

Если передать sorted словарь, функция вернет отсортированный список ключей:

```
In [9]: dict_for_sort = {
   ...:         'id': 1,
   ...:         'name': 'London',
   ...:         'IT_VLAN': 320,
   ...:         'User_VLAN': 1010,
   ...:         'Mngmt_VLAN': 99,
   ...:         'to_name': None,
   ...:         'to_id': None,
   ...:         'port': 'G1/0/11'
   ...: }

In [10]: sorted(dict_for_sort)
Out[10]:
['IT_VLAN',
 'Mngmt_VLAN',
 'User_VLAN',
 'id',
 'name',
 'port',
 'to_id',
 'to_name']
```

reverse

Флаг reverse позволяет управлять порядком сортировки. По умолчанию сортировка будет по возрастанию элементов.

Указав флаг reverse, можно поменять порядок:

```
In [11]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [12]: sorted(list_of_words)
Out[12]: ['', 'dict', 'list', 'one', 'two']

In [13]: sorted(list_of_words, reverse=True)
Out[13]: ['two', 'one', 'list', 'dict', '']
```

key

С помощью параметра key можно указывать, как именно выполнять сортировку. Параметр key ожидает функцию, с помощью которой должно быть выполнено сравнение.

Например, таким образом можно отсортировать список строк по длине строки:

```
In [14]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [15]: sorted(list_of_words, key=len)
Out[15]: ['', 'one', 'two', 'list', 'dict']
```

Если нужно отсортировать ключи словаря, но при этом игнорировать регистр строк:

```
In [16]: dict_for_sort = {
    ...:         'id': 1,
    ...:         'name':'London',
    ...:         'IT_VLAN':320,
    ...:         'User_VLAN':1010,
    ...:         'Mngmt_VLAN':99,
    ...:         'to_name': None,
    ...:         'to_id': None,
    ...:         'port':'G1/0/11'
    ...: }

In [17]: sorted(dict_for_sort, key=str.lower)
Out[17]:
['id',
 'IT_VLAN',
 'Mngmt_VLAN',
 'name',
 'port',
 'to_id',
 'to_name',
 'User_VLAN']
```

Параметру key можно передавать любые функции, не только встроенные. Также тут удобно использовать анонимную функцию lambda.

С помощью параметра key можно сортировать объекты не по первому элементу, а по любому другому. Но для этого надо использовать или функцию lambda, или специальные функции из модуля operator.

Например, чтобы отсортировать список кортежей из двух элементов по второму элементу, надо использовать такой прием:

```
In [18]: from operator import itemgetter
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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)]
```

Пример сортировки разных объектов

Сортировка выполняется по первому элементу, например, по первому символу в списке строк, если он одинаковый, по второму и так далее. Сортировка выполняется по коду Unicode символа. Для символов из одного алфавита, это значит что сортировка по сути будет по алфавиту.

Пример сортировки списка строк:

```
In [6]: data = ["test1", "test2", "text1", "text2"]

In [7]: sorted(data)
Out[7]: ['test1', 'test2', 'text1', 'text2']
```

Некоторые данные будут сортироваться неправильно, например, список IP-адресов:

```
In [11]: ip_list = ["10.1.1.1", "10.1.10.1", "10.1.2.1", "10.1.11.1"]

In [12]: sorted(ip_list)
Out[12]: ['10.1.1.1', '10.1.10.1', '10.1.11.1', '10.1.2.1']
```

Это происходит потому используется лексикографическая сортировка. Чтобы в данном случае сортировка была нормальной, надо или использовать отдельный модуль с натуральной сортировкой (модуль natsort) или сортировать, например, по двоичному/десятичному значению адреса.

Пример сортировки IP-адресов по двоичному значению. Сначала создаем функцию, которая преобразует IP-адреса в двоичный формат:

```
In [15]: def bin_ip(ip):
    ...:     octets = [int(o) for o in ip.split(".")]
    ...:     return ("{:08b}"*4).format(*octets)
    ...:

In [16]: bin_ip("10.1.1.1")
Out[16]: '00001010000000010000000100000001'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [17]: bin_ip("160.1.1.1")
Out[17]: '10100000000000010000000100000001'
```

Сортировка с использованием функции bin_ip:

```
In [18]: ip_list = ["10.1.1.1", "10.1.10.1", "10.1.2.1", "10.1.11.1"]

In [19]: sorted(ip_list, key=bin_ip)
Out[19]: ['10.1.1.1', '10.1.2.1', '10.1.10.1', '10.1.11.1']
```

Примечание: Также дальше будет рассматриваться модуль ipaddress, который позволит создавать специальные объекты, которые соответствуют IP-адресу и они уже сортируются правильно по десятичному значению.

enumerate

Иногда, при переборе объектов в цикле for, нужно не только получить сам объект, но и его порядковый номер. Это можно сделать, создав дополнительную переменную, которая будет расти на единицу с каждым прохождением цикла. Однако, гораздо удобнее это делать с помощью итератора enumerate.

Базовый пример:

```
In [15]: list1 = ['str1', 'str2', 'str3']

In [16]: for position, string in enumerate(list1):
    ...:     print(position, string)
    ...:
0 str1
1 str2
2 str3
```

enumerate умеет считать не только с нуля, но и с любого значение, которое ему указали после объекта:

```
In [17]: list1 = ['str1', 'str2', 'str3']

In [18]: for position, string in enumerate(list1, 100):
    ...:     print(position, string)
    ...:
100 str1
101 str2
102 str3
```

Иногда нужно проверить, что сгенерировал итератор, как правило, на стадии написания скрипта. Если необходимо увидеть содержимое, которое сгенерирует итератор, полностью, можно воспользоваться функцией list:

```
In [19]: list1 = ['str1', 'str2', 'str3']

In [20]: list(enumerate(list1, 100))
Out[20]: [(100, 'str1'), (101, 'str2'), (102, 'str3')]
```

Пример использования enumerate для EEM

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 down"
 action 1 cli command "enable"
 action 2 cli command "conf t"
 action 3 cli command "interface fa0/1"
 action 4 cli command "no sh"
```

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 Python-скрипта можно сгенерировать команды EEM на основании существующего списка команд (файл enumerate_eem.py):

```
import sys

config = sys.argv[1]

with open(config, 'r') as f:
    for i, command in enumerate(f, 1):
        print('action {:04} cli command "{}"'.format(i, command.rstrip()))
```

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

Файл с командами выглядит так (r1_config.txt):

```
en
conf t
no int Gi0/0/0.300
```

(continues on next page)

(продолжение с предыдущей страницы)

```
no int Gi0/0/0.301
no int Gi0/0/0.302
int range gi0/0/0-2
 channel-group 1 mode active
interface Port-channel1.300
 encapsulation dot1Q 300
 vrf forwarding Management
 ip address 10.16.19.35 255.255.255.248
```

Вывод будет таким:

```
$ python enumerate_eem.py r1_config.txt
action 0001 cli command "en"
action 0002 cli command "conf t"
action 0003 cli command "no int Gi0/0/0.300"
action 0004 cli command "no int Gi0/0/0.301"
action 0005 cli command "no int Gi0/0/0.302"
action 0006 cli command "int range gi0/0/0-2"
action 0007 cli command " channel-group 1 mode active"
action 0008 cli command "interface Port-channel1.300"
action 0009 cli command " encapsulation dot1Q 300"
action 0010 cli command " vrf forwarding Management"
action 0011 cli command " ip address 10.16.19.35 255.255.255.248"
```

Функция zip

Функция `zip`:

- на вход функции передаются последовательности
- `zip` возвращает итератор с кортежами, в котором n-ый кортеж состоит из n-ых элементов последовательностей, которые были переданы как аргументы
- например, десятый кортеж будет содержать десятый элемент каждой из переданных последовательностей
- если на вход были переданы последовательности разной длины, то все они будут отрезаны по самой короткой последовательности
- порядок элементов соблюдается

Примечание: Так как `zip` - это итератор, для отображение его содержимого используется `list`

Пример использования `zip`:

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

In [2]: b = [100, 200, 300]

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

Использование zip со списками разной длины:

```
In [4]: a = [1, 2, 3, 4, 5]

In [5]: b = [10, 20, 30, 40, 50]

In [6]: c = [100, 200, 300]

In [7]: list(zip(a, b, c))
Out[7]: [(1, 10, 100), (2, 20, 200), (3, 30, 300)]
```

Использование zip для создания словаря

Пример использования zip для создания словаря:

```
In [4]: d_keys = ['hostname', 'location', 'vendor', 'model', 'IOS', 'IP']
In [5]: d_values = ['london_r1', '21 New Globe Walk', 'Cisco', '4451', '15.4', '10.255.0.1
↪']

In [6]: list(zip(d_keys, d_values))
Out[6]:
[('hostname', 'london_r1'),
 ('location', '21 New Globe Walk'),
 ('vendor', 'Cisco'),
 ('model', '4451'),
 ('IOS', '15.4'),
 ('IP', '10.255.0.1')]

In [7]: dict(zip(d_keys, d_values))
Out[7]:
{'IOS': '15.4',
 'IP': '10.255.0.1',
 'hostname': 'london_r1',
 'location': '21 New Globe Walk',
 'model': '4451',
 'vendor': 'Cisco'}
In [8]: r1 = dict(zip(d_keys,d_values))
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  'location': '21 New Globe Walk',
  'model': '3850',
  'vendor': 'Cisco'}}
```

Функция all

Функция `all` возвращает True, если все элементы истинные (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 истинный.

```
In [7]: any([False, True, True])
Out[7]: True

In [8]: any([False, False, False])
Out[8]: False

In [9]: any([])
Out[9]: False

In [10]: any(i.isdigit() for i in '10.1.1.a'.split('.'))
Out[10]: True
```

Например, с помощью any, можно заменить функцию ignore_command:

```
def ignore_command(command):
    '''
    Функция проверяет содержится ли в команде слово из списка ignore.
    * command - строка. Команда, которую надо проверить
    * Возвращает True, если в команде содержится слово из списка ignore, False - если нет
    '''
    ignore = ['duplex', 'alias', 'Current configuration']

    for word in ignore:
        if word in command:
            return True
    return False
```

На такой вариант:

```
def ignore_command(command):
    '''
    Функция проверяет содержится ли в команде слово из списка ignore.
    command - строка. Команда, которую надо проверить
    Возвращает True, если в команде содержится слово из списка ignore, False - если нет
    '''
    ignore = ['duplex', 'alias', 'Current configuration']

    return any([word in command for word in ignore])
```

Анонимная функция (лямбда-выражение)

В Python лямбда-выражение позволяет создавать анонимные функции - функции, которые не привязаны к имени.

В анонимной функции:

- может содержаться только одно выражение
- могут передаваться сколько угодно аргументов

Стандартная функция:

```
In [1]: def sum_arg(a, b): return a + b

In [2]: sum_arg(1, 2)
Out[2]: 3
```

Аналогичная анонимная функция, или лямбда-функция:

```
In [3]: sum_arg = lambda a, b: a + b

In [4]: sum_arg(1, 2)
Out[4]: 3
```

Обратите внимание, что в определении лямбда-функции нет оператора return, так как в этой функции может быть только одно выражение, которое всегда возвращает значение и завершает работу функции.

Лямбда-функцию удобно использовать в выражениях, где требуется написать небольшую функцию для обработки данных.

Например, в функции sorted лямбда-выражение 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 лямбда-функция пригодится в функциях map и filter, которые будут рассматриваться в следующих разделах.

Функция map

Функция map применяет функцию к каждому элементу последовательности и возвращает итератор с результатами.

Например, с помощью map можно выполнять преобразования элементов. Перевести все строки в верхний регистр:

```
In [1]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [2]: map(str.upper, list_of_words)
Out[2]: <map at 0xb45eb7ec>

In [3]: list(map(str.upper, list_of_words))
Out[3]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

Примечание: `str.upper("aaa")` делает то же самое что `"aaa".upper()`.

Конвертация в числа:

```
In [3]: list_of_str = ['1', '2', '5', '10']

In [4]: list(map(int, list_of_str))
Out[4]: [1, 2, 5, 10]
```

Вместе с map удобно использовать лямбда-выражения:

```
In [5]: vlans = [100, 110, 150, 200, 201, 202]

In [6]: list(map(lambda x: 'vlan {}'.format(x), vlans))
Out[6]: ['vlan 100', 'vlan 110', 'vlan 150', 'vlan 200', 'vlan 201', 'vlan 202']
```

Если функция, которую использует map(), ожидает два аргумента, то передаются два списка:

```
In [7]: nums = [1, 2, 3, 4, 5]

In [8]: nums2 = [100, 200, 300, 400, 500]

In [9]: list(map(lambda x, y: x*y, nums, nums2))
Out[9]: [100, 400, 900, 1600, 2500]
```

List comprehension вместо map

Как правило, вместо map можно использовать list comprehension. Чаще всего, вариант с list comprehension более понятный, а в некоторых случаях даже быстрее.

> Ответ Alex Martelli со сравнением map и list comprehension

Но map может быть эффективней в том случае, когда надо сгенерировать большое количество элементов, так как map - итератор, а list comprehension генерирует список.

Примеры, аналогичные приведенным выше, в варианте с list comprehension.

Перевести все строки в верхний регистр:

```
In [48]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [49]: [word.upper() for word in list_of_words]
Out[49]: ['ONE', 'TWO', 'LIST', '', 'DICT']
```

Конвертация в числа:

```
In [50]: list_of_str = ['1', '2', '5', '10']

In [51]: [int(i) for i in list_of_str]
Out[51]: [1, 2, 5, 10]
```

Форматирование строк:

```
In [52]:  vlans = [100, 110, 150, 200, 201, 202]

In [53]: [f'vlan {x}' for x in vlans]
Out[53]: ['vlan 100', 'vlan 110', 'vlan 150', 'vlan 200', 'vlan 201', 'vlan 202']
```

Для получения пар элементов используется zip:

```
In [54]: nums = [1, 2, 3, 4, 5]

In [55]: nums2 = [100, 200, 300, 400, 500]

In [56]: [x * y for x, y in zip(nums, nums2)]
Out[56]: [100, 400, 900, 1600, 2500]
```

Функция filter

Функция `filter` применяет функцию ко всем элементам последовательности и возвращает итератор с теми объектами, для которых функция вернула True.

Например, вернуть только те строки, в которых находятся числа:

```
In [1]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']

In [2]: filter(str.isdigit, list_of_strings)
Out[2]: <filter at 0xb45eb1cc>

In [3]: list(filter(str.isdigit, list_of_strings))
Out[3]: ['100', '1', '50']
```

Из списка чисел оставить только нечетные:

```
In [3]: list(filter(lambda x: x % 2 == 1, [10, 111, 102, 213, 314, 515]))
Out[3]: [111, 213, 515]
```

Аналогично, только четные:

```
In [4]: list(filter(lambda x: x % 2 == 0, [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']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [6]: list(filter(lambda x: len(x) > 2, list_of_words))
Out[6]: ['one', 'two', 'list', 'dict']
```

List comprehension вместо filter

Как правило, вместо filter можно использовать list comprehension.

Примеры, аналогичные приведенным выше, в варианте с list comprehension.

Вернуть только те строки, в которых находятся числа:

```
In [7]: list_of_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']

In [8]: [s for s in list_of_strings if s.isdigit()]
Out[8]: ['100', '1', '50']
```

Нечетные/четные числа:

```
In [9]: nums = [10, 111, 102, 213, 314, 515]

In [10]: [n for n in nums if n % 2 == 1]
Out[10]: [111, 213, 515]

In [11]: [n for n in nums if n % 2 == 0]
Out[11]: [10, 102, 314]
```

Из списка слов оставить только те, у которых количество букв больше двух:

```
In [12]: list_of_words = ['one', 'two', 'list', '', 'dict']

In [13]: [word for word in list_of_words if len(word) > 2]
Out[13]: ['one', 'two', 'list', 'dict']
```

11. 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',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '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.

Примечание: 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\n--- 8.8.8.8␣
↪ping statistics ---\n2 packets transmitted, 2 packets received, 0.0% packet loss\nround-
↪trip min/avg/max/stddev = 46.875/48.377/49.880/1.503 ms\n'
```

from module import object

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_CANTCREAT',
 'EX_CONFIG',
 ...
 'wait',
 'wait3',
 'wait4',
 'waitpid',
 'walk',
 'write']

In [3]: len(dir())
Out[3]: 218
```

В модуле os очень много объектов, поэтому вывод сокращен. В конце указана длина списка имен текущего именного пространства.

Такой вариант импорта лучше не использовать. При таком импорте по коду непонятно, что какая-то функция взята, например, из модуля os. Это заметно усложняет понимание кода.

Создание своих модулей

Модуль - это файл с расширением .py и кодом Python.

Пример создания своих модулей и импорта функции из одного модуля в другой.

Файл check_ip_function.py:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


ip1 = '10.1.1.1'
ip2 = '10.1.1'

print('Проверка IP...')
print(ip1, check_ip(ip1))
print(ip2, check_ip(ip2))
```

В файле check_ip_function.py создана функция check_ip, которая проверяет, что аргумент является IP-адресом. Тут проверка выполняется с помощью модуля ipaddress, который будет рассматриваться в следующем разделе.

Функция ipaddress.ip_address сама проверяет правильность IP-адреса и генерирует исключение ValueError, если адрес не прошел проверку. Функция check_ip возвращает True, если адрес прошел проверку и False - если нет.

Если запустить скрипт check_ip_function.py вывод будет таким:

```
$ python check_ip_function.py
Проверка IP...
10.1.1.1 True
10.1.1 False
```

Второй скрипт импортирует функцию check_ip и использует ее для того чтобы из списка адресов отобрать только те, которые прошли проверку (файл get_correct_ip.py):

```
from check_ip_function import check_ip


def return_correct_ip(ip_addresses):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    correct = []
    for ip in ip_addresses:
        if check_ip(ip):
            correct.append(ip)
    return correct

print('Проверка списка IP-адресов')
ip_list = ['10.1.1.1', '8.8.8.8', '2.2.2']
correct = return_correct_ip(ip_list)
print(correct)
```

В первой строке выполняется импорт функции check_ip из модуля check_ip_function.py.

Результат выполнения скрипта:

```
$ python get_correct_ip.py
Проверка IP...
10.1.1.1 True
10.1.1 False
Проверка списка IP-адресов
['10.1.1.1', '8.8.8.8']
```

Обратите внимание, что выведена не только информация из скрипта get_correct_ip.py, но и информация из скрипта check_ip_function.py. Так происходит из-за того, что любая разновидность import выполняет весь скрипт. То есть, даже когда импорт выглядит как `from check_ip_function import check_ip`, выполняется весь скрипт check_ip_function.py, а не только функция check_ip. В итоге будут выводиться все сообщения импортируемого скрипта.

Сообщения из импортируемого скрипта не страшны, они мешают и только, хуже когда скрипт выполнял что-то типа подключения к оборудованию и при импорте функции из него, придется ждать пока это подключение выполнится.

В Python есть возможность указать, что некоторые строки не должны выполняться при импорте. Это рассматривается в следующем подразделе.

Функцию return_correct_ip можно заменить filter или генератором списка, выше используется более длинный, но скорее всего, пока что более понятный вариант:

```
In [19]: list(filter(check_ip, ip_list))
Out[19]: ['10.1.1.1', '8.8.8.8']

In [20]: [ip for ip in ip_list if check_ip(ip)]
Out[20]: ['10.1.1.1', '8.8.8.8']

In [21]: def return_correct_ip(ip_addresses):
    ...:     return [ip for ip in ip_addresses if check_ip(ip)]
    ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [22]: return_correct_ip(ip_list)
Out[22]: ['10.1.1.1', '8.8.8.8']
```

if __name__ == "__main__"

Достаточно часто скрипт может выполняться и самостоятельно, и может быть импортирован как модуль другим скриптом. Так как импорт скрипта запускает этот скрипт, часто надо указать, что какие-то строки не должны выполняться при импорте.

В предыдущем примере было два скрипта: check_ip_function.py и get_correct_ip.py. И при запуске get_correct_ip.py, отображались print из check_ip_function.py.

В Python есть специальный прием, который позволяет указать, что какой-то код не должен выполняться при импорте: все строки, которые находятся в блоке `if __name__ == '__main__'` не выполняются при импорте.

Переменная __name__ - это специальная переменная, которая будет равна "__main__", только если файл запускается как основная программа, и выставляется равной имени модуля при импорте модуля. То есть, условие `if __name__ == '__main__'` проверяет, был ли файл запущен напрямую.

Как правило, в блок `if __name__ == '__main__'` заносят все вызовы функций и вывод информации на стандартный поток вывода. То есть, в скрипте check_ip_function.py в этом блоке будет все, кроме импорта и функции return_correct_ip:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


if __name__ == '__main__':
    ip1 = '10.1.1.1'
    ip2 = '10.1.1'

    print('Проверка IP...')
    print(ip1, check_ip(ip1))
    print(ip2, check_ip(ip2))
```

Результат выполнения скрипта:

```
$ python check_ip_function.py
Проверка IP...
10.1.1.1 True
10.1.1 False
```

При запуске скрипта check_ip_function.py напрямую, выполняются все строки, так как переменная `__name__` в этом случае равна `'__main__'`.

Скрипт get_correct_ip.py остается без изменений

```
from check_ip_function import check_ip


def return_correct_ip(ip_addresses):
    correct = []
    for ip in ip_addresses:
        if check_ip(ip):
            correct.append(ip)
    return correct


print('Проверка списка IP-адресов')
ip_list = ['10.1.1.1', '8.8.8.8', '2.2.2']
correct = return_correct_ip(ip_list)
print(correct)
```

Выполнение скрипта get_correct_ip.py выглядит таким образом:

```
$ python get_correct_ip.py
Проверка списка IP-адресов
['10.1.1.1', '8.8.8.8']
```

Теперь вывод содержит только информацию из скрипта get_correct_ip.py.

В целом, лучше привыкнуть писать весь код, который вызывает функции и выводит что-то на стандартный поток вывода, внутри блока `if __name__ == '__main__'`.

Предупреждение: Начиная с 9 раздела, для заданий есть программные тесты, с помощью которых можно проверить правильность выполнения заданий. Для корректной работы с тестами, надо всегда писать вызов функции в файле задания внутри блока `if __name__ == '__main__'`. Отсутствие этого блока будет вызывать ошибки, не во всех заданиях, однако это все равно позволит избежать проблем.

Пути поиска модулей

При импорте модуля, Python сначала ищет модуль в стандартной библиотеке. Если модуль не найден в стандартной библиотеке, поиск модуля идет в каталогах, которые указаны в sys.path.

Содержимое sys.path состоит из:

- текущего каталога
- каталогов, которые указаны в переменной PYTHONPATH
- пути по умолчанию (зависят от установки Python)

Пути поиска модулей хранятся в переменной sys.path:

```
In [1]: import sys

In [2]: sys.path
Out[2]:
['',
 '/usr/local/bin',
 '/usr/local/lib/python36.zip',
 '/usr/local/lib/python3.6',
 '/usr/local/lib/python3.6/lib-dynload',
 '/home/vagrant/.local/lib/python3.6/site-packages',
 '/usr/local/lib/python3.6/site-packages',
 '/usr/local/lib/python3.6/site-packages/IPython/extensions',
 '/home/vagrant/.ipython']
```

Аналогичный вывод, но внутри виртуального окружения:

```
In [1]: import sys

In [2]: sys.path
Out[2]:
['/home/vagrant/venv/pyneng-py3-8-0/bin',
 '/home/vagrant/venv/pyneng-py3-8-0/lib/python38.zip',
 '/home/vagrant/venv/pyneng-py3-8-0/lib/python3.8',
 '/home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/lib-dynload',
 '/usr/local/lib/python3.8',
 '',
 '/home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/site-packages',
 '/home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/site-packages/IPython/extensions',
 '/home/vagrant/.ipython']
```

Добавление своих скриптов в пути поиска модулей

Добавить свой скрипт в пути поиска модулей нужно в том случае, если этот скрипт нужно использовать в других скриптах, которые находятся в разных каталогах.

Для добавления модулей в пути поиска есть несколько вариантов:

1. Переместить скрипт в каталог site-packages
2. Создать специальный файл с расширением pth в каталоге site-packages и написать в этом файлы пути поиска модулей

Конкретный путь каталога site-packages зависит от версии Python и того используете ли вы виртуальное окружение. Например, в последнем выводе `sys.path` путь будет `'/home/vagrant/venv/pyneng-py3-8-0/lib/python3.8/site-packages'`. Если переместить туда скрипт, его можно будет импортировать из любого другого скрипта.

Так как переносить файлы не всегда удобно, есть второй вариант - файлы pth. Для этого варианта надо создать файл с любым именем в каталоге site-packages, например, `my_scripts.pth` и написать в нем пути к нужным скриптам:

```
/home/vagrant/repos/pyneng/examples/11_modules
/home/vagrant/repos/pyneng/exercises/09_functions
/home/vagrant/repos/pyneng/exercises/11_modules
/home/vagrant/repos/pyneng/exercises/12_userful_modules
```

Примечание: Обратите внимание, что тут речь именно об использовании функций какого-то вашего скрипта в других скриптах, не о запуске модуля из любого места в файловой системе как утилиты. Это тоже можно сделать, коротко о том как установить скрипт с CLI интерфейсом как утилиту в ОС. Например, так устанавливалась утилита pyneng.

Рекомендации по поводу расположения функций в коде

В PEP8 нет рекомендаций по этому поводу.

Если скрипт в одном файле, обычно порядок такой:

1. shebang, file encoding
2. docstring модуля
3. импорт (модули стандартной библиотеки, сторонние модули, свои скрипты)
4. константы
5. все функции в условно произвольном порядке, тут уже надо самостоятельно решить как удобнее

6. функции/код для создания CLI если есть
7. Часто, если есть код который надо писать глобально создают функцию main
8. `if __name__ == "__main__":` и вызов функции main или глобального кода, который вызывает функции

При этом среди функций обычно выбирают для себя какой-то порядок, чтобы он был плюс-минус однотипным в разных файлах. Например, сначала пишутся общие функцие, которые не зависят от других функций в файле, потом те что зависят. При этом обычно есть какой-то порядок выполнения действий: подключились на оборудование и считали вывод, парсим его, записали результат в файл - тогда соблюдаем этот порядок в функциях.

Примечание: О структуре больших проектов. И еще одна ссылка по этой же теме, с примерами структуры проектов Flask/Django.

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 11.1

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

У функции должен быть один параметр command_output, который ожидает как аргумент вывод команды одной строкой (не имя файла). Для этого надо считать все содержимое файла в строку, а затем передать строку как аргумент функции (как передать вывод команды показано в коде ниже).

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")}
```

В словаре интерфейсы должны быть записаны без пробела между типом и именем. То есть так Fa0/0, а не так Fa 0/0.

Проверить работу функции на содержимом файла sh_cdp_n_sw1.txt. При этом функция работать и на других файлах (тест проверяет работу функции на выводе из sh_cdp_n_sw1.txt и sh_cdp_n_r3.txt).

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
def parse_cdp_neighbors(command_output):
    """
    Тут мы передаем вывод команды одной строкой потому что именно в таком виде
    будет получен вывод команды с оборудования. Принимая как аргумент вывод
    команды, вместо имени файла, мы делаем функцию более универсальной: она может
    работать и с файлами и с выводом с оборудования.
    Плюс учимся работать с таким выводом.
    """


if __name__ == "__main__":
    with open("sh_cdp_n_sw1.txt") as f:
        print(parse_cdp_neighbors(f.read()))
```

Задание 11.2

Создать функцию create_network_map, которая обрабатывает вывод команды show cdp neighbors из нескольких файлов и объединяет его в одну общую топологию.

У функции должен быть один параметр filenames, который ожидает как аргумент список с именами файлов, в которых находится вывод команды show cdp neighbors.

Функция должна возвращать словарь, который описывает соединения между устройствами. Структура словаря такая же, как в задании 11.1:

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

Сгенерировать топологию, которая соответствует выводу 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. Если функция parse_cdp_neighbors не может обработать вывод одного из файлов с выводом команды, надо исправить код функции в задании 11.1.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
infiles = [
    "sh_cdp_n_sw1.txt",
    "sh_cdp_n_r1.txt",
    "sh_cdp_n_r2.txt",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "sh_cdp_n_r3.txt",
]
```

Задание 11.2a

Примечание: 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 create_network_map из задания 11.2 создать словарь topology с описанием топологии для файлов:

- sh_cdp_n_sw1.txt
- sh_cdp_n_r1.txt
- sh_cdp_n_r2.txt
- sh_cdp_n_r3.txt

С помощью функции draw_topology из файла draw_network_graph.py нарисовать схему для словаря topology, полученного с помощью create_network_map. Как работать с функцией draw_topology надо разобраться самостоятельно, почитав описание функции в файле draw_network_graph.py. Полученная схема будет записана в файл svg - его можно открыть браузером.

С текущим словарем topology на схеме нарисованы лишние соединения. Они возникают потому что в одном файле CDP (sh_cdp_n_r1.txt) описывается соединение

```
("R1", "Eth0/0"): ("SW1", "Eth0/1")
```

а в другом (sh_cdp_n_sw1.txt)

```
("SW1", "Eth0/1"): ("R1", "Eth0/0")
```

В этом задании надо создать новую функцию unique_network_map, которая из этих двух соединений будет оставлять только одно, для корректного рисования схемы. При этом все равно какое из соединений оставить.

У функции unique_network_map должен быть один параметр topology_dict, который ожидает как аргумент словарь. Это должен быть словарь полученный в результате выполнения функции create_network_map из задания 11.2.

Пример словаря:

```
{
    ("R1", "Eth0/0"): ("SW1", "Eth0/1"),
    ("R2", "Eth0/0"): ("SW1", "Eth0/2"),
    ("R2", "Eth0/1"): ("SW2", "Eth0/11"),
    ("R3", "Eth0/0"): ("SW1", "Eth0/3"),
    ("R3", "Eth0/1"): ("R4", "Eth0/0"),
    ("R3", "Eth0/2"): ("R5", "Eth0/0"),
    ("SW1", "Eth0/1"): ("R1", "Eth0/0"),
    ("SW1", "Eth0/2"): ("R2", "Eth0/0"),
    ("SW1", "Eth0/3"): ("R3", "Eth0/0"),
    ("SW1", "Eth0/5"): ("R6", "Eth0/1"),
}
```

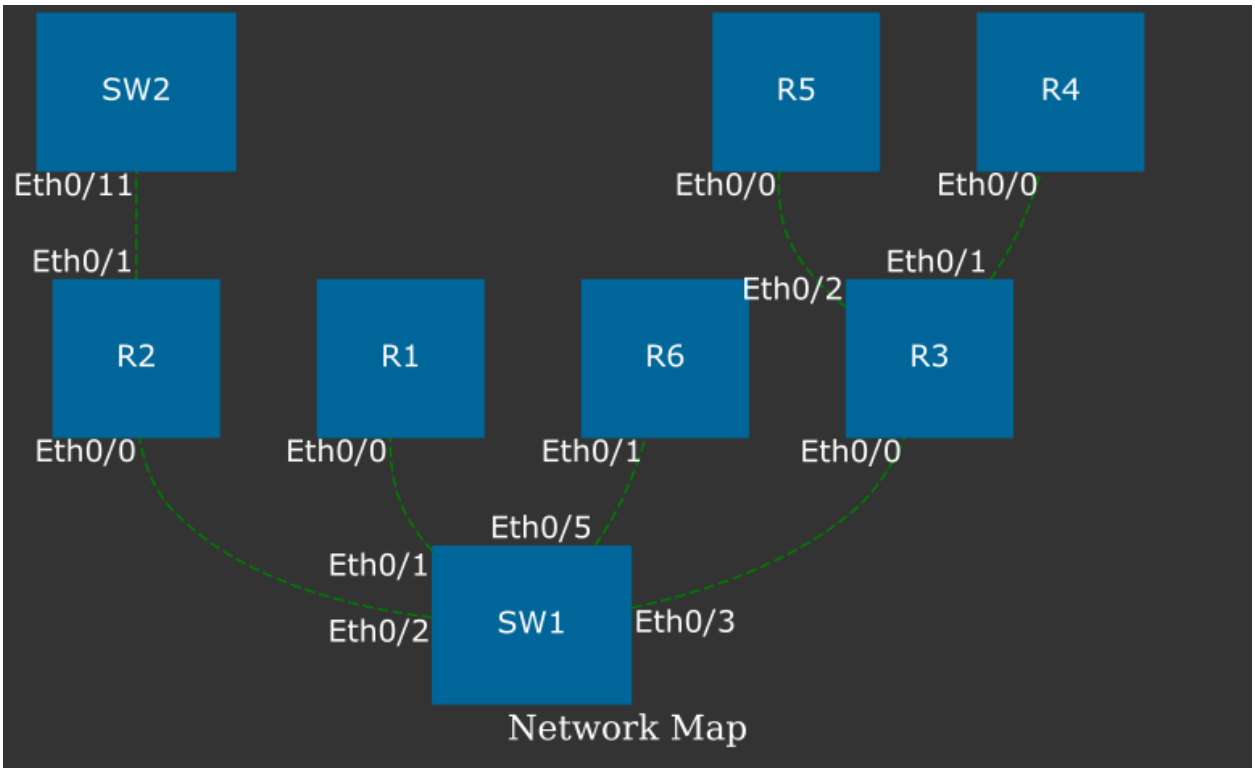
Функция должна возвращать словарь, который описывает соединения между устройствами. В словаре надо избавиться от «дублирующих» соединений и оставлять только одно из них.

Структура итогового словаря такая же, как в задании 11.2:

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

После создания функции, попробовать еще раз нарисовать топологию, теперь уже для словаря, который возвращает функция unique_network_map.

Результат должен выглядеть так же, как схема в файле task_11_2a_topology.svg



При этом:

- Расположение устройств на схеме может быть другим
- Соединения должны соответствовать схеме

Не копировать код функций create_network_map и draw_topology.

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
infiles = [
    "sh_cdp_n_sw1.txt",
    "sh_cdp_n_r1.txt",
    "sh_cdp_n_r2.txt",
    "sh_cdp_n_r3.txt",
]
```

12. Полезные модули

В этом разделе описаны такие модули:

- subprocess
- os
- ipaddress
- pprint
- tabulate

Модуль subprocess

Модуль subprocess позволяет создавать новые процессы. При этом он может подключаться к стандартным потокам ввода/вывода/ошибок и получать код возврата.

С помощью subprocess можно, например, выполнять любые команды Linux из скрипта. И в зависимости от ситуации получать вывод или только проверять, что команда выполнилась без ошибок.

Примечание: В Python 3.5 синтаксис модуля subprocess изменился.

Функция `subprocess.run`

Функция `subprocess.run` - основной способ работы с модулем subprocess.

Самый простой вариант использования функции - запуск её таким образом:

```
In [1]: import subprocess

In [2]: result = subprocess.run('ls')
ipython_as_mngmt_console.md  README.md         version_control.md
module_search.md             useful_functions
naming_conventions           useful_modules
```

В переменной result теперь содержится специальный объект CompletedProcess. Из этого объекта можно получить информацию о выполнении процесса, например, о коде возврата:

```
In [3]: result
Out[3]: CompletedProcess(args='ls', returncode=0)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [4]: result.returncode
Out[4]: 0
```

Код 0 означает, что программа выполнилась успешно.

Если необходимо вызвать команду с аргументами, её нужно передавать таким образом (как список):

```
In [5]: result = subprocess.run(['ls', '-ls'])
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:28 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

При попытке выполнить команду с использованием wildcard-выражений, например, использовать *, возникнет ошибка:

```
In [6]: result = subprocess.run(['ls', '-ls', '*md'])
ls: cannot access *md: No such file or directory
```

Чтобы вызывать команды, в которых используются wildcard-выражения, нужно добавлять аргумент shell и вызывать команду таким образом:

```
In [7]: result = subprocess.run('ls -ls *md', shell=True)
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Ещё одна особенность функции run - она ожидает завершения выполнения команды. Если попробовать, например, запустить команду ping, то этот аспект будет заметен:

```
In [8]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=55.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=54.4 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 54.498/54.798/55.116/0.252 ms
```

Получение результата выполнения команды

По умолчанию функция run возвращает результат выполнения команды на стандартный поток вывода. Если нужно получить результат выполнения команды, надо добавить аргумент stdout и указать ему значение subprocess.PIPE:

```
In [9]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE)
```

Теперь можно получить результат выполнения команды таким образом:

```
In [10]: print(result.stdout)
b'total 28\n4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md\
↪n4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md\n4 drwxr-xr-x 2␣
↪vagrant vagrant 4096 Jun  7 19:35 naming_conventions\n4 -rw-r--r-- 1 vagrant vagrant ␣
↪277 Jun  7 19:35 README.md\n4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_
↪functions\n4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:30 useful_modules\n4 -rw-r--r--
↪ 1 vagrant vagrant   49 Jun  7 19:35 version_control.md\n'
```

Обратите внимание на букву b перед строкой. Она означает, что модуль вернул вывод в виде байтовой строки. Для перевода её в unicode есть два варианта:

- выполнить decode полученной строки
- указать аргумент encoding

Вариант с decode:

```
In [11]: print(result.stdout.decode('utf-8'))
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:30 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Вариант с encoding:

```
In [12]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.PIPE, encoding='utf-8')

In [13]: print(result.stdout)
total 28
4 -rw-r--r-- 1 vagrant vagrant   56 Jun  7 19:35 ipython_as_mngmt_console.md
4 -rw-r--r-- 1 vagrant vagrant 1638 Jun  7 19:35 module_search.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun  7 19:35 naming_conventions
4 -rw-r--r-- 1 vagrant vagrant  277 Jun  7 19:35 README.md
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 16 05:11 useful_functions
```

(continues on next page)

(продолжение с предыдущей страницы)

```
4 drwxr-xr-x 2 vagrant vagrant 4096 Jun 17 16:31 useful_modules
4 -rw-r--r-- 1 vagrant vagrant   49 Jun  7 19:35 version_control.md
```

Отключение вывода

Иногда достаточно получения кода возврата и нужно отключить вывод результата выполнения на стандартный поток вывода, и при этом сам результат не нужен. Это можно сделать, передав функции run аргумент stdout со значением subprocess.DEVNULL:

```
In [14]: result = subprocess.run(['ls', '-ls'], stdout=subprocess.DEVNULL)

In [15]: print(result.stdout)
None

In [16]: print(result.returncode)
0
```

Работа со стандартным потоком ошибок

Если команда была выполнена с ошибкой или не отработала корректно, вывод команды попадет на стандартный поток ошибок.

Получить этот вывод можно так же, как и стандартный поток вывода:

```
In [17]: result = subprocess.run(['ping', '-c', '3', '-n', 'a'], stderr=subprocess.PIPE,␣
↪encoding='utf-8')
```

Теперь в result.stdout пустая строка, а в result.stderr находится стандартный поток вывода:

```
In [18]: print(result.stdout)
None

In [19]: print(result.stderr)
ping: unknown host a


In [20]: print(result.returncode)
2
```

Примеры использования модуля

Пример использования модуля subprocess (файл subprocess_run_basic.py):

```
import subprocess

reply = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'])

if reply.returncode == 0:
    print('Alive')
else:
    print('Unreachable')
```

Результат выполнения будет таким:

```
$ python subprocess_run_basic.py
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=54.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 53.962/54.145/54.461/0.293 ms
Alive
```

То есть, результат выполнения команды выводится на стандартный поток вывода.

Функция ping_ip проверяет доступность IP-адреса и возвращает True и stdout, если адрес доступен, или False и stderr, если адрес недоступен (файл subprocess_ping_function.py):

```
import subprocess


def ping_ip(ip_address):
    """
    Ping IP address and return tuple:
    On success:
        * True
        * command output (stdout)
    On failure:
        * False
        * error output (stderr)
    """
    reply = subprocess.run(['ping', '-c', '3', '-n', ip_address],
                           stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                            encoding='utf-8')
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stderr

print(ping_ip('8.8.8.8'))
print(ping_ip('a'))
```

Результат выполнения будет таким:

```
$ python subprocess_ping_function.py
(True, 'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8: icmp_seq=1␣
↪ttl=43 time=63.8 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=55.6 ms\n64 bytes␣
↪from 8.8.8.8: icmp_seq=3 ttl=43 time=55.9 ms\n\n--- 8.8.8.8 ping statistics ---\n3␣
↪packets transmitted, 3 received, 0% packet loss, time 2003ms\nrtt min/avg/max/mdev = 55.
↪643/58.492/63.852/3.802 ms\n')
(False, 'ping: unknown host a\n')
```

Модуль os

Модуль os позволяет работать с файловой системой, с окружением, управлять процессами.

В этом подразделе рассматриваются лишь несколько полезных возможностей. За более полным описанием возможностей модуля можно обратиться к документации или статье на сайте PyMOTW.

Импорт модуля:

```
In [1]: import os
```

os.environ

os.environ возвращает словарь с переменными окружения и их значениями. Можно использовать синтаксис обращения по ключу через квадратные скобки, если переменная окружения точно существует (если переменной нет возникнет исключение).

```
In [2]: os.environ["HOME"]
Out[2]: '/home/nata'

In [3]: os.environ["TOKEN"]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Input In [3], in <cell line: 1>()
----> 1 os.environ["TOKEN"]

File ~/venv/pyneng-py3-8-0/lib/python3.8/os.py:673, in _Environ.__getitem__(self, key)
    670     value = self._data[self.encodekey(key)]
    671 except KeyError:
    672     # raise KeyError with the original key value
--> 673     raise KeyError(key) from None
    674 return self.decodevalue(value)

KeyError: 'TOKEN'
```

Или использовать get, тогда при отстутствии переменной окружения, возвращается None:

```
In [3]: os.environ.get("HOME")
Out[3]: '/home/nata'

In [4]: os.environ.get("TOKEN")
```

Примечание: Технически os.environ возвращает объект типа mapping, но на данном этапе проще считать его словарем.

Переменные окружения считываются в момент импорта модуля os, если какие-то переменные были добавлены во время работы скрипта, они не будут доступны через `os.environ`.

os.mkdir

os.mkdir позволяет создать каталог:

```
In [2]: os.mkdir('test')

In [3]: ls -ls
total 0
0 drwxr-xr-x  2 nata  nata  68 Jan 23 18:58 test/
```

os.listdir

Функция listdir возвращает список файлов и подкаталогов в указанном каталоге. Порядок файлов в списке произвольный, если нужно получить их в порядке сортировки имен, можно использовать sorted.

```
In [2]: os.listdir("09_functions")
Out[2]:
['test_task_9_4.py',
 'task_9_2a.py',
 'task_9_1a.py',
 'test_task_9_2.py',
 'task_9_3a.py',
 'test_task_9_3a.py',
 'task_9_3.py',
 'test_task_9_3.py',
 'config_sw2.txt',
 'test_task_9_2a.py',
 'config_sw1.txt',
 'test_task_9_1a.py',
 'test_task_9_1.py',
 'task_9_4.py',
 'task_9_1.py',
 'config_r1.txt',
 'task_9_2.py']

In [3]: sorted(os.listdir("09_functions"))
Out[3]:
['config_r1.txt',
 'config_sw1.txt',
 'config_sw2.txt',
 'task_9_1.py',
 'task_9_1a.py',
 'task_9_2.py',
 'task_9_2a.py',
 'task_9_3.py',
 'task_9_3a.py',
 'task_9_4.py',
 'test_task_9_1.py',
 'test_task_9_1a.py',
 'test_task_9_2.py',
 'test_task_9_2a.py',
 'test_task_9_3.py',
 'test_task_9_3a.py',
 'test_task_9_4.py']
```

Текущий каталог можно указать так `"."` или вызывать listdir без аргументов:

```
In [7]: os.listdir('.')
Out[7]: ['cover3.png', 'dir2', 'dir3', 'README.txt', 'test']

In [7]: os.listdir()
Out[7]: ['cover3.png', 'dir2', 'dir3', 'README.txt', 'test']
```

os.path

Разные операционные системы (ОС) имеют разные соглашения об именах путей, поэтому в стандартной библиотеке есть несколько версий модуля `os.path`. Модуль os автоматически подгружает нужную часть для работы с текущей ОС. Например, при запуске одних и тех же функций модуля os на Windows и Linux, разделителем пути будут считаться разные значения.

При необходимости работы на Linux с путями Windows и наоборот, можно использовать модули `posixpath`, `ntpath` вместо `os.path`.

os.path.exists

Функция `os.path.exists` проверяет существует ли указанный путь и возвращает True, если пусть существует и False иначе:

```
In [5]: os.path.exists('test')
Out[5]: True

In [6]: if not os.path.exists('test'):
   ...:     os.mkdir('test')
   ...:
```

os.path.isdir, os.path.isfile

Функция `os.path.isdir` возвращает True, если путь ведет к каталогу и False иначе:

```
In [4]: os.path.isdir("09_functions")
Out[4]: True

In [5]: os.path.isdir("/home/nata/repos/pyneng-tasks/exercises/09_functions/")
Out[5]: True

In [6]: os.path.isdir("/home/nata/repos/pyneng-tasks/exercises/09_functions/task_9_1.py")
Out[6]: False

In [7]: os.path.isdir("09_functions/task_9_1.py")
Out[7]: False
```

Функция os.path.isfile возвращает True, если путь ведет к файлу и False иначе:

```
In [9]: os.path.isfile("09_functions/task_9_1.py")
Out[9]: True

In [10]: os.path.isfile("09_functions/")
Out[10]: False
```

С помощью проверок os.path.isdir и os.path.isfile и os.listdir можно получить списки файлов и каталогов (в примере для текущего каталога).

Список каталогов в текущем каталоге:

```
In [8]: dirs = [d for d in os.listdir('.') if os.path.isdir(d)]

In [9]: dirs
Out[9]: ['dir2', 'dir3', 'test']
```

Список файлов в текущем каталоге:

```
In [10]: files = [f for f in os.listdir('.') if os.path.isfile(f)]

In [11]: files
Out[11]: ['cover3.png', 'README.txt']
```

os.path.split

Функция os.path.split делает разделение пути на «основную часть» и конец пути по последнему / и возвращает кортеж из двух элементов. При этом для Windows автоматически будет использоваться обратный слеш.

Если в конце пути не слеша, разделение будет таким

```
In [6]: os.path.split("book/25_additional_info/README.md")
Out[6]: ('book/25_additional_info', 'README.md')

In [8]: os.path.split("book/25_additional_info")
Out[8]: ('book', '25_additional_info')
```

Если путь заканчивается на слеш, второй элемент кортежа будет пустой строкой:

```
In [7]: os.path.split("book/25_additional_info/")
Out[7]: ('book/25_additional_info', '')

In [9]: os.path.split("book/")
Out[9]: ('book', '')
```

Если в пути нет слеш, первый элемент кортежа будет пустой строкой:

```
In [39]: os.path.split("README.md")
Out[39]: ('', 'README.md')
```

os.path.abspath

Функция os.path.abspath возвращает абсолютный путь для указанного файла или каталога:

```
In [40]: os.path.abspath("09_functions")
Out[40]: '/home/nata/repos/pyneng-tasks/exercises/09_functions'

In [41]: os.path.abspath(".")
Out[41]: '/home/nata/repos/pyneng-tasks/exercises'
```

Модуль ipaddress

Модуль ipaddress упрощает работу с IP-адресами.

Примечание: С версии Python 3.3 модуль ipaddress входит в стандартную библиотеку Python.

---

ipaddress.ip_address

Функция `ipaddress.ip_address` позволяет создавать объект IPv4Address или IPv6Address соответственно:

```
In [1]: import ipaddress

In [2]: ipv4 = ipaddress.ip_address('10.0.1.1')

In [3]: ipv4
Out[3]: IPv4Address('10.0.1.1')

In [4]: print(ipv4)
10.0.1.1
```

У объекта есть несколько методов и атрибутов:

```
In [5]: ipv4.
ipv4.compressed       ipv4.is_loopback      ipv4.is_unspecified   ipv4.version
ipv4.exploded         ipv4.is_multicast     ipv4.max_prefixlen
```

(continues on next page)

(продолжение с предыдущей страницы)

```
ipv4.is_global        ipv4.is_private       ipv4.packed
ipv4.is_link_local    ipv4.is_reserved      ipv4.reverse_pointer
```

С помощью атрибутов `is_` можно проверить, к какому диапазону принадлежит адрес:

```
In [6]: ipv4.is_loopback
Out[6]: False

In [7]: ipv4.is_multicast
Out[7]: False

In [8]: ipv4.is_reserved
Out[8]: False

In [9]: ipv4.is_private
Out[9]: True
```

С полученными объектами можно выполнять различные операции:

```
In [10]: ip1 = ipaddress.ip_address('10.0.1.1')

In [11]: ip2 = ipaddress.ip_address('10.0.2.1')

In [12]: ip1 > ip2
Out[12]: False

In [13]: ip2 > ip1
Out[13]: True

In [14]: ip1 == ip2
Out[14]: False

In [15]: ip1 != ip2
Out[15]: True

In [16]: str(ip1)
Out[16]: '10.0.1.1'

In [17]: int(ip1)
Out[17]: 167772417

In [18]: ip1 + 5
Out[18]: IPv4Address('10.0.1.6')

In [19]: ip1 - 5
Out[19]: IPv4Address('10.0.0.252')
```

ipaddress.ip_network

Функция ipaddress.ip_network позволяет создать объект, который описывает сеть (IPv4 или IPv6):

```
In [20]: subnet1 = ipaddress.ip_network('80.0.1.0/28')
```

Как и у адреса, у сети есть различные атрибуты и методы:

```
In [21]: subnet1.broadcast_address
Out[21]: IPv4Address('80.0.1.15')

In [22]: subnet1.with_netmask
Out[22]: '80.0.1.0/255.255.255.240'

In [23]: subnet1.with_hostmask
Out[23]: '80.0.1.0/0.0.0.15'

In [24]: subnet1.prefixlen
Out[24]: 28

In [25]: subnet1.num_addresses
Out[25]: 16
```

Метод hosts возвращает генератор, поэтому, чтобы посмотреть все хосты, надо применить функцию list:

```
In [26]: list(subnet1.hosts())
Out[26]:
[IPv4Address('80.0.1.1'),
 IPv4Address('80.0.1.2'),
 IPv4Address('80.0.1.3'),
 IPv4Address('80.0.1.4'),
 IPv4Address('80.0.1.5'),
 IPv4Address('80.0.1.6'),
 IPv4Address('80.0.1.7'),
 IPv4Address('80.0.1.8'),
 IPv4Address('80.0.1.9'),
 IPv4Address('80.0.1.10'),
 IPv4Address('80.0.1.11'),
 IPv4Address('80.0.1.12'),
 IPv4Address('80.0.1.13'),
 IPv4Address('80.0.1.14')]
```

Метод subnets позволяет разбивать на подсети. По умолчанию он разбивает сеть на две подсети:

```
In [27]: list(subnet1.subnets())
Out[27]: [IPv4Network('80.0.1.0/29'), IPv4Network('80.0.1.8/29')]
```

Параметр prefixlen_diff позволяет указать количество бит для подсетей:

```
In [28]: list(subnet1.subnets(prefixlen_diff=2))
Out[28]:
[IPv4Network('80.0.1.0/30'),
 IPv4Network('80.0.1.4/30'),
 IPv4Network('80.0.1.8/30'),
 IPv4Network('80.0.1.12/30')]
```

С помощью параметра new_prefix можно указать, какая маска должна быть у подсетей:

```
In [29]: list(subnet1.subnets(new_prefix=30))
Out[29]:
[IPv4Network('80.0.1.0/30'),
 IPv4Network('80.0.1.4/30'),
 IPv4Network('80.0.1.8/30'),
 IPv4Network('80.0.1.12/30')]

In [30]: list(subnet1.subnets(new_prefix=29))
Out[30]: [IPv4Network('80.0.1.0/29'), IPv4Network('80.0.1.8/29')]
```

По IP-адресам в сети можно проходиться в цикле:

```
In [31]: for ip in subnet1:
   ....:     print(ip)
   ....:
80.0.1.0
80.0.1.1
80.0.1.2
80.0.1.3
80.0.1.4
80.0.1.5
80.0.1.6
80.0.1.7
80.0.1.8
80.0.1.9
80.0.1.10
80.0.1.11
80.0.1.12
80.0.1.13
80.0.1.14
80.0.1.15
```

Или обращаться к конкретному адресу:

```
In [32]: subnet1[0]
Out[32]: IPv4Address('80.0.1.0')

In [33]: subnet1[5]
Out[33]: IPv4Address('80.0.1.5')
```

Таким образом можно проверять, находится ли IP-адрес в сети:

```
In [34]: ip1 = ipaddress.ip_address('80.0.1.3')

In [35]: ip1 in subnet1
Out[35]: True
```

ipaddress.ip_interface

Функция `ipaddress.ip_interface` позволяет создавать объект IPv4Interface или IPv6Interface соответственно:

```
In [36]: int1 = ipaddress.ip_interface('10.0.1.1/24')
```

Используя методы объекта IPv4Interface, можно получать адрес, маску или сеть интерфейса:

```
In [37]: int1.ip
Out[37]: IPv4Address('10.0.1.1')

In [38]: int1.network
Out[38]: IPv4Network('10.0.1.0/24')

In [39]: int1.netmask
Out[39]: IPv4Address('255.255.255.0')
```

Пример использования модуля

Так как в модуль встроены проверки корректности адресов, можно ими пользоваться, например, чтобы проверить, является ли адрес адресом сети или хоста:

```
In [40]: IP1 = '10.0.1.1/24'

In [41]: IP2 = '10.0.1.0/24'

In [42]: def check_if_ip_is_network(ip_address):
   ....:     try:
   ....:         ipaddress.ip_network(ip_address)
   ....:         return True
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ....:     except ValueError:
   ....:         return False
   ....:

In [43]: check_if_ip_is_network(IP1)
Out[43]: False

In [44]: check_if_ip_is_network(IP2)
Out[44]: True
```

Модуль tabulate

tabulate - это модуль, который позволяет красиво отображать табличные данные. Он не входит в стандартную библиотеку Python, поэтому tabulate нужно установить:

```
pip install tabulate
```

Модуль поддерживает такие типы табличных данных:

- список списков (в общем случае iterable of iterables)
- список словарей (или любой другой итерируемый объект со словарями). Ключи используются как имена столбцов
- словарь с итерируемыми объектами. Ключи используются как имена столбцов

Для генерации таблицы используется функция tabulate:

```
In [1]: from tabulate import tabulate

In [2]: sh_ip_int_br = [('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
   ...:  ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
   ...:  ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
   ...:  ('Loopback0', '10.1.1.1', 'up', 'up'),
   ...:  ('Loopback100', '100.0.0.1', 'up', 'up')]
   ...:

In [4]: print(tabulate(sh_ip_int_br))
---------------  ---------  --  --
FastEthernet0/0  15.0.15.1  up  up
FastEthernet0/1  10.0.12.1  up  up
FastEthernet0/2  10.0.13.1  up  up
Loopback0        10.1.1.1   up  up
Loopback100      100.0.0.1  up  up
---------------  ---------  --  --
```

headers

Параметр headers позволяет передавать дополнительный аргумент, в котором указаны имена столбцов:

```
In [8]: columns = ['Interface', 'IP', 'Status', 'Protocol']

In [9]: print(tabulate(sh_ip_int_br, headers=columns))
Interface        IP         Status    Protocol
---------------  ---------  --------  ----------
FastEthernet0/0  15.0.15.1  up        up
FastEthernet0/1  10.0.12.1  up        up
FastEthernet0/2  10.0.13.1  up        up
Loopback0        10.1.1.1   up        up
Loopback100      100.0.0.1  up        up
```

Достаточно часто первый набор данных - это заголовки. Тогда достаточно указать headers равным «firstrow»:

```
In [18]: data
Out[18]:
[('Interface', 'IP', 'Status', 'Protocol'),
 ('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]

In [20]: print(tabulate(data, headers='firstrow'))
Interface        IP         Status    Protocol
---------------  ---------  --------  ----------
FastEthernet0/0  15.0.15.1  up        up
FastEthernet0/1  10.0.12.1  up        up
FastEthernet0/2  10.0.13.1  up        up
Loopback0        10.1.1.1   up        up
Loopback100      100.0.0.1  up        up
```

Если данные в виде списка словарей, надо указать headers равным «keys»:

```
In [22]: list_of_dict
Out[22]:
[{'IP': '15.0.15.1',
  'Interface': 'FastEthernet0/0',
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '10.0.12.1',
  'Interface': 'FastEthernet0/1',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '10.0.13.1',
  'Interface': 'FastEthernet0/2',
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '10.1.1.1',
  'Interface': 'Loopback0',
  'Protocol': 'up',
  'Status': 'up'},
 {'IP': '100.0.0.1',
  'Interface': 'Loopback100',
  'Protocol': 'up',
  'Status': 'up'}]

In [23]: print(tabulate(list_of_dict, headers='keys'))
Interface        IP         Status    Protocol
---------------  ---------  --------  ----------
FastEthernet0/0  15.0.15.1  up        up
FastEthernet0/1  10.0.12.1  up        up
FastEthernet0/2  10.0.13.1  up        up
Loopback0        10.1.1.1   up        up
Loopback100      100.0.0.1  up        up
```

Отображение словаря:

```
In [6]: vlans = {"sw1": [10, 20, 30, 40], "sw2": [1, 2, 10], "sw3": [1, 2, 3, 4, 5, 10,␣
↪11, 12]}

In [7]: print(tabulate(vlans, headers="keys"))
  sw1    sw2    sw3
-----  -----  -----
   10      1      1
   20      2      2
   30     10      3
   40             4
                  5
                 10
                 11
                 12
```

Стиль таблицы

tabulate поддерживает разные стили отображения таблицы.

Формат grid:

```
In [24]: print(tabulate(list_of_dict, headers='keys', tablefmt="grid"))
+-----------------+-----------+----------+------------+
| Interface       | IP        | Status   | Protocol   |
+=================+===========+==========+============+
| FastEthernet0/0 | 15.0.15.1 | up       | up         |
+-----------------+-----------+----------+------------+
| FastEthernet0/1 | 10.0.12.1 | up       | up         |
+-----------------+-----------+----------+------------+
| FastEthernet0/2 | 10.0.13.1 | up       | up         |
+-----------------+-----------+----------+------------+
| Loopback0       | 10.1.1.1  | up       | up         |
+-----------------+-----------+----------+------------+
| Loopback100     | 100.0.0.1 | up       | up         |
+-----------------+-----------+----------+------------+
```

Таблица в формате Markdown:

```
In [25]: print(tabulate(list_of_dict, headers='keys', tablefmt='pipe'))
| Interface       | IP        | Status   | Protocol   |
|:----------------|:----------|:---------|:-----------|
| FastEthernet0/0 | 15.0.15.1 | up       | up         |
| FastEthernet0/1 | 10.0.12.1 | up       | up         |
| FastEthernet0/2 | 10.0.13.1 | up       | up         |
| Loopback0       | 10.1.1.1  | up       | up         |
| Loopback100     | 100.0.0.1 | up       | up         |
```

Таблица в формате HTML:

```
In [26]: print(tabulate(list_of_dict, headers='keys', tablefmt='html'))
<table>
<thead>
<tr><th>Interface      </th><th>IP       </th><th>Status  </th><th>Protocol  </th></tr>
</thead>
<tbody>
<tr><td>FastEthernet0/0</td><td>15.0.15.1</td><td>up      </td><td>up        </td></tr>
<tr><td>FastEthernet0/1</td><td>10.0.12.1</td><td>up      </td><td>up        </td></tr>
<tr><td>FastEthernet0/2</td><td>10.0.13.1</td><td>up      </td><td>up        </td></tr>
<tr><td>Loopback0      </td><td>10.1.1.1 </td><td>up      </td><td>up        </td></tr>
<tr><td>Loopback100    </td><td>100.0.0.1</td><td>up      </td><td>up        </td></tr>
</tbody>
</table>
```

Выравнивание столбцов

Можно указывать выравнивание для столбцов:

```
In [27]: print(tabulate(list_of_dict, headers='keys', tablefmt='pipe', stralign='center'))
|    Interface    |    IP     |  Status  |  Protocol  |
|:---------------:|:---------:|:--------:|:----------:|
| FastEthernet0/0 | 15.0.15.1 |    up    |     up     |
| FastEthernet0/1 | 10.0.12.1 |    up    |     up     |
| FastEthernet0/2 | 10.0.13.1 |    up    |     up     |
|    Loopback0    | 10.1.1.1  |    up    |     up     |
|   Loopback100   | 100.0.0.1 |    up    |     up     |
```

Обратите внимание, что тут не только столбцы отобразились с выравниванием по центру, но и соответственно изменился синтаксис Markdown.

Модуль pprint

Модуль pprint позволяет красиво отображать объекты Python. При этом сохраняется структура объекта и отображение, которое выводит pprint, можно использовать для создания объекта. Модуль pprint входит в стандартную библиотеку Python.

Самый простой вариант использования модуля - функция `pprint`. Например, словарь с вложенными словарями отобразится так:

```
In [6]: london_co = {'r1': {'hostname': 'london_r1', 'location': '21 New Globe Wal
   ...: k', 'vendor': 'Cisco', 'model': '4451', 'IOS': '15.4', 'IP': '10.255.0.1'}
   ...: , 'r2': {'hostname': 'london_r2', 'location': '21 New Globe Walk', 'vendor
   ...: ': 'Cisco', 'model': '4451', 'IOS': '15.4', 'IP': '10.255.0.2'}, 'sw1': {'
   ...: hostname': 'london_sw1', 'location': '21 New Globe Walk', 'vendor': 'Cisco
   ...: ', 'model': '3850', 'IOS': '3.6.XE', 'IP': '10.255.0.101'}}
   ...:

In [7]: from pprint import pprint

In [8]: pprint(london_co)
{'r1': {'IOS': '15.4',
        'IP': '10.255.0.1',
        'hostname': 'london_r1',
        'location': '21 New Globe Walk',
        'model': '4451',
        'vendor': 'Cisco'},
 'r2': {'IOS': '15.4',
        'IP': '10.255.0.2',
        'hostname': 'london_r2',
        'location': '21 New Globe Walk',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        'model': '4451',
        'vendor': 'Cisco'},
 'sw1': {'IOS': '3.6.XE',
         'IP': '10.255.0.101',
         'hostname': 'london_sw1',
         'location': '21 New Globe Walk',
         'model': '3850',
         'vendor': 'Cisco'}}
```

Список списков:

```
In [13]: interfaces = [['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up
    ...: '], ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'], ['FastE
    ...: thernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]
    ...:

In [14]: pprint(interfaces)
[['FastEthernet0/0', '15.0.15.1', 'YES', 'manual', 'up', 'up'],
 ['FastEthernet0/1', '10.0.1.1', 'YES', 'manual', 'up', 'up'],
 ['FastEthernet0/2', '10.0.2.1', 'YES', 'manual', 'up', 'down']]
```

Строка:

```
In [18]: tunnel
Out[18]: '\ninterface Tunnel0\n ip address 10.10.10.1 255.255.255.0\n ip mtu 1416\n ip␣
↪ospf hello-interval 5\n tunnel source FastEthernet1/0\n tunnel protection ipsec profile␣
↪DMVPN\n'

In [19]: pprint(tunnel)
('\n'
 'interface Tunnel0\n'
 ' ip address 10.10.10.1 255.255.255.0\n'
 ' ip mtu 1416\n'
 ' ip ospf hello-interval 5\n'
 ' tunnel source FastEthernet1/0\n'
 ' tunnel protection ipsec profile DMVPN\n')
```

Ограничение вложенности

У функции pprint есть дополнительный параметр depth, который позволяет ограничивать глубину отображения структуры данных.

Например, есть такой словарь:

```
In [3]: result = {
   ...:  'interface Tunnel0': [' ip unnumbered Loopback0',
   ...:   ' tunnel mode mpls traffic-eng',
   ...:   ' tunnel destination 10.2.2.2',
   ...:   ' tunnel mpls traffic-eng priority 7 7',
   ...:   ' tunnel mpls traffic-eng bandwidth 5000',
   ...:   ' tunnel mpls traffic-eng path-option 10 dynamic',
   ...:   ' no routing dynamic'],
   ...:  'ip access-list standard LDP': [' deny   10.0.0.0 0.0.255.255',
   ...:   ' permit 10.0.0.0 0.255.255.255'],
   ...:  'router bgp 100': {' address-family vpnv4': ['  neighbor 10.2.2.2 activat
   ...: e',
   ...:    '  neighbor 10.2.2.2 send-community both',
   ...:    '  exit-address-family'],
   ...:   ' bgp bestpath igp-metric ignore': [],
   ...:   ' bgp log-neighbor-changes': [],
   ...:   ' neighbor 10.2.2.2 next-hop-self': [],
   ...:   ' neighbor 10.2.2.2 remote-as 100': [],
   ...:   ' neighbor 10.2.2.2 update-source Loopback0': [],
   ...:   ' neighbor 10.4.4.4 remote-as 40': []},
   ...:  'router ospf 1': [' mpls ldp autoconfig area 0',
   ...:   ' mpls traffic-eng router-id Loopback0',
   ...:   ' mpls traffic-eng area 0',
   ...:   ' network 10.0.0.0 0.255.255.255 area 0']}
   ...:
```

Можно отобразить только ключи, указав глубину равной 1:

```
In [5]: pprint(result, depth=1)
{'interface Tunnel0': [...],
 'ip access-list standard LDP': [...],
 'router bgp 100': {...},
 'router ospf 1': [...]}
```

Скрытые уровни вложенности заменяются `...`.

Если указать глубину равной 2, отобразится следующий уровень:

```
In [6]: pprint(result, depth=2)
{'interface Tunnel0': [' ip unnumbered Loopback0',
                       ' tunnel mode mpls traffic-eng',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                        ' tunnel destination 10.2.2.2',
                        ' tunnel mpls traffic-eng priority 7 7',
                        ' tunnel mpls traffic-eng bandwidth 5000',
                        ' tunnel mpls traffic-eng path-option 10 dynamic',
                        ' no routing dynamic'],
 'ip access-list standard LDP': [' deny   10.0.0.0 0.0.255.255',
                                 ' permit 10.0.0.0 0.255.255.255'],
 'router bgp 100': {' address-family vpnv4': [...],
                    ' bgp bestpath igp-metric ignore': [],
                    ' bgp log-neighbor-changes': [],
                    ' neighbor 10.2.2.2 next-hop-self': [],
                    ' neighbor 10.2.2.2 remote-as 100': [],
                    ' neighbor 10.2.2.2 update-source Loopback0': [],
                    ' neighbor 10.4.4.4 remote-as 40': []},
 'router ospf 1': [' mpls ldp autoconfig area 0',
                   ' mpls traffic-eng router-id Loopback0',
                   ' mpls traffic-eng area 0',
                   ' network 10.0.0.0 0.255.255.255 area 0']}
```

pformat

pformat - это функция, которая отображает результат в виде строки. Ее удобно использовать, если необходимо записать структуру данных в какой-то файл, например, для логирования.

```
In [15]: from pprint import pformat

In [16]: formatted_result = pformat(result)

In [17]: print(formatted_result)
{'interface Tunnel0': [' ip unnumbered Loopback0',
                        ' tunnel mode mpls traffic-eng',
                        ' tunnel destination 10.2.2.2',
                        ' tunnel mpls traffic-eng priority 7 7',
                        ' tunnel mpls traffic-eng bandwidth 5000',
                        ' tunnel mpls traffic-eng path-option 10 dynamic',
                        ' no routing dynamic'],
 'ip access-list standard LDP': [' deny   10.0.0.0 0.0.255.255',
                                 ' permit 10.0.0.0 0.255.255.255'],
 'router bgp 100': {' address-family vpnv4': ['  neighbor 10.2.2.2 activate',
                                              '  neighbor 10.2.2.2 '
                                              'send-community both',
                                              '  exit-address-family'],
                    ' bgp bestpath igp-metric ignore': [],
                    ' bgp log-neighbor-changes': [],
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    ' neighbor 10.2.2.2 next-hop-self': [],
                    ' neighbor 10.2.2.2 remote-as 100': [],
                    ' neighbor 10.2.2.2 update-source Loopback0': [],
                    ' neighbor 10.4.4.4 remote-as 40': []},
 'router ospf 1': [' mpls ldp autoconfig area 0',
                   ' mpls traffic-eng router-id Loopback0',
                   ' mpls traffic-eng area 0',
                   ' network 10.0.0.0 0.255.255.255 area 0']}
```

Сортировка словарей

pprint сортирует словари при выводе, что не всегда удобно. Начиная с версии Python 3.8, появилась возможность отключить сортировку:

```
In [3]: r1 = {"ios": "16.4", "hostname": "R1", "ip": "10.1.1.1", "vendor": "Cisco IOS"}

In [4]: pprint(r1)
{'hostname': 'R1', 'ios': '16.4', 'ip': '10.1.1.1', 'vendor': 'Cisco IOS'}

In [5]: pprint(r1, sort_dicts=False)
{'ios': '16.4', 'hostname': 'R1', 'ip': '10.1.1.1', 'vendor': 'Cisco IOS'}
```

Дополнительные материалы

pprint

- pprint — Data pretty printer
- PyMOTW. pprint — Pretty-Print Data Structures

tabulate

Документация tabulate

Статьи от автора tabulate:

- Pretty printing tables in Python
- Tabulate 0.7.1 with LaTeX & MediaWiki tables

Stack Overflow:

- Printing Lists as Tabular Data. Обратите внимание на ответ - в нём указаны другие аналоги tabulate.

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 12.1

Создать функцию ping_ip_addresses, которая проверяет пингуются ли IP-адреса. Функция ожидает как аргумент список IP-адресов.

Функция должна возвращать кортеж с двумя списками:

- список доступных IP-адресов
- список недоступных IP-адресов

Для проверки доступности IP-адреса, используйте команду ping.

Ограничение: Все задания надо выполнять используя только пройденные темы.

Задание 12.2

Функция ping_ip_addresses из задания 12.1 принимает только список адресов, но было бы удобно иметь возможность указывать адреса с помощью диапазона, например, 192.168.100.1-10.

В этом задании необходимо создать функцию convert_ranges_to_ip_list, которая конвертирует список IP-адресов в разных форматах в список, где каждый IP-адрес указан отдельно.

Функция ожидает как аргумент список IP-адресов и/или диапазонов IP-адресов.

Элементы списка могут быть в формате:

- 10.1.1.1
- 10.1.1.1-10.1.1.10
- 10.1.1.1-10

Если адрес указан в виде диапазона, надо развернуть диапазон в отдельные адреса, включая последний адрес диапазона. Для упрощения задачи, можно считать, что в диапазоне всегда меняется только последний октет адреса.

Функция возвращает список IP-адресов.

Например, если передать функции convert_ranges_to_ip_list такой список:

```
['8.8.4.4', '1.1.1.1-3', '172.21.41.128-172.21.41.132']
```

Функция должна вернуть такой список:

```
['8.8.4.4', '1.1.1.1', '1.1.1.2', '1.1.1.3', '172.21.41.128',
 '172.21.41.129', '172.21.41.130', '172.21.41.131', '172.21.41.132']
```

Задание 12.3

Создать функцию print_ip_table, которая отображает таблицу доступных и недоступных IP-адресов.

Функция ожидает как аргументы два списка:

- список доступных IP-адресов
- список недоступных IP-адресов

Результат работы функции - вывод на стандартный поток вывода таблицы вида:

```
Reachable    Unreachable
-----------  -------------
10.1.1.1     10.1.1.7
10.1.1.2     10.1.1.8
             10.1.1.9
```

13. Итераторы, итерируемые объекты и генераторы

В этом разделе рассматриваются:

- итерируемые объекты (iterable)
- итераторы (iterator)
- генераторные выражения (generator expression)

Итерируемый объект

Итерация - это общий термин, который описывает процедуру взятия элементов чего-то по очереди.

В более общем смысле, это последовательность инструкций, которая повторяется определенное количество раз или до выполнения указанного условия.

Итерируемый объект (iterable) - это объект, который способен возвращать элементы по одному. Кроме того, это объект, из которого можно получить итератор.

Примеры итерируемых объектов:

- все последовательности: список, строка, кортеж
- словари
- файлы

В Python за получение итератора отвечает функция iter():

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

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

Функция `iter()` отработает на любом объекте, у которого есть метод `__iter__` или метод `__getitem__`.

Метод `__iter__` возвращает итератор. Если этого метода нет, функция iter() проверяет, нет ли метода `__getitem__` - метода, который позволяет получать элементы по индексу.

Если метод `__getitem__` есть, возвращается итератор, который проходится по элементам, используя индекс (начиная с 0).

На практике использование метода `__getitem__` означает, что все последовательности элементов - это итерируемые объекты. Например, список, кортеж, строка. Хотя у этих типов данных есть и метод `__iter__`.

Итераторы

Итератор (iterator) - это объект, который возвращает свои элементы по одному за раз.

С точки зрения Python - это любой объект, у которого есть метод __next__. Этот метод возвращает следующий элемент, если он есть, или возвращает исключение StopIteration, когда элементы закончились.

Кроме того, итератор запоминает, на каком объекте он остановился в последнюю итерацию.

В Python у каждого итератора присутствует метод __iter__ - то есть, любой итератор является итерируемым объектом. Этот метод просто возвращает сам итератор.

Пример создания итератора из списка:

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

In [4]: i = iter(numbers)
```

Теперь можно использовать функцию next(), которая вызывает метод __next__, чтобы взять следующий элемент:

```
In [5]: next(i)
Out[5]: 1

In [6]: next(i)
Out[6]: 2

In [7]: next(i)
Out[7]: 3

In [8]: next(i)
---------------------------------------------------------------
StopIteration              Traceback (most recent call last)
<ipython-input-8-bed2471d02c1> in <module>()
----> 1 next(i)

StopIteration:
```

После того, как элементы закончились, возвращается исключение StopIteration.

Для того, чтобы итератор снова начал возвращать элементы, его надо заново создать.

Аналогичные действия выполняются, когда цикл for проходится по списку:

```
In [9]: for item in numbers:
   ...:     print(item)
   ...:
1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
2
3
```

Когда мы перебираем элементы списка, к списку сначала применяется функция iter(), чтобы создать итератор, а затем вызывается его метод __next__ до тех пор, пока не возникнет исключение StopIteration.

Итераторы полезны тем, что они отдают элементы по одному. Например, при работе с файлом это полезно тем, что в памяти будет находиться не весь файл, а только одна строка файла.

Файл как итератор

Один из самых распространенных примеров итератора - файл.

Файл r1.txt:

```
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Если открыть файл обычной функцией open, мы получим объект, который представляет файл:

```
In [10]: f = open('r1.txt')
```

Этот объект является итератором, что можно проверить, вызвав метод __next__:

```
In [11]: f.__next__()
Out[11]: '!\n'

In [12]: f.__next__()
Out[12]: 'service timestamps debug datetime msec localtime show-timezone year\n'
```

Аналогичным образом можно перебирать строки в цикле for:

```
In [13]: for line in f:
    ...:     print(line.rstrip())
    ...:
service timestamps log datetime msec localtime show-timezone year
```

(continues on next page)

(продолжение с предыдущей страницы)

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

При работе с файлами, использование файла как итератора не просто позволяет перебирать файл построчно - в каждую итерацию загружена только одна строка. Это очень важно при работе с большими файлами на тысячи и сотни тысяч строк, например, с лог-файлами.

Поэтому при работе с файлами в Python чаще всего используется конструкция вида:

```
In [14]: with open('r1.txt') as f:
    ...:     for line in f:
    ...:         print(line.rstrip())
    ...:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Генератор (generator)

Генераторы - это специальный класс функций, который позволяет легко создавать свои итераторы. В отличие от обычных функций, генератор не просто возвращает значение и завершает работу, а возвращает итератор, который отдает элементы по одному.

Обычная функция завершает работу, если:

- встретилось выражение `return`
- закончился код функции (это срабатывает как выражение `return None`)
- возникло исключение

После выполнения функции управление возвращается, и программа выполняется дальше. Все аргументы, которые передавались в функцию, локальные переменные, все это теряется. Остается только результат, который вернула функция.

Функция может возвращать список элементов, несколько объектов или возвращать разные результаты в зависимости от аргументов, но она всегда возвращает какой-то один результат.

Генератор же генерирует значения. При этом значения возвращаются по запросу, и после возврата одного значения выполнение функции-генератора приостанавливается до запроса следующего значения. Между запросами генератор сохраняет свое состояние.

Python позволяет создавать генераторы двумя способами:

- генераторное выражение
- функция-генератор

Ниже пример генераторного выражения, а по функциям-генераторам - отдельная заметка

generator expression (генераторное выражение)

Генераторное выражение использует такой же синтаксис, как list comprehensions, но возвращает итератор, а не список.

Генераторное выражение выглядит точно так же, как list comprehensions, но используются круглые скобки:

```
In [1]: genexpr = (x**2 for x in range(10000))

In [2]: genexpr
Out[2]: <generator object <genexpr> at 0xb571ec8c>

In [3]: next(genexpr)
Out[3]: 0

In [4]: next(genexpr)
Out[4]: 1

In [5]: next(genexpr)
Out[5]: 4
```

Обратите внимание, что это не tuple comprehensions, а генераторное выражение.

Оно полезно в том случае, когда надо работать с большим итерируемым объектом или бесконечным итератором.

Дополнительные материалы

Документация Python:

- Sequence types
- Iterator types
- Functional Programming HOWTO

Статьи:

- Iterables vs. Iterators vs. Generators

4

III. Регулярные выражения

Регулярное выражение - это последовательность из обычных и специальных символов. Эта последовательность задает шаблон, который позже используется для поиска подстрок.

При работе с сетевым оборудованием регулярные выражения могут использоваться, например, для:

- получения информации из вывода команд show
- отбора части строк из вывода команд show, которые совпадают с шаблоном
- проверки, есть ли определенные настройки в конфигурации

Несколько примеров:

- обработав вывод команды show version, можно собрать информацию про версию ОС и uptime оборудования.
- получить из log-файла те строки, которые соответствуют шаблону.
- получить из конфигурации те интерфейсы, на которых нет описания (description)

Кроме того, в самом сетевом оборудовании регулярные выражения можно использовать для фильтрации вывода любых команд show.

В целом, использование регулярных выражений будет связано с получением части текста из большого вывода. Но это не единственное, в чем они могут пригодиться. Например, с помощью регулярных выражений можно выполнять замены в строках или разделение строки на части.

Эти области применения пересекаются с методами, которые применяются к строкам. И, если задача понятно и просто решается с помощью методов строк, лучше использовать их. Такой код будет проще понять и, кроме того, методы строк быстрее работают.

Но методы строк могут справиться не со всеми задачами или могут сильно усложнить решение задачи. В этом случае могут помочь регулярные выражения.

14. Синтаксис регулярных выражений

Синтаксис регулярных выражений

В Python для работы с регулярными выражениями используется модуль re. Соответственно, для начала работы с регулярными выражениями надо его импортировать.

В Python некоторые символы строки надо экранировать, чтобы они воспринимались правильно. К таким символам относится, например, \. Чтобы написать правильно строку, в которой находятся два символа \\, оба символа надо экранировать и в итоге получится строка вида: \\\\data. Вместо этого, можно использовать raw-строку и тогда каждый символ будет восприниматься как написано. Raw-строки отличаются от обычных тем, что при создании строки, в начале пишется буква r:

```
In [3]: r"\\data"
Out[3]: '\\\\data'
```

Так как в регулярных выражениях постоянно используется \, всегда используйте raw-строки для описания регулярных выражений. Некоторые выражения правильно отработают и без raw-строк, но, использование raw-строк для регулярных выражений это хороший тон.

В этом разделе для всех примеров будет использоваться функция search. А в следующем подразделе будут рассматриваться остальные функции модуля re.

Синтаксис функции search такой:

```
match = re.search(regex, string)
```

У функции search два обязательных параметра:

- regex - регулярное выражение
- string - строка, в которой ищется совпадение

Если совпадение было найдено, функция вернет специальный объект Match. Если же совпадения не было, функция вернет None.

При этом особенность функции search в том, что она ищет только первое совпадение. То есть, если в строке есть несколько подстрок, которые соответствуют регулярному выражению, search вернет только первое найденное совпадение.

Чтобы получить представление о регулярных выражениях, рассмотрим несколько примеров.

Самый простой пример регулярного выражения - подстрока:

```
In [1]: import re

In [2]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [3]: match = re.search(r'MTU', int_line)
```

Примечание: Технически тут не нужна raw-строка, так как тут регулярное выражение это обычная подстрока MTU, но лучше сразу привыкать к тому, что в регулярных выражениях надо использовать raw строки.

В этом примере:

- сначала импортируется модуль re
- затем идет пример строки int_line
- и в 3 строке функции search передается выражение, которое надо искать, и строка int_line, в которой ищется совпадение

В данном случае мы ищем, есть ли подстрока „MTU“ в строке int_line. Если она есть, в переменной match будет находиться специальный объект Match:

```
In [4]: print(match)
<_sre.SRE_Match object; span=(2, 5), match='MTU'>
```

У объекта Match есть несколько методов, которые позволяют получать разную информацию о полученном совпадении. Например, метод group показывает, что в строке совпало с описанным выражением.

В данном случае это подстрока „MTU“:

```
In [5]: match.group()
Out[5]: 'MTU'
```

Если совпадения не было, в переменной match будет значение None:

```
In [6]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [7]: match = re.search(r'MU', int_line)

In [8]: print(match)
None
```

Полностью возможности регулярных выражений проявляются при использовании специальных символов. Например, символ \d означает цифру, а + означает повторение предыдущего символа один или более раз. Если их совместить \d+, получится выражение, которое означает одну или более цифр.

Используя это выражение, можно получить часть строки, в которой описана пропускная способность:

```
In [9]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [10]: match = re.search(r'BW \d+', int_line)

In [11]: match.group()
Out[11]: 'BW 10000'
```

Особенно полезны регулярные выражения в получении определенных подстрок из строки. Например, необходимо получить VLAN, MAC и порты из вывода такого лог-сообщения:

```
In [12]: log2 = 'Oct  3 12:49:15.941: %SW_MATM-4-MACFLAP_NOTIF: Host f04d.a206.7fd6 in␣
↪vlan 1 is flapping between port Gi0/5 and port Gi0/16'
```

Это можно сделать с помощью такого регулярного выражения:

```
In [13]: re.search(r'Host (\S+) in vlan (\d+) is flapping between port (\S+) and port (\
↪S+)', log2).groups()
Out[13]: ('f04d.a206.7fd6', '1', 'Gi0/5', 'Gi0/16')
```

Метод groups возвращает только те части исходной строки, которые попали в круглые скобки. Таким образом, заключив часть выражения в скобки, можно указать, какие части строки надо запомнить.

Выражение \d+ уже использовалось ранее - оно описывает одну или более цифр. А выражение \S+ описывает все символы, кроме whitespace (пробел, таб и другие).

В следующих подразделах рассматриваются специальные символы, которые используются в регулярных выражениях.

Примечание: Если вы знаете, что означают специальные символы в регулярных выражениях, можно пропустить следующий подраздел и сразу переключиться на подраздел о модуле re.

Наборы символов

В Python есть специальные обозначения для наборов символов:

- \d - любая цифра
- \D - любое нечисловое значение
- \s - пробельные символы
- \S - все, кроме пробельных символов
- \w - любая буква, цифра или нижнее подчеркивание

- \W - все, кроме букв, цифр или нижнего подчеркивания

Примечание: Это не все наборы символов, которые поддерживает Python. Подробнее смотрите в документации.

Наборы символов позволяют писать более короткие выражения без необходимости перечислять все нужные символы.

Например, получим время из строки лог-файла:

```
In [1]: log = '*Jul  7 06:15:18.695: %LINEPROTO-5-UPDOWN: Line protocol on Interface␣
↪Ethernet0/3, changed state to down'

In [2]: re.search(r'\d\d:\d\d:\d\d', log).group()
Out[2]: '06:15:18'
```

Выражение \d\d:\d\d:\d\d описывает 3 пары чисел, разделенных двоеточиями.

Получение MAC-адреса из лог-сообщения:

```
In [3]: log2 = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [4]: re.search(r'\w\w\w\w\.\w\w\w\w\.\w\w\w\w', log2).group()
Out[4]: 'f03a.b216.7ad7'
```

Выражение \w\w\w\w\.\w\w\w\w\.\w\w\w\w описывает 12 букв или цифр, которые разделены на три группы по четыре символа точками.

Группы символов очень удобны, но пока что приходится вручную указывать повторение символа. В следующем подразделе рассматриваются символы повторения, которые упростят описание выражений.

Символы повторения

- regex+ - одно или более повторений предшествующего элемента
- regex* - ноль или более повторений предшествующего элемента
- regex? - ноль или одно повторение предшествующего элемента
- regex{n} - ровно n повторений предшествующего элемента
- regex{n,m} - от n до m повторений предшествующего элемента
- regex{n,} - n или более повторений предшествующего элемента

+

Плюс указывает, что предыдущее выражение может повторяться сколько угодно раз, но, как минимум, один раз.

Например, тут повторение относится к букве a:

```
In [1]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [2]: re.search(r'a+', line).group()
Out[2]: 'aa'
```

А в этом выражении повторяется строка „a1“:

```
In [3]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [4]: re.search(r'(a1)+', line).group()
Out[4]: 'a1a1'

В выражении ``(a1)+`` скобки используются для того, чтобы указать,
что повторение относится к последовательности символов 'a1'.
```

IP-адрес можно описать выражением `\d+\.\d+\.\d+\.\d+`. Тут плюс используется, чтобы указать, что цифр может быть несколько. А также встречается выражение `\.`.

Оно необходимо из-за того, что точка является специальным символом (она обозначает любой символ). И чтобы указать, что нас интересует именно точка, надо ее экранировать - поместить перед точкой обратный слеш.

Используя это выражение, можно получить IP-адрес из строки sh_ip_int_br:

```
In [5]: sh_ip_int_br = 'Ethernet0/1    192.168.200.1   YES NVRAM  up          up'

In [6]: re.search(r'\d+\.\d+\.\d+\.\d+', sh_ip_int_br).group()
Out[6]: '192.168.200.1'
```

Еще один пример выражения: `\d+\s+\S+` - оно описывает строку, в которой идут сначала цифры, после них пробельные символы, а затем непробельные символы (все, кроме пробела, таба и других подобных символов). С его помощью можно получить VLAN и MAC-адрес из строки:

```
In [7]: line = '1500     aab1.a1a1.a5d3    FastEthernet0/1'

In [8]: re.search(r'\d+\s+\S+', line).group()
Out[8]: '1500     aab1.a1a1.a5d3'
```

*

Звездочка указывает, что предыдущее выражение может повторяться 0 или более раз.

Например, если звездочка стоит после символа, она означает повторение этого символа.

Выражение ba* означает b, а затем ноль или более повторений a:

```
In [9]: line = '100     a011.baaa.a5d3    FastEthernet0/1'

In [10]: re.search(r'ba*', line).group()
Out[10]: 'baaa'
```

Если в строке line до подстроки baaa встретится b, то совпадением будет b:

```
In [11]: line = '100     ab11.baaa.a5d3    FastEthernet0/1'

In [12]: re.search(r'ba*', line).group()
Out[12]: 'b'
```

Допустим, необходимо написать регулярное выражение, которое описывает электронные адреса в двух форматах: user@example.com и user.test@example.com. То есть, в левой части адреса может быть или одно слово, или два слова, разделенные точкой.

Первый вариант на примере адреса без точки:

```
In [13]: email1 = 'user1@gmail.com'
```

Этот адрес можно описать таким выражением \w+@\w+\.\w+:

```
In [14]: re.search(r'\w+@\w+\.\w+', email1).group()
Out[14]: 'user1@gmail.com'
```

Но такое выражение не подходит для электронного адреса с точкой:

```
In [15]: email2 = 'user2.test@gmail.com'

In [16]: re.search(r'\w+@\w+\.\w+', email2).group()
Out[16]: 'test@gmail.com'
```

Регулярное выражение для адреса с точкой:

```
In [17]: re.search(r'\w+\.\w+@\w+\.\w+', email2).group()
Out[17]: 'user2.test@gmail.com'
```

Чтобы описать оба варианта адресов, надо указать, что точка в адресе опциональна:

```
'\w+\.*\w+@\w+\.\w+'
```

Такое регулярное выражение описывает оба варианта:

```
In [18]: email1 = 'user1@gmail.com'

In [19]: email2 = 'user2.test@gmail.com'

In [20]: re.search(r'\w+\.*\w+@\w+\.\w+', email1).group()
Out[20]: 'user1@gmail.com'

In [21]: re.search(r'\w+\.*\w+@\w+\.\w+', email2).group()
Out[21]: 'user2.test@gmail.com'
```

?

В последнем примере регулярное выражение указывает, что точка необязательна, но в то же время определяет, что она может появиться много раз.

В этой ситуации логичней использовать знак вопроса. Он обозначает ноль или одно повторение предыдущего выражения или символа. Теперь регулярное выражение выглядит так `\w+\.?\w+@\w+\.\w+`:

```
In [22]: mail_log = ['Jun 18 14:10:35 client-ip=154.10.180.10 from=user1@gmail.com,␣
↪size=551',
    ...:             'Jun 18 14:11:05 client-ip=150.10.180.10 from=user2.test@gmail.com,␣
↪size=768']

In [23]: for message in mail_log:
     ...:     match = re.search(r'\w+\.?\w+@\w+\.\w+', message)
     ...:     if match:
     ...:         print("Found email: ", match.group())
     ...:
Found email:  user1@gmail.com
Found email:  user2.test@gmail.com
```

{n}

С помощью фигурных скобок можно указать, сколько раз должно повторяться предшествующее выражение.

Например, выражение `\w{4}\.\w{4}\.\w{4}` описывает 12 букв или цифр, которые разделены на три группы по четыре символа точками. Таким образом можно получить MAC-адрес:

```
In [24]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [25]: re.search(r'\w{4}\.\w{4}\.\w{4}', line).group()
Out[25]: 'aab1.a1a1.a5d3'
```

В фигурных скобках можно указывать и диапазон повторений. Например, попробуем получить все номера VLAN из строки mac_table:

```
In [26]: mac_table = '''
    ...: sw1#sh mac address-table
    ...:           Mac Address Table
    ...: -------------------------------------------
    ...:
    ...: Vlan    Mac Address       Type        Ports
    ...: ----    -----------       --------    -----
    ...:  100    a1b2.ac10.7000    DYNAMIC     Gi0/1
    ...:  200    a0d4.cb20.7000    DYNAMIC     Gi0/2
    ...:  300    acb4.cd30.7000    DYNAMIC     Gi0/3
    ...: 1100    a2bb.ec40.7000    DYNAMIC     Gi0/4
    ...:  500    aa4b.c550.7000    DYNAMIC     Gi0/5
    ...: 1200    a1bb.1c60.7000    DYNAMIC     Gi0/6
    ...: 1300    aa0b.cc70.7000    DYNAMIC     Gi0/7
    ...: '''
```

Так как search ищет только первое совпадение, в выражение `\d{1,4}` попадет номер VLAN:

```
In [27]: for line in mac_table.split('\n'):
    ...:     match = re.search(r'\d{1,4}', line)
    ...:     if match:
    ...:         print('VLAN: ', match.group())
    ...:
VLAN:  1
VLAN:  100
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

Выражение `\d{1,4}` описывает от одной до четырех цифр.

Обратите внимание, что в выводе команды с оборудования нет VLAN с номером 1. При этом регулярное выражение получило откуда-то число 1. Цифра 1 попала в вывод из имени хоста в строке `sw1#sh mac address-table`.

Чтобы исправить это, достаточно дополнить выражение и указать, что после цифр должен идти хотя бы один пробел:

```
In [28]: for line in mac_table.split('\n'):
    ...:     match = re.search(r'\d{1,4} +', line)
    ...:     if match:
    ...:         print('VLAN: ', match.group())
    ...:
VLAN:  100
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

Специальные символы

- `.` - любой символ, кроме символа новой строки
- `^` - начало строки
- `$` - конец строки
- `[abc]` - любой символ в скобках
- `[^abc]` - любой символ, кроме тех, что в скобках
- `a|b` - элемент a или b
- `(regex)` - выражение рассматривается как один элемент. Кроме того, подстрока, которая совпала с выражением, запоминается

.

Точка обозначает любой символ.

Чаще всего точка используется с символами повторения + и *, чтобы указать, что между определенными выражениями могут находиться любые символы.

Например, с помощью выражения `Interface.+Port ID.+` можно описать строку с интерфейсами в выводе sh cdp neighbors detail:

```
In [1]: cdp = '''
   ...: SW1#show cdp neighbors detail
   ...: -------------------------
   ...: Device ID: SW2
   ...: Entry address(es):
   ...:   IP address: 10.1.1.2
   ...: Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...: Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
   ...: Holdtime : 164 sec
   ...: '''

In [2]: re.search(r'Interface.+Port ID.+', cdp).group()
Out[2]: 'Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1'
```

В результат попала только одна строка, так как точка обозначает любой символ, кроме символа перевода строки. Кроме того, символы повторения + и * по умолчанию захватывают максимально длинную строку. Этот аспект рассматривается в подразделе «Жадность символов повторения».

^

Символ ^ означает начало строки. Выражению ^\d+ соответствует подстрока:

```
In [3]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [4]: re.search(r'^\d+', line).group()
Out[4]: '100'
```

Символы с начала строки и до решетки (включая решетку):

```
In [5]: prompt = 'SW1#show cdp neighbors detail'

In [6]: re.search(r'^.+#', prompt).group()
Out[6]: 'SW1#'
```

$

Символ $ обозначает конец строки.

Выражение \S+$ описывает любые символы, кроме whitespace в конце строки:

```
In [7]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [8]: re.search(r'\S+$', line).group()
Out[8]: 'FastEthernet0/1'
```

[]

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

```
In [9]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [10]: re.search(r'[Ff]ast', line).group()
Out[10]: 'Fast'

In [11]: re.search(r'[Ff]ast[Ee]thernet', line).group()
Out[11]: 'FastEthernet'
```

С помощью квадратных скобок можно указать, какие символы могут встречаться на конкретной позиции. Например, выражение `^.+[>#]` описывает символы с начала строки и до решетки или знака больше (включая их). С помощью такого выражения можно получить имя устройства:

```
In [12]: commands = ['SW1#show cdp neighbors detail',
    ...:             'SW1>sh ip int br',
    ...:             'r1-london-core# sh ip route']
    ...:

In [13]: for line in commands:
    ...:     match = re.search(r'^.+[>#]', line)
    ...:     if match:
    ...:         print(match.group())
    ...:
SW1#
SW1>
r1-london-core#
```

В квадратных скобках можно указывать диапазоны символов. Например, таким образом можно указать, что нас интересует любая цифра от 0 до 9:

```
In [14]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [15]: re.search(r'[0-9]+', line).group()
Out[15]: '100'
```

Аналогичным образом можно указать буквы:

```
In [16]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [17]: re.search(r'[a-z]+', line).group()
Out[17]: 'aa'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [18]: re.search(r'[A-Z]+', line).group()
Out[18]: 'F'
```

В квадратных скобках можно указывать несколько диапазонов:

```
In [19]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [20]: re.search(r'[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+', line).group()
Out[20]: 'aa12.35fe.a5d3'
```

Выражение `[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+` описывает три группы символов, разделенных точкой. Символами в каждой группе могут быть буквы a-f или цифры 0-9. Это выражение описывает MAC-адрес.

Еще одна особенность квадратных скобок - специальные символы внутри квадратных скобок теряют свое специальное значение и обозначают просто символ. Например, точка внутри квадратных скобок будет обозначать точку, а не любой символ.

Выражение `[a-f0-9]+[./][a-f0-9]+` описывает три группы символов:

1. буквы a-f или цифры от 0 до 9
2. точка или слеш
3. буквы a-f или цифры от 0 до 9

Для строки line совпадением будет такая подстрока:

```
In [21]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [22]: re.search(r'[a-f0-9]+[./][a-f0-9]+', line).group()
Out[22]: 'aa12.35fe'
```

Если после открывающейся квадратной скобки указан символ ^, совпадением будет любой символ, кроме указанных в скобках:

```
In [23]: line = 'FastEthernet0/0    15.0.15.1       YES manual up         up'

In [24]: re.search(r'[^a-zA-Z]+', line).group()
Out[24]: '0/0    15.0.15.1       '
```

В данном случае выражение описывает все, кроме букв.

|

Вертикальная черта работает как „или“:

```
In [25]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [26]: re.search(r'Fast|0/1', line).group()
Out[26]: 'Fast'
```

Обратите внимание на то, как срабатывает | - Fast и 0/1 воспринимаются как целое выражение. То есть, в итоге выражение означает, что мы ищем Fast или 0/1, а не то, что мы ищем Fas, затем t или 0 и 0/1.

()

Скобки используются для группировки выражений. Как и в математических выражениях, с помощью скобок можно указать, к каким элементам применяется операция.

Например, выражение `[0-9]([a-f]|[0-9])[0-9]` описывает три символа: цифра, потом буква или цифра и цифра:

```
In [27]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [28]: re.search(r'[0-9]([a-f]|[0-9])[0-9]', line).group()
Out[28]: '100'
```

Скобки позволяют указывать, какое выражение является одним целым. Это особенно полезно при использовании символов повторения:

```
In [29]: line = 'FastEthernet0/0    15.0.15.1       YES manual up         up'

In [30]: re.search(r'([0-9]+\.)+[0-9]+', line).group()
Out[30]: '15.0.15.1'
```

Скобки позволяют не только группировать выражения. Строка, которая совпала с выражением в скобках, запоминается. Ее можно получить отдельно с помощью специальных методов groups и group(n). Это рассматривается в подразделе «Группировка выражений».

Жадность символов повторения

По умолчанию символы повторения в регулярных выражениях жадные (greedy). Это значит, что результирующая подстрока, которая соответствует шаблону, будет наиболее длинной.

Пример жадного поведения:

```
In [1]: import re

In [2]: line = '<text line> some text>'

In [3]: match = re.search(r'<.*>', 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(r'<.*?>', line)

In [7]: match.group()
Out[7]: '<text line>'
```

Зачастую жадность наоборот полезна. Например, без отключения жадности последнего плюса, выражение \d+\s+\S+ описывает такую строку:

```
In [8]: line = '1500     aab1.a1a1.a5d3    FastEthernet0/1'

In [9]: re.search(r'\d+\s+\S+', line).group()
Out[9]: '1500     aab1.a1a1.a5d3'
```

Символ \S обозначает все, кроме пробельных символов. Поэтому выражение \S+ с жадным символом повторения описывает максимально длинную строку до первого whitespace символа. В данном случае - до первого пробела.

Если отключить жадность, результат будет таким:

```
In [10]: re.search(r'\d+\s+\S+?', line).group()
Out[10]: '1500     a'
```

Группировка выражений

Группировка выражений указывает, что последовательность символов надо рассматривать как одно целое. Однако это не единственное преимущество группировки.

Кроме этого, с помощью групп можно получать только определенную часть строки, которая была описана выражением. Это очень полезно в ситуациях, когда надо описать строку достаточно подробно, чтобы отобрать нужные строки, но в то же время из самой строки надо получить только определенное значение.

Например, из log-файла надо отобрать строки, в которых встречается «%SW_MATM-4-MACFLAP_NOTIF», а затем из каждой такой строки получить MAC-адрес, VLAN и интерфейсы. В этом случае регулярное выражение просто должно описывать строку, а все части строки, которые надо получить в результате, просто заключаются в скобки.

В Python есть два варианта использования групп:

- Нумерованные группы
- Именованные группы

Нумерованные группы

Группа определяется помещением выражения в круглые скобки `()`.

Внутри выражения группы нумеруются слева направо, начиная с 1. Затем к группам можно обращаться по номерам и получать текст, который соответствует выражению в группе.

Пример использования групп:

```
In [8]: line = "FastEthernet0/1            10.0.12.1       YES manual up                 ␣
↪  up"
In [9]: match = re.search(r'(\S+)\s+([\w.]+)\s+.*', line)
```

В данном примере указаны две группы:

- первая группа - любые символы, кроме пробельных
- 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'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [11]: match.group(1)
Out[11]: 'FastEthernet0/1'

In [12]: match.group(2)
Out[12]: '10.0.12.1'
```

При необходимости можно перечислить несколько номеров групп:

```
In [13]: match.group(1, 2)
Out[13]: ('FastEthernet0/1', '10.0.12.1')

In [14]: match.group(2, 1, 2)
Out[14]: ('10.0.12.1', 'FastEthernet0/1', '10.0.12.1')
```

Начиная с версии Python 3.6, к группам можно обращаться таким образом:

```
In [15]: match[0]
Out[15]: 'FastEthernet0/1            10.0.12.1       YES manual up                    up'

In [16]: match[1]
Out[16]: 'FastEthernet0/1'

In [17]: match[2]
Out[17]: '10.0.12.1'
```

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

```
In [18]: match.groups()
Out[18]: ('FastEthernet0/1', '10.0.12.1')
```

Именованные группы

Когда выражение сложное, не очень удобно определять номер группы. Плюс, при дополнении выражения, может получиться так, что порядок групп изменился, и придется изменить и код, который ссылается на группы.

Именованные группы позволяют задавать группе имя.

Синтаксис именованной группы `(?P<name>regex)`:

```
In [19]: line = "FastEthernet0/1            10.0.12.1       YES manual up                    ␣
↪   up"

In [20]: match = re.search(r'(?P<intf>\S+)\s+(?P<address>[\d.]+)\s+', line)
```

Теперь к этим группам можно обращаться по имени:

```
In [21]: match.group('intf')
Out[21]: 'FastEthernet0/1'

In [22]: match.group('address')
Out[22]: '10.0.12.1'
```

Также очень полезно то, что с помощью метода groupdict(), можно получить словарь, где ключи - имена групп, а значения - подстроки, которые им соответствуют:

```
In [23]: match.groupdict()
Out[23]: {'address': '10.0.12.1', 'intf': 'FastEthernet0/1'}
```

И в таком случае можно добавить группы в регулярное выражение и полагаться на их имя, а не на порядок:

```
In [24]: match = re.search(r'(?P<intf>\S+)\s+(?P<address>[\d\.]+)\s+\w+\s+\w+\s+(?P
↪<status>up|down)\s+(?P<protocol>up|down)', line)

In [25]: match.groupdict()
Out[25]:
{'address': '10.0.12.1',
 'intf': 'FastEthernet0/1',
 'protocol': 'up',
 'status': 'up'}
```

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     FastEthernet0/10
00:05:B3:7E:9B:60   10.1.5.4      63253       dhcp-snooping   5     FastEthernet0/9
00:09:BC:3F:A6:50   10.1.10.6     76260       dhcp-snooping   10    FastEthernet0/3
Total number of bindings: 4
```

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(r'(?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 пробельных. Получается, что выражение описывает последовательность любых символов до пробела
- `(?P<ip>\S+) +` - тут аналогично: последовательность любых непробельных символов 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 пробельных - 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<port>\S+)` - именованная группа „port“ - сюда попадают любые символы, кроме whitespace

В результате, метод groupdict вернет такой словарь:

```
In [3]: match.groupdict()
Out[3]:
{'int': 'FastEthernet0/1',
 'ip': '10.1.10.2',
 'mac': '00:09:BB:3D:D6:58',
 'vlan': '10'}
```

Так как регулярное выражение отработало как нужно, можно создавать скрипт. В скрипте перебираются все строки файла dhcp_snooping.txt, и на стандартный поток вывода выводится информация об устройствах.

Файл parse_dhcp_snooping.py:

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

#'00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10    FastEthernet0/1'
regex = r'(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P<port>\S+)'
result = []
```

(continues on next page)

(продолжение с предыдущей страницы)

```
with open('dhcp_snooping.txt') as data:
    for line in data:
        match = re.search(regex, line)
        if match:
            result.append(match.groupdict())

print('К коммутатору подключено {} устройства'.format(len(result)))

for num, comp in enumerate(result, 1):
    print('Параметры устройства {}:'.format(num))
    for key in comp:
        print('{:10}: {:10}'.format(key, comp[key]))
```

Результат выполнения:

```
$ python parse_dhcp_snooping.py
К коммутатору подключено 4 устройства
Параметры устройства 1:
    int:    FastEthernet0/1
    ip:    10.1.10.2
    mac:    00:09:BB:3D:D6:58
    vlan:    10
Параметры устройства 2:
    int:    FastEthernet0/10
    ip:    10.1.5.2
    mac:    00:04:A3:3E:5B:69
    vlan:    5
Параметры устройства 3:
    int:    FastEthernet0/9
    ip:    10.1.5.4
    mac:    00:05:B3:7E:9B:60
    vlan:    5
Параметры устройства 4:
    int:    FastEthernet0/3
    ip:    10.1.10.6
    mac:    00:09:BC:3F:A6:50
    vlan:    10
```

Группа без захвата

По умолчанию все, что попало в группу, запоминается. Это называется группа с захватом.

Иногда скобки нужны для указания части выражения, которое повторяется. И, при этом, не нужно запоминать выражение.

Например, надо получить MAC-адрес, VLAN и порты из такого лог-сообщения:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in vlan␣
↪10 is flapping between port Gi0/5 and port Gi0/15'
```

Регулярное выражение, которое описывает нужные подстроки:

```
In [2]: match = re.search(r'((\w{4}\.){2}\w{4}).+vlan (\d+).+port (\S+).+port (\S+)', log)
```

Выражение состоит из таких частей:

- `((\w{4}\.){2}\w{4})` - сюда попадет MAC-адрес
- `\w{4}\.` - эта часть описывает 4 буквы или цифры и точку
- `(\w{4}\.){2}` - тут скобки нужны, чтобы указать, что 4 буквы или цифры и точка повторяются два раза
- `\w{4}` - затем 4 буквы или цифры
- `.+vlan (\d+)` - в группу попадет номер VLAN
- `.+port (\S+)` - первый интерфейс
- `.+port (\S+)` - второй интерфейс

Метод groups вернет такой результат:

```
In [3]: match.groups()
Out[3]: ('f03a.b216.7ad7', 'b216.', '10', 'Gi0/5', 'Gi0/15')
```

Второй элемент, по сути, лишний. Он попал в вывод из-за скобок в выражении `(\w{4}\.){2}`.

В этом случае нужно отключить захват в группе. Это делается добавлением `?:` после открывающейся скобки группы.

Теперь выражение выглядит так:

```
In [4]: match = re.search(r'((?:\w{4}\.){2}\w{4}).+vlan (\d+).+port (\S+).+port (\S+)',␣
↪log)
```

И, соответственно, группы:

```
In [5]: match.groups()
Out[5]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Повторение захваченного результата

При работе с группами можно использовать результат, который попал в группу, дальше в этом же выражении.

Например, в выводе sh ip bgp последний столбец описывает атрибут AS Path (через какие автономные системы прошел маршрут):

```
In [1]: bgp = '''
   ...: R9# sh ip bgp | be Network
   ...:    Network          Next Hop       Metric LocPrf Weight Path
   ...: *  192.168.66.0/24  192.168.79.7                       0 500 500 500 i
   ...: *>                  192.168.89.8                       0 800 700 i
   ...: *  192.168.67.0/24  192.168.79.7         0             0 700 700 700 i
   ...: *>                  192.168.89.8                       0 800 700 i
   ...: *  192.168.88.0/24  192.168.79.7                       0 700 700 700 i
   ...: *>                  192.168.89.8         0             0 800 800 i
   ...: '''
```

Допустим, надо получить те префиксы, у которых в пути несколько раз повторяется один и тот же номер AS.

Это можно сделать с помощью ссылки на результат, который был захвачен группой. Например, такое выражение отображает все строки, в которых один и тот же номер повторяется хотя бы два раза:

```
In [2]: for line in bgp.split('\n'):
   ...:     match = re.search(r'(\d+) \1', line)
   ...:     if match:
   ...:         print(line)
   ...:
*  192.168.66.0/24  192.168.79.7                       0 500 500 500 i
*  192.168.67.0/24  192.168.79.7         0             0 700 700 700 i
*  192.168.88.0/24  192.168.79.7                       0 700 700 700 i
*>                  192.168.89.8         0             0 800 800 i
```

В этом выражении обозначение \1 подставляет результат, который попал в группу. Номер один указывает на конкретную группу. В данном случае это группа 1, она же единственная.

Аналогичным образом можно описать строки, в которых один и тот же номер встречается три раза:

```
In [3]: for line in bgp.split('\n'):
   ...:     match = re.search(r'(\d+) \1 \1', line)
   ...:     if match:
   ...:         print(line)
   ...:
*  192.168.66.0/24  192.168.79.7                       0 500 500 500 i
```

(continues on next page)

(продолжение с предыдущей страницы)

```
*  192.168.67.0/24  192.168.79.7         0             0 700 700 700 i
*  192.168.88.0/24  192.168.79.7                       0 700 700 700 i
```

Аналогичным образом можно ссылаться на результат, который попал в именованную группу:

```
In [129]: for line in bgp.split('\n'):
     ...:     match = re.search(r'(?P<as>\d+) (?P=as)', line)
     ...:     if match:
     ...:         print(line)
     ...:
*  192.168.66.0/24   192.168.79.7                        0 500 500 500 i
*  192.168.67.0/24   192.168.79.7         0              0 700 700 700 i
*  192.168.88.0/24   192.168.79.7                        0 700 700 700 i
*>                   192.168.89.8         0              0 800 800 i
```

Дополнительные материалы

Сайты для проверки регулярных выражений:

- regex101

Общие руководства по использованию регулярных выражений:

- Множество примеров использования регулярных выражений от основ до более сложных тем
- Книга Mastering Regular Expressions

Помощь в изучении регулярных выражений:

- regexlearn.com
- Визуализация регулярного выражения
- Regex Crossword

15. Модуль re

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

Основные функции модуля **re**:

- `match` - ищет последовательность в начале строки
- `search` - ищет первое совпадение с шаблоном
- `findall` - ищет все совпадения с шаблоном. Возвращает результирующие строки в виде списка
- `finditer` - ищет все совпадения с шаблоном. Возвращает итератор
- `compile` - компилирует регулярное выражение. К этому объекту затем можно применять все перечисленные функции
- `fullmatch` - вся строка должна соответствовать описанному регулярному выражению

Кроме функций для поиска совпадений, в модуле есть такие функции:

- `re.sub` - для замены в строках
- `re.split` - для разделения строки на части

Объект Match

В модуле re несколько функций возвращают объект Match, если было найдено совпадение:

- search
- match
- finditer возвращает итератор с объектами Match

В этом подразделе рассматриваются методы объекта Match.

Пример объекта Match:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in vlan␣
↪10 is flapping between port Gi0/5 and port Gi0/15'

In [2]: match = re.search(r'Host (\S+) in vlan (\d+) .* port (\S+) and port (\S+)', log)

In [3]: match
Out[3]: <_sre.SRE_Match object; span=(47, 124), match='Host f03a.b216.7ad7 in vlan 10 is␣
↪flapping betwee>'
```

Вывод в 3 строке просто отображает информацию об объекте. Поэтому не стоит полагаться на то, что отображается в части match, так как отображаемая строка обрезается по фиксированному количеству знаков.

group

Метод group возвращает подстроку, которая совпала с выражением или с выражением в группе.

Если метод вызывается без аргументов, отображается вся подстрока:

```
In [4]: match.group()
Out[4]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'
```

Аналогичный вывод возвращает группа 0:

```
In [5]: match.group(0)
Out[5]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'
```

Другие номера отображают только содержимое соответствующей группы:

```
In [6]: match.group(1)
Out[6]: 'f03a.b216.7ad7'

In [7]: match.group(2)
Out[7]: '10'

In [8]: match.group(3)
Out[8]: 'Gi0/5'

In [9]: match.group(4)
Out[9]: 'Gi0/15'
```

Если вызвать метод group с номером группы, который больше, чем количество существующих групп, возникнет ошибка:

```
In [10]: match.group(5)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-18-9df93fa7b44b> in <module>()
----> 1 match.group(5)

IndexError: no such group
```

Если вызвать метод с несколькими номерами групп, результатом будет кортеж со строками, которые соответствуют совпадениям:

```
In [11]: match.group(1, 2, 3)
Out[11]: ('f03a.b216.7ad7', '10', 'Gi0/5')
```

В группу может ничего не попасть, тогда ей будет соответствовать пустая строка:

```
In [12]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [13]: match = re.search(r'Host (\S+) in vlan (\D*)', log)

In [14]: match.group(2)
Out[14]: ''
```

Если группа описывает часть шаблона и совпадений было несколько, метод отобразит последнее совпадение:

```
In [15]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [16]: match = re.search(r'Host (\w{4}\.)+', log)

In [17]: match.group(1)
Out[17]: 'b216.'
```

Такой вывод получился из-за того, что выражение в скобках описывает 4 буквы или цифры, точка и после этого стоит плюс. Соответственно, сначала с выражением в скобках совпала первая часть MAC-адреса, потом вторая. Но запоминается и возвращается только последнее выражение.

Если в выражении использовались именованные группы, методу group можно передать имя группы и получить соответствующую подстроку:

```
In [18]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [19]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [20]: match.group('mac')
Out[20]: 'f03a.b216.7ad7'

In [21]: match.group('int2')
Out[21]: 'Gi0/15'
```

При этом группы доступны и по номеру:

```
In [22]: match.group(3)
Out[22]: 'Gi0/5'

In [23]: match.group(4)
Out[23]: 'Gi0/15'
```

groups

Метод groups возвращает кортеж со строками, в котором элементы - это те подстроки, которые попали в соответствующие группы:

```
In [24]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [25]: match = re.search(r'Host (\S+) '
    ...:                   r'in vlan (\d+) .* '
    ...:                   r'port (\S+) '
    ...:                   r'and port (\S+)',
    ...:                   log)
    ...:

In [26]: match.groups()
Out[26]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

У метода groups есть опциональный параметр - default. Он срабатывает в ситуации, когда все, что попадает в группу, опционально.

Например, при такой строке, совпадение будет и в первой группе, и во второй:

```
In [26]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [27]: match = re.search(r'(\d+) +(\w+)?', line)

In [28]: match.groups()
Out[28]: ('100', 'aab1')
```

Если же в строке нет ничего после пробела, в группу ничего не попадет. Но совпадение будет, так как в регулярном выражении описано, что группа опциональна:

```
In [30]: line = '100     '

In [31]: match = re.search(r'(\d+) +(\w+)?', line)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [32]: match.groups()
Out[32]: ('100', None)
```

Соответственно, для второй группы значением будет None.

Если передать методу groups значение default, оно будет возвращаться вместо None:

```
In [33]: line = '100     '

In [34]: match = re.search(r'(\d+) +(\w+)?', line)

In [35]: match.groups(default=0)
Out[35]: ('100', 0)

In [36]: match.groups(default='No match')
Out[36]: ('100', 'No match')
```

groupdict

Метод `groupdict` возвращает словарь, в котором ключи - имена групп, а значения - соответствующие строки:

```
In [37]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [38]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [39]: match.groupdict()
Out[39]: {'int1': 'Gi0/5', 'int2': 'Gi0/15', 'mac': 'f03a.b216.7ad7', 'vlan': '10'}
```

start, end

Методы `start` и end возвращают индексы начала и конца совпадения с регулярным выражением.

Если методы вызываются без аргументов, они возвращают индексы для всего совпадения:

```
In [40]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [41]: match = re.search(r'(\d+) +([0-9a-f.]+) +(\S+)', line)

In [42]: match.start()
Out[42]: 2

In [43]: match.end()
Out[43]: 42

In [45]: line[match.start():match.end()]
Out[45]: '10     aab1.a1a1.a5d3    FastEthernet0/1'
```

Методам можно передавать номер или имя группы. Тогда они возвращают индексы для этой группы:

```
In [46]: match.start(2)
Out[46]: 9

In [47]: match.end(2)
Out[47]: 23

In [48]: line[match.start(2):match.end(2)]
Out[48]: 'aab1.a1a1.a5d3'
```

Аналогично для именованных групп:

```
In [49]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [50]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [51]: match.start('mac')
Out[51]: 52

In [52]: match.end('mac')
Out[52]: 66
```

span

Метод span возвращает кортеж с индексом начала и конца подстроки. Он работает аналогично методам start, end, но возвращает пару чисел.

Без аргументов метод span возвращает индексы для всего совпадения:

```
In [53]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [54]: match = re.search(r'(\d+) +([0-9a-f.]+) +(\S+)', line)

In [55]: match.span()
Out[55]: (2, 42)
```

Но ему также можно передать номер группы:

```
In [56]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [57]: match = re.search(r'(\d+) +([0-9a-f.]+) +(\S+)', line)

In [58]: match.span(2)
Out[58]: (9, 23)
```

Аналогично для именованных групп:

```
In [59]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in␣
↪vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [60]: match = re.search(r'Host (?P<mac>\S+) '
    ...:                   r'in vlan (?P<vlan>\d+) .* '
    ...:                   r'port (?P<int1>\S+) '
    ...:                   r'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [64]: match.span('mac')
Out[64]: (52, 66)

In [65]: match.span('vlan')
Out[65]: (75, 77)
```

Функция search

Функция search:

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

Функция search подходит в том случае, когда надо найти только одно совпадение в строке, например, когда регулярное выражение описывает всю строку или часть строки.

Рассмотрим пример использования функции search в разборе лог-файла.

В файле log.txt находятся лог-сообщения с информацией о том, что один и тот же MAC слишком быстро переучивается то на одном, то на другом интерфейсе. Одна из причин таких сообщений - петля в сети.

Содержимое файла log.txt:

```
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/16␣
↪and port Gi0/24
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/16␣
↪and port Gi0/24
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/24␣
↪and port Gi0/19
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/24␣
↪and port Gi0/16
```

При этом MAC-адрес может прыгать между несколькими портами. В таком случае очень важно знать, с каких портов прилетает MAC.

Попробуем вычислить, между какими портами и в каком VLAN образовалась проблема. Проверка регулярного выражения с одной строкой из log-файла:

```
In [1]: import re

In [2]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping␣
↪between port Gi0/16 and port Gi0/24'

In [3]: match = re.search(r'Host \S+ '
   ...:                   r'in vlan (\d+) '
   ...:                   r'is flapping between port '
   ...:                   r'(\S+) and port (\S+)', log)
   ...:
```

Регулярное выражение для удобства чтения разбито на части. В нём есть три группы:

- (\d+) - описывает номер VLAN

- `(\S+) and port (\S+)` - в это выражение попадают номера портов

В итоге, в группы попали такие части строки:

```
In [4]: match.groups()
Out[4]: ('10', 'Gi0/16', 'Gi0/24')
```

В итоговом скрипте файл log.txt обрабатывается построчно, и из каждой строки собирается информация о портах. Так как порты могут дублироваться, сразу добавляем их в множество, чтобы получить подборку уникальных интерфейсов (файл parse_log_search.py):

```
import re

regex = (r'Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for line in f:
        match = re.search(regex, line)
        if match:
            vlan = match.group(1)
            ports.add(match.group(2))
            ports.add(match.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат выполнения скрипта такой:

```
$ python parse_log_search.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Обработка вывода show cdp neighbors detail

Попробуем получить параметры устройств из вывода sh cdp neighbors detail.

Пример вывода информации для одного соседа:

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE␣
↪SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

Задача получить такие поля:

- имя соседа (Device ID: SW2)
- IP-адрес соседа (IP address: 10.1.1.2)
- платформу соседа (Platform: cisco WS-C2960-8TC-L)
- версию IOS (Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE SOFTWARE (fc1))

И для удобства надо получить данные в виде словаря. Пример итогового словаря для коммутатора SW2:

```
{'SW2': {'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L',
         'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9'}}
```

Пример проверяется на файле sh_cdp_neighbors_sw1.txt.

Первый вариант решения (файл parse_sh_cdp_neighbors_detail_ver1.py):

```
import re
from pprint import pprint


def parse_cdp(filename):
    result = {}

    with open(filename) as f:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        for line in f:
            if line.startswith('Device ID'):
                neighbor = re.search(r'Device ID: (\S+)', line).group(1)
                result[neighbor] = {}
            elif line.startswith('  IP address'):
                ip = re.search(r'IP address: (\S+)', line).group(1)
                result[neighbor]['ip'] = ip
            elif line.startswith('Platform'):
                platform = re.search(r'Platform: (\S+ \S+),', line).group(1)
                result[neighbor]['platform'] = platform
            elif line.startswith('Cisco IOS Software'):
                ios = re.search(r'Cisco IOS Software, (.+), RELEASE',
                                line).group(1)
                result[neighbor]['ios'] = ios

    return result


pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Тут нужные строки отбираются с помощью метода строк startswith. И в строке с помощью регулярного выражения получается требуемая часть строки. В итоге все собирается в словарь.

Результат выглядит так:

```
$ python parse_sh_cdp_neighbors_detail_ver1.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

Все получилось как нужно, но эту задачу можно решить более компактно.

Вторая версия решения (файл parse_sh_cdp_neighbors_detail_ver2.py):

```
import re
from pprint import pprint


def parse_cdp(filename):
    regex = (r'Device ID: (?P<device>\S+)'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            r'|IP address: (?P<ip>\S+)'
            r'|Platform: (?P<platform>\S+ \S+),'
            r'|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open(filename) as f:
        for line in f:
            match = re.search(regex, line)
            if match:
                if match.lastgroup == 'device':
                    device = match.group(match.lastgroup)
                    result[device] = {}
                else:
                    result[device][match.lastgroup] = match.group(
                        match.lastgroup)

    return result


pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Пояснения ко второму варианту:

- в регулярном выражении описаны все варианты строк через знак или |
- без проверки строки ищется совпадение
- если совпадение найдено, проверяется метод lastgroup
- метод lastgroup возвращает имя последней именованной группы в регулярном выражении, для которой было найдено совпадение
- если было найдено совпадение для группы device, в переменную device записывается значение, которое попало в эту группу
- иначе в словарь записывается соответствие „имя группы“: соответствующее значение

Результат будет таким же:

```
$ python parse_sh_cdp_neighbors_detail_ver2.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

Функция match

Функция `match()`:

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

Функция match отличается от search тем, что match всегда ищет совпадение в начале строки. Например, если повторить пример, который использовался для функции search, но уже с match:

```
In [2]: import re

In [3]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping␣
↪between port Gi0/16 and port Gi0/24'

In [4]: match = re.match(r'Host \S+ '
   ...:                  r'in vlan (\d+) '
   ...:                  r'is flapping between port '
   ...:                  r'(\S+) and port (\S+)', log)
   ...:
```

Результатом будет None:

```
In [6]: print(match)
None
```

Так получилось из-за того, что match ищет слово Host в начале строки. Но это сообщение находится в середине.

В данном случае можно легко исправить выражение, чтобы функция match находила совпадение:

```
In [4]: match = re.match(r'\S+: Host \S+ '
   ...:                  r'in vlan (\d+) '
   ...:                  r'is flapping between port '
   ...:                  r'(\S+) and port (\S+)', log)
   ...:
```

Перед словом Host добавлено выражение `\S+:`. Теперь совпадение будет найдено:

```
In [11]: print(match)
<_sre.SRE_Match object; span=(0, 104), match='%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.
↪0156 in '>

In [12]: match.groups()
Out[12]: ('10', 'Gi0/16', 'Gi0/24')
```

Пример аналогичен тому, который использовался в функции search, с небольшими изменениями (файл parse_log_match.py):

```
import re

regex = (r'\S+: Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for line in f:
        match = re.match(regex, line)
        if match:
            vlan = match.group(1)
            ports.add(match.group(2))
            ports.add(match.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат:

```
$ python parse_log_match.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Функция finditer

Функция `finditer()`:

- используется для поиска всех непересекающихся совпадений в шаблоне
- возвращает итератор с объектами Match
- finditer возвращает итератор даже в том случае, когда совпадение не найдено

Функция finditer отлично подходит для обработки тех команд, вывод которых отображается столбцами. Например, sh ip int br, sh mac address-table и др. В этом случае его можно применять ко всему выводу команды.

Пример вывода sh ip int br:

```
In [8]: sh_ip_int_br = '''
   ...: R1#show ip interface brief
   ...: Interface              IP-Address      OK? Method Status           Protocol
   ...: FastEthernet0/0        15.0.15.1       YES manual up               up
   ...: FastEthernet0/1        10.0.12.1       YES manual up               up
   ...: FastEthernet0/2        10.0.13.1       YES manual up               up
   ...: FastEthernet0/3        unassigned      YES unset  up               up
   ...: Loopback0              10.1.1.1        YES manual up               up
   ...: Loopback100            100.0.0.1       YES manual up               up
   ...: '''
```

Регулярное выражение для обработки вывода:

```
In [9]: result = re.finditer(r'(\S+) +'
   ...:                      r'([\d.]+) +'
   ...:                      r'\w+ +\w+ +'
   ...:                      r'(up|down|administratively down) +'
   ...:                      r'(up|down)',
   ...:                      sh_ip_int_br)
   ...:
```

В переменной result находится итератор:

```
In [12]: result
Out[12]: <callable_iterator at 0xb583f46c>
```

В итераторе находятся объекты Match:

```
In [16]: groups = []

In [18]: for match in result:
    ...:     print(match)
    ...:     groups.append(match.groups())
    ...:
<_sre.SRE_Match object; span=(103, 171), match='FastEthernet0/0       15.0.15.1       YES␣
↪manual >
<_sre.SRE_Match object; span=(172, 240), match='FastEthernet0/1       10.0.12.1       YES␣
↪manual >
<_sre.SRE_Match object; span=(241, 309), match='FastEthernet0/2       10.0.13.1       YES␣
↪manual >
<_sre.SRE_Match object; span=(379, 447), match='Loopback0             10.1.1.1        YES␣
↪manual >
<_sre.SRE_Match object; span=(448, 516), match='Loopback100           100.0.0.1       YES␣
↪manual >'
```

Теперь в списке groups находятся кортежи со строками, которые попали в группы:

```
In [19]: groups
Out[19]:
[('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]
```

Аналогичный результат можно получить с помощью генератора списков:

```
In [20]: regex = r'(\S+) +([\d.]+) +\w+ +\w+ +(up|down|administratively down) +(up|down)'

In [21]: result = [match.groups() for match in re.finditer(regex, sh_ip_int_br)]

In [22]: result
Out[22]:
[('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]
```

Теперь разберем тот же лог-файл, который использовался в подразделах search и match.

В этом случае вывод можно не перебирать построчно, а передать все содержимое файла (файл parse_log_finditer.py):

```
import re

regex = (r'Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for m in re.finditer(regex, f.read()):
        vlan = m.group(1)
        ports.add(m.group(2))
        ports.add(m.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Предупреждение: В реальной жизни log-файл может быть очень большим. В таком слу-

чае, его лучше обрабатывать построчно.

Вывод будет таким же:

```
$ python parse_log_finditer.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Обработка вывода show cdp neighbors detail

С помощью finditer можно обработать вывод sh cdp neighbors detail, так же, как и в подразделе re.search.

Скрипт почти полностью аналогичен варианту с re.search (файл parse_sh_cdp_neighbors_detail_finditer.py):

```
import re
from pprint import pprint


def parse_cdp(filename):
    regex = (r'Device ID: (?P<device>\S+)'
             r'|IP address: (?P<ip>\S+)'
             r'|Platform: (?P<platform>\S+ \S+),'
             r'|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open(filename) as f:
        match_iter = re.finditer(regex, f.read())
        for match in match_iter:
            if match.lastgroup == 'device':
                device = match.group(match.lastgroup)
                result[device] = {}
            elif device:
                result[device][match.lastgroup] = match.group(match.lastgroup)

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Теперь совпадения ищутся во всем файле, а не в каждой строке отдельно:

```
with open(filename) as f:
    match_iter = re.finditer(regex, f.read())
```

Затем перебираются совпадения:

```
with open(filename) as f:
    match_iter = re.finditer(regex, f.read())
    for match in match_iter:
```

Остальное аналогично.

Результат будет таким:

```
$ python parse_sh_cdp_neighbors_detail_finditer.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

Хотя результат аналогичный, с finditer больше возможностей, так как можно указывать не только то, что должно находиться в нужной строке, но и в строках вокруг.

Например, можно точнее указать, какой именно IP-адрес надо взять:

```
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP

...

Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

Например, если нужно взять первый IP-адрес, можно так дополнить регулярное выражение:

```
regex = (r'Device ID: (?P<device>\S+)'
         r'|Entry address.*\n +IP address: (?P<ip>\S+)'
         r'|Platform: (?P<platform>\S+ \S+),'
         r'|Cisco IOS Software, (?P<ios>.+), RELEASE')
```

Функция findall

Функция `findall()`:

- используется для поиска всех непересекающихся совпадений в шаблоне
- возвращает:
  - список строк, которые описаны регулярным выражением, если в регулярном выражении нет групп
  - список строк, которые совпали с регулярным выражением в группе, если в регулярном выражении одна группа
  - список кортежей, в которых находятся строки, которые совпали с выражением в группе, если групп несколько

Рассмотрим работу findall на примере вывода команды sh mac address-table:

```
In [2]: mac_address_table = open('CAM_table.txt').read()

In [3]: print(mac_address_table)
sw1#sh mac address-table
          Mac Address Table
-------------------------------------------

Vlan    Mac Address       Type        Ports
----    -----------       --------    -----
 100    a1b2.ac10.7000    DYNAMIC     Gi0/1
 200    a0d4.cb20.7000    DYNAMIC     Gi0/2
 300    acb4.cd30.7000    DYNAMIC     Gi0/3
 100    a2bb.ec40.7000    DYNAMIC     Gi0/4
 500    aa4b.c550.7000    DYNAMIC     Gi0/5
 200    a1bb.1c60.7000    DYNAMIC     Gi0/6
 300    aa0b.cc70.7000    DYNAMIC     Gi0/7
```

Первый пример - регулярное выражение без групп. В этом случае findall возвращает список строк, которые совпали с регулярным выражением.

Например, с помощью findall можно получить список строк с соответствиями vlan - mac - interface и избавиться от заголовка в выводе команды:

```
In [4]: re.findall(r'\d+ +\S+ +\w+ +\S+', mac_address_table)
Out[4]:
['100    a1b2.ac10.7000    DYNAMIC     Gi0/1',
 '200    a0d4.cb20.7000    DYNAMIC     Gi0/2',
 '300    acb4.cd30.7000    DYNAMIC     Gi0/3',
 '100    a2bb.ec40.7000    DYNAMIC     Gi0/4',
 '500    aa4b.c550.7000    DYNAMIC     Gi0/5',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 '200    a1bb.1c60.7000    DYNAMIC     Gi0/6',
 '300    aa0b.cc70.7000    DYNAMIC     Gi0/7']
```

Обратите внимание, что findall возвращает список строк, а не объект Match.

Как только в регулярном выражении появляется группа, findall ведет себя по-другому. Если в выражении используется одна группа, findall возвращает список строк, которые совпали с выражением в группе:

```
In [5]: re.findall(r'\d+ +(\S+) +\w+ +\S+', mac_address_table)
Out[5]:
['a1b2.ac10.7000',
 'a0d4.cb20.7000',
 'acb4.cd30.7000',
 'a2bb.ec40.7000',
 'aa4b.c550.7000',
 'a1bb.1c60.7000',
 'aa0b.cc70.7000']
```

При этом findall ищет совпадение всей строки, но возвращает результат, похожий на метод groups() в объекте Match.

Если же групп несколько, findall вернет список кортежей:

```
In [6]: re.findall(r'(\d+) +(\S+) +\w+ +(\S+)', mac_address_table)
Out[6]:
[('100', 'a1b2.ac10.7000', 'Gi0/1'),
 ('200', 'a0d4.cb20.7000', 'Gi0/2'),
 ('300', 'acb4.cd30.7000', 'Gi0/3'),
 ('100', 'a2bb.ec40.7000', 'Gi0/4'),
 ('500', 'aa4b.c550.7000', 'Gi0/5'),
 ('200', 'a1bb.1c60.7000', 'Gi0/6'),
 ('300', 'aa0b.cc70.7000', 'Gi0/7')]
```

Если такие особенности работы функции findall мешают получить необходимый результат, то лучше использовать функцию finditer, но иногда такое поведение подходит и удобно использовать.

Пример использования findall в разборе лог-файла (файл parse_log_findall.py):

```
import re

regex = (r'Host \S+ '
         r'in vlan (\d+) '
         r'is flapping between port '
         r'(\S+) and port (\S+)')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
ports = set()

with open('log.txt') as f:
    result = re.findall(regex, f.read())
    for vlan, port1, port2 in result:
        ports.add(port1)
        ports.add(port2)

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат:

```
$ python parse_log_findall.py
Петля между портами Gi0/19, Gi0/16, Gi0/24 в VLAN 10
```

Функция compile

В Python есть возможность заранее скомпилировать регулярное выражение, а затем использовать его. Это особенно полезно в тех случаях, когда регулярное выражение много используется в скрипте.

Использование компилированного выражения может ускорить обработку, и, как правило, такой вариант удобней использовать, так как в программе разделяется создание регулярного выражения и его использование. Кроме того, при использовании функции `re.compile` создается объект `RegexObject`, у которого есть несколько дополнительных возможностей, которых нет в объекте `MatchObject`.

Для компиляции регулярного выражения используется функция `re.compile`:

```
In [52]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')
```

У объекта который возвращает re.compile, доступны методы search, match, finditer, findall. Это те же функции, которые доступны в модуле глобально, но теперь их надо применять к объекту.

Пример использования метода search:

```
In [67]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [68]: match = regex.search(line)
```

Теперь search надо вызывать как метод объекта regex. И передать как аргумент строку.

Результатом будет объект Match:

```
In [69]: match
Out[69]: <_sre.SRE_Match object; span=(1, 43), match='100    a1b2.ac10.7000    DYNAMIC    ␣
↪ Gi0/1'>

In [70]: match.group()
Out[70]: '100    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

При использовании re.compile, флаги надо указывать внутри re.compile, не в методах search/finditer:

```
regex = re.compile(r'^Device ID: \S+.+?Cisco IOS', re.MULTILINE | re.DOTALL)
match_all = regex.finditer(output)
```

Пример компиляции регулярного выражения и его использования на примере разбора лог-файла (файл parse_log_compile.py):

```
import re

regex = re.compile(r'Host \S+ '
                   r'in vlan (\d+) '
                   r'is flapping between port '
                   r'(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for m in regex.finditer(f.read()):
        vlan = m.group(1)
        ports.add(m.group(2))
        ports.add(m.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Это модифицированный пример с использованием finditer. Тут изменилось описание регулярного выражения:

```
regex = re.compile(r'Host \S+ '
                   r'in vlan (\d+) '
                   r'is flapping between port '
                   r'(\S+) and port (\S+)')
```

И вызов finditer теперь выполняется как метод объекта regex:

```
for m in regex.finditer(f.read()):
```

Параметры, которые доступны только при использовании re.compile

При использовании функции re.compile в методах search, match, findall, finditer и fullmatch появляются дополнительные параметры:

- pos - позволяет указывать индекс в строке, с которого надо начать искать совпадение
- endpos - указывает, до какого индекса надо выполнять поиск

Их использование аналогично выполнению среза строки.

Например, таким будет результат без указания параметров pos, endpos:

```
In [75]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [76]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [77]: match = regex.search(line)

In [78]: match.group()
Out[78]: '100    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

В этом случае указывается начальная позиция поиска:

```
In [79]: match = regex.search(line, 2)

In [80]: match.group()
Out[80]: '00    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

Указание начальной позиции аналогично срезу строки:

```
In [81]: match = regex.search(line[2:])

In [82]: match.group()
Out[82]: '00    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

И последний пример - использование двух индексов:

```
In [90]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [91]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [92]: match = regex.search(line, 2, 40)

In [93]: match.group()
Out[93]: '00    a1b2.ac10.7000    DYNAMIC     Gi'
```

И аналогичный срез строки:

```
In [94]: match = regex.search(line[2:40])

In [95]: match.group()
Out[95]: '00    a1b2.ac10.7000    DYNAMIC     Gi'
```

В методах match, findall, finditer и fullmatch параметры pos и endpos работают аналогично.

Флаги

При использовании функций или создании скомпилированного регулярного выражения можно указывать дополнительные флаги, которые влияют на поведение регулярного выражения.

Модуль re поддерживает такие флаги (в скобках короткий вариант обозначения флага):

- re.ASCII (re.A)
- re.IGNORECASE (re.I)
- re.MULTILINE (re.M)
- re.DOTALL (re.S)
- re.VERBOSE (re.X)
- re.LOCALE (re.L)
- re.DEBUG

В этом подразделе для примера рассматривается флаг re.DOTALL. Информация об остальных флагах доступна в документации.

re.DOTALL

С помощью регулярных выражений можно работать и с многострочной строкой.

Например, из строки sh_cdp надо получить имя устройства, платформу и IOS:

```
In [2]: sh_cdp = '''
   ...: Device ID: SW2
   ...: Entry address(es):
   ...:   IP address: 10.1.1.2
   ...: Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
   ...: Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
   ...: Holdtime : 164 sec
   ...:
   ...: Version :
   ...: Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9,␣
↪RELEASE SOFTWARE (fc1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...: 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
   ...: '''
```

Конечно, в этом случае можно разделить строку на части и работать с каждой строкой отдельно, но можно получить нужные данные и без разделения.

В этом выражении описаны строки с нужными данными:

```
In [3]: regex = r'Device ID: (\S+).+Platform: \w+ (\S+),.+Cisco IOS Software.+ Version (\
↪S+),'
```

В таком случае, совпадения не будет, потому что по умолчанию точка означает любой символ, кроме перевода строки:

```
In [4]: print(re.search(regex, sh_cdp))
None
```

Изменить поведение по умолчанию, можно с помощью флага re.DOTALL:

```
In [5]: match = re.search(regex, sh_cdp, re.DOTALL)

In [6]: match.groups()
Out[6]: ('SW2', 'WS-C2960-8TC-L', '12.2(55)SE9')
```

Так как теперь в точку входит и перевод строки, комбинация `.+` захватывает все, между нужными данными.

Теперь попробуем с помощью этого регулярного выражения, получить информацию про всех соседей из файла sh_cdp_neighbors_sw1.txt (вывод сокращен).

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE␣
↪SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport

-------------------------
Device ID: R1
Entry address(es):
  IP address: 10.1.1.1
Platform: Cisco 3825,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/22,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1, RELEASE␣
↪SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport

-------------------------
Device ID: R2
Entry address(es):
  IP address: 10.2.2.2
Platform: Cisco 2911,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/21,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1, RELEASE␣
↪SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport
```

Поиск всех совпадений с регулярным выражением:

```
In [7]: with open('sh_cdp_neighbors_sw1.txt') as f:
   ...:     sh_cdp = f.read()
   ...:

In [8]: regex = r'Device ID: (\S+).+Platform: \w+ (\S+),.+Cisco IOS Software.+ Version (\
↪S+),'

In [9]: match = re.finditer(regex, sh_cdp, re.DOTALL)

In [10]: for m in match:
    ...:     print(m.groups())
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:
('SW2', '2911', '15.2(2)T1')
```

На первый взгляд, кажется, что вместо трех устройств, в вывод попало только одно. Однако, если присмотреться к результатам, окажется, что кортеже находится Device ID от первого соседа, а платформа и IOS от последнего.

Короткий вариант вывода, чтобы легче было ориентироваться в результатах:

```
Device ID        Local Intrfce     Holdtme    Capability  Platform  Port ID
SW2              Gi 1/0/16         171              R S   C2960     Gi 0/1
R1               Gi 1/0/22         158              R     C3825     Gi 0/0
R2               Gi 1/0/21         177              R     C2911     Gi 0/0
```

Так получилось из-за того, что между нужными частями вывода, указана комбинация `.+`. Без флага `re.DOTALL` такое выражение захватило бы все до перевода строки, но с флагом оно захватывает максимально длинный кусок текста, так как + жадный. В итоге регулярное выражение описывает строку от первого Device ID до последнего места где встречается `Cisco IOS Software.+ Version`.

Такая ситуация возникает очень часто при использовании `re.DOTALL` и чтобы исправить ее, надо не забыть отключить жадность:

```
In [10]: regex = r'Device ID: (\S+).+?Platform: \w+ (\S+),.+?Cisco IOS Software.+?␣
↪Version (\S+),'

In [11]: match = re.finditer(regex, sh_cdp, re.DOTALL)

In [12]: for m in match:
    ...:     print(m.groups())
    ...:
('SW2', 'WS-C2960-8TC-L', '12.2(55)SE9')
('R1', '3825', '12.4(24)T1')
('R2', '2911', '15.2(2)T1')
```

Функция re.split

Функция split работает аналогично методу split в строках, но в функции re.split можно использовать регулярные выражения, а значит, разделять строку на части по более сложным условиям.

Например, строку ospf_route надо разбить на элементы по пробелам (как в методе str.split):

```
In [1]: ospf_route = 'O     10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [2]: re.split(r' +', ospf_route)
Out[2]:
['O',
 '10.0.24.0/24',
 '[110/41]',
 'via',
 '10.0.13.3,',
 '3d18h,',
 'FastEthernet0/0']
```

Аналогичным образом можно избавиться и от запятых:

```
In [3]: re.split(r'[ ,]+', ospf_route)
Out[3]:
['O',
 '10.0.24.0/24',
 '[110/41]',
 'via',
 '10.0.13.3',
 '3d18h',
 'FastEthernet0/0']
```

И, если нужно, от квадратных скобок:

```
In [4]: re.split(r'[ ,\[\]]+', ospf_route)
Out[4]: ['O', '10.0.24.0/24', '110/41', 'via', '10.0.13.3', '3d18h', 'FastEthernet0/0']
```

У функции split есть особенность работы с группами (выражения в круглых скобках). Если указать то же выражение с помощью круглых скобок, в итоговый список попадут и разделители.

Например, в выражении как разделитель добавлено слово via:

```
In [5]: re.split(r'(via|[ ,\[\]])+', ospf_route)
Out[5]:
['O',
 ' ',
 '10.0.24.0/24',
 '[',
 '110/41',
 ' ',
 '10.0.13.3',
 ' ',
 '3d18h',
 ' ',
 'FastEthernet0/0']
```

Для отключения такого поведения надо сделать группу noncapture. То есть, отключить запоминание элементов группы:

```
In [6]: re.split(r'(?:via|[ ,\[\]])+', ospf_route)
Out[6]: ['O', '10.0.24.0/24', '110/41', '10.0.13.3', '3d18h', 'FastEthernet0/0']
```

Функция re.sub

Функция re.sub работает аналогично методу replace в строках. Но в функции re.sub можно использовать регулярные выражения, а значит, делать замены по более сложным условиям.

Заменим запятые, квадратные скобки и слово via на пробел в строке ospf_route:

```
In [7]: ospf_route = 'O    10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0'

In [8]: re.sub(r'(via|[,\[\]])', ' ', ospf_route)
Out[8]: 'O        10.0.24.0/24  110/41    10.0.13.3  3d18h  FastEthernet0/0'
```

С помощью re.sub можно трансформировать строку. Например, преобразовать строку mac_table таким образом:

```
In [9]: mac_table = '''
   ...:  100    aabb.cc10.7000    DYNAMIC     Gi0/1
   ...:  200    aabb.cc20.7000    DYNAMIC     Gi0/2
   ...:  300    aabb.cc30.7000    DYNAMIC     Gi0/3
   ...:  100    aabb.cc40.7000    DYNAMIC     Gi0/4
   ...:  500    aabb.cc50.7000    DYNAMIC     Gi0/5
   ...:  200    aabb.cc60.7000    DYNAMIC     Gi0/6
   ...:  300    aabb.cc70.7000    DYNAMIC     Gi0/7
   ...: '''

In [4]: print(re.sub(r' *(\d+) +'
   ...:              r'([a-f0-9]+)\.'
   ...:              r'([a-f0-9]+)\.'
   ...:              r'([a-f0-9]+) +\w+ +'
   ...:              r'(\S+)',
   ...:              r'\1 \2:\3:\4 \5',
   ...:              mac_table))
   ...:

100 aabb:cc10:7000 Gi0/1
200 aabb:cc20:7000 Gi0/2
300 aabb:cc30:7000 Gi0/3
100 aabb:cc40:7000 Gi0/4
500 aabb:cc50:7000 Gi0/5
```

(continues on next page)

(продолжение с предыдущей страницы)

```
200 aabb:cc60:7000 Gi0/6
300 aabb:cc70:7000 Gi0/7
```

Регулярное выражение разделено на группы:

- `(\d+)` - первая группа. Сюда попадет номер VLAN
- `([a-f0-9]+).([a-f0-9]+).([a-f0-9]+)` - три следующие группы (2, 3, 4) описывают MAC-адрес
- `(\S+)` - пятая группа. Описывает интерфейс.

Во втором регулярном выражении эти группы используются. Для того, чтобы сослаться на группу, используется обратный слеш и номер группы. Чтобы не пришлось экранировать обратный слеш, используется raw строка.

В итоге вместо номеров групп будут подставлены соответствующие подстроки. Для примера, также изменен формат записи MAC-адреса.

Дополнительные материалы

Регулярные выражения в Python:

- Регулярные выражения в Python от простого к сложному. Подробности, примеры, картинки, упражнения
- Regular Expression HOWTO
- Python 3 Module of the Week. Модуль re

Сайты для проверки регулярных выражений:

- regex101
- для Python - тут можно указывать и методы search, match, findall, и флаги. Пример регулярного выражения. К сожалению, иногда не все выражения воспринимает.
- Еще один сайт для Python - не поддерживает методы, но хорошо работает и отработал те выражения, на которые ругнулся предыдущий сайт. Подходит для однострочного текста отлично. С многострочным надо учитывать, что в Python будет другая ситуация.

Общие руководства по использованию регулярных выражений:

- Множество примеров использования регулярных выражений от основ до более сложных тем
- Книга Mastering Regular Expressions

Помощь в изучении регулярных выражений:

- regexlearn.com

- Визуализация регулярного выражения
- Regex Crossword

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 15.1

Создать функцию get_ip_from_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, можно не проверять корректность IP-адреса, диапазоны адресов и так далее, так как обрабатывается вывод команды, а не ввод пользователя.

Задание 15.1a

Скопировать функцию get_ip_from_cfg из задания 15.1 и переделать ее таким образом, чтобы она возвращала словарь:

- ключ: имя интерфейса
- значение: кортеж с двумя строками:
  - IP-адрес
  - маска

В словарь добавлять только те интерфейсы, на которых настроены IP-адреса.

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.

Обратите внимание, что в данном случае, можно не проверять корректность IP-адреса, диапазоны адресов и так далее, так как обрабатывается вывод команды, а не ввод пользователя.

Задание 15.1b

Проверить работу функции get_ip_from_cfg из задания 15.1a 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 get_ip_from_cfg, интерфейсу Ethernet0/1 соответствует только один из них (второй).

Скопировать функцию get_ip_from_cfg из задания 15.1a и переделать ее таким образом, чтобы она возвращала список кортежей для каждого интерфейса. Если на интерфейсе назначен только один адрес, в списке будет один кортеж. Если же на интерфейсе настроены несколько IP-адресов, то в списке будет несколько кортежей.

Проверьте функцию на конфигурации config_r2.txt и убедитесь, что интерфейсу Ethernet0/1 соответствует список из двух кортежей.

Обратите внимание, что в данном случае, можно не проверять корректность IP-адреса, диапазоны адресов и так далее, так как обрабатывается вывод команды, а не ввод пользователя.

Задание 15.2

Создать функцию parse_sh_ip_int_br, которая ожидает как аргумент имя файла, в котором находится вывод команды show ip int br

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", "down", "down")]
```

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла sh_ip_int_br.txt.

Задание 15.2a

Создать функцию convert_to_dict, которая ожидает два аргумента:

- список с названиями полей
- список кортежей со значениями

Функция возвращает результат в виде списка словарей, где ключи - взяты из первого списка, а значения подставлены из второго.

Например, если функции передать как аргументы список headers и список

```
[("R1", "12.4(24)T1", "Cisco 3825"),
 ("R2", "15.2(2)T1", "Cisco 2911")]
```

Функция должна вернуть такой список со словарями:

```
[{"hostname": "R1", "ios": "12.4(24)T1", "platform": "Cisco 3825"},
 {"hostname": "R2", "ios": "15.2(2)T1", "platform": "Cisco 2911"}]
```

Функция не должна быть привязана к конкретным данным или количеству заголовков/данных в кортежах.

Проверить работу функции:

- первый аргумент - список headers
- второй аргумент - список data

Ограничение: Все задания надо выполнять используя только пройденные темы.

```
headers = ["hostname", "ios", "platform"]

data = [
    ("R1", "12.4(24)T1", "Cisco 3825"),
    ("R2", "15.2(2)T1", "Cisco 2911"),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ("SW1", "12.2(55)SE9", "Cisco WS-C2960-8TC-L"),
]
```

Задание 15.3

Создать функцию convert_ios_nat_to_asa, которая конвертирует правила NAT из синтаксиса cisco IOS в cisco ASA.

Функция ожидает такие аргументы:

- имя файла, в котором находится правила NAT Cisco IOS
- имя файла, в который надо записать полученные правила NAT для ASA

Функция ничего не возвращает.

Проверить функцию на файле cisco_nat_config.txt.

Пример правил NAT cisco IOS

```
ip nat inside source static tcp 10.1.2.84 22 interface GigabitEthernet0/1 20022
ip nat inside source static tcp 10.1.9.5 22 interface GigabitEthernet0/1 20023
```

И соответствующие правила NAT для ASA:

```
object network LOCAL_10.1.2.84
 host 10.1.2.84
 nat (inside,outside) static interface service tcp 22 20022
object network LOCAL_10.1.9.5
 host 10.1.9.5
 nat (inside,outside) static interface service tcp 22 20023
```

В файле с правилами для ASA:

- не должно быть пустых строк между правилами
- перед строками «object network» не должны быть пробелы
- перед остальными строками должен быть один пробел

Во всех правилах для ASA интерфейсы будут одинаковыми (inside,outside).

Задание 15.4

Создать функцию get_ints_without_description, которая ожидает как аргумент имя файла, в котором находится конфигурация устройства.

Функция должна обрабатывать конфигурацию и возвращать список имен интерфейсов, на которых нет описания (команды description).

Пример интерфейса с описанием:

```
interface Ethernet0/2
 description To P_r9 Ethernet0/2
 ip address 10.0.19.1 255.255.255.0
 mpls traffic-eng tunnels
 ip rsvp bandwidth
```

Интерфейс без описания:

```
interface Loopback0
 ip address 10.1.1.1 255.255.255.255
```

Проверить работу функции на примере файла config_r1.txt.

Задание 15.5

Создать функцию generate_description_from_cdp, которая ожидает как аргумент имя файла, в котором находится вывод команды show cdp neighbors.

Функция должна обрабатывать вывод команды show cdp neighbors и генерировать на основании вывода команды описание для интерфейсов.

Например, если у R1 такой вывод команды:

```
R1>show cdp neighbors
Capability Codes: R - Router, T - Trans Bridge, B - Source Route Bridge
                  S - Switch, H - Host, I - IGMP, r - Repeater

Device ID        Local Intrfce     Holdtme    Capability  Platform  Port ID
SW1              Eth 0/0           140          S I      WS-C3750-  Eth 0/1
```

Для интерфейса Eth 0/0 надо сгенерировать такое описание `description Connected to SW1 port Eth 0/1`.

Функция должна возвращать словарь, в котором ключи - имена интерфейсов, а значения - команда задающая описание интерфейса:

```
"Eth 0/0": "description Connected to SW1 port Eth 0/1"
```

Проверить работу функции на файле sh_cdp_n_sw1.txt.

5

IV. Запись и передача данных

В этой части книги рассматриваются вопросы сохранения и передачи данных. Данными могут быть, например:

- вывод команд
- обработанный вывод команд в виде словаря, списка и подобного
- информация полученная из системы мониторинга

До сих пор рассматривался только самый простой вариант - запись информации в обычный текстовый файл.

В этой части рассматривается чтение и запись данных в форматах CSV, JSON и YAML:

- CSV - это табличный формат представления данных. Он может быть получен, например, при экспорте данных из таблицы или базе данных. Аналогичным образом данные могут быть записаны в этом формате для последующего импорта в таблицу.
- JSON - это формат, который очень часто используется в API. Кроме того, этот формат позволит сохранить такие структуры данных как словари или списки в структурированном формате и затем прочитать их из файла в формате JSON и получить те же структуры данных в Python.
- Формат YAML очень часто используется для описания сценариев. Например, он используется в Ansible. Кроме того, в этом формате удобно записывать вручную параметры, которые должны считывать скрипты.

Примечание: Python позволяет записывать объекты самого языка в файлы и считывать их с помощью модуля Pickle, но этот аспект в книге не рассматривается.

Также в этой части рассматриваются базы данных. Хотя данные можно записать с соблюдением структуры и в CSV или JSON, запрашивать нужную информацию из файлов в этом

формате не всегда удобно. Особенно, когда речь идет о более сложных запросах, в которых указаны несколько критериев.

Для задач такого рода отлично подходят базы данных. В разделе 25 рассматривается СУБД SQLite, а также основы языка SQL.

16. Unicode

Программы, которые мы пишем, не изолированы в себе. Они скачивают данные из Интернета, читают и записывают данные на диск, передают данные через сеть.

Поэтому очень важно понимать разницу между тем, как компьютер хранит и передает данные, и как эти данные воспринимает человек. Мы воспринимаем текст, а компьютер - байты.

В Python 3, соответственно, есть две концепции:

- текст - неизменяемая последовательность Unicode-символов. Для хранения этих символов используется тип строка (str)
- данные - неизменяемая последовательность байтов. Для хранения используется тип bytes

---

Примечание: Более корректно будет сказать, что текст - это неизменяемая последовательность кодов (codepoints) Unicode.

Стандарт Юникод

Юникод - это стандарт, который описывает представление и кодировку почти всех языков и других символов.

Несколько фактов про Юникод:

- стандарт версии 12.1 (май 2019) описывает 137 994 кодов
- каждый код - это номер, который соответствует определенному символу
- стандарт также определяет кодировки - способ представления кода символа в байтах

Каждому символу в Юникод соответствует определенный код. Это число, которое обычно записывается таким образом: `U+0073`, где 0073 - это шестнадцатеричные цифры.

Кроме кода, у каждого символа есть свое уникальное имя. Например, букве «s» соответствует код `U+0073` и имя «LATIN SMALL LETTER S».

Примеры кодов, имен и соответствующих символов:

- `U+0073`, «LATIN SMALL LETTER S» - s
- `U+00F6`, «LATIN SMALL LETTER O WITH DIAERESIS» - ö
- U+1F383, «JACK-O-LANTERN» - 🎃
- U+2615, «HOT BEVERAGE» - ☕
- `U+1f600`, «GRINNING FACE» - 😀

Кодировки

Кодировки позволяют записывать код символа в байтах.

Юникод поддерживает несколько кодировок:

- UTF-8
- UTF-16
- UTF-32

Одна из самых популярных кодировок на сегодняшний день - UTF-8. Эта кодировка использует переменное количество байт для записи символов Юникод.

Примеры символов Юникод и их представление в байтах в кодировке UTF-8:

- H - 48
- i - 69
- 𝄀 - 01 f6 c0
- 𝄁 - 01 f6 80
- ☃ - 26 03

Юникод в Python 3

В Python 3 есть:

- строки - неизменяемая последовательность Unicode-символов. Для хранения этих символов используется тип строка (str)
- байты - неизменяемая последовательность байтов. Для хранения используется тип bytes

Строки

Примеры строк:

```
In [11]: hi = 'привет'

In [12]: hi
Out[12]: 'привет'

In [15]: type(hi)
Out[15]: str

In [13]: beautiful = 'schön'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [14]: beautiful
Out[14]: 'schön'
```

Так как строки - это последовательность кодов Юникод, можно записать строку разными способами.

Символ Юникод можно записать, используя его имя:

```
In [1]: "\N{LATIN SMALL LETTER O WITH DIAERESIS}"
Out[1]: 'ö'
```

Или использовав такой формат:

```
In [4]: "\u00F6"
Out[4]: 'ö'
```

Строку можно записать как последовательность кодов Юникод:

```
In [19]: hi1 = 'привет'

In [20]: hi2 = '\u043f\u0440\u0438\u0432\u0435\u0442'

In [21]: hi2
Out[21]: 'привет'

In [22]: hi1 == hi2
Out[22]: True

In [23]: len(hi2)
Out[23]: 6
```

Функция ord возвращает значение кода Unicode для символа:

```
In [6]: ord('ö')
Out[6]: 246
```

Функция chr возвращает символ Юникод, который соответствует коду:

```
In [7]: chr(246)
Out[7]: 'ö'
```

Байты

Тип bytes - это неизменяемая последовательность байтов.

Байты обозначаются так же, как строки, но с добавлением буквы «b» перед строкой:

```
In [30]: b1 = b'\xd0\xb4\xd0\xb0'

In [31]: b2 = b"\xd0\xb4\xd0\xb0"

In [32]: b3 = b'''\xd0\xb4\xd0\xb0'''

In [36]: type(b1)
Out[36]: bytes

In [37]: len(b1)
Out[37]: 4
```

В Python байты, которые соответствуют символам ASCII, отображаются как эти символы, а не как соответствующие им байты. Это может немного путать, но всегда можно распознать тип bytes по букве b:

```
In [38]: bytes1 = b'hello'

In [39]: bytes1
Out[39]: b'hello'

In [40]: len(bytes1)
Out[40]: 5

In [41]: bytes1.hex()
Out[41]: '68656c6c6f'

In [42]: bytes2 = b'\x68\x65\x6c\x6c\x6f'

In [43]: bytes2
Out[43]: b'hello'
```

Если попытаться написать не ASCII-символ в байтовом литерале, возникнет ошибка:

```
In [44]: bytes3 = b'привет'
  File "<ipython-input-44-dc8b23504fa7>", line 1
    bytes3 = b'привет'
             ^
SyntaxError: bytes can only contain ASCII literal characters.
```

Конвертация между байтами и строками

Избежать работы с байтами нельзя. Например, при работе с сетью или файловой системой, чаще всего, результат возвращается в байтах.

Соответственно, надо знать, как выполнять преобразование байтов в строку и наоборот. Для этого и нужна кодировка.

Кодировку можно представлять как ключ шифрования, который указывает:

- как «зашифровать» строку в байты (str -> bytes). Используется метод encode (похож на encrypt)
- как «расшифровать» байты в строку (bytes -> str). Используется метод decode (похож на decrypt)

Эта аналогия позволяет понять, что преобразования строка-байты и байты-строка должны использовать одинаковую кодировку.

encode, decode

Для преобразования строки в байты используется метод **encode**:

```
In [1]: hi = 'привет'

In [2]: hi.encode('utf-8')
Out[2]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [3]: hi_bytes = hi.encode('utf-8')
```

Чтобы получить строку из байт, используется метод **decode**:

```
In [4]: hi_bytes
Out[4]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [5]: hi_bytes.decode('utf-8')
Out[5]: 'привет'
```

str.encode, bytes.decode

Метод encode есть также в классе str (как и другие методы работы со строками):

```
In [6]: hi
Out[6]: 'привет'

In [7]: str.encode(hi, encoding='utf-8')
Out[7]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
```

А метод decode есть у класса bytes (как и другие методы):

```
In [8]: hi_bytes
Out[8]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [9]: bytes.decode(hi_bytes, encoding='utf-8')
Out[9]: 'привет'
```

В этих методах кодировка может указываться как ключевой аргумент (примеры выше) или как позиционный:

```
In [10]: hi_bytes
Out[10]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [11]: bytes.decode(hi_bytes, 'utf-8')
Out[11]: 'привет'
```

Как работать с Юникодом и байтами

Есть очень простое правило, придерживаясь которого, можно избежать, как минимум, части проблем. Оно называется «Юникод-сэндвич»:

- байты, которые программа считывает, надо как можно раньше преобразовать в Юникод (строку)
- внутри программы работать с Юникод
- Юникод надо преобразовать в байты как можно позже, перед передачей

Примеры конвертации между байтами и строками

Рассмотрим несколько примеров работы с байтами и конвертации байт в строки.

subprocess

Модуль subprocess возвращает результат команды в виде байт:

```
In [1]: import subprocess

In [2]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'],
   ...:                         stdout=subprocess.PIPE)
   ...:

In [3]: result.stdout
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[3]: b'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8: icmp_seq=1␣
↪ttl=43 time=59.4 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms\n64 bytes␣
↪from 8.8.8.8: icmp_seq=3 ttl=43 time=55.1 ms\n\n--- 8.8.8.8 ping statistics ---\n3␣
↪packets transmitted, 3 received, 0% packet loss, time 2002ms\nrtt min/avg/max/mdev = 54.
↪470/56.346/59.440/2.220 ms\n'
```

Если дальше необходимо работать с этим выводом, надо сразу конвертировать его в строку:

```
In [4]: output = result.stdout.decode('utf-8')

In [5]: print(output)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=59.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=55.1 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 54.470/56.346/59.440/2.220 ms
```

Модуль subprocess поддерживает еще один вариант преобразования - параметр encoding. Если указать его при вызове функции run, результат будет получен в виде строки:

```
In [6]: result = subprocess.run(['ping', '-c', '3', '-n', '8.8.8.8'],
   ...:                         stdout=subprocess.PIPE, encoding='utf-8')
   ...:

In [7]: result.stdout
Out[7]: 'PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.\n64 bytes from 8.8.8.8: icmp_seq=1␣
↪ttl=43 time=55.5 ms\n64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.6 ms\n64 bytes␣
↪from 8.8.8.8: icmp_seq=3 ttl=43 time=53.3 ms\n\n--- 8.8.8.8 ping statistics ---\n3␣
↪packets transmitted, 3 received, 0% packet loss, time 2003ms\nrtt min/avg/max/mdev = 53.
↪368/54.534/55.564/0.941 ms\n'

In [8]: print(result.stdout)
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=43 time=55.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=43 time=54.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=43 time=53.3 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 53.368/54.534/55.564/0.941 ms
```

telnetlib

В зависимости от модуля, преобразование между строками и байтами может выполняться автоматически, а может требоваться явно.

Например, в модуле telnetlib необходимо передавать байты в методах read_until и write:

```
import telnetlib
import time

t = telnetlib.Telnet('192.168.100.1')

t.read_until(b'Username:')
t.write(b'cisco\n')

t.read_until(b'Password:')
t.write(b'cisco\n')
t.write(b'sh ip int br\n')

time.sleep(5)

output = t.read_very_eager().decode('utf-8')
print(output)
```

И возвращает метод байты, поэтому в предпоследней строке используется decode.

pexpect

Модуль pexpect как аргумент ожидает строку, а возвращает байты:

```
In [9]: import pexpect

In [10]: output = pexpect.run('ls -ls')

In [11]: output
Out[11]: b'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_futures\
↪r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug  3 07:59 iterator_generator\r\n'

In [12]: output.decode('utf-8')
Out[12]: 'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_futures\
↪r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug  3 07:59 iterator_generator\r\n'
```

И также поддерживает вариант передачи кодировки через параметр encoding:

```
In [13]: output = pexpect.run('ls -ls', encoding='utf-8')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [14]: output
Out[14]: 'total 8\r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug 28 12:16 concurrent_futures\
↪r\n4 drwxr-xr-x 2 vagrant vagrant 4096 Aug  3 07:59 iterator_generator\r\n'
```

Работа с файлами

До сих пор при работе с файлами использовалась такая конструкция:

```
with open(filename) as f:
    for line in f:
        print(line)
```

Но на самом деле, при чтении файла происходит конвертация байт в строки. И при этом использовалась кодировка по умолчанию:

```
In [1]: import locale

In [2]: locale.getpreferredencoding()
Out[2]: 'UTF-8'
```

Кодировка по умолчанию в файле:

```
In [2]: f = open('r1.txt')

In [3]: f
Out[3]: <_io.TextIOWrapper name='r1.txt' mode='r' encoding='UTF-8'>
```

При работе с файлами лучше явно указывать кодировку, так как в разных ОС она может отличаться:

```
In [4]: with open('r1.txt', encoding='utf-8') as f:
   ...:     for line in f:
   ...:         print(line, end='')
   ...:
!
service timestamps debug datetime msec localtime show-timezone year
service timestamps log datetime msec localtime show-timezone year
service password-encryption
service sequence-numbers
!
no ip domain lookup
!
ip ssh version 2
!
```

Выводы

Эти примеры показаны тут для того, чтобы показать, что разные модули могут по-разному подходить к вопросу конвертации между строками и байтами. И разные функции и методы этих модулей могут ожидать аргументы и возвращать значения разных типов. Однако все эти вещи написаны в документации.

Ошибки при конвертации

При конвертации между строками и байтами очень важно точно знать, какая кодировка используется, а также знать о возможностях разных кодировок.

Например, кодировка ASCII не может преобразовать в байты кириллицу:

```
In [32]: hi_unicode = 'привет'

In [33]: hi_unicode.encode('ascii')
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-33-ec69c9fd2dae> in <module>()
----> 1 hi_unicode.encode('ascii')

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-5: ordinal not in␣
↪range(128)
```

Аналогично, если строка «привет» преобразована в байты, и попробовать преобразовать ее в строку с помощью ascii, тоже получим ошибку:

```
In [34]: hi_unicode = 'привет'

In [35]: hi_bytes = hi_unicode.encode('utf-8')

In [36]: hi_bytes.decode('ascii')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-36-aa0ada5e44e9> in <module>()
----> 1 hi_bytes.decode('ascii')

UnicodeDecodeError: 'ascii' codec can't decode byte 0xd0 in position 0: ordinal not in␣
↪range(128)
```

Еще один вариант ошибки, когда используются разные кодировки для преобразований:

```
In [37]: de_hi_unicode = 'grüezi'

In [38]: utf_16 = de_hi_unicode.encode('utf-16')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [39]: utf_16.decode('utf-8')
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-39-4b4c731e69e4> in <module>()
----> 1 utf_16.decode('utf-8')

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte
```

Наличие ошибок - это хорошо. Они явно говорят, в чем проблема. Хуже, когда получается так:

```
In [40]: hi_unicode = 'привет'

In [41]: hi_bytes = hi_unicode.encode('utf-8')

In [42]: hi_bytes
Out[42]: b'\xd0\xbf\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'

In [43]: hi_bytes.decode('utf-16')
Out[43]: '□□□□□□'
```

Обработка ошибок

У методов encode и decode есть режимы обработки ошибок, которые указывают, как реагировать на ошибку преобразования.

Параметр errors в encode

По умолчанию encode использует режим `strict` - при возникновении ошибок кодировки генерируется исключение UnicodeError. Примеры такого поведения были выше.

Вместо этого режима можно использовать replace, чтобы заменить символ знаком вопроса:

```
In [44]: de_hi_unicode = 'grüezi'

In [45]: de_hi_unicode.encode('ascii', 'replace')
Out[45]: b'gr?ezi'
```

Или namereplace, чтобы заменить символ именем:

```
In [46]: de_hi_unicode = 'grüezi'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [47]: de_hi_unicode.encode('ascii', 'namereplace')
Out[47]: b'gr\\N{LATIN SMALL LETTER U WITH DIAERESIS}ezi'
```

Кроме того, можно полностью игнорировать символы, которые нельзя закодировать:

```
In [48]: de_hi_unicode = 'grüezi'

In [49]: de_hi_unicode.encode('ascii', 'ignore')
Out[49]: b'grezi'
```

Параметр errors в decode

В методе decode по умолчанию тоже используется режим strict и генерируется исключение UnicodeDecodeError.

Если изменить режим на ignore, как и в encode, символы будут просто игнорироваться:

```
In [50]: de_hi_unicode = 'grüezi'

In [51]: de_hi_utf8 = de_hi_unicode.encode('utf-8')

In [52]: de_hi_utf8
Out[52]: b'gr\xc3\xbcezi'

In [53]: de_hi_utf8.decode('ascii', 'ignore')
Out[53]: 'grezi'
```

Режим replace заменит символы:

```
In [54]: de_hi_unicode = 'grüezi'

In [55]: de_hi_utf8 = de_hi_unicode.encode('utf-8')

In [56]: de_hi_utf8.decode('ascii', 'replace')
Out[56]: 'gr??ezi'
```

Дополнительные материалы

Документация Python:

- What’s New In Python 3: Text Vs. Data Instead Of Unicode Vs. 8-bit
- Unicode HOWTO

Статьи:

- Pragmatic Unicode - статья, презентация и видео
- Раздел «Strings» книги «Dive Into Python 3» - очень хорошо написано о Unicode, кодировках и как все это работает в Python

Без привязки к Python:

- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
- The Unicode Consortium
- Unicode (Wikipedia)
- UTF-8 (Wikipedia)
- CRLF vs. LF: Normalizing Line Endings in Git

17. Работа с файлами в формате CSV, JSON, YAML

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

Например, это могут быть:

- файлы в формате YAML или JSON
- файлы в формате CSV
- база данных

Кроме того, Python позволяет записывать объекты самого языка (этот аспект в курсе не рассматривается, но, если вам интересно, посмотрите на модуль Pickle).

В этом разделе рассматриваются форматы CSV, JSON, YAML, а в следующем разделе - базы данных.

Для чего могут пригодится форматы YAML, JSON, CSV:

- у вас могут быть данные об IP-адресах и подобной информации, которую нужно обработать, в таблицах
  - таблицу можно экспортировать в формат CSV и обрабатывать её с помощью Python
- управляющий софт может возвращать данные в JSON. Соответственно, преобразовав эти данные в объект Python, с ними можно работать и делать что угодно
- YAML очень удобно использовать для описания параметров, так как у него довольно приятный синтаксис
  - например, это могут быть параметры настройки различных объектов (IP-адреса, VLAN и др.)
  - как минимум, знание формата YAML пригодится при использовании Ansible

Для каждого из этих форматов в Python есть модуль, который существенно упрощает работу с ними.

Работа с файлами в формате CSV

CSV (comma-separated value) - это формат представления табличных данных (например, это могут быть данные из таблицы или данные из БД).

В этом формате каждая строка файла - это строка таблицы. Несмотря на название формата, разделителем может быть не только запятая.

И хотя у форматов с другим разделителем может быть и собственное название, например, TSV (tab separated values), тем не менее, под форматом CSV понимают, как правило, любые разделители.

Пример файла в формате CSV (sw_data.csv):

```
hostname,vendor,model,location
sw1,Cisco,3750,London
sw2,Cisco,3850,Liverpool
sw3,Cisco,3650,Liverpool
sw4,Cisco,3650,London
```

В стандартной библиотеке Python есть модуль csv, который позволяет работать с файлами в CSV формате.

Чтение

Пример чтения файла в формате CSV (файл csv_read.py):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Вывод будет таким:

```
$ python csv_read.py
['hostname', 'vendor', 'model', 'location']
['sw1', 'Cisco', '3750', 'London']
['sw2', 'Cisco', '3850', 'Liverpool']
['sw3', 'Cisco', '3650', 'Liverpool']
['sw4', 'Cisco', '3650', 'London']
```

В первом списке находятся названия столбцов, а в остальных соответствующие значения.

Обратите внимание, что сам csv.reader возвращает итератор:

```
In [1]: import csv

In [2]: with open('sw_data.csv') as f:
   ...:     reader = csv.reader(f)
   ...:     print(reader)
   ...:
<_csv.reader object at 0x10385b050>
```

При необходимости его можно превратить в список таким образом:

```
In [3]: with open('sw_data.csv') as f:
   ...:     reader = csv.reader(f)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:     print(list(reader))
   ...:
[['hostname', 'vendor', 'model', 'location'], ['sw1', 'Cisco', '3750', 'London'], ['sw2',
↪'Cisco', '3850', 'Liverpool'], ['sw3', 'Cisco', '3650', 'Liverpool'], ['sw4', 'Cisco',
↪'3650', 'London']]
```

Чаще всего заголовки столбцов удобней получить отдельным объектом. Это можно сделать таким образом (файл csv_read_headers.py):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.reader(f)
    headers = next(reader)
    print('Headers: ', headers)
    for row in reader:
        print(row)
```

Иногда в результате обработки гораздо удобней получить словари, в которых ключи - это названия столбцов, а значения - значения столбцов.

Для этого в модуле есть **DictReader** (файл csv_read_dict.py):

```
import csv

with open('sw_data.csv') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row)
        print(row['hostname'], row['model'])
```

Вывод будет таким:

```
$ python csv_read_dict.py
{'hostname': 'sw1', 'vendor': 'Cisco', 'model': '3750', 'location': 'London, Globe Str 1
↪'}
sw1 3750
{'hostname': 'sw2', 'vendor': 'Cisco', 'model': '3850', 'location': 'Liverpool'}
sw2 3850
{'hostname': 'sw3', 'vendor': 'Cisco', 'model': '3650', 'location': 'Liverpool'}
sw3 3650
{'hostname': 'sw4', 'vendor': 'Cisco', 'model': '3650', 'location': 'London, Grobe Str 1'}
sw4 3650
```

Примечание: До Python 3.8 возвращался отдельный тип упорядоченные словари

(OrderedDict).

Запись

Аналогичным образом с помощью модуля csv можно и записать файл в формате CSV (файл csv_write.py):

```
import csv

data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f)
    for row in data:
        writer.writerow(row)

with open('sw_data_new.csv') as f:
    print(f.read())
```

В примере выше строки из списка сначала записываются в файл, а затем содержимое файла выводится на стандартный поток вывода.

Вывод будет таким:

```
$ python csv_write.py
hostname,vendor,model,location
sw1,Cisco,3750,"London, Best str"
sw2,Cisco,3850,"Liverpool, Better str"
sw3,Cisco,3650,"Liverpool, Better str"
sw4,Cisco,3650,"London, Best str"
```

Обратите внимание на интересную особенность: строки в последнем столбце взяты в кавычки, а остальные значения - нет.

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

Иногда лучше, чтобы все строки были в кавычках. Конечно, в данном случае достаточно простой пример, но когда в строках больше значений, то кавычки позволяют указать, где начинается и заканчивается значение.

Модуль csv позволяет управлять этим. Для того, чтобы все строки записывались в CSV-файл с кавычками, надо изменить скрипт таким образом (файл csv_write_quoting.py):

```
import csv


data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
    writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    for row in data:
        writer.writerow(row)

with open('sw_data_new.csv') as f:
    print(f.read())
```

Теперь вывод будет таким:

```
$ python csv_write_quoting.py
"hostname","vendor","model","location"
"sw1","Cisco","3750","London, Best str"
"sw2","Cisco","3850","Liverpool, Better str"
"sw3","Cisco","3650","Liverpool, Better str"
"sw4","Cisco","3650","London, Best str"
```

Теперь все значения с кавычками. И поскольку номер модели задан как строка в изначальном списке, тут он тоже в кавычках.

Кроме метода writerow, поддерживается метод writerows. Ему можно передать любой итерируемый объект.

Например, предыдущий пример можно записать таким образом (файл csv_writerows.py):

```
import csv

data = [['hostname', 'vendor', 'model', 'location'],
        ['sw1', 'Cisco', '3750', 'London, Best str'],
        ['sw2', 'Cisco', '3850', 'Liverpool, Better str'],
        ['sw3', 'Cisco', '3650', 'Liverpool, Better str'],
        ['sw4', 'Cisco', '3650', 'London, Best str']]


with open('sw_data_new.csv', 'w') as f:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    writer = csv.writer(f, quoting=csv.QUOTE_NONNUMERIC)
    writer.writerows(data)

with open('sw_data_new.csv') as f:
    print(f.read())
```

DictWriter

С помощью DictWriter можно записать словари в формат CSV.

В целом DictWriter работает так же, как writer, но так как словари не упорядочены, надо указывать явно в каком порядке будут идти столбцы в файле. Для этого используется параметр fieldnames (файл csv_write_dict.py):

```
import csv

data = [{
    'hostname': 'sw1',
    'location': 'London',
    'model': '3750',
    'vendor': 'Cisco'
}, {
    'hostname': 'sw2',
    'location': 'Liverpool',
    'model': '3850',
    'vendor': 'Cisco'
}, {
    'hostname': 'sw3',
    'location': 'Liverpool',
    'model': '3650',
    'vendor': 'Cisco'
}, {
    'hostname': 'sw4',
    'location': 'London',
    'model': '3650',
    'vendor': 'Cisco'
}]

with open('csv_write_dictwriter.csv', 'w') as f:
    writer = csv.DictWriter(
        f, fieldnames=list(data[0].keys()), quoting=csv.QUOTE_NONNUMERIC)
    writer.writeheader()
    for d in data:
        writer.writerow(d)
```

Указание разделителя

Иногда в качестве разделителя используются другие значения. В таком случае должна быть возможность подсказать модулю, какой именно разделитель использовать.

Например, если в файле используется разделитель `;` (файл sw_data2.csv):

```
hostname;vendor;model;location
sw1;Cisco;3750;London
sw2;Cisco;3850;Liverpool
sw3;Cisco;3650;Liverpool
sw4;Cisco;3650;London
```

Достаточно просто указать, какой разделитель используется в reader (файл csv_read_delimiter.py):

```
import csv

with open('sw_data2.csv') as f:
    reader = csv.reader(f, delimiter=';')
    for row in reader:
        print(row)
```

Работа с файлами в формате JSON

JSON (JavaScript Object Notation) - это текстовый формат для хранения и обмена данными.

JSON по синтаксису очень похож на Python и достаточно удобен для восприятия.

Как и в случае с CSV, в Python есть модуль, который позволяет легко записывать и читать данные в формате JSON.

Чтение

Файл sw_templates.json:

```
{
  "access": [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable"
  ],
  "trunk": [
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "switchport trunk encapsulation dot1q",
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
  ]
}
```

Для чтения в модуле json есть два метода:

- `json.load` - метод считывает файл в формате JSON и возвращает объекты Python
- `json.loads` - метод считывает строку в формате JSON и возвращает объекты Python

`json.load`

Чтение файла в формате JSON в объект Python (файл json_read_load.py):

```
import json

with open('sw_templates.json') as f:
    templates = json.load(f)

print(templates)

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

Вывод будет таким:

```
$ python json_read_load.py
{'access': ['switchport mode access', 'switchport access vlan', 'switchport nonegotiate',
↪'spanning-tree portfast', 'spanning-tree bpduguard enable'], 'trunk': ['switchport␣
↪trunk encapsulation dot1q', 'switchport mode trunk', 'switchport trunk native vlan 999',
↪ 'switchport trunk allowed vlan']}
access
switchport mode access
switchport access vlan
switchport nonegotiate
spanning-tree portfast
spanning-tree bpduguard enable
trunk
switchport trunk encapsulation dot1q
switchport mode trunk
switchport trunk native vlan 999
switchport trunk allowed vlan
```

json.loads

Считывание строки в формате JSON в объект Python (файл json_read_loads.py):

```
import json

with open('sw_templates.json') as f:
    file_content = f.read()
    templates = json.loads(file_content)

print(templates)

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

Результат будет аналогичен предыдущему выводу.

Запись

Запись файла в формате JSON также осуществляется достаточно легко.

Для записи информации в формате JSON в модуле json также два метода:

- `json.dump` - метод записывает объект Python в файл в формате JSON
- `json.dumps` - метод возвращает строку в формате JSON

json.dumps

Преобразование объекта в строку в формате JSON (json_write_dumps.py):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

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

to_json = {'trunk': trunk_template, 'access': access_template}
```

(continues on next page)

(продолжение с предыдущей страницы)

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

with open('sw_templates.json') as f:
    print(f.read())
```

Метод `json.dumps` подходит для ситуаций, когда надо вернуть строку в формате JSON. Например, чтобы передать ее API.

json.dump

Запись объекта Python в файл в формате JSON (файл json_write_dump.py):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

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

to_json = {'trunk': trunk_template, 'access': access_template}

with open('sw_templates.json', 'w') as f:
    json.dump(to_json, f)

with open('sw_templates.json') as f:
    print(f.read())
```

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

Дополнительные параметры методов записи

Методам dump и dumps можно передавать дополнительные параметры для управления форматом вывода.

По умолчанию эти методы записывают информацию в компактном представлении. Как правило, когда данные используются другими программами, визуальное представление данных не важно. Если же данные в файле нужно будет считать человеку, такой формат не очень удобно воспринимать.

К счастью, модуль json позволяет управлять подобными вещами.

Передав дополнительные параметры методу dump (или методу dumps), можно получить более удобный для чтения вывод (файл json_write_indent.py):

```
import json

trunk_template = [
    'switchport trunk encapsulation dot1q', 'switchport mode trunk',
    'switchport trunk native vlan 999', 'switchport trunk allowed vlan'
]

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

to_json = {'trunk': trunk_template, 'access': access_template}

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

with open('sw_templates.json') as f:
    print(f.read())
```

Теперь содержимое файла sw_templates.json выглядит так:

```
{
  "access": [
    "switchport mode access",
    "switchport access vlan",
    "switchport nonegotiate",
    "spanning-tree portfast",
    "spanning-tree bpduguard enable"
  ],
  "trunk": [
    "switchport trunk encapsulation dot1q",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "switchport mode trunk",
    "switchport trunk native vlan 999",
    "switchport trunk allowed vlan"
  ]
}
```

Изменение типа данных

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

Например, кортежи при записи в JSON превращаются в списки:

```
In [1]: import json

In [2]: trunk_template = ('switchport trunk encapsulation dot1q',
   ...:                   'switchport mode trunk',
   ...:                   'switchport trunk native vlan 999',
   ...:                   'switchport trunk allowed vlan')

In [3]: print(type(trunk_template))
<class 'tuple'>

In [4]: with open('trunk_template.json', 'w') as f:
   ...:     json.dump(trunk_template, f, sort_keys=True, indent=2)
   ...:

In [5]: cat trunk_template.json
[
  "switchport trunk encapsulation dot1q",
  "switchport mode trunk",
  "switchport trunk native vlan 999",
  "switchport trunk allowed vlan"
]
In [6]: templates = json.load(open('trunk_template.json'))

In [7]: type(templates)
Out[7]: list

In [8]: print(templates)
['switchport trunk encapsulation dot1q', 'switchport mode trunk', 'switchport trunk␣
↪native vlan 999', 'switchport trunk allowed vlan']
```

Так происходит из-за того, что в JSON используются другие типы данных и не для всех типов данных Python есть соответствия.

Таблица конвертации данных Python в JSON:

Python	JSON
dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

Таблица конвертации JSON в данные Python:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Ограничение по типам данных

В формат JSON нельзя записать словарь, у которого ключи - кортежи:

```
In [23]: to_json = {('trunk', 'cisco'): trunk_template, 'access': access_template}

In [24]: with open('sw_templates.json', 'w') as f:
    ...:     json.dump(to_json, f)
    ...:
...
TypeError: key ('trunk', 'cisco') is not a string
```

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)
    ...:
    ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [27]: cat sw_templates.json
{"access": ["switchport mode access", "switchport access vlan", "switchport nonegotiate",
↪"spanning-tree portfast", "spanning-tree bpduguard enable"]}
```

Кроме того, в JSON ключами словаря могут быть только строки. Но, если в словаре Python использовались числа, ошибки не будет. Вместо этого выполнится конвертация чисел в строки:

```
In [28]: d = {1: 100, 2: 200}

In [29]: json.dumps(d)
Out[29]: '{"1": 100, "2": 200}'
```

Работа с файлами в формате YAML

YAML (YAML Ain't Markup Language) - еще один текстовый формат для записи данных.

YAML более приятен для восприятия человеком, чем JSON, поэтому его часто используют для описания сценариев в ПО. Например, в Ansible.

Синтаксис YAML

Как и Python, YAML использует отступы для указания структуры документа. Но в YAML можно использовать только пробелы и нельзя использовать знаки табуляции.

Еще одна схожесть с Python: комментарии начинаются с символа # и продолжаются до конца строки.

Список

Список может быть записан в одну строку:

```
[switchport mode access, switchport access vlan, switchport nonegotiate]
```

Или каждый элемент списка в своей строке:

```
- switchport mode access
- switchport access vlan
- switchport nonegotiate
```

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
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  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
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Чтение из YAML (файл yaml_read.py):

```
import yaml
from pprint import pprint

with open('info.yaml') as f:
    templates = yaml.safe_load(f)

pprint(templates)
```

Результат:

```
$ python yaml_read.py
[{'BS': 1550,
  'IT': 791,
  'id': 11,
  'name': 'Liverpool',
  'to_id': 1,
  'to_name': 'LONDON'},
 {'BS': 1510,
  'IT': 793,
  'id': 12,
  'name': 'Bristol',
  'to_id': 1,
  'to_name': 'LONDON'},
 {'BS': 1650,
  'IT': 892,
  'id': 14,
  'name': 'Coventry',
  'to_id': 2,
  'to_name': 'Manchester'}]
```

Формат YAML очень удобен для хранения различных параметров, особенно, если они заполняются вручную.

Запись в YAML

Запись объектов Python в YAML (файл yaml_write.py):

```
import yaml

to_yaml = {
   'access': ['switchport mode access',
              'switchport access vlan',
              'switchport nonegotiate',
              'spanning-tree portfast',
              'spanning-tree bpduguard enable'],
   'trunk': ['switchport trunk encapsulation dot1q',
             'switchport mode trunk',
             'switchport trunk native vlan 999',
             'switchport trunk allowed vlan'],
}

with open('sw_templates.yaml', 'w') as f:
    yaml.dump(to_yaml, f)

with open('sw_templates.yaml') as f:
    print(f.read())
```

Файл sw_templates.yaml выглядит таким образом:

```
access:
- switchport mode access
- switchport access vlan
- switchport nonegotiate
- spanning-tree portfast
- spanning-tree bpduguard enable
trunk:
- switchport trunk encapsulation dot1q
- switchport mode trunk
- switchport trunk native vlan 999
- switchport trunk allowed vlan
```

yaml.full_load()

Дополнительные материалы

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

- CSV
- JSON
- YAML

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

- CSV
- JSON

Пример получения данных в формате JSON через GitHub API:

- Пример работы с GitHub API с помощью requests
- Запись кириллицы и других не ASCII символов в формате JSON

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

> **Предупреждение:** Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 17.1

Создать функцию write_dhcp_snooping_to_csv, которая обрабатывает вывод команды show dhcp snooping binding из разных файлов и записывает обработанные данные в csv файл.

Аргументы функции:

- filenames - список с именами файлов с выводом show dhcp snooping binding
- output - имя файла в формате csv, в который будет записан результат

Функция ничего не возвращает.

Например, если как аргумент был передан список с одним файлом sw3_dhcp_snooping.txt:

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  --------------------
00:E9:BC:3F:A6:50   100.1.1.6        76260       dhcp-snooping   3    FastEthernet0/20
00:E9:22:11:A6:50   100.1.1.7        76260       dhcp-snooping   3    FastEthernet0/21
Total number of bindings: 2
```

В итоговом csv файле должно быть такое содержимое:

```
switch,mac,ip,vlan,interface
sw3,00:E9:BC:3F:A6:50,100.1.1.6,3,FastEthernet0/20
sw3,00:E9:22:11:A6:50,100.1.1.7,3,FastEthernet0/21
```

Первый столбец в csv файле имя коммутатора надо получить из имени файла, остальные - из содержимого в файлах.

Проверить работу функции на содержимом файлов sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt.

Задание 17.2

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 как аргумент вывод команды sh version одной строкой (не имя файла)
- обрабатывает вывод, с помощью регулярных выражений
- возвращает кортеж из трёх элементов:
  - ios - в формате «12.4(5)T»
  - image - в формате «flash:c2800-advipservicesk9-mz.124-5.T.bin»
  - uptime - в формате «5 days, 3 hours, 3 minutes»

У функции write_inventory_to_csv должно быть два параметра:

- data_filenames - ожидает как аргумент список имен файлов с выводом sh version
- csv_filename - ожидает как аргумент имя файла (например, routers_inventory.csv), в который будет записана информация в формате CSV

Функция write_inventory_to_csv записывает содержимое в файл, в формате CSV и ничего не возвращает

Функция write_inventory_to_csv должна делать следующее:

- 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 CSV файл

В файле routers_inventory.csv должны быть такие столбцы: `hostname, ios, image, uptime`

В скрипте, с помощью модуля glob, создан список файлов, имя которых начинается на sh_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 17.3

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

Функция ожидает, как аргумент, вывод команды одной строкой (не имя файла). Функция должна возвращать словарь, который описывает соединения между устройствами.

Например, если как аргумент был передан такой вывод:

```
R4>show cdp neighbors

Device ID    Local Intrfce   Holdtme     Capability       Platform    Port ID
R5           Fa 0/1          122           R S I           2811       Fa 0/1
R6           Fa 0/2          143           R S I           2811       Fa 0/0
```

Функция должна вернуть такой словарь:

```
{"R4": {"Fa 0/1": {"R5": "Fa 0/1"},
        "Fa 0/2": {"R6": "Fa 0/0"}}}
```

Интерфейсы должны быть записаны с пробелом. То есть, так Fa 0/0, а не так Fa0/0.

Проверить работу функции на содержимом файла sh_cdp_n_sw1.txt

Задание 17.3a

Создать функцию generate_topology_from_cdp, которая обрабатывает вывод команды show cdp neighbor из нескольких файлов и записывает итоговую топологию в один словарь.

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_filename - имя файла в формате YAML, в который сохранится топология.
  - значение по умолчанию - None. По умолчанию, топология не сохраняется в файл

  - топология сохраняется только, если save_to_filename как аргумент указано имя файла

Функция должна возвращать словарь, который описывает соединения между устройствами, независимо от того сохраняется ли топология в файл.

Структура словаря должна быть такой:

```
{"R4": {"Fa 0/1": {"R5": "Fa 0/1"},
        "Fa 0/2": {"R6": "Fa 0/0"}},
 "R5": {"Fa 0/1": {"R4": "Fa 0/1"}},
 "R6": {"Fa 0/0": {"R4": "Fa 0/2"}}}
```

Интерфейсы должны быть записаны с пробелом. То есть, так Fa 0/0, а не так Fa0/0.

Проверить работу функции generate_topology_from_cdp на списке 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

Проверить работу параметра save_to_filename и записать итоговый словарь в файл topology.yaml.

Задание 17.3b

Создать функцию transform_topology, которая преобразует топологию в формат подходящий для функции draw_topology.

Функция ожидает как аргумент имя файла в формате YAML, в котором хранится топология.

Функция должна считать данные из YAML файла, преобразовать их соответственно, чтобы функция возвращала словарь такого вида:

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

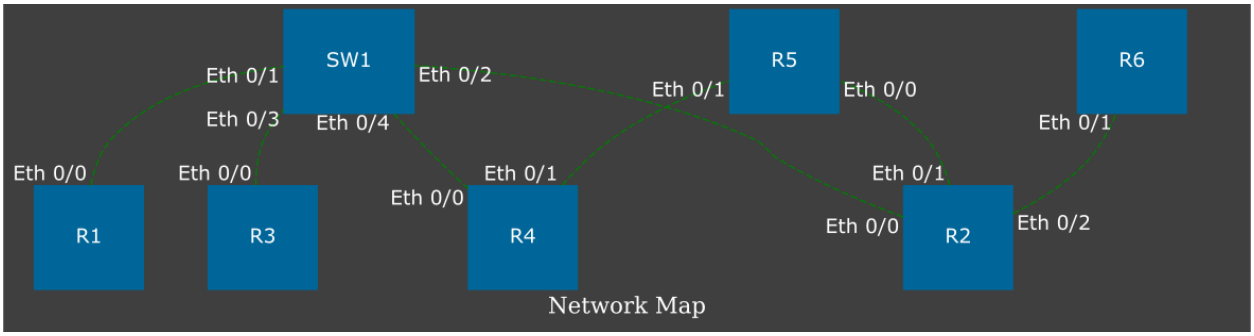
Функция transform_topology должна не только менять формат представления топологии, но и удалять «дублирующиеся» соединения (их лучше всего видно на схеме, которую генерирует функция draw_topology из файла draw_network_graph.py). Тут «дублирующиеся» соединения, это ситуация когда в словаре есть два соединения:

```
("R1", "Eth0/0"): ("SW1", "Eth0/1")
("SW1", "Eth0/1"): ("R1", "Eth0/0")
```

Из-за того что один и тот же линк описывается дважды, на схеме будут лишние соединения. Задача оставить только один из этих линков в итоговом словаре, не важно какой.

Проверить работу функции на файле topology.yaml (должен быть создан в задании 17.3a). На основании полученного словаря надо сгенерировать изображение топологии с помощью функции draw_topology. Не копировать код функции draw_topology из файла draw_network_graph.py.

Результат должен выглядеть так же, как схема в файле task_17_3b_topology.svg



При этом:

- Интерфейсы должны быть записаны с пробелом Fa 0/0
- Расположение устройств на схеме может быть другим
- Соединения должны соответствовать схеме
- На схеме не должно быть дублирующихся линков

Примечание: Для выполнения этого задания, должен быть установлен graphviz: apt-get install graphviz

И модуль python для работы с graphviz: pip install graphviz

Задание 17.4

Создать функцию write_last_log_to_csv.

Аргументы функции:

- source_log - имя файла в формате csv, из которого читаются данные (пример mail_log.csv)
- output - имя файла в формате csv, в который будет записан результат

Функция ничего не возвращает.

Функция write_last_log_to_csv обрабатывает csv файл mail_log.csv. В файле mail_log.csv находятся логи изменения имени пользователя. При этом, email пользователь менять не может, только имя.

Функция write_last_log_to_csv должна отбирать из файла mail_log.csv только самые свежие записи для каждого пользователя и записывать их в другой csv файл. В файле output первой строкой должны быть заголовки столбцов, такие же как в файле source_log.

Для части пользователей запись только одна и тогда в итоговый файл надо записать только ее. Для некоторых пользователей есть несколько записей с разными именами. Например пользователь с email c3po@gmail.com несколько раз менял имя:

```
C=3PO,c3po@gmail.com,16/12/2019 17:10
C3PO,c3po@gmail.com,16/12/2019 17:15
C-3PO,c3po@gmail.com,16/12/2019 17:24
```

Из этих трех записей, в итоговый файл должна быть записана только одна - самая свежая:

```
C-3PO,c3po@gmail.com,16/12/2019 17:24
```

Для сравнения дат удобно использовать объекты datetime из модуля datetime. Чтобы упростить работу с датами, создана функция convert_str_to_datetime - она конвертирует строку с датой в формате 11/10/2019 14:05 в объект datetime. Полученные объекты datetime можно сравнивать между собой. Вторая функция convert_datetime_to_str делает обратную операцию - превращает объект datetime в строку.

Функции convert_str_to_datetime и convert_datetime_to_str использовать не обязательно.

```
import datetime


def convert_str_to_datetime(datetime_str):
    """
    Конвертирует строку с датой в формате 11/10/2019 14:05 в объект datetime.
    """
    return datetime.datetime.strptime(datetime_str, "%d/%m/%Y %H:%M")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def convert_datetime_to_str(datetime_obj):
    """
    Конвертирует строку с датой в формате 11/10/2019 14:05 в объект datetime.
    """
    return datetime.datetime.strftime(datetime_obj, "%d/%m/%Y %H:%M")
```

6

V. Работа с сетевым оборудованием

В этой части рассматриаются:

- подключение к оборудованию по SSH и Telnet
- одновременное подключение к нескольким устройствам
- создание шаблонов конфигурации с помощью Jinja2
- обработка вывода команд с помощью TextFSM

18. 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:

- `pexpect` - 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 pexpect 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 pexpect, 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
- `scrapli` - это модуль, который позволяет подключаться к сетевому оборудованию используя Telnet, SSH или NETCONF.

В этом разделе рассматриваются все 5 модулей, а также как подключаться к нескольким устройствам параллельно. В примерах раздела используются три маршрутизатора. К ним нет никаких требований, только настроенный SSH.

Параметры, которые используются в разделе:

- пользователь: cisco
- пароль: cisco
- пароль на режим enable: cisco

- SSH версии 2
- IP-адреса: 192.168.100.1, 192.168.100.2, 192.168.100.3

Ввод пароля

При подключении к оборудованию вручную, как правило, пароль также вводится вручную.

При автоматизации подключения надо решить, каким образом будет передаваться пароль:

- Запрашивать пароль при старте скрипта и считывать ввод пользователя. Минус в том, что будет видно, какие символы вводит пользователь
- Записывать логин и пароль в каком-то файле (это не очень безопасно).

Как правило, один и тот же пользователь использует одинаковый логин и пароль для подключения к оборудованию. И, как правило, будет достаточно запросить логин и пароль при старте скрипта, а затем использовать их для подключения на разные устройства.

К сожалению, если использовать `input()`, набираемый пароль будет виден. А хотелось бы, чтобы при вводе пароля вводимые символы не отображались.

Модуль getpass

Модуль getpass позволяет запрашивать пароль, не отображая вводимые символы:

```
In [1]: import getpass

In [2]: password = getpass.getpass()
Password:

In [3]: print(password)
testpass
```

Переменные окружения

Еще один вариант хранения пароля (а можно и пользователя) - переменные окружения.

Например, таким образом логин и пароль записываются в переменные:

```
$ export SSH_USER=user
$ export SSH_PASSWORD=userpass
```

А затем в Python считываются 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

Примечание: Pexpect - это реализация expect на Python.

Для начала, модуль pexpect нужно установить:

```
pip install pexpect
```

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.

pexpect.spawn

Класс spawn позволяет взаимодействовать с вызванной программой, отправляя данные и ожидая ответ.

Например, таким образом можно инициировать соединение SSH:

```
In [5]: ssh = pexpect.spawn('ssh cisco@192.168.100.1')
```

После выполнения этой строки, подключение готово. Теперь необходимо указать какую строку ожидать. В данном случае, надо дождаться запроса о пароле:

```
In [6]: ssh.expect('[Pp]assword')
Out[6]: 0
```

Обратите внимание как описана строка, которую ожидает pexpect: `[Pp]assword`. Это регулярное выражение, которое описывает строку password или Password. То есть, методу expect можно передавать регулярное выражение как аргумент.

Метод expect вернул число 0 в результате работы. Это число указывает, что совпадение было найдено и что это элемент с индексом ноль. Индекс тут фигурирует из-за того, что expect можно передавать список строк. Например, можно передать список с двумя элементами:

```
In [7]: ssh = pexpect.spawn('ssh cisco@192.168.100.1')

In [8]: ssh.expect(['password', 'Password'])
Out[8]: 1
```

Обратите внимание, что теперь возвращается 1. Это значит, что совпадением было слово Password.

Теперь можно отправлять пароль. Для этого используется команда sendline:

```
In [9]: ssh.sendline('cisco')
Out[9]: 6
```

Команда sendline отправляет строку, автоматически добавляет к ней перевод строки на основе значения os.linesep, а затем возвращает число указывающее сколько байт было записано.

Примечание: В pexpect есть несколько вариантов отправки команд, не только sendline.

Для того чтобы попасть в режим enable цикл expect-sendline повторяется:

```
In [10]: ssh.expect('[>#]')
Out[10]: 0

In [11]: ssh.sendline('enable')
Out[11]: 7

In [12]: ssh.expect('[Pp]assword')
Out[12]: 0

In [13]: ssh.sendline('cisco')
Out[13]: 6

In [14]: ssh.expect('[>#]')
Out[14]: 0
```

Теперь можно отправлять команду:

```
In [15]: ssh.sendline('sh ip int br')
Out[15]: 13
```

После отправки команды, pexpect надо указать до какого момента считать вывод. Указываем, что считать надо до #:

```
In [16]: ssh.expect('#')
Out[16]: 0
```

Вывод команды находится в атрибуте before:

```
In [17]: ssh.before
Out[17]: b'sh ip int br\r\nInterface                  IP-Address      OK? Method Status  ␣
↪             Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up      ␣
↪             up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up      ␣
↪             up      \r\nEthernet0/2                19.1.1.1        YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3                192.168.230.1   YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3.100            10.100.0.1      YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3.200            10.200.0.1      YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3.300            10.30.0.1       YES NVRAM  up      ␣
↪             up      \r\nR1'
```

Так как результат выводится в виде последовательности байтов, надо конвертировать ее в строку:

```
In [18]: show_output = ssh.before.decode('utf-8')

In [19]: print(show_output)
sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                19.1.1.1        YES NVRAM  up                    up
Ethernet0/3                192.168.230.1   YES NVRAM  up                    up
Ethernet0/3.100            10.100.0.1      YES NVRAM  up                    up
Ethernet0/3.200            10.200.0.1      YES NVRAM  up                    up
Ethernet0/3.300            10.30.0.1       YES NVRAM  up                    up
R1
```

Завершается сессия вызовом метода close:

```
In [20]: ssh.close()
```

Специальные символы в shell

Pexpect не интерпретирует специальные символы shell, такие как >, |, *.

Для того, чтобы, например, команда `ls -ls | grep SUMMARY` отработала, нужно запустить shell таким образом:

```
In [1]: import pexpect

In [2]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep pexpect"')

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

In [4]: print(p.before)
b'4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py\r\n'

In [5]: print(p.before.decode('utf-8'))
4 -rw-r--r-- 1 vagrant vagrant 3203 Jul 14 07:15 1_pexpect.py
```

pexpect.EOF

В предыдущем примере встретилось использование pexpect.EOF.

Примечание: 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')
...
```

Если передать в expect EOF, ошибки не будет.

Метод pexpect.expect

В pexpect.expect как шаблон может использоваться:

- регулярное выражение
- EOF - этот шаблон позволяет среагировать на исключение EOF
- TIMEOUT - исключение timeout (по умолчанию значение timeout = 30 секунд)
- compiled re

Еще одна очень полезная возможность pexpect.expect: можно передавать не одно значение, а список.

Например:

```
In [7]: p = pexpect.spawn('/bin/bash -c "ls -ls | grep netmiko"')

In [8]: p.expect(['py3_convert', pexpect.TIMEOUT, pexpect.EOF])
Out[8]: 2
```

Тут несколько важных моментов:

- когда pexpect.expect вызывается со списком, можно указывать разные ожидаемые строки
- кроме строк, можно указывать исключения
- pexpect.expect возвращает номер элемента списка, который сработал
  - в данном случае номер 2, так как исключение EOF находится в списке под номером два
- за счет такого формата можно делать ответвления в программе, в зависимости от того, с каким элементом было совпадение

Пример использования pexpect

Пример использования pexpect для подключения к оборудованию и передачи команды show (файл 1_pexpect.py):

```
import pexpect
import re
from pprint import pprint


def send_show_command(ip, username, password, enable, commands, prompt="#"):
    with pexpect.spawn(f"ssh {username}@{ip}", timeout=10, encoding="utf-8") as ssh:
        ssh.expect("[Pp]assword")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        ssh.sendline(password)
        enable_status = ssh.expect([">", "#"])
        if enable_status == 0:
            ssh.sendline("enable")
            ssh.expect("[Pp]assword")
            ssh.sendline(enable)
            ssh.expect(prompt)

        ssh.sendline("terminal length 0")
        ssh.expect(prompt)

        result = {}
        for command in commands:
            ssh.sendline(command)
            match = ssh.expect([prompt, pexpect.TIMEOUT, pexpect.EOF])
            if match == 1:
                print(
                    f"Символ {prompt} не найден в выводе. Полученный вывод записан в␣
↪словарь"
                )
            if match == 2:
                print("Соединение разорвано со стороны сервера")
                return result
            else:
                output = ssh.before
                result[command] = output.replace("\r\n", "\n")
        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh clock", "sh int desc"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", commands)
        pprint(result, width=120)
```

Эта часть функции отвечает за переход в режим enable:

```
enable_status = ssh.expect([">", "#"])
if enable_status == 0:
    ssh.sendline("enable")
    ssh.expect("[Pp]assword")
    ssh.sendline(enable)
    ssh.expect(prompt)
```

Если `ssh.expect([">", "#"])` возвращает индекс 0, значит при подключении не было авто-

матического перехода в режим enable и его надо выполнить. Если возвращается индекс 1 - значит мы уже находимся в режиме enable, например, потому что на оборудовании настроено privilege 15.

Еще один интересный момент в функции:

```
for command in commands:
    ssh.sendline(command)
    match = ssh.expect([prompt, pexpect.TIMEOUT, pexpect.EOF])
    if match == 1:
        print(
            f"Символ {prompt} не найден в выводе. Полученный вывод записан в словарь"
        )
    if match == 2:
        print("Соединение разорвано со стороны сервера")
        return result
    else:
        output = ssh.before
        result[command] = output.replace("\r\n", "\n")
return result
```

Тут по очереди отправляются команды и expect ждет три варианта: приглашение, таймаут или EOF. Если метод expect не дождался #, будет возвращено значение 1 и в этом случае выводится сообщение, что символ не найден. При этом, и когда совпадение найдено и когда был таймаут, полученный вывод записывается в словарь. Таким образом можно увидеть, что было получено с устройства, даже если приглашение не найдено.

Вывод при запуске скрипта:

```
{'sh clock': 'sh clock\n*13:13:47.525 UTC Sun Jul 19 2020\n',
 'sh int desc': 'sh int desc\n'
                'Interface                      Status         Protocol Description\n'
                'Et0/0                          up             up       \n'
                'Et0/1                          up             up       \n'
                'Et0/2                          up             up       \n'
                'Et0/3                          up             up       \n'
                'Lo22                           up             up       \n'
                'Lo33                           up             up       \n'
                'Lo45                           up             up       \n'
                'Lo55                           up             up       \n'}
{'sh clock': 'sh clock\n*13:13:50.450 UTC Sun Jul 19 2020\n',
 'sh int desc': 'sh int desc\n'
                'Interface                      Status         Protocol Description\n'
                'Et0/0                          up             up       \n'
                'Et0/1                          up             up       \n'
                'Et0/2                          admin down     down     \n'
                'Et0/3                          admin down     down     \n'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                'Lo0                            up             up       \n'
                'Lo9                            up             up       \n'
                'Lo19                           up             up       \n'
                'Lo33                           up             up       \n'
                'Lo100                          up             up       \n'}
{'sh clock': 'sh clock\n*13:13:53.360 UTC Sun Jul 19 2020\n',
 'sh int desc': 'sh int desc\n'
                'Interface                      Status         Protocol Description\n'
                'Et0/0                          up             up       \n'
                'Et0/1                          up             up       \n'
                'Et0/2                          admin down     down     \n'
                'Et0/3                          admin down     down     \n'
                'Lo33                           up             up       \n'}
```

Работа с pexpect без отключения постраничного вывода команд

Иногда вывод команды сильно большой и его не получается полностью считать или оборудование не дает возможность отключить постраничный вывод. В этом случае необходим немного другой подход.

Примечание: Эта же задача будет повторяться и для других модулей этого раздела.

Пример использования pexpect для работы с постраничным выводом команд show (файл 1_pexpect_more.py):

```
import pexpect
import re
from pprint import pprint


def send_show_command(ip, username, password, enable, command, prompt="#"):
    with pexpect.spawn(f"ssh {username}@{ip}", timeout=10, encoding="utf-8") as ssh:
        ssh.expect("[Pp]assword")
        ssh.sendline(password)
        enable_status = ssh.expect([">", "#"])
        if enable_status == 0:
            ssh.sendline("enable")
            ssh.expect("[Pp]assword")
            ssh.sendline(enable)
            ssh.expect(prompt)

        ssh.sendline(command)
        output = ""
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        while True:
            match = ssh.expect([prompt, "--More--", pexpect.TIMEOUT])
            page = ssh.before.replace("\r\n", "\n")
            page = re.sub(" +\x08+ +\x08+", "\n", page)
            output += page
            if match == 0:
                break
            elif match == 1:
                ssh.send(" ")
            else:
                print("Ошибка: timeout")
                break
        output = re.sub("\n +\n", "\n", output)
        return output


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", "sh run")
        with open(f"{ip}_result.txt", "w") as f:
            f.write(result)
```

Теперь после отправки команды, метод expect ждет еще один вариант `--More--` - признак, что дальше идет еще одна страница. Так как заранее не известно сколько именно страниц будет в выводе, чтение выполняется в цикле `while True`. Цикл прерывается если встретилось приглашение # или в течение 10 секунд не появилось приглашение или `--More--`.

Если встретилось `--More--`, страницы еще не закончились и надо пролистнуть следующую. В Cisco для этого надо нажать пробел (без перевода строки). Поэтому тут используется метод send, а не sendline - sendline автоматически добавляет перевод строки.

Эта строка `page = re.sub(" +\x08+ +\x08+", "\n", page)` удаляет backspace символы, которые находятся вокруг `--More--` чтобы они не попали в итоговый вывод.

Модуль telnetlib

Модуль telnetlib входит в стандартную библиотеку Python. Это реализация клиента telnet.

Примечание: Подключиться по telnet можно и используя pexpect. Плюс telnetlib в том, что этот модуль входит в стандартную библиотеку Python.

Принцип работы telnetlib напоминает pexpect, но есть несколько отличий. Самое заметное

отличие в том, что telnetlib требует передачи байтовой строки, а не обычной.

Подключение выполняется таким образом:

```
In [1]: telnet = telnetlib.Telnet('192.168.100.1')
```

Метод read_until

С помощью метода read_until указывается до какой строки считать вывод. При этом, как аргумент надо передавать не обычную строку, а байты:

```
In [2]: telnet.read_until(b'Username')
Out[2]: b'\r\n\r\nUser Access Verification\r\n\r\nUsername'
```

Метод read_until возвращает все, что он считал до указанной строки.

Метод write

Для передачи данных используется метод write. Ему нужно передавать байтовую строку:

```
In [3]: telnet.write(b'cisco\n')
```

Читаем вывод до слова Password и передаем пароль:

```
In [4]: telnet.read_until(b'Password')
Out[4]: b': cisco\r\nPassword'

In [5]: telnet.write(b'cisco\n')
```

Теперь можно указать, что надо считать вывод до приглашения, а затем отправить команду:

```
In [6]: telnet.read_until(b'>')
Out[6]: b': \r\nR1>'

In [7]: telnet.write(b'sh ip int br\n')
```

После отправки команды можно продолжать использовать метод read_until:

```
In [8]: telnet.read_until(b'>')
Out[8]: b'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
↪            Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up
↪            up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up
↪            up      \r\nEthernet0/2                19.1.1.1        YES NVRAM  up
↪            up      \r\nEthernet0/3                192.168.230.1   YES NVRAM  up
↪            up      \r\nEthernet0/3.100            10.100.0.1      YES NVRAM  up
```

(continues on next page)

(продолжение с предыдущей страницы)

```
↪            up      \r\nEthernet0/3.200            10.200.0.1      YES NVRAM  up
↪            up      \r\nEthernet0/3.300            10.30.0.1       YES NVRAM  up
↪            up      \r\nR1>'
```

Метод read_very_eager

Или использовать еще один метод для чтения read_very_eager. При использовании метода read_very_eager, можно отправить несколько команд, а затем считать весь доступный вывод:

```
In [9]: telnet.write(b'sh arp\n')

In [10]: telnet.write(b'sh clock\n')

In [11]: telnet.write(b'sh ip int br\n')

In [12]: all_result = telnet.read_very_eager().decode('utf-8')

In [13]: print(all_result)
sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.30.0.1               -   aabb.cc00.6530  ARPA   Ethernet0/3.300
Internet  10.100.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.100
Internet  10.200.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.200
Internet  19.1.1.1                -   aabb.cc00.6520  ARPA   Ethernet0/2
Internet  192.168.100.1           -   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2         124   aabb.cc00.6600  ARPA   Ethernet0/0
Internet  192.168.100.3         143   aabb.cc00.6700  ARPA   Ethernet0/0
Internet  192.168.100.100       160   aabb.cc80.c900  ARPA   Ethernet0/0
Internet  192.168.200.1           -   0203.e800.6510  ARPA   Ethernet0/1
Internet  192.168.200.100        13   0800.27ac.16db  ARPA   Ethernet0/1
Internet  192.168.230.1           -   aabb.cc00.6530  ARPA   Ethernet0/3
R1>sh clock
*19:18:57.980 UTC Fri Nov 3 2017
R1>sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                19.1.1.1        YES NVRAM  up                    up
Ethernet0/3                192.168.230.1   YES NVRAM  up                    up
Ethernet0/3.100            10.100.0.1      YES NVRAM  up                    up
Ethernet0/3.200            10.200.0.1      YES NVRAM  up                    up
Ethernet0/3.300            10.30.0.1       YES NVRAM  up                    up
R1>
```

Предупреждение: Перед методом read_very_eager всегда надо ставить time.sleep(n).

С read_until будет немного другой подход. Можно выполнить те же три команды, но затем получать вывод по одной за счет чтения до строки с приглашением:

```
In [14]: telnet.write(b'sh arp\n')

In [15]: telnet.write(b'sh clock\n')

In [16]: telnet.write(b'sh ip int br\n')

In [17]: telnet.read_until(b'>')
Out[17]: b'sh arp\r\nProtocol  Address          Age (min)  Hardware Addr   Type  ␣
↪Interface\r\nInternet  10.30.0.1               -   aabb.cc00.6530  ARPA   Ethernet0/3.
↪300\r\nInternet  10.100.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.100\r\
↪nInternet  10.200.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.200\r\
↪nInternet  19.1.1.1                -   aabb.cc00.6520  ARPA   Ethernet0/2\r\nInternet ␣
↪192.168.100.1           -   aabb.cc00.6500  ARPA   Ethernet0/0\r\nInternet  192.168.100.
↪2         126   aabb.cc00.6600  ARPA   Ethernet0/0\r\nInternet  192.168.100.3        ␣
↪145   aabb.cc00.6700  ARPA   Ethernet0/0\r\nInternet  192.168.100.100       162   aabb.
↪cc80.c900  ARPA   Ethernet0/0\r\nInternet  192.168.200.1           -   0203.e800.6510 ␣
↪ARPA   Ethernet0/1\r\nInternet  192.168.200.100        15   0800.27ac.16db  ARPA  ␣
↪Ethernet0/1\r\nInternet  192.168.230.1           -   aabb.cc00.6530  ARPA   Ethernet0/3\
↪r\nR1>'

In [18]: telnet.read_until(b'>')
Out[18]: b'sh clock\r\n*19:20:39.388 UTC Fri Nov 3 2017\r\nR1>'

In [19]: telnet.read_until(b'>')
Out[19]: b'sh ip int br\r\nInterface                  IP-Address      OK? Method Status  ␣
↪             Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up      ␣
↪             up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up      ␣
↪             up      \r\nEthernet0/2                19.1.1.1        YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3                192.168.230.1   YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3.100            10.100.0.1      YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3.200            10.200.0.1      YES NVRAM  up      ␣
↪             up      \r\nEthernet0/3.300            10.30.0.1       YES NVRAM  up      ␣
↪             up      \r\nR1>'
```

read_until vs read_very_eager

Важное отличие между read_until и read_very_eager заключается в том, как они реагируют на отсутствие вывода.

Метод read_until ждет определенную строку. По умолчанию, если ее нет, метод «зависнет». Опциональный параметр timeout позволяет указать сколько ждать нужную строку:

```
In [20]: telnet.read_until(b'>', timeout=5)
Out[20]: b''
```

Если за указанное время строка не появилась, возвращается пустая строка.

Метод read_very_eager просто вернет пустую строку, если вывода нет:

```
In [21]: telnet.read_very_eager()
Out[21]: b''
```

Метод expect

Метод expect позволяет указывать список с регулярными выражениями. Он работает похоже на rexpect, но в модуле telnetlib всегда надо передавать список регулярных выражений.

При этом, можно передавать байтовые строки или компилированные регулярные выражения:

```
In [22]: telnet.write(b'sh clock\n')

In [23]: telnet.expect([b'[>#]'])
Out[23]:
(0,
 <_sre.SRE_Match object; span=(46, 47), match=b'>'>,
 b'sh clock\r\n*19:35:10.984 UTC Fri Nov 3 2017\r\nR1>')
```

Метод expect возвращает кортеж их трех элементов:

- индекс выражения, которое совпало
- объект Match
- байтовая строка, которая содержит все считанное до регулярного выражения и включая его

Соответственно, при необходимости, с этими элементами можно дальше работать:

```
In [24]: telnet.write(b'sh clock\n')

In [25]: regex_idx, match, output = telnet.expect([b'[>#]'])
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [26]: regex_idx
Out[26]: 0

In [27]: match.group()
Out[27]: b'>'

In [28]: match
Out[28]: <_sre.SRE_Match object; span=(46, 47), match=b'>'>

In [29]: match.group()
Out[29]: b'>'

In [30]: output
Out[30]: b'sh clock\r\n*19:37:21.577 UTC Fri Nov 3 2017\r\nR1>'

In [31]: output.decode('utf-8')
Out[31]: 'sh clock\r\n*19:37:21.577 UTC Fri Nov 3 2017\r\nR1>'
```

Метод close

Закрывается соединение методом close, но лучше открывать и закрывать сессию с помощью менеджера контекста:

```
In [32]: telnet.close()
```

Примечание: Использование объекта Telnet как менеджера контекса добавлено в версии 3.6

Пример использования telnetlib

Файл 2_telnetlib.py:

```
import telnetlib
import time
from pprint import pprint


def to_bytes(line):
    return f"{line}\n".encode("utf-8")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def send_show_command(ip, username, password, enable, commands):
    with telnetlib.Telnet(ip) as telnet:
        telnet.read_until(b"Username")
        telnet.write(to_bytes(username))
        telnet.read_until(b"Password")
        telnet.write(to_bytes(password))
        index, m, output = telnet.expect([b">", b"#"])
        if index == 0:
            telnet.write(b"enable\n")
            telnet.read_until(b"Password")
            telnet.write(to_bytes(enable))
            telnet.read_until(b"#", timeout=5)
        telnet.write(b"terminal length 0\n")
        telnet.read_until(b"#", timeout=5)
        time.sleep(3)
        telnet.read_very_eager()

        result = {}
        for command in commands:
            telnet.write(to_bytes(command))
            output = telnet.read_until(b"#", timeout=5).decode("utf-8")
            result[command] = output.replace("\r\n", "\n")
        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh ip int br", "sh arp"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", commands)
        pprint(result, width=120)
```

Так как методу write надо передавать байты и добавлять каждый раз перевод строки, создана небольшая функция to_bytes, которая выполняет преобразование в байты и добавление перевода строки.

Выполнение скрипта:

```
{'sh int desc': 'sh int desc\n'
                'Interface              Status         Protocol Description\n'
                'Et0/0                  up             up       \n'
                'Et0/1                  up             up       \n'
                'Et0/2                  up             up       \n'
                'Et0/3                  up             up       \n'
                'R1#',
 'sh ip int br': 'sh ip int br\n'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                 'Interface         IP-Address      OK? Method Status                 ␣
↪Protocol\n'
                 'Ethernet0/0       192.168.100.1   YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/1       192.168.200.1   YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/2       unassigned      YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/3       192.168.130.1   YES NVRAM  up                    up  ␣
↪   \n'
                 'R1#'}
{'sh int desc': 'sh int desc\n'
                'Interface             Status         Protocol Description\n'
                'Et0/0                 up             up       \n'
                'Et0/1                 up             up       \n'
                'Et0/2                 admin down     down     \n'
                'Et0/3                 admin down     down     \n'
                'R2#',
 'sh ip int br': 'sh ip int br\n'
                 'Interface         IP-Address      OK? Method Status                 ␣
↪Protocol\n'
                 'Ethernet0/0       192.168.100.2   YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/1       unassigned      YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/2       unassigned      YES NVRAM  administratively down down␣
↪   \n'
                 'Ethernet0/3       unassigned      YES NVRAM  administratively down down␣
↪   \n'
                 'R2#'}
{'sh int desc': 'sh int desc\n'
                'Interface             Status         Protocol Description\n'
                'Et0/0                 up             up       \n'
                'Et0/1                 up             up       \n'
                'Et0/2                 admin down     down     \n'
                'Et0/3                 admin down     down     \n'
                'R3#',
 'sh ip int br': 'sh ip int br\n'
                 'Interface         IP-Address      OK? Method Status                 ␣
↪Protocol\n'
                 'Ethernet0/0       192.168.100.3   YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/1       unassigned      YES NVRAM  up                    up  ␣
↪   \n'
                 'Ethernet0/2       unassigned      YES NVRAM  administratively down down␣
```

(continues on next page)

(продолжение с предыдущей страницы)

```
↪   \n'
                'Ethernet0/3       unassigned      YES NVRAM  administratively down down␣
↪   \n'
```

Постраничный вывод команд

Пример использования telnetlib для работы с постраничным выводом команд show (файл 2_telnetlib_more.py):

```
import telnetlib
import time
from pprint import pprint
import re


def to_bytes(line):
    return f"{line}\n".encode("utf-8")


def send_show_command(ip, username, password, enable, command):
    with telnetlib.Telnet(ip) as telnet:
        telnet.read_until(b"Username")
        telnet.write(to_bytes(username))
        telnet.read_until(b"Password")
        telnet.write(to_bytes(password))
        index, m, output = telnet.expect([b">", b"#"])
        if index == 0:
            telnet.write(b"enable\n")
            telnet.read_until(b"Password")
            telnet.write(to_bytes(enable))
            telnet.read_until(b"#", timeout=5)
        time.sleep(3)
        telnet.read_very_eager()

        telnet.write(to_bytes(command))
        result = ""

        while True:
            index, match, output = telnet.expect([b"--More--", b"#"], timeout=5)
            output = output.decode("utf-8")
            output = re.sub(" +--More--| +\x08+ +\x08+", "\n", output)
            result += output
            if index in (1, -1):
                break
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            telnet.write(b" ")
            time.sleep(1)
            result.replace("\r\n", "\n")

        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    for ip in devices:
        result = send_show_command(ip, "cisco", "cisco", "cisco", "sh run")
        pprint(result, width=120)
```

Модуль paramiko

Paramiko - это реализация протокола SSHv2 на Python. Paramiko предоставляет функциональность клиента и сервера. В книге рассматривается только функциональность клиента.

Так как Paramiko не входит в стандартную библиотеку модулей Python, его нужно установить:

```
pip install paramiko
```

Подключение выполняется таким образом: сначала создается клиент и выполняются настройки клиента, затем выполняется подключение и получение интерактивной сессии:

```
In [2]: client = paramiko.SSHClient()

In [3]: client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

In [4]: client.connect(hostname="192.168.100.1", username="cisco", password="cisco",
   ...: look_for_keys=False, allow_agent=False)

In [5]: ssh = client.invoke_shell()
```

SSHClient это класс, который представляет соединение к SSH-серверу. Он выполняет аутентификацию клиента. Следующая настройка `set_missing_host_key_policy` не является обязательной, она указывает какую политику использовать, когда выполнятся подключение к серверу, ключ которого неизвестен. Политика `paramiko.AutoAddPolicy()` автоматически добавляет новое имя хоста и ключ в локальный объект HostKeys.

Метод `connect` выполняет подключение к SSH-серверу и аутентифицирует подключение. Параметры:

- `look_for_keys` - по умолчанию paramiko выполняет аутентификацию по ключам. Чтобы отключить это, надо поставить флаг в False

- allow_agent - paramiko может подключаться к локальному SSH агенту ОС. Это нужно при работе с ключами, а так как в данном случае аутентификация выполняется по логину/паролю, это нужно отключить.

После выполнения предыдущей команды уже есть подключение к серверу. Метод invoke_shell позволяет установить интерактивную сессию SSH с сервером.

Метод send

Метод send - отправляет указанную строку в сессию и возвращает количество отправленных байт или ноль если сессия закрыта и не удалось отправить команду:

```
In [7]: ssh.send("enable\n")
Out[7]: 7

In [8]: ssh.send("cisco\n")
Out[8]: 6

In [9]: ssh.send("sh ip int br\n")
Out[9]: 13
```

Предупреждение: В коде после send надо будет ставить time.sleep, особенно между send и recv. Так как это интерактивная сессия и команды набираются медленно, все работает и без пауз.

Метод recv

Метод recv получает данные из сессии. В скобках указывается максимальное значение в байтах, которое нужно получить. Этот метод возвращает считанную строку.

```
In [10]: ssh.recv(3000)
Out[10]: b'\r\nR1>enable\r\nPassword: \r\nR1#sh ip int br\r\nInterface
↪IP-Address      OK? Method Status                Protocol\r\nEthernet0/0
↪192.168.100.1   YES NVRAM  up                    up      \r\nEthernet0/1
↪192.168.200.1   YES NVRAM  up                    up      \r\nEthernet0/2
↪unassigned      YES NVRAM  up                    up      \r\nEthernet0/3
↪192.168.130.1   YES NVRAM  up                    up      \r\nLoopback22
↪10.2.2.2        YES manual up                    up      \r\nLoopback33
↪unassigned      YES unset  up                    up      \r\nLoopback45
↪unassigned      YES unset  up                    up      \r\nLoopback55
↪5.5.5.5         YES manual up                    up      \r\nR1#'
```

Метод close

Метод close закрывает сессию:

```
In [11]: ssh.close()
```

Пример использования paramiko

Пример использования paramiko (файл 3_paramiko.py):

```
import paramiko
import time
import socket
from pprint import pprint


def send_show_command(
    ip,
    username,
    password,
    enable,
    command,
    max_bytes=60000,
    short_pause=1,
    long_pause=5,
):
    cl = paramiko.SSHClient()
    cl.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    cl.connect(
        hostname=ip,
        username=username,
        password=password,
        look_for_keys=False,
        allow_agent=False,
    )
    with cl.invoke_shell() as ssh:
        ssh.send("enable\n")
        ssh.send(f"{enable}\n")
        time.sleep(short_pause)
        ssh.send("terminal length 0\n")
        time.sleep(short_pause)
        ssh.recv(max_bytes)

        result = {}
        for command in commands:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            ssh.send(f"{command}\n")
            ssh.settimeout(5)

            output = ""
            while True:
                try:
                    part = ssh.recv(max_bytes).decode("utf-8")
                    output += part
                    time.sleep(0.5)
                except socket.timeout:
                    break
            result[command] = output

        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh clock", "sh arp"]
    result = send_show_command("192.168.100.1", "cisco", "cisco", "cisco", commands)
    pprint(result, width=120)
```

Результат выполнения скрипта:

```
{'sh arp': 'sh arp\r\n'
           'Protocol  Address          Age (min)  Hardware Addr   Type   Interface\r\n'
           'Internet  192.168.100.1           -   aabb.cc00.6500  ARPA   Ethernet0/0\r\n'
           'Internet  192.168.100.2         124   aabb.cc00.6600  ARPA   Ethernet0/0\r\n'
           'Internet  192.168.100.3         183   aabb.cc00.6700  ARPA   Ethernet0/0\r\n'
           'Internet  192.168.100.100       208   aabb.cc80.c900  ARPA   Ethernet0/0\r\n'
           'Internet  192.168.101.1           -   aabb.cc00.6500  ARPA   Ethernet0/0\r\n'
           'Internet  192.168.102.1           -   aabb.cc00.6500  ARPA   Ethernet0/0\r\n'
           'Internet  192.168.130.1           -   aabb.cc00.6530  ARPA   Ethernet0/3\r\n'
           'Internet  192.168.200.1           -   0203.e800.6510  ARPA   Ethernet0/1\r\n'
           'Internet  192.168.200.100        18   6ee2.6d8c.e75d  ARPA   Ethernet0/1\r\n'
           'R1#',
 'sh clock': 'sh clock\r\n*08:25:22.435 UTC Mon Jul 20 2020\r\nR1#'}
```

Постраничный вывод команд

Пример использования paramiko для работы с постраничным выводом команд show (файл 3_paramiko_more.py):

```
import paramiko
import time
import socket
from pprint import pprint
import re


def send_show_command(
    ip,
    username,
    password,
    enable,
    command,
    max_bytes=60000,
    short_pause=1,
    long_pause=5,
):
    cl = paramiko.SSHClient()
    cl.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    cl.connect(
        hostname=ip,
        username=username,
        password=password,
        look_for_keys=False,
        allow_agent=False,
    )
    with cl.invoke_shell() as ssh:
        ssh.send("enable\n")
        ssh.send(enable + "\n")
        time.sleep(short_pause)
        ssh.recv(max_bytes)

        result = {}
        for command in commands:
            ssh.send(f"{command}\n")
            ssh.settimeout(5)

            output = ""
            while True:
                try:
                    page = ssh.recv(max_bytes).decode("utf-8")
                    output += page
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    time.sleep(0.5)
                except socket.timeout:
                    break
                if "More" in page:
                    ssh.send(" ")
            output = re.sub(" +--More--| +\x08+ +\x08+", "\n", output)
            result[command] = output

        return result


if __name__ == "__main__":
    devices = ["192.168.100.1", "192.168.100.2", "192.168.100.3"]
    commands = ["sh run"]
    result = send_show_command("192.168.100.1", "cisco", "cisco", "cisco", commands)
    pprint(result, width=120)
```

Модуль netmiko

Netmiko - это модуль, который позволяет упростить использование paramiko для сетевых устройств. Netmiko использует paramiko, но при этом создает интерфейс и методы, которые нужны для работы с сетевым оборудованием.

Сначала netmiko нужно установить:

```
pip install netmiko
```

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',
    'host': '192.168.1.1',
    'username': 'user',
    'password': 'userpass',
    'secret': 'enablepass',
    'port': 20022,
}
```

Подключение по SSH

```
ssh = ConnectHandler(**cisco_router)
```

Режим enable

Перейти в режим enable:

```
ssh.enable()
```

Выйти из режима enable:

```
ssh.exit_enable_mode()
```

Отправка команд

В netmiko есть несколько способов отправки команд:

- `send_command` - отправить одну команду
- `send_config_set` - отправить список команд или команду в конфигурационном режиме
- `send_config_from_file` - отправить команды из файла (использует внутри метод `send_config_set`)
- `send_command_timing` - отправить команду и подождать вывод на основании таймера

send_command

Метод send_command позволяет отправить одну команду на устройство.

Например:

```
result = ssh.send_command('show ip int br')
```

Метод работает таким образом:

- отправляет команду на устройство и получает вывод до строки с приглашением или до указанной строки
  - приглашение определяется автоматически
  - если на вашем устройстве оно не определилось, можно просто указать строку, до которой считывать вывод
  - ранее так работал метод send_command_expect, но с версии 1.0.0 так работает send_command, а метод send_command_expect оставлен для совместимости
- метод возвращает вывод команды
- методу можно передавать такие параметры:
  - command_string - команда
  - expect_string - до какой строки считывать вывод
  - delay_factor - параметр позволяет увеличить задержку до начала поиска строки
  - max_loops - количество итераций, до того как метод выдаст ошибку (исключение). По умолчанию 500
  - strip_prompt - удалить приглашение из вывода. По умолчанию удаляется
  - strip_command - удалить саму команду из вывода

В большинстве случаев достаточно будет указать только команду.

send_config_set

Метод send_config_set позволяет отправить команду или несколько команд конфигурационного режима.

Пример использования:

```
commands = ['router ospf 1',
            'network 10.0.0.0 0.255.255.255 area 0',
            'network 192.168.100.0 0.0.0.255 area 1']

result = ssh.send_config_set(commands)
```

Метод работает таким образом:

- заходит в конфигурационный режим,
- затем передает все команды
- и выходит из конфигурационного режима
- в зависимости от типа устройства, выхода из конфигурационного режима может и не быть. Например, для IOS-XR выхода не будет, так как сначала надо закоммитить изменения

send_config_from_file

Метод send_config_from_file отправляет команды из указанного файла в конфигурационный режим.

Пример использования:

```
result = ssh.send_config_from_file('config_ospf.txt')
```

Метод открывает файл, считывает команды и передает их методу send_config_set.

Дополнительные методы

Кроме перечисленных методов для отправки команд, netmiko поддерживает такие методы:

- config_mode - перейти в режим конфигурации: ssh.config_mode()
- exit_config_mode - выйти из режима конфигурации: ssh.exit_config_mode()
- check_config_mode - проверить, находится ли netmiko в режиме конфигурации (возвращает True, если в режиме конфигурации, и False - если нет): ssh.check_config_mode()
- find_prompt - возвращает текущее приглашение устройства: ssh.find_prompt()
- commit - выполнить commit на IOS-XR и Juniper: ssh.commit()
- disconnect - завершить соединение SSH

Примечание: Выше ssh - это созданное предварительно соединение SSH: ssh = ConnectHandler(**cisco_router)

Поддержка 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 = {
    "device_type": "cisco_ios_telnet",
    "host": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}
```

В остальном, методы, которые применимы к SSH, применимы и к Telnet. Пример, аналогичный примеру с SSH (файл 4_netmiko_telnet.py):

```
from pprint import pprint
import yaml
from netmiko import (
    ConnectHandler,
    NetmikoTimeoutException,
    NetmikoAuthenticationException,
)


def send_show_command(device, commands):
    result = {}
    try:
        with ConnectHandler(**device) as ssh:
            ssh.enable()
            for command in commands:
                output = ssh.send_command(command)
                result[command] = output
        return result
    except (NetmikoTimeoutException, NetmikoAuthenticationException) as error:
        print(error)


if __name__ == "__main__":
    device = {
        "device_type": "cisco_ios_telnet",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "secret": "cisco",
    }
    result = send_show_command(device, ["sh clock", "sh ip int br"])
    pprint(result, width=120)
```

Аналогично работают и методы:

- send_command_timing()
- find_prompt()
- send_config_set()
- send_config_from_file()
- check_enable_mode()
- disconnect()

Пример использования netmiko

Пример использования netmiko (файл 4_netmiko.py):

```
from pprint import pprint
import yaml
from netmiko import (
    ConnectHandler,
    NetmikoTimeoutException,
    NetmikoAuthenticationException,
)


def send_show_command(device, commands):
    result = {}
    try:
        with ConnectHandler(**device) as ssh:
            ssh.enable()
            for command in commands:
                output = ssh.send_command(command)
                result[command] = output
        return result
    except (NetmikoTimeoutException, NetmikoAuthenticationException) as error:
        print(error)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)
    for device in devices:
        result = send_show_command(device, ["sh clock", "sh ip int br"])
        pprint(result, width=120)
```

В этом примере не передается команда terminal length, так как netmiko по умолчанию выполняет эту команду.

Результат выполнения скрипта:

```
{'sh clock': '*09:12:15.210 UTC Mon Jul 20 2020',
 'sh ip int br': 'Interface     IP-Address      OK? Method Status                Protocol\
↪n'
                 'Ethernet0/0   192.168.100.1   YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/1   192.168.200.1   YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/2   unassigned      YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/3   192.168.130.1   YES NVRAM  up                    up      \
↪n'}
{'sh clock': '*09:12:24.507 UTC Mon Jul 20 2020',
 'sh ip int br': 'Interface     IP-Address      OK? Method Status                Protocol\
↪n'
                 'Ethernet0/0   192.168.100.2   YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/1   unassigned      YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/2   unassigned      YES NVRAM  administratively down down    \
↪n'
                 'Ethernet0/3   unassigned      YES NVRAM  administratively down down    \
↪n'}
{'sh clock': '*09:12:33.573 UTC Mon Jul 20 2020',
 'sh ip int br': 'Interface     IP-Address      OK? Method Status                Protocol\
↪n'
                 'Ethernet0/0   192.168.100.3   YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/1   unassigned      YES NVRAM  up                    up      \
↪n'
                 'Ethernet0/2   unassigned      YES NVRAM  administratively down down    \
↪n'
                 'Ethernet0/3   unassigned      YES NVRAM  administratively down down    \
↪n'}
```

Постраничный вывод команд

Пример использования paramiko для работы с постраничным выводом команд show (файл 4_netmiko_more.py):

```
from netmiko import ConnectHandler, NetmikoTimeoutException
import yaml


def send_show_command(device_params, command):
    with ConnectHandler(**device_params) as ssh:
        ssh.enable()
        prompt = ssh.find_prompt()
        ssh.send_command("terminal length 100")
        ssh.write_channel(f"{command}\n")
        output = ""
        while True:
            try:
                page = ssh.read_until_pattern(f"More|{prompt}")
                output += page
                if "More" in page:
                    ssh.write_channel(" ")
                elif prompt in output:
                    break
            except NetmikoTimeoutException:
                break
    return output


if __name__ == "__main__":
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)
    print(send_show_command(devices[0], "sh run"))
```

Модуль scrapli

scrapli это модуль, который позволяет подключаться к сетевому оборудованию используя Telnet, SSH или NETCONF.

Также как и netmiko, scrapli может использовать paramiko или telnetlib (и другие модули) для самого подключения, но при этом предоставляет одинаковый интерфейс работы для разных типов подключения и разного оборудования.

Установка scrapli:

```
pip install scrapli
```

Примечание: Рассматривается scrapli версии 2021.1.30.

Три основные составляющие части scrapli:

- transport - это конкретный способ подключения к оборудованию
- channel - следующий уровень над транспортом, который отвечает за отправку команд, получение вывода и другими взаимодействиями с оборудованием
- driver - это интерфейс, который предоставляется пользователю для работы со scrapli. Тут есть как специфические драйверы типа `IOSXEDriver`, который понимает как взаимодействовать с оборудованием конкретного типа, так и базовый драйвер Driver, который предоставляет минимальный интерфейс для работы через SSH/Telnet.

Доступные варианты транспорта:

- system - используется встроенный SSH клиент, подразумевается использование клиента на Linux/MacOS
- paramiko - модуль paramiko
- ssh2 - используется модуль ssh2-python (обертка вокруг C библиотеки libssh2)
- telnet - будет использоваться telnetlib
- asyncssh - модуль asyncssh
- asynctelnet - telnet клиент написанный с использованием asyncio

Основные примеры будут с использованием транспорта system. Так как интерфейс модуля одинаковый для всех синхронных вариантов транспорта, для использовании другого вида транспорта, надо только указать его (для транспорта telnet также обязательно указать порт).

Примечание: Асинхронные варианты транспорта (asyncssh, asynctelnet) рассматриваются в книге Advanced Python для сетевых инженеров

Поддерживаемые платформы:

- Cisco IOS-XE
- Cisco NX-OS
- Juniper JunOS
- Cisco IOS-XR
- Arista EOS

Кроме этих платформ, есть также платформы scrapli community. И одно из преимуществ scrapli то, что добавлять платформы относительно легко.

В scrapli глобально есть два варианта подключения: используя общий класс Scrapli, который выбирает нужный driver по параметру platform или конкретный driver, например, IOSXEDriver. При этом параметры передаются те же самые и конкретному драйверу и Scrapli.

Примечание: Кроме этих вариантов, есть также общие (базовые) драйверы.

Если в scrapli (или scrapli community) нет поддержки необходимой платформы, можно добавить платформу в scrapli community или использовать (не рассматривается в книге):

- Driver
- GenericDriver
- NetworkDriver

Параметры подключения

Основные параметры подключения:

- host - IP-адрес или имя хоста
- auth_username - имя пользователя
- auth_password - пароль
- auth_secondary - пароль на enable
- auth_strict_key - контролирует проверку SSH ключей сервера, а именно разрешать ли подключаться к серверам ключ которых не сохранен в ssh/known_hosts. False - разрешить подключение (по умолчанию значение True)
- platform - нужно указывать при использовании Scrapli
- transport - какой транспорт использовать
- transport_options - опции для конкретного транспорта

Процесс подключения немного отличается в зависимости от того используется менеджер контекста или нет. При подключении без менеджера контекста, сначала надо передать параметры драйверу или Scrapli, а затем вызвать метод open:

```
from scrapli import Scrapli

r1 = {
   "host": "192.168.100.1",
   "auth_username": "cisco",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   "auth_password": "cisco",
   "auth_secondary": "cisco",
   "auth_strict_key": False,
   "platform": "cisco_iosxe"
}

In [2]: ssh = Scrapli(**r1)

In [3]: ssh.open()
```

После этого можно отправлять команды:

```
In [4]: ssh.get_prompt()
Out[4]: 'R1#'

In [5]: ssh.close()
```

При использовании менеджера контекста, open вызывать не надо:

```
In [8]: with Scrapli(**r1) as ssh:
   ...:     print(ssh.get_prompt())
   ...:
R1#
```

Использование драйвера

Доступные драйверы

Оборудование	Драйвер	Параметр platform
Cisco IOS-XE	IOSXEDriver	cisco_iosxe
Cisco NX-OS	NXOSDriver	cisco_nxos
Cisco IOS-XR	IOSXRDriver	cisco_iosxr
Arista EOS	EOSDriver	arista_eos
Juniper JunOS	JunosDriver	juniper_junos

Пример подключения с использованием драйвера IOSXEDriver (технически подключение выполняется к Cisco IOS):

```
In [11]: from scrapli.driver.core import IOSXEDriver

In [12]: r1_driver = {
    ...:    "host": "192.168.100.1",
    ...:    "auth_username": "cisco",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:    "auth_password": "cisco",
    ...:    "auth_secondary": "cisco",
    ...:    "auth_strict_key": False,
    ...: }

In [13]: with IOSXEDriver(**r1_driver) as ssh:
    ...:     print(ssh.get_prompt())
    ...:
R1#
```

Отправка команд

В scrapli есть несколько методов для отправки команд:

- send_command - отправить одну show команду
- send_commands - отправить список show команд
- send_commands_from_file - отправить show команды из файла
- send_config - отправить одну команду в конфигурационном режиме
- send_configs - отправить список команд в конфигурационном режиме
- send_configs_from_file - отправить команды из файла в конфигурационном режиме
- send_interactive

Все эти методы возвращают объект Response, а не вывод команды в виде строки.

Объект Response

Метод send_command и другие методы для отправки команд на оборудование возвращают объект Response (не вывод команды). Response позволяет получить не только вывод команды, но и такие вещи как время работы команды, выполнилась команда с ошибками или без, структурированный вывод с помощью textfsm и так далее.

```
In [15]: reply = ssh.send_command("sh clock")

In [16]: reply
Out[16]: Response(host='192.168.100.1',channel_input='sh clock',textfsm_platform='cisco_
↪iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '%␣
↪Incomplete command', '% Invalid input detected', '% Unknown command'])
```

Получить вывод команды можно обратившись к атрибуту result:

```
In [17]: reply.result
Out[17]: '*17:31:54.232 UTC Wed Mar 31 2021'
```

Атрибут raw_result содержит байтовую строку с полным выводом:

```
In [18]: reply.raw_result
Out[18]: b'\n*17:31:54.232 UTC Wed Mar 31 2021\nR1#'
```

Для команд, которые выполняются дольше обычных show, может быть необходимо знать время выполнения команды:

```
In [18]: r = ssh.send_command("ping 10.1.1.1")

In [19]: r.result
Out[19]: 'Type escape sequence to abort.\nSending 5, 100-byte ICMP Echos to 10.1.1.1,␣
↪timeout is 2 seconds:\n.....\nSuccess rate is 0 percent (0/5)'

In [20]: r.elapsed_time
Out[20]: 10.047594

In [21]: r.start_time
Out[21]: datetime.datetime(2021, 4, 1, 7, 10, 56, 63697)

In [22]: r.finish_time
Out[22]: datetime.datetime(2021, 4, 1, 7, 11, 6, 111291)
```

Атрибут channel_input возвращает команду, которая была отправлена на оборудование:

```
In [23]: r.channel_input
Out[23]: 'ping 10.1.1.1'
```

Метод send_command

Метод send_command позволяет отправить одну команду на устройство.

```
In [14]: reply = ssh.send_command("sh clock")
```

Параметры метода (все эти параметры надо передавать как ключевые):

- `strip_prompt` - удалить приглашение из вывода. По умолчанию удаляется
- `failed_when_contains` - если вывод содержит указанную строку или одну из строк в списке, будет считаться, что команда выполнилась с ошибкой
- `timeout_ops` - максимальное время на выполнение команды, по умолчанию равно 30 секунд для IOSXEDriver

Пример вызова метода send_command:

```
In [15]: reply = ssh.send_command("sh clock")

In [16]: reply
Out[16]: Response(host='192.168.100.1',channel_input='sh clock',textfsm_platform='cisco_
↪iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '%␣
↪Incomplete command', '% Invalid input detected', '% Unknown command'])
```

Параметр timeout_ops указывает сколько ждать выполнения команды:

```
In [19]: ssh.send_command("ping 8.8.8.8", timeout_ops=20)
Out[19]: Response <Success: True>
```

Если команда не выполнилась за указанное время, сгенерируется исключение ScrapliTimeout (вывод сокращен):

```
In [20]: ssh.send_command("ping 8.8.8.8", timeout_ops=2)
---------------------------------------------------------------------------
ScrapliTimeout                            Traceback (most recent call last)
<ipython-input-20-e062fb19f0e6> in <module>
----> 1 ssh.send_command("ping 8.8.8.8", timeout_ops=2)
```

Кроме получения обычного вывода команды, scrapli также позволяет получить структурированный вывод, например, с помощью метода textfsm_parse_output:

```
In [21]: reply = ssh.send_command("sh ip int br")

In [22]: reply.textfsm_parse_output()
Out[22]:
[{'intf': 'Ethernet0/0',
  'ipaddr': '192.168.100.1',
  'status': 'up',
  'proto': 'up'},
 {'intf': 'Ethernet0/1',
  'ipaddr': '192.168.200.1',
  'status': 'up',
  'proto': 'up'},
 {'intf': 'Ethernet0/2',
  'ipaddr': 'unassigned',
  'status': 'up',
  'proto': 'up'},
 {'intf': 'Ethernet0/3',
  'ipaddr': '192.168.130.1',
  'status': 'up',
  'proto': 'up'}]
```

Примечание: Что такое TextFSM и как с ним работать рассматривается в 21 разделе. Scrapli использует готовые шаблоны для того чтобы получать структурированный вывод и в базовых случаях не требует знания TextFSM.

Обнаружение ошибок

Методы для отправки команд автоматически проверяют вывод на наличие ошибок. Для каждого вендора/типа оборудования это свои ошибки, плюс можно самостоятельно указать наличие каких строк в выводе будет считаться ошибкой. По умолчанию для IOSXEDriver ошибками будут считаться такие строки:

```
In [21]: ssh.failed_when_contains
Out[21]:
['% Ambiguous command',
 '% Incomplete command',
 '% Invalid input detected',
 '% Unknown command']
```

Атрибут failed у объекта Response возвращает True, если команда отработала с ошибкой и False, если без ошибки:

```
In [23]: reply = ssh.send_command("sh clck")

In [24]: reply.result
Out[24]: "        ^\n% Invalid input detected at '^' marker."

In [25]: reply
Out[25]: Response(host='192.168.100.1',channel_input='sh clck',textfsm_platform='cisco_
↪iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '%␣
↪Incomplete command', '% Invalid input detected', '% Unknown command'])

In [26]: reply.failed
Out[26]: True
```

Метод send_config

Метод send_config позволяет отправить одну команду конфигурационного режима.

Пример использования:

```
In [33]: r = ssh.send_config("username user1 password password1")
```

Так как scrapli удаляет команду из вывода, по умолчанию, при использовании send_config, в атрибуте result будет пустая строка (если не было ошибки при выполнении команды):

```
In [34]: r.result
Out[34]: ''
```

Можно добавлять параметр `strip_prompt=False` и тогда в выводе появится приглашение:

```
In [37]: r = ssh.send_config("username user1 password password1", strip_prompt=False)

In [38]: r.result
Out[38]: 'R1(config)#'
```

Методы send_commands, send_configs

Методы send_commands, send_configs отличаются от send_command, send_config тем, что могут отправлять несколько команд. Кроме того, эти методы возвращают не Response, а MultiResponse, который можно в целом воспринимать как список Response, по одному для каждой команды.

```
In [44]: reply = ssh.send_commands(["sh clock", "sh ip int br"])

In [45]: reply
Out[45]: [Response(host='192.168.100.1',channel_input='sh clock',textfsm_platform='cisco_
↪iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '%␣
↪Incomplete command', '% Invalid input detected', '% Unknown command']), Response(host=
↪'192.168.100.1',channel_input='sh ip int br',textfsm_platform='cisco_iosxe',genie_
↪platform='iosxe',failed_when_contains=['% Ambiguous command', '% Incomplete command', '
↪% Invalid input detected', '% Unknown command'])]


In [46]: for r in reply:
    ...:     print(r)
    ...:     print(r.result)
    ...:
Response <Success: True>
*08:38:20.115 UTC Thu Apr 1 2021
Response <Success: True>
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
Ethernet0/2                unassigned      YES NVRAM  up                    up
Ethernet0/3                192.168.130.1   YES NVRAM  up                    up

In [47]: reply.result
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[47]: 'sh clock\n*08:38:20.115 UTC Thu Apr 1 2021sh ip int br\nInterface
↪   IP-Address      OK? Method Status                Protocol\nEthernet0/0
↪ 192.168.100.1   YES NVRAM  up                    up\nEthernet0/1                192.
↪168.200.1   YES NVRAM  up                    up\nEthernet0/2                unassigned
↪    YES NVRAM  up                    up\nEthernet0/3                192.168.130.1   YES
↪NVRAM  up                    up'

In [48]: reply[0]
Out[48]: Response(host='192.168.100.1',channel_input='sh clock',textfsm_platform='cisco_
↪iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '%
↪Incomplete command', '% Invalid input detected', '% Unknown command'])

In [49]: reply[1]
Out[49]: Response(host='192.168.100.1',channel_input='sh ip int br',textfsm_platform=
↪'cisco_iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '%
↪Incomplete command', '% Invalid input detected', '% Unknown command'])


In [50]: reply[0].result
Out[50]: '*08:38:20.115 UTC Thu Apr 1 2021'
```

При отправке нескольких команд также очень удобно использовать параметр stop_on_failed. По умолчанию он равен False, поэтому выполняются все команды, но если указать stop_on_failed=True, после возникновения ошибки в какой-то команде, следующие команды не будут выполняться:

```
In [59]: reply = ssh.send_commands(["ping 192.168.100.2", "sh clck", "sh ip int br"],
↪stop_on_failed=True)

In [60]: reply
Out[60]: [Response(host='192.168.100.1',channel_input='ping 192.168.100.2',textfsm_
↪platform='cisco_iosxe',genie_platform='iosxe',failed_when_contains=['% Ambiguous command
↪', '% Incomplete command', '% Invalid input detected', '% Unknown command']),
↪Response(host='192.168.100.1',channel_input='sh clck',textfsm_platform='cisco_iosxe',
↪genie_platform='iosxe',failed_when_contains=['% Ambiguous command', '% Incomplete
↪command', '% Invalid input detected', '% Unknown command'])]

In [61]: print(reply)
MultiResponse <Success: False; Response Elements: 2>

In [62]: reply.result
Out[62]: "ping 192.168.100.2\nType escape sequence to abort.\nSending 5, 100-byte ICMP
↪Echos to 192.168.100.2, timeout is 2 seconds:\n!!!!!\nSuccess rate is 100 percent (5/5),
↪ round-trip min/avg/max = 1/2/6 mssh clck\n        ^\n% Invalid input detected at '^'
↪marker."
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [63]: for r in reply:
    ...:     print(r)
    ...:     print(r.result)
    ...:
Response <Success: True>
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 192.168.100.2, timeout is 2 seconds:
!!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 1/2/6 ms
Response <Success: False>
        ^
% Invalid input detected at '^' marker.
```

Подключение telnet

Для подключения к оборудовани по Telnet надо указать transport равным telnet и обязательно указать параметр port равным 23 (или тому порту который используется у вас для подключения по Telnet):

```
from scrapli.driver.core import IOSXEDriver
from scrapli.exceptions import ScrapliException
import socket

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco2",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "transport": "telnet",
    "port": 23,  # обязательно указывать при подключении telnet
}


def send_show(device, show_command):
    try:
        with IOSXEDriver(**device) as ssh:
            reply = ssh.send_command(show_command)
            return reply.result
    except socket.timeout as error:
        print(error)
    except ScrapliException as error:
        print(error, device["host"])
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    output = send_show(r1, "sh ip int br")
    print(output)
```

Примеры использования scrapli

```
from scrapli.driver.core import IOSXEDriver
from scrapli.exceptions import ScrapliException


r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "timeout_socket": 5,  # timeout for establishing socket/initial connection
    "timeout_transport": 10,  # timeout for ssh|telnet transport
}


def send_show(device, show_command):
    try:
        with IOSXEDriver(**device) as ssh:
            reply = ssh.send_command(show_command)
            return reply.result
    except ScrapliException as error:
        print(error, device["host"])


if __name__ == "__main__":
    output = send_show(r1, "sh ip int br")
    print(output)
```

```
from pprint import pprint
from scrapli import Scrapli

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    "auth_strict_key": False,
    "platform": "cisco_iosxe",
}


def send_show(device, show_commands):
    if type(show_commands) == str:
        show_commands = [show_commands]
    cmd_dict = {}
    with Scrapli(**device) as ssh:
        for cmd in show_commands:
            reply = ssh.send_command(cmd)
            cmd_dict[cmd] = reply.result
    return cmd_dict


if __name__ == "__main__":
    print("show".center(20, "#"))
    output = send_show(r1, ["sh ip int br", "sh ver | i uptime"])
    pprint(output, width=120)
```

```
from pprint import pprint
from scrapli import Scrapli

r1 = {
    "host": "192.168.100.1",
    "auth_username": "cisco",
    "auth_password": "cisco",
    "auth_secondary": "cisco",
    "auth_strict_key": False,
    "platform": "cisco_iosxe",
}


def send_cfg(device, cfg_commands, strict=False):
    output = ""
    if type(cfg_commands) == str:
        cfg_commands = [cfg_commands]
    with Scrapli(**device) as ssh:
        reply = ssh.send_configs(cfg_commands, stop_on_failed=strict)
        for cmd_reply in reply:
            if cmd_reply.failed:
                print(f"При выполнении команды возникла ошибка:\n{reply.result}\n")
        output = reply.result
    return output
```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == "__main__":
    output_cfg = send_cfg(
        r1, ["interfacelo11", "ip address 11.1.1.1 255.255.255.255"], strict=True
    )
    print(output_cfg)
```

Дополнительные материалы

Документация:

- pexpect
- telnetlib
- paramiko Client
- paramiko Channel
- netmiko
- scrapli
- scrapli-cfg
- time
- datetime
- getpass

Статьи:

- Netmiko Library
- Automate SSH connections with netmiko
- Network Automation Using Python: BGP Configuration
- A Tale of Five Python SSH Libraries Commentary

Примеры кода:

- netmiko
- scrapli
- netmiko, paramiko, telnetlib, scrapli, pexpect

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 18.1

Создать функцию send_show_command.

Функция подключается по SSH (с помощью netmiko) к ОДНОМУ устройству и выполняет указанную команду.

Параметры функции:

- device - словарь с параметрами подключения к устройству
- command - команда, которую надо выполнить

Функция возвращает строку с выводом команды.

Скрипт должен отправлять команду command на все устройства из файла devices.yaml с помощью функции send_show_command (эта часть кода написана).

```
import yaml



if __name__ == "__main__":
    command = "sh ip int br"
    with open("devices.yaml") as f:
        devices = yaml.safe_load(f)

    for dev in devices:
        print(send_show_command(dev, command))
```

Задание 18.1a

Скопировать функцию send_show_command из задания 18.1 и переделать ее таким образом, чтобы обрабатывалось исключение, которое генерируется при ошибке аутентификации на устройстве.

При возникновении ошибки, на стандартный поток вывода должно выводиться сообщение исключения.

Для проверки измените пароль на устройстве или в файле devices.yaml.

Задание 18.1b

Скопировать функцию send_show_command из задания 18.1a и переделать ее таким образом, чтобы обрабатывалось не только исключение, которое генерируется при ошибке аутентификации на устройстве, но и исключение, которое генерируется, когда IP-адрес устройства недоступен.

При возникновении ошибки, на стандартный поток вывода должно выводиться сообщение исключения.

Для проверки измените IP-адрес на устройстве или в файле devices.yaml.

Задание 18.2

Создать функцию send_config_commands

Функция подключается по SSH (с помощью netmiko) к одному устройству и выполняет перечень команд в конфигурационном режиме на основании переданных аргументов.

Параметры функции:

- device - словарь с параметрами подключения к устройству
- config_commands - список команд, которые надо выполнить

Функция возвращает строку с результатами выполнения команды:

```
In [7]: r1
Out[7]:
{'device_type': 'cisco_ios',
 'ip': '192.168.100.1',
 'username': 'cisco',
 'password': 'cisco',
 'secret': 'cisco'}

In [8]: commands
Out[8]: ['logging 10.255.255.1', 'logging buffered 20010', 'no logging console']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [9]: result = send_config_commands(r1, commands)

In [10]: result
Out[10]: 'config term\nEnter configuration commands, one per line.  End with CNTL/Z.\
↪nR1(config)#logging 10.255.255.1\nR1(config)#logging buffered 20010\nR1(config)#no␣
↪logging console\nR1(config)#end\nR1#'

In [11]: print(result)
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#logging 10.255.255.1
R1(config)#logging buffered 20010
R1(config)#no logging console
R1(config)#end
R1#
```

Скрипт должен отправлять команду command на все устройства из файла devices.yaml с помощью функции send_config_commands.

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

Задание 18.2a

Скопировать функцию send_config_commands из задания 18.2 и добавить параметр log, который контролирует будет ли выводится на стандартный поток вывода информация о том к какому устройству выполняется подключение.

По умолчанию, результат должен выводиться.

Пример работы функции:

```
In [13]: result = send_config_commands(r1, commands)
Подключаюсь к 192.168.100.1...

In [14]: result = send_config_commands(r1, commands, log=False)

In [15]:
```

Скрипт должен отправлять список команд commands на все устройства из файла devices.yaml с помощью функции send_config_commands.

Задание 18.2b

Скопировать функцию send_config_commands из задания 18.2a и добавить проверку на ошибки.

При выполнении каждой команды, скрипт должен проверять результат на такие ошибки:

- Invalid input detected
- Incomplete command
- Ambiguous command

Если при выполнении какой-то из команд возникла ошибка, функция должна выводить сообщение на стандартный поток вывода с информацией о том, какая ошибка возникла, при выполнении какой команды и на каком устройстве, например: Команда «logging» выполнилась с ошибкой «Incomplete command.» на устройстве 192.168.100.1

Ошибки должны выводиться всегда, независимо от значения параметра log. При этом, log по-прежнему должен контролировать будет ли выводиться сообщение: Подключаюсь к 192.168.100.1...

Функция send_config_commands теперь должна возвращать кортеж из двух словарей:

- первый словарь с выводом команд, которые выполнились без ошибки
- второй словарь с выводом команд, которые выполнились с ошибками

Оба словаря в формате (примеры словарей ниже):

- ключ - команда
- значение - вывод с выполнением команд

Проверить работу функции можно на одном устройстве.

Пример работы функции send_config_commands:

```
In [16]: commands
Out[16]:
['logging 0255.255.1',
 'logging',
 'a',
 'logging buffered 20010',
 'ip http server']

In [17]: result = send_config_commands(r1, commands)
Подключаюсь к 192.168.100.1...
Команда "logging 0255.255.1" выполнилась с ошибкой "Invalid input detected at '^' marker.
↪" на устройстве 192.168.100.1
Команда "logging" выполнилась с ошибкой "Incomplete command." на устройстве 192.168.100.1
Команда "a" выполнилась с ошибкой "Ambiguous command:  "a"" на устройстве 192.168.100.1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [18]: pprint(result, width=120)
({'ip http server': 'config term\n'
                    'Enter configuration commands, one per line.  End with CNTL/Z.\n'
                    'R1(config)#ip http server\n'
                    'R1(config)#',
  'logging buffered 20010': 'config term\n'
                            'Enter configuration commands, one per line.  End with CNTL/Z.
↪\n'
                            'R1(config)#logging buffered 20010\n'
                            'R1(config)#'},
 {'a': 'config term\n'
       'Enter configuration commands, one per line.  End with CNTL/Z.\n'
       'R1(config)#a\n'
       '% Ambiguous command:  "a"\n'
       'R1(config)#',
  'logging': 'config term\n'
             'Enter configuration commands, one per line.  End with CNTL/Z.\n'
             'R1(config)#logging\n'
             '% Incomplete command.\n'
             '\n'
             'R1(config)#',
  'logging 0255.255.1': 'config term\n'
                        'Enter configuration commands, one per line.  End with CNTL/Z.\n'
                        'R1(config)#logging 0255.255.1\n'
                        '                   ^\n'
                        "% Invalid input detected at '^' marker.\n"
                        '\n'
                        'R1(config)#'})

In [19]: good, bad = result

In [20]: good.keys()
Out[20]: dict_keys(['logging buffered 20010', 'ip http server'])

In [21]: bad.keys()
Out[21]: dict_keys(['logging 0255.255.1', 'logging', 'a'])
```

Примеры команд с ошибками:

```
R1(config)#logging 0255.255.1
                   ^
% Invalid input detected at '^' marker.

R1(config)#logging
% Incomplete command.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
R1(config)#a
% Ambiguous command:  "a"
```

Списки команд с ошибками и без:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
correct_commands = ['logging buffered 20010', 'ip http server']

commands = commands_with_errors + correct_commands
```

Задание 18.2c

Скопировать функцию send_config_commands из задания 18.2b и переделать ее таким образом: Если при выполнении команды возникла ошибка, спросить пользователя надо ли выполнять остальные команды.

Варианты ответа [y]/n:

- y - выполнять остальные команды. Это значение по умолчанию, поэтому нажатие любой комбинации воспринимается как y
- n или no - не выполнять остальные команды

Функция send_config_commands по-прежнему должна возвращать кортеж из двух словарей:

- первый словарь с выводом команд, которые выполнились без ошибки
- второй словарь с выводом команд, которые выполнились с ошибками

Оба словаря в формате

- ключ - команда
- значение - вывод с выполнением команд

Проверить работу функции можно на одном устройстве.

Пример работы функции:

```
In [11]: result = send_config_commands(r1, commands)
Подключаюсь к 192.168.100.1...
Команда "logging 0255.255.1" выполнилась с ошибкой "Invalid input detected at '^' marker.
↪" на устройстве 192.168.100.1
Продолжать выполнять команды? [y]/n: y
Команда "logging" выполнилась с ошибкой "Incomplete command." на устройстве 192.168.100.1
Продолжать выполнять команды? [y]/n: n
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [12]: pprint(result)
({},
 {'logging': 'config term\n'
             'Enter configuration commands, one per line.  End with CNTL/Z.\n'
             'R1(config)#logging\n'
             '% Incomplete command.\n'
             '\n'
             'R1(config)#',
  'logging 0255.255.1': 'config term\n'
                        'Enter configuration commands, one per line.  End with '
                        'CNTL/Z.\n'
                        'R1(config)#logging 0255.255.1\n'
                        '                   ^\n'
                        "% Invalid input detected at '^' marker.\n"
                        '\n'
                        'R1(config)#'})
```

Списки команд с ошибками и без:

```
commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
correct_commands = ['logging buffered 20010', 'ip http server']

commands = commands_with_errors + correct_commands
```

Задание 18.3

Создать функцию send_commands (для подключения по SSH используется netmiko).

Параметры функции:

- device - словарь с параметрами подключения к одному устройству
- show - одна команда show (строка)
- config - список с командами, которые надо выполнить в конфигурационном режиме

Аргументы show и config должны передаваться только как ключевые. При передачи этих аргументов как позиционных, должно генерироваться исключение TypeError.

```
In [4]: send_commands(r1, 'sh clock')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-75adcfb4a005> in <module>
----> 1 send_commands(r1, 'sh clock')

TypeError: send_commands() takes 1 positional argument but 2 were given
```

В зависимости от того, какой аргумент был передан, функция вызывает разные функции внутри. При вызове функции send_commands, всегда должен передаваться только один из аргументов show, config. Если передаются оба аргумента, должно генерироваться исключение ValueError.

Далее комбинация из аргумента и соответствующей функции:

- show - функция send_show_command из задания 18.1
- config - функция send_config_commands из задания 18.2

Функция возвращает строку с результатами выполнения команд или команды.

Проверить работу функции:

- со списком команд commands
- командой command

Пример работы функции:

```
In [14]: send_commands(r1, show='sh clock')
Out[14]: '*17:06:12.278 UTC Wed Mar 13 2019'

In [15]: commands = ['username user5 password pass5', 'username user6 password pass6']

In [16]: send_commands(r1, config=commands)
Out[16]: 'config term\nEnter configuration commands, one per line.  End with CNTL/Z.\
↪nR1(config)#username user5 password pass5\nR1(config)#username user6 password pass6\
↪nR1(config)#end\nR1#'
```

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

19. Одновременное подключение к нескольким устройствам

Когда нужно опросить много устройств, выполнение подключений поочередно будет достаточно долгим. Конечно, это будет быстрее, чем подключение вручную, но хотелось бы получать отклик как можно быстрее.

Примечание: Все эти «долго» и «быстрее» относительные понятия, но в этом разделе мы научимся и конкретно измерять, сколько отрабатывал скрипт, чтобы сравнить, насколько быстрее будет выполняться подключение.

Для параллельного подключения к устройствам в этом разделе используется модуль concurrent.futures.

Измерение времени выполнения скрипта

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

- утилита Linux time
- и модуль Python datetime

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. Для использования утилиты time достаточно написать time перед строкой запуска скрипта:

```
$ 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.

datetime

Второй вариант - модуль datetime. Этот модуль позволяет работать со временем и датами в Python.

Пример использования:

```
from datetime import datetime
import time

start_time = datetime.now()

#Тут выполняются действия
time.sleep(5)

print(datetime.now() - start_time)
```

Результат выполнения:

```
$ python test.py
0:00:05.004949
```

Процессы и потоки в Python (CPython)

Для начала нам нужно разобраться с терминами:

- процесс (process) - это, грубо говоря, запущенная программа. Процессу выделяются отдельные ресурсы: память, процессорное время
- поток (thread) - это единица исполнения в процессе. Потоки разделяют ресурсы процесса, к которому они относятся.

Python (а точнее, CPython - реализация, которая используется в книге) 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. Однако, если в программе есть некое «ожидание»: пакетов из сети, запроса пользователя, пауза типа time.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.

То есть, потоки отлично подходят для задач, которые связаны с операциями ввода-вывода:

- Подключение к оборудованию и подключение по сети в целом
- Работа с файловой системой
- Скачивание файлов по сети

Примечание: В интернете часто можно встретить выражения «В Python лучше вообще не использовать потоки». К сожалению, такие фразы не всегда пишут с контекстом, а именно, что речь о конкретных задачах, которые завязаны на CPU.

В следующих разделах рассматривается, как использовать потоки для подключения по Telnet/SSH. И проверяется, какое суммарное время будет занимать исполнение скрипта, по сравнению с последовательным исполнением и с использованием процессов.

Процессы

Процессы позволяют выполнять задачи на разных ядрах компьютера. Это важно для задач, которые завязаны на CPU. Для каждого процесса создается своя копия ресурсов, выделяется память, у каждого процесса свой GIL. Это же делает процессы более тяжеловесными, по сравнению с потоками.

Кроме того, количество процессов, которые запускаются параллельно, зависит от количества ядер и CPU и обычно исчисляется в десятках, тогда как количество потоков для операций ввода-вывода может исчисляться в сотнях.

Процессы и потоки можно совмещать, но это усложняет программу и на базовом уровне для операций ввода-вывода лучше остановиться на потоках.

Примечание: Совмещение потоков и процессов, то есть запуск процесса в программе и внутри него уже запуск потоков - сильно усложняет траблшутинг программы. И лучше такой вариант не использовать.

Несмотря на то, что, как правило, для задач ввода-вывода лучше использовать потоки с некоторыми модулями надо использовать процессы, так как они могут некорректно работать с потоками.

Примечание: Помимо процессов и потоков есть еще один вариант одновременного подключения к оборудованию: асинхронное программирование. Этот вариант не рассматривается в

книге.

Количество потоков

Сколько потоков нужно использовать при подключении к оборудованию? На этот вопрос нет однозначного ответа. Количество потоков, как минимум, зависит от того на каком компьютере выполняется скрипт (ОС, память, процессор), от самой сети (задержек).

Поэтому, вместо поиска идеального количества потоков, надо замерить количество на своем компьютере, сети, скрипте. Например, в примерах к этому разделу есть скрипт netmiko_count_threads.py, который запускает одну и ту же функцию с разным количеством потоков и выводит информацию о времени выполнения. В функции по умолчанию используется небольшое количество устройств из файла devices_all.yaml и небольшое количество потоков, однако его можно адаптировать к любому количеству для своей сети.

Пример подключения к 5000 устройств с разным количеством потоков:

```
Количество устройств: 5460

#30 потоков
----------------------------------------
Время выполнения: 0:09:17.187867

#50 потоков
----------------------------------------
Время выполнения: 0:09:17.604252

#70 потоков
----------------------------------------
Время выполнения: 0:09:17.117332

#90 потоков
----------------------------------------
Время выполнения: 0:09:16.693774

#100 потоков
----------------------------------------
Время выполнения: 0:09:17.083294

#120 потоков
----------------------------------------
Время выполнения: 0:09:17.945270

#140 потоков
----------------------------------------
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Время выполнения: 0:09:18.114993

#200 потоков
----------------------------------------
Время выполнения: 0:11:12.951247

#300 потоков
----------------------------------------
Время выполнения: 0:14:03.790432
```

В этом случае время выполнения с 30 потоками и 120 потоков одинаковое, а после время только увеличивается. Так происходит потому что на переключение между потоками также тратится много времени, чем больше потоков, тем больше переключений. И с какого-то момента нет смысла увеличивать количество потоков. Тут оптимальным количеством можно считать, например, 50 потоков. Тут мы берем не 30 чтобы сделать запас.

Потоковая безопасность

При работе с потоками есть несколько рекомендаций, а также правил. Если их соблюдать, работать с потоками будет проще и, скорее всего, не будет проблем именно из-за потоков. Конечно, время от времени, будут появляться задачи, которые потребуют нарушения рекомендаций. Однако, прежде чем делать это, лучше попытаться решить задачу соблюдая рекомендации. Если это невозможно, тогда надо искать как обезопасить решение, чтобы не были повреждены данные.

Очень важная особенность работы с потоками: на небольшом количестве потоков и небольших тестовых задачах «все работает». Например, вывод информации с помощью print, при подключении к 20 устройствам в 5 потоков, будет работать нормально. А при подключении к большому количеству устройств с большим количеством потоков окажется, что иногда сообщения «налазят» друг на друга. Такая особенность проявляется очень часто, поэтому не доверяйте варианту когда «все работает» на базовых примерах, соблюдайте правила работы с потоками.

Прежде чем разбираться с правилами, надо разобраться с термином «потоковая безопасность». Потоковая безопасность - это концепция, которая описывает работу с многопоточными программами. Код считается потокобезопасным (thread-safe), если он может работать нормально при использовании нескольких потоков.

Например, функция print не является потокобезопасной. Это проявляется в том, что когда код выполняет print из разных потоков, сообщения на стандартном потоке вывода могут смешиваться. Может выводиться сначала часть сообщения из одного потока, потом часть из второго, потом часть из первого и так далее. То есть, функция print не работает нормально (как положено) в потоках. В этом случае говорят, что функция print не является потокобезопасной (not thread-safe).

В общем случае, проблем не будет, если каждый поток работает со своими ресурсами. Например, каждый поток пишет данные в свой файл. Однако, это не всегда возможно или может усложнять решение.

Примечание: С print проблемы потому что из разных потоков пишем в один стандартный поток вывода, а print не потокобезопасен.

В том случае, если надо все же из разных потоков писать в один и тот же ресурс, есть два варианта:

1. Писать в один и тот же ресурс после того как работа в потоке закончилась. Например, в потоках 1, 2 и 3 выполнилась функция, ее результат по очереди (последовательно) получен из каждого потока, а затем записан в файл.
2. Использовать потокобезопасную альтернативу (не всегда доступна и/или не всегда простая). Например, вместо функции print использовать модуль logging.

Рекомендации при работе с потоками:

1. Не пишите в один и тот же ресурс из разных потоков, если ресурс или то, чем пишете не предназначено для многопоточной работы. Выяснить это, проще всего, погуглив что-то вроде «python write to file from threads».

- В этой рекомендации есть нюансы. Например, можно писать из разных потоков в один и тот же файл, если использовать Lock или использовать потокобезопасную очередь. Эти варианты, чаще всего, непросты в использовании и не рассматриваются в книге. Скорее всего, 95% задач, с которыми вы будете сталкиваться, можно решить без них.
- К этой категории относится запись/изменение списков/словарей/множеств из разных потоков. Сами по себе эти объекты потокобезопасны, но нет гарантии, что при изменении с одного и того же списка из разных потоков, данные в списке будут правильные. В случае, если надо использовать общий контейнер для данных для разных потоков, надо использовать очередь queue из модуля Queue. Она потокобезопасна и с ней можно работать из разных потоков.

2. Если есть воможность, избегайте коммуникаций между потоками в процессе их работы. Это непростая задача и лучше постараться обойтись без нее.
3. Соблюдайте принцип KISS (Keep it simple, stupid) - постарайтесь, чтобы решение было максимально простым.

Примечание: Эти рекомендации, в целом, написаны для тех, кто только начинает программировать на Python. Однако, как правило, они актуальны для большиства программистов, которые пишут приложения для пользователей, а не фреймворки.

Модуль concurrent.futures, который будет рассматриваться дальше, упрощает соблюдение первого принципа «Не пишите в один и тот же ресурс из разных потоков...». Сам интерфейс

работы с модулем к этому подталкивает, но конечно не запрещает его нарушать.

Однако, перед знакомством с concurrent.futures, надо рассмотреть основы работы с модулем logging. Он будет использоваться вместо функции print, которая не является потокобезопасной.

Модуль logging

Модуль logging - это модуль из стандартной библиотеки Python, который позволяет настраивать логирование из скрипта. У модуля logging очень много возможностей и огромное количество вариантов настройки. В этом разделе рассматривается только базовый вариант настройки.

Самый простой вариант настройки логирования в скрипте, использовать logging.basicConfig:

```
import logging


logging.basicConfig(
    format='%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)
```

В таком варианте настройки:

- все сообщения будут выводиться на стандартный поток вывода,
- будут выводиться сообщения уровня INFO и выше,
- в каждом сообщении будет информация о потоке, имя логера, уровень сообщения и само сообщение.

Теперь, чтобы вывести log-сообщение в этом скрипте, надо написать так `logging.info("тест")`.

Пример скрипта с настройкой логирования: (файл logging_basics.py)

```
from datetime import datetime
import logging
import netmiko
import yaml


# эта строка указывает, что лог-сообщения paramiko будут выводиться
# только если они уровня WARNING и выше
logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    level=logging.INFO)


def send_show(device, show):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device["ip"]
    logging.info(start_msg.format(datetime.now().time(), ip))

    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result =  ssh.send_command(show)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return result


if __name__ == "__main__":
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    for dev in devices:
        print(send_show(dev, 'sh clock'))
```

При выполнении скрипта, вывод будет таким:

```
$ python logging_basics.py
MainThread root INFO: ===> 12:26:12.767168 Connection: 192.168.100.1
MainThread root INFO: <=== 12:26:18.307017 Received:   192.168.100.1
*12:26:18.137 UTC Wed Jun 5 2019
MainThread root INFO: ===> 12:26:18.413913 Connection: 192.168.100.2
MainThread root INFO: <=== 12:26:23.991715 Received:   192.168.100.2
*12:26:23.819 UTC Wed Jun 5 2019
MainThread root INFO: ===> 12:26:24.095452 Connection: 192.168.100.3
MainThread root INFO: <=== 12:26:29.478553 Received:   192.168.100.3
*12:26:29.308 UTC Wed Jun 5 2019
```

Примечание: В модуле logging еще очень много возможностей. В этом разделе используется только базовый вариант настройки. Узнать больше о возможностях модуля можно в Logging HOWTO

Модуль concurrent.futures

Модуль concurrent.futures предоставляет высокоуровневый интерфейс для работы с процессами и потоками. При этом и для потоков, и для процессов используется одинаковый интерфейс, что позволяет легко переключаться между ними.

Если сравнивать этот модуль с threading или multiprocessing, то у него меньше возможностей, но с concurrent.futures работать проще и интерфейс более понятный.

Модуль concurrent.futures позволяет решить задачу запуска нескольких потоков/процессов и получения из них данных. Для этого в модуле используются два класса:

- **ThreadPoolExecutor** - для работы с потоками
- **ProcessPoolExecutor** - для работы с процессами

Оба класса используют одинаковый интерфейс, поэтому достаточно разобраться с одним и затем просто переключиться на другой при необходимости.

Создание объекта Executor на примере ThreadPoolExecutor:

```
executor = ThreadPoolExecutor(max_workers=5)
```

После создания объекта Executor, у него есть три метода: shutdown, map и submit. Метод shutdown отвечает за завершение потоков/процессов, а методы map и submit за запуск функций в разных потоках/процессах.

Примечание: На самом деле, map и submit могут запускать не только функции, а и любой вызываемый объект. Однако далее будут рассматриваться только функции.

Метод shutdown указывает, что объекту Executor надо завершить работу. При этом, если методу shutdown передать значение `wait=True` (значение по умолчанию), он не вернет результат пока не завершатся все функции, которые запущены в потоках. Если же `wait=False`, метод shutdown завершает работу мгновенно, но при этом сам скрипт не завершит работу пока все функции не отработают.

Как правило, метод shutdown не используется явно, так как при создании объекта Executor в менеджере контекста, метод shutdown автоматически вызывается в конце блока with с wait равным True.

```
with ThreadPoolExecutor(max_workers=5) as executor:
    ...
```

Так как методы map и submit запускают какую-то функцию в потоках или процессах, в коде должна присутствовать, как минимум, функция которая выполняет одно действие и которую надо запустить в разных потоках с разными аргументами функции.

Например, если необходимо пинговать несколько IP-адресов в разных потоках, надо создать функцию, которая будет пинговать один IP-адрес, а затем запустить эту функцию в разных потоках для разных IP-адресов с помощью concurrent.futures.

Метод map

Синтаксис метода:

```
map(func, *iterables, timeout=None)
```

Метод map - работает похоже на встроенную функцию map: применяет функцию func к одному или более итерируемых объектов. При этом, каждый вызов функции запускается в отдельном потоке/процессе. Метод map возвращает итератор с результатами выполнения функции для каждого элемента итерируемого объекта. Результаты расположены в том же порядке, что и элементы в итерируемом объекте.

При работе с пулами потоков/процессов, создается определенное количество потоков/процессов и затем код выполняется в этих потоках. Например, если при создании пула указано, что надо создать 5 потоков, а функцию надо запустить для 10 разных устройств, подключение будет выполняться сначала к первым пяти устройствам, а затем, по мере освобождения, к остальным.

Пример использования функции map с ThreadPoolExecutor (файл netmiko_threads_map_basics.py):

```
from datetime import datetime
import time
from itertools import repeat
from concurrent.futures import ThreadPoolExecutor
import logging

import netmiko
import yaml


logging.getLogger('paramiko').setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO,
)


def send_show(device, show):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ip = device['ip']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(show)
        logging.info(received_msg.format(datetime.now().time(), ip))
        return result


with open('devices.yaml') as f:
    devices = yaml.safe_load(f)

with ThreadPoolExecutor(max_workers=3) as executor:
    result = executor.map(send_show, devices, repeat('sh clock'))
    for device, output in zip(devices, result):
        print(device['ip'], output)
```

Так как методу map надо передавать функцию, создана функция send_show, которая подключается к оборудованию, передает указанную команду show и возвращает результат с выводом команды.

```
def send_show(device, show):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device['ip']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with netmiko.ConnectHandler(**device) as ssh:
        ssh.enable()
        result = ssh.send_command(show)
        logging.info(received_msg.format(datetime.now().time(), ip))
        return result
```

Функция send_show выводит лог сообщения в начале и в конце работы. Это позволит определить когда функция отработала для конкретного устройства. Также внутри функции указано, что при подключении к устройству с адресом 192.168.100.1, надо сделать паузу на 5 секунд - таким образом маршрутизатор с этим адресом будет отрабатывать дольше.

Последние 4 строки кода отвечают за подключение к устройствам в отдельных потоках:

```
with ThreadPoolExecutor(max_workers=3) as executor:
    result = executor.map(send_show, devices, repeat('sh clock'))
    for device, output in zip(devices, result):
        print(device['ip'], output)
```

- `with ThreadPoolExecutor(max_workers=3) as executor:` - класс ThreadPoolExecutor инициируется в блоке with с указанием количества потоков.
- **`result = executor.map(send_show, devices, repeat('sh clock'))` - метод map похож на** функцию map, но тут функция send_show вызывается в разных потоках. При этом в разных потоках функция будет вызываться с разными аргументами:
  - элементами итерируемого объекта devices и одной и той же командой sh clock.
  - так как вместо списка команд, тут используется только одна команда, ее надо каким-то образом повторять, чтобы метод map подставлял эту команду разным устройствам. Для этого используется функция repeat - она повторяет команду ровно столько раз, сколько запрашивает map
- метод map возвращает генератор. В этом генераторе содержатся результаты выполнения функций. Результаты находятся в том же порядке, что и устройства в списке devices, поэтому для совмещения IP-адресов устройств и вывода команды используется функция zip.

Результат выполнения:

```
$ python netmiko_threads_map_basics.py
ThreadPoolExecutor-0_0 root INFO: ===> 08:28:55.950254 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 08:28:55.963198 Connection: 192.168.100.2
ThreadPoolExecutor-0_2 root INFO: ===> 08:28:55.970269 Connection: 192.168.100.3
ThreadPoolExecutor-0_1 root INFO: <=== 08:29:11.968796 Received:   192.168.100.2
ThreadPoolExecutor-0_2 root INFO: <=== 08:29:15.497324 Received:   192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 08:29:16.854344 Received:   192.168.100.1
192.168.100.1 *08:29:16.663 UTC Thu Jul 4 2019
192.168.100.2 *08:29:11.744 UTC Thu Jul 4 2019
192.168.100.3 *08:29:15.374 UTC Thu Jul 4 2019
```

Первые три сообщения указывают когда было выполнено подключение и к какому устройству:

```
ThreadPoolExecutor-0_0 root INFO: ===> 08:28:55.950254 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 08:28:55.963198 Connection: 192.168.100.2
ThreadPoolExecutor-0_2 root INFO: ===> 08:28:55.970269 Connection: 192.168.100.3
```

Следующие три сообщения показывают время получения информации и завершения функции:

```
ThreadPoolExecutor-0_1 root INFO: <=== 08:29:11.968796 Received:   192.168.100.2
ThreadPoolExecutor-0_2 root INFO: <=== 08:29:15.497324 Received:   192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 08:29:16.854344 Received:   192.168.100.1
```

Так как для первого устройства был добавлен sleep на 5 секунд, информация с первого маршрутизатора фактически была получена позже всего. Однако, так как метод map возвращает значения в том же порядке, что и устройства в списке device, итоговый результат выглядит так:

```
192.168.100.1 *08:29:16.663 UTC Thu Jul 4 2019
192.168.100.2 *08:29:11.744 UTC Thu Jul 4 2019
192.168.100.3 *08:29:15.374 UTC Thu Jul 4 2019
```

Обработка исключений с map

Пример использования map с обработкой исключений:

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time
from itertools import repeat
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger('paramiko').setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO,
)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received:   {}'
    ip = device_dict['ip']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1': time.sleep(5)

    try:
        with ConnectHandler(**device_dict) as ssh:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            ssh.enable()
            result = ssh.send_command(command)
            logging.info(received_msg.format(datetime.now().time(), ip))
        return result
    except NetMikoAuthenticationException as err:
        logging.warning(err)


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        result = executor.map(send_show, devices, repeat(command))
        for device, output in zip(devices, result):
            data[device['ip']] = output
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh ip int br'))
```

Пример в целом аналогичен предыдущему, но в функции send_show появилась обработка ошибки NetMikoAuthenticationException, а код, который запускал функцию send_show в потоках, теперь находится в функции send_command_to_devices.

При использовании метода map, обработку исключений лучше делать внутри функции, которая запускается в потоках, в данном случае это функция send_show.

Метод submit и работа с futures

Метод submit отличается от метода map:

- submit запускает в потоке только одну функцию
- с помощью submit можно запускать разные функции с разными несвязанными аргументами, а map надо обязательно запускать с итерируемым объектами в роли аргументов
- submit сразу возвращает результат, не дожидаясь выполнения функции
- submit возвращает специальный объект Future, который представляет выполнение функции.
  - submit возвращает Future для того чтобы вызов submit не блокировал код. Как только submit вернул Future, код может выполняться дальше. И как только запущены все функции в потоках, можно начинать запрашивать Future о том готовы ли результаты. Или воспользоваться специальной функцией as_completed, которая сама

запрашивает результат, а код получает его по мере готовности

- submit можно передавать ключевые аргументы, а map только позиционные

Метод submit использует объект Future - это объект, который представляет отложенное вычисление. Этот объект можно запрашивать о состоянии (завершена работа или нет), можно получать результаты или исключения, которые возникли в процессе работы. Future не нужно создавать вручную, эти объекты создаются методом submit.

Пример запуска функции в потоках с помощью submit (файл netmiko_threads_submit_basics.py):

```
from concurrent.futures import ThreadPoolExecutor
from pprint import pprint
from datetime import datetime
import time
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received: {}'
    ip = device_dict['ip']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return {ip: result}


with open('devices.yaml') as f:
    devices = yaml.safe_load(f)

with ThreadPoolExecutor(max_workers=2) as executor:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    future_list = []
    for device in devices:
        future = executor.submit(send_show, device, 'sh clock')
        future_list.append(future)
    for f in future_list:
        print(f.result())
```

Остальной код не изменился, поэтому разобраться надо только с блоком, который запускает функцию send_show в потоках:

```
with ThreadPoolExecutor(max_workers=2) as executor:
    future_list = []
    for device in devices:
        future = executor.submit(send_show, device, 'sh clock')
        future_list.append(future)
    for f in future_list:
        print(f.result())
```

Теперь в блоке with два цикла:

- future_list - это список объектов future:
  - для создания future используется функция submit
  - ей как аргументы передаются: имя функции, которую надо выполнить, и ее аргументы
- следующий цикл проходится по списку future и получает результат с помощью метода result

Результат выполнения:

```
ThreadPoolExecutor-0_0 root INFO: ===> 12:13:39.194657 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 12:13:39.195841 Connection: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: <=== 12:13:40.024725 Received: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 12:13:40.257218 Connection: 192.168.100.3
ThreadPoolExecutor-0_1 root INFO: <=== 12:13:41.085220 Received: 192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 12:13:45.025395 Received: 192.168.100.1
{'192.168.100.1': '*12:13:45.017 UTC Tue Mar 16 2021'}
{'192.168.100.2': '*12:13:40.019 UTC Tue Mar 16 2021'}
{'192.168.100.3': '*12:13:41.077 UTC Tue Mar 16 2021'}
```

В этом случае результаты возвращаются в порядке создания Future, но при использовании submit можно также использовать функцию as_completed, которая позволяет получать результаты по мере того как функции завершают работу.

Пример запуска функции в потоках с помощью submit и as_completed (полный код в файле netmiko_threads_submit_basics_as_completed.py):

```
from concurrent.futures import ThreadPoolExecutor, as_completed


with ThreadPoolExecutor(max_workers=2) as executor:
    future_list = []
    for device in devices:
        future = executor.submit(send_show, device, 'sh clock')
        future_list.append(future)
    for f in as_completed(future_list):
        print(f.result())
```

Теперь цикл проходится по списку future с помощью функции as_completed. Эта функция возвращает future только когда они завершили работу или были отменены. При этом future возвращаются по мере завершения работы, не в порядке добавления в список future_list.

Примечание: Создание списка с future можно сделать с помощью list comprehensions: `future_list = [executor.submit(send_show, device, 'sh clock') for device in devices]`

Результат выполнения:

```
$ python netmiko_threads_submit_basics.py
ThreadPoolExecutor-0_0 root INFO: ===> 17:32:59.088025 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 17:32:59.094103 Connection: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: <=== 17:33:11.639672 Received: 192.168.100.2
{'192.168.100.2': '*17:33:11.429 UTC Thu Jul 4 2019'}
ThreadPoolExecutor-0_1 root INFO: ===> 17:33:11.849132 Connection: 192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 17:33:17.735761 Received: 192.168.100.1
{'192.168.100.1': '*17:33:17.694 UTC Thu Jul 4 2019'}
ThreadPoolExecutor-0_1 root INFO: <=== 17:33:23.230123 Received: 192.168.100.3
{'192.168.100.3': '*17:33:23.188 UTC Thu Jul 4 2019'}
```

Обратите внимание, что порядок не сохраняется и зависит от того, какие функции раньше завершили работу.

Future

Пример запуска функции send_show с помощью submit и вывод информации о Future (обратите внимание на статус future в разные моменты времени):

```
In [1]: from concurrent.futures import ThreadPoolExecutor

In [2]: from netmiko_threads_submit_futures import send_show

In [3]: executor = ThreadPoolExecutor(max_workers=2)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [4]: f1 = executor.submit(send_show, r1, 'sh clock')
   ...: f2 = executor.submit(send_show, r2, 'sh clock')
   ...: f3 = executor.submit(send_show, r3, 'sh clock')
   ...:
ThreadPoolExecutor-0_0 root INFO: ===> 17:53:19.656867 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 17:53:19.657252 Connection: 192.168.100.2

In [5]: print(f1, f2, f3, sep='\n')
<Future at 0xb488e2ac state=running>
<Future at 0xb488ef2c state=running>
<Future at 0xb488e72c state=pending>

ThreadPoolExecutor-0_1 root INFO: <=== 17:53:25.757704 Received: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 17:53:25.869368 Connection: 192.168.100.3

In [6]: print(f1, f2, f3, sep='\n')
<Future at 0xb488e2ac state=running>
<Future at 0xb488ef2c state=finished returned dict>
<Future at 0xb488e72c state=running>

ThreadPoolExecutor-0_0 root INFO: <=== 17:53:30.431207 Received: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: <=== 17:53:31.636523 Received: 192.168.100.3

In [7]: print(f1, f2, f3, sep='\n')
<Future at 0xb488e2ac state=finished returned dict>
<Future at 0xb488ef2c state=finished returned dict>
<Future at 0xb488e72c state=finished returned dict>
```

Чтобы посмотреть на future, в скрипт добавлены несколько строк с выводом информации (netmiko_threads_submit_futures.py):

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
import logging

import yaml
from netmiko import ConnectHandler, NetMikoAuthenticationException


logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    level=logging.INFO)


def send_show(device_dict, command):
    start_msg = '===> {} Connection: {}'
    received_msg = '<=== {} Received: {}'
    ip = device_dict['ip']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1':
        time.sleep(5)

    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return {ip: result}


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        future_list = []
        for device in devices:
            future = executor.submit(send_show, device, command)
            future_list.append(future)
            print('Future: {} for device {}'.format(future, device['ip']))
        for f in as_completed(future_list):
            result = f.result()
            print('Future done {}'.format(f))
            data.update(result)
    return data


if __name__ == '__main__':
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh clock'))
```

Результат выполнения:

```
$ python netmiko_threads_submit_futures.py
Future: <Future at 0xb5ed938c state=running> for device 192.168.100.1
ThreadPoolExecutor-0_0 root INFO: ===> 07:14:26.298007 Connection: 192.168.100.1
Future: <Future at 0xb5ed96cc state=running> for device 192.168.100.2
Future: <Future at 0xb5ed986c state=pending> for device 192.168.100.3
ThreadPoolExecutor-0_1 root INFO: ===> 07:14:26.299095 Connection: 192.168.100.2
```

(continues on next page)

(продолжение с предыдущей страницы)

```
ThreadPoolExecutor-0_1 root INFO: <=== 07:14:32.056003 Received: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 07:14:32.164774 Connection: 192.168.100.3
Future done <Future at 0xb5ed96cc state=finished returned dict>
ThreadPoolExecutor-0_0 root INFO: <=== 07:14:36.714923 Received: 192.168.100.1
Future done <Future at 0xb5ed938c state=finished returned dict>
ThreadPoolExecutor-0_1 root INFO: <=== 07:14:37.577327 Received: 192.168.100.3
Future done <Future at 0xb5ed986c state=finished returned dict>
{'192.168.100.1': '*07:14:36.546 UTC Fri Jul 26 2019',
 '192.168.100.2': '*07:14:31.865 UTC Fri Jul 26 2019',
 '192.168.100.3': '*07:14:37.413 UTC Fri Jul 26 2019'}
```

Так как по умолчанию используется ограничение в два потока, только два из трех future показывают статус running. Третий находится в состоянии pending и ждет, пока до него дойдет очередь.

Обработка исключений

Если при выполнении функции возникло исключение, оно будет сгенерировано при получении результата

Например, в файле devices.yaml пароль для устройства 192.168.100.2 изменен на неправильный:

```
$ python netmiko_threads_submit.py
===> 06:29:40.871851 Connection to device: 192.168.100.1
===> 06:29:40.872888 Connection to device: 192.168.100.2
===> 06:29:43.571296 Connection to device: 192.168.100.3
<=== 06:29:48.921702 Received result from device: 192.168.100.3
<=== 06:29:56.269284 Received result from device: 192.168.100.1
Traceback (most recent call last):
...
  File "/home/vagrant/venv/py3_convert/lib/python3.6/site-packages/netmiko/base_
↪connection.py", line 500, in establish_connection
    raise NetMikoAuthenticationException(msg)
netmiko.ssh_exception.NetMikoAuthenticationException: Authentication failure: unable to␣
↪connect cisco_ios 192.168.100.2:22
Authentication failed.
```

Так как исключение возникает при получении результата, легко добавить обработку исключений (файл netmiko_threads_submit_exception.py):

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from pprint import pprint
from datetime import datetime
import time
```

(continues on next page)

(продолжение с предыдущей страницы)

```
from itertools import repeat
import logging

import yaml
from netmiko import ConnectHandler
from netmiko.ssh_exception import NetMikoAuthenticationException

logging.getLogger("paramiko").setLevel(logging.WARNING)

logging.basicConfig(
    format = '%(threadName)s %(name)s %(levelname)s: %(message)s',
    level=logging.INFO)

start_msg = '===> {} Connection: {}'
received_msg = '<=== {} Received: {}'


def send_show(device_dict, command):
    ip = device_dict['ip']
    logging.info(start_msg.format(datetime.now().time(), ip))
    if ip == '192.168.100.1': time.sleep(5)
    with ConnectHandler(**device_dict) as ssh:
        ssh.enable()
        result = ssh.send_command(command)
        logging.info(received_msg.format(datetime.now().time(), ip))
    return {ip: result}


def send_command_to_devices(devices, command):
    data = {}
    with ThreadPoolExecutor(max_workers=2) as executor:
        future_ssh = [
            executor.submit(send_show, device, command) for device in devices
        ]
        for f in as_completed(future_ssh):
            try:
                result = f.result()
            except NetMikoAuthenticationException as e:
                print(e)
            else:
                data.update(result)
    return data


if __name__ == '__main__':
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    with open('devices.yaml') as f:
        devices = yaml.safe_load(f)
    pprint(send_command_to_devices(devices, 'sh clock'))
```

Результат выполнения:

```
$ python netmiko_threads_submit_exception.py
ThreadPoolExecutor-0_0 root INFO: ===> 07:21:21.190544 Connection: 192.168.100.1
ThreadPoolExecutor-0_1 root INFO: ===> 07:21:21.191429 Connection: 192.168.100.2
ThreadPoolExecutor-0_1 root INFO: ===> 07:21:23.672425 Connection: 192.168.100.3
Authentication failure: unable to connect cisco_ios 192.168.100.2:22
Authentication failed.
ThreadPoolExecutor-0_1 root INFO: <=== 07:21:29.095289 Received: 192.168.100.3
ThreadPoolExecutor-0_0 root INFO: <=== 07:21:31.607635 Received: 192.168.100.1
{'192.168.100.1': '*07:21:31.436 UTC Fri Jul 26 2019',
 '192.168.100.3': '*07:21:28.930 UTC Fri Jul 26 2019'}
```

Конечно, обработка исключения может выполняться и внутри функции send_show, но это просто пример того, как можно работать с исключениями при использовании future.

Дополнительные материалы

GIL

- Can't we get rid of the Global Interpreter Lock?
- GIL (на русском)
- Understanding the Python GIL
- Python threads and the GIL

concurrent.futures

Документация Python:

- concurrent.futures — Launching parallel tasks
- PEP 3148
- PyMOTW. concurrent.futures — Manage Pools of Concurrent Tasks
- threading
- multiprocessing
- queue

Статьи:

- A quick introduction to the concurrent.futures module
- Python - paralellizing CPU-bound tasks with concurrent.futures
- concurrent.futures in Python 3

Полезные вопросы и ответы на stackoverflow

- How many processes should I run in parallel?
- How many threads is too many?

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 19.1

Создать функцию ping_ip_addresses, которая проверяет пингуются ли IP-адреса. Проверка IP-адресов должна выполняться параллельно в разных потоках.

Параметры функции:

- ip_list - список IP-адресов
- limit - максимальное количество параллельных потоков (по умолчанию 3)

Функция должна возвращать кортеж с двумя списками:

- список доступных IP-адресов
- список недоступных IP-адресов

Для выполнения задания можно создавать любые дополнительные функции.

Для проверки доступности IP-адреса, используйте ping.

Примечание: Подсказка о работе с concurrent.futures: Если необходимо пинговать несколько IP-адресов в разных потоках, надо создать функцию, которая будет пинговать один IP-адрес, а затем запустить эту функцию в разных потоках для разных IP-адресов с помощью concurrent.futures (это надо сделать в функции ping_ip_addresses).

Задание 19.2

Создать функцию send_show_command_to_devices, которая отправляет одну и ту же команду show на разные устройства в параллельных потоках, а затем записывает вывод команд в файл. Вывод с устройств в файле может быть в любом порядке.

Параметры функции:

- devices - список словарей с параметрами подключения к устройствам
- command - команда
- filename - имя текстового файла, в который будут записаны выводы всех команд
- limit - максимальное количество параллельных потоков (по умолчанию 3)

Функция ничего не возвращает.

Вывод команд должен быть записан в обычный текстовый файл в таком формате (перед выводом команды надо написать имя хоста и саму команду):

```
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
R2#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.2   YES NVRAM  up                    up
Ethernet0/1                10.1.1.1        YES NVRAM  administratively down down
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
```

Для выполнения задания можно создавать любые дополнительные функции.

Проверить работу функции на устройствах из файла devices.yaml

Задание 19.3

Создать функцию send_command_to_devices, которая отправляет разные команды show на разные устройства в параллельных потоках, а затем записывает вывод команд в файл. Вывод с устройств в файле может быть в любом порядке.

Параметры функции:

- devices - список словарей с параметрами подключения к устройствам
- commands_dict - словарь в котором указано на какое устройство отправлять какую команду. Пример словаря - commands

- filename - имя файла, в который будут записаны выводы всех команд
- limit - максимальное количество параллельных потоков (по умолчанию 3)

Функция ничего не возвращает.

Вывод команд должен быть записан в файл в таком формате (перед выводом команды надо написать имя хоста и саму команду):

```
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
R2#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  192.168.100.1          76   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2           -   aabb.cc00.6600  ARPA   Ethernet0/0
Internet  192.168.100.3         173   aabb.cc00.6700  ARPA   Ethernet0/0
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
```

Для выполнения задания можно создавать любые дополнительные функции.

Проверить работу функции на устройствах из файла devices.yaml и словаре commands

```
# Этот словарь нужен только для проверки работа кода, в нем можно менять IP-адреса
# тест берет адреса из файла devices.yaml
commands = {
    "192.168.100.3": "sh run | s ^router ospf",
    "192.168.100.1": "sh ip int br",
    "192.168.100.2": "sh int desc",
}
```

Задание 19.3a

Создать функцию send_command_to_devices, которая отправляет список указанных команды show на разные устройства в параллельных потоках, а затем записывает вывод команд в файл. Вывод с устройств в файле может быть в любом порядке.

Параметры функции:

- devices - список словарей с параметрами подключения к устройствам
- commands_dict - словарь в котором указано на какое устройство отправлять какие команды. Пример словаря - commands
- filename - имя файла, в который будут записаны выводы всех команд

- limit - максимальное количество параллельных потоков (по умолчанию 3)

Функция ничего не возвращает.

Вывод команд должен быть записан в файл в таком формате (перед выводом каждой команды надо написать имя хоста и саму команду):

```
R2#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  192.168.100.1          87   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2           -   aabb.cc00.6600  ARPA   Ethernet0/0
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
R1#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.30.0.1               -   aabb.cc00.6530  ARPA   Ethernet0/3.300
Internet  10.100.0.1              -   aabb.cc00.6530  ARPA   Ethernet0/3.100
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
R3#sh ip route | ex -

Gateway of last resort is not set

      10.0.0.0/8 is variably subnetted, 4 subnets, 2 masks
O        10.1.1.1/32 [110/11] via 192.168.100.1, 07:12:03, Ethernet0/0
O        10.30.0.0/24 [110/20] via 192.168.100.1, 07:12:03, Ethernet0/0
```

Порядок команд в файле может быть любым.

Для выполнения задания можно создавать любые дополнительные функции, а также использовать функции созданные в предыдущих заданиях.

Проверить работу функции на устройствах из файла devices.yaml и словаре commands

```
# Этот словарь нужен только для проверки работа кода, в нем можно менять IP-адреса
# тест берет адреса из файла devices.yaml
commands = {
    "192.168.100.3": ["sh ip int br", "sh ip route | ex -"],
    "192.168.100.1": ["sh ip int br", "sh int desc"],
    "192.168.100.2": ["sh int desc"],
}
```

Задание 19.4

Создать функцию send_commands_to_devices, которая отправляет команду show или config на разные устройства в параллельных потоках, а затем записывает вывод команд в файл.

Параметры функции:

- devices - список словарей с параметрами подключения к устройствам
- filename - имя файла, в который будут записаны выводы всех команд
- show - команда show, которую нужно отправить (по умолчанию, значение None)
- config - команды конфигурационного режима, которые нужно отправить (по умолчанию None)
- limit - максимальное количество параллельных потоков (по умолчанию 3)

Функция ничего не возвращает.

Аргументы show, config и limit должны передаваться только как ключевые. При передачи этих аргументов как позиционных, должно генерироваться исключение TypeError.

```
In [4]: send_commands_to_devices(devices, 'result.txt', 'sh clock')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-75adcfb4a005> in <module>
----> 1 send_commands_to_devices(devices, 'result.txt', 'sh clock')

TypeError: send_commands_to_devices() takes 2 positional argument but 3 were given
```

При вызове функции send_commands_to_devices, всегда должен передаваться только один из аргументов show, config. Если передаются оба аргумента, должно генерироваться исключение ValueError.

Вывод команд должен быть записан в файл в таком формате (перед выводом команды надо написать имя хоста и саму команду):

```
R1#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
Ethernet0/0                192.168.100.1   YES NVRAM  up                    up
Ethernet0/1                192.168.200.1   YES NVRAM  up                    up
R2#sh arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  192.168.100.1          76   aabb.cc00.6500  ARPA   Ethernet0/0
Internet  192.168.100.2           -   aabb.cc00.6600  ARPA   Ethernet0/0
Internet  192.168.100.3         173   aabb.cc00.6700  ARPA   Ethernet0/0
R3#sh ip int br
Interface                  IP-Address      OK? Method Status                Protocol
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Ethernet0/0                192.168.100.3   YES NVRAM  up                    up
Ethernet0/1                unassigned      YES NVRAM  administratively down down
```

Пример вызова функции:

```
In [5]: send_commands_to_devices(devices, 'result.txt', show='sh clock')

In [6]: cat result.txt
R1#sh clock
*04:56:34.668 UTC Sat Mar 23 2019
R2#sh clock
*04:56:34.687 UTC Sat Mar 23 2019
R3#sh clock
*04:56:40.354 UTC Sat Mar 23 2019

In [11]: send_commands_to_devices(devices, 'result.txt', config='logging 10.5.5.5')

In [12]: cat result.txt
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#logging 10.5.5.5
R1(config)#end
R1#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R2(config)#logging 10.5.5.5
R2(config)#end
R2#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R3(config)#logging 10.5.5.5
R3(config)#end
R3#

In [13]: commands = ['router ospf 55', 'network 0.0.0.0 255.255.255.255 area 0']

In [13]: send_commands_to_devices(devices, 'result.txt', config=commands)

In [14]: cat result.txt
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#router ospf 55
R1(config-router)#network 0.0.0.0 255.255.255.255 area 0
R1(config-router)#end
R1#
config term
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Enter configuration commands, one per line.  End with CNTL/Z.
R2(config)#router ospf 55
R2(config-router)#network 0.0.0.0 255.255.255.255 area 0
R2(config-router)#end
R2#
config term
Enter configuration commands, one per line.  End with CNTL/Z.
R3(config)#router ospf 55
R3(config-router)#network 0.0.0.0 255.255.255.255 area 0
R3(config-router)#end
R3#
```

Для выполнения задания можно создавать любые дополнительные функции.

20. Шаблоны конфигураций с Jinja2

Jinja2 - это язык шаблонов, который используется в Python. Jinja - это не единственный язык шаблонов (шаблонизатор) для Python и 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

Начало работы с Jinja2

Установить Jinja2 можно с помощью pip:

```
pip install jinja2
```

Примечание: Далее термины Jinja и Jinja2 используются взаимозаменяемо.

Главная идея Jinja: 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 (cfg_template.txt):

```
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 Environment, FileSystemLoader
import yaml


env = Environment(loader=FileSystemLoader("."))
templ = env.get_template("cfg_template.txt")

liverpool = {"id": "11", "name": "Liverpool", "int": "Gi1/0/17", "ip": "10.1.1.10"}
print(templ.render(liverpool))
```

Скрипт 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 текущем каталоге
- Environment - класс для описания параметров окружения. В данном случае указан только загрузчик, но в нём можно указывать методы обработки шаблона

Комментарии к файлу basic_generator.py:

- в шаблоне используются переменные в синтаксисе Jinja
- в словаре liverpool ключи должны быть такими же, как имена переменных в шаблоне
- значения, которые соответствуют ключам - это те данные, которые будут подставлены на место переменных
- последняя строка рендерит шаблон, используя словарь liverpool, то есть, подставляет значения в переменные.

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

```
$ python basic_generator.py

hostname Liverpool
!
interface Loopback255
 description Management loopback
 ip address 10.255.11.1 255.255.255.255
!
interface GigabitEthernet0/0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 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 Jinja

Шаблон templates/router_template.txt - это обычный текстовый файл:

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

(continues on next page)

(продолжение с предыдущей страницы)

```
!
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

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

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

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

Скрипт для генерации конфигураций router_config_generator_ver2.py

```python
from jinja2 import Environment, FileSystemLoader
import yaml

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

with open('routers_info.yml') as f:
    routers = yaml.safe_load(f)

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

Файл router_config_generator.py импортирует из модуля jinja2:

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

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

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.

В этом разделе рассматриваются только основы этих возможностей. Подробнее о шаблонах Jinja2 можно почитать в документации.

Примечание: Все файлы, которые используются как примеры в этом подразделе, находятся в каталоге 3_template_syntax/

Для генерации шаблонов будет использоваться скрипт cfg_gen.py

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

(continues on next page)

(продолжение с предыдущей страницы)

```
#$ python cfg_gen.py templates/for.txt data_files/for.yml
template_dir, template_file = os.path.split(sys.argv[1])

vars_file = sys.argv[2]

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

with open(vars_file) as f:
    vars_dict = yaml.safe_load(f)

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

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

- шаблон
- файл с переменными в формате YAML

Результат будет выведен на стандартный поток вывода.

Пример запуска скрипта:

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

Параметры trim_blocks и lstrip_blocks описаны в следующем подразделе.

Контроль символов whitespace

trim_blocks, lstrip_blocks

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

Эффект применения флага рассматривается 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
```

Переводы строк появляются из-за блока for.

```
{% for ibgp in bgp.ibgp_neighbors %}
```

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.

Перед строками `neighbor ... remote-as` появились два пробела. Так получилось из-за того, что перед блоком for стоит пробел. После того, как был отключен лишний перевод строки, пробелы и табы перед блоком добавляются к первой строке блока.

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 %}
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Обратите внимание на минус в начале второго блока. Минус 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 минус 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 %}
```

Удалится пустая строка и в конце блока:

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

 neighbor 10.0.0.2 remote-as 100
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 neighbor 10.0.0.2 update-source lo100

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


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

Попробуйте добавить минус в конце выражений, описывающих блок, и посмотреть на результат:

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

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

Переменные

Переменные в шаблоне указываются в двойных фигурных скобках:

```
hostname {{ name }}

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

Значения переменных подставляются на основе словаря, который передается шаблону.

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

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

```
hostname {{ name }}

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

vlan {{ vlans[0] }}

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

И соответствующий файл data_files/vars.yml с переменными:

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

Обратите внимание на использование переменной vlans в шаблоне: так как переменная vlans это список, можно указывать, какой именно элемент из списка нам нужен

Если передается словарь (как в случае с переменной ospf), то внутри шаблона можно обращаться к объектам словаря, используя один из вариантов: `ospf.network` или `ospf['network']`

Результат запуска скрипта будет таким:

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

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

vlan 10

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

Цикл for

Цикл for позволяет проходиться по элементам последовательности.

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

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

Пример шаблона templates/for.txt с использованием цикла:

```
hostname {{ name }}

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

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

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

Файл data_files/for.yml с переменными:

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

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

Результат выполнения будет таким:

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

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

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

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

if/elif/else

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

Конструкция if также должна находиться внутри `{% %}`. Нужно явно указывать окончание условия:

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

Пример шаблона templates/if.txt:

```
hostname {{ name }}

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
{% endfor %}

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

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

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

Конфигурация будет генерироваться с двумя вариантами данных.

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

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

Результат будет таким:

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

hostname R3

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

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

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

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

Теперь результат выполнения будет таким:

```
hostname R3

interface Loopback0
 ip address 10.0.0.3 255.255.255.255

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

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

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

Пример шаблона templates/if_vlans.txt:

```
{% for intf, params in trunks.items() %}
interface {{ intf }}
 {% if params.action == 'add' %}
 switchport trunk allowed vlan add {{ params.vlans }}
 {% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove {{ params.vlans }}
 {% else %}
 switchport trunk allowed vlan {{ params.vlans }}
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Файл data_files/if_vlans.yml с данными:

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

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

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

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

В итоге будет сгенерирована такая конфигурация:

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

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

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

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

Файл с данными (data_files/if_for.yml):

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

Результат выполнения:

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

Фильтры

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

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

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

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

default

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

Пример шаблона templates/filter_default.txt:

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

(continues on next page)

(продолжение с предыдущей страницы)

```
 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
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 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 %}
```

Примечание: 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 отсортировал 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 }}
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Результат выполнения будет таким (интерфейсы упорядочены):

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

join

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

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

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(',') }}
```

(continues on next page)

(продолжение с предыдущей страницы)

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

Файл с данными (data_files/filter_join.yml):

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

Результат выполнения:

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

Тесты

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

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

Тесты, как и фильтры, можно создавать самостоятельно.

defined

Тест defined позволяет проверить, есть ли переменная в словаре данных.

Пример шаблона templates/test_defined.txt:

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

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

Файл с данными (data_files/test_defined.yml):

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

Результат выполнения:

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

iterable

Тест iterable проверяет, является ли объект итератором.

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

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

```
{% for intf, params in trunks | dictsort %}
interface {{ intf }}
 {% if params.vlans is iterable %}
   {% if params.action == 'add' %}
 switchport trunk allowed vlan add {{ params.vlans | join(',') }}
   {% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove {{ params.vlans | join(',') }}
   {% else %}
 switchport trunk allowed vlan {{ params.vlans | join(',') }}
   {% endif %}
 {% else %}
   {% if params.action == 'add' %}
 switchport trunk allowed vlan add {{ params.vlans }}
   {% elif params.action == 'delete' %}
 switchport trunk allowed vlan remove {{ params.vlans }}
   {% else %}
 switchport trunk allowed vlan {{ params.vlans }}
   {% endif %}
 {% endif %}
{% endfor %}
```

Файл с данными (data_files/test_iterable.yml):

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

Обратите внимание на последнюю строку: `vlans: 10`. В данном случае 10 уже не находится

в списке, и фильтр join в таком случае не работает. Но, за счет теста `is iterable` (в этом случае результат будет false), в этом случае шаблон уходит в ветку else.

Результат выполнения:

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

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

set

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

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

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

```
{% for intf, params in trunks | dictsort %}
 {% set vlans = params.vlans %}
 {% set action = params.action %}

interface {{ intf }}
 {% if vlans is iterable %}
  {% if action == 'add' %}
 switchport trunk allowed vlan add {{ vlans | join(',') }}
  {% elif action == 'delete' %}
 switchport trunk allowed vlan remove {{ vlans | join(',') }}
  {% else %}
 switchport trunk allowed vlan {{ vlans | join(',') }}
  {% endif %}
 {% else %}
  {% if action == 'add' %}
 switchport trunk allowed vlan add {{ vlans }}
  {% elif action == 'delete' %}
 switchport trunk allowed vlan remove {{ vlans }}
  {% else %}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 switchport trunk allowed vlan {{ vlans }}
  {% endif %}
 {% endif %}
{% endfor %}
```

Обратите внимание на вторую и третью строки:

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

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

Файл с данными (data_files/set.yml):

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

Результат выполнения:

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

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

interface Fa0/2
 switchport trunk allowed vlan 10,30

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

include

Выражение include позволяет добавить один шаблон в другой.

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

Шаблон templates/vlans.txt:

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

Шаблон templates/ospf.txt:

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

Шаблон templates/bgp.txt:

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

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

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

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

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
  - 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
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  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. 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 %}
```

(continues on next page)

(продолжение с предыдущей страницы)

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

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

Дочерний шаблон templates/hq_router.txt:

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

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

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

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

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

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

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

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

В базовом шаблоне четыре блока: services, ospf, bgp, alias. В дочернем шаблоне заполнены только два из них: ospf и alias. В этом удобство наследования. Не обязательно заполнять все блоки в каждом дочернем шаблоне.

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

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

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

В данном случае конфигурация дополняется. Именно поэтому в дочернем шаблоне templates/hq_router.txt блок ospf начинается с выражения `{{ super() }}`:

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

`{{ super() }}` переносит в дочерний шаблон содержимое этого блока из родительского шаблона. За счет этого в дочерний шаблон перенесутся строки из родительского.

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

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

Подытожим правила работы с блоками. Если в родительском шаблоне создан блок:

- без содержимого - в дочернем шаблоне можно заполнить этот блок или игнорировать. Если блок заполнен, в нём будет только то, что было написано в дочернем шаблоне (пример - блок alias)
- с содержимым - то в дочернем шаблоне можно выполнить такие действия:

- игнорировать блок - в таком случае в дочерний шаблон попадет содержимое, которое находилось в этом блоке в родительском шаблоне (пример - блок services)
- переписать блок - тогда в дочернем шаблоне будет только то, что указано в нём
- перенести содержимое блока из родительского шаблона и дополнить его - тогда в дочернем шаблоне будет и содержимое блока из родительского шаблона, и содержимое из дочернего шаблона. Для переноса содержимого из родительского шаблона используется выражение `{{ super() }}` (пример - блок ospf)

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

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

Результат выполнения будет таким:

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

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

(continues on next page)

(продолжение с предыдущей страницы)

```
alias exec desc sh int desc | ex down
!
line con 0
 logging synchronous
 history size 100
line vty 0 4
 logging synchronous
 history size 100
 transport input ssh
!
```

Обратите внимание, что в блоке ospf есть и команды из базового шаблона, и команды из дочернего шаблона.

Дополнительные материалы

Документация:

- Общая документация Jinja2
- Синтаксис шаблонов

Статьи:

- Network Configuration Templates Using Jinja2. Matt Oswalt
- Python And Jinja2 Tutorial. Jeremy Schulman
- Configuration Generator with Python and Jinja2
- Custom filters for a Jinja2 based Config Generator

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 20.1

Создать функцию generate_config.

Параметры функции:

- template - путь к файлу с шаблоном (например, «templates/for.txt»)
- data_dict - словарь со значениями, которые надо подставить в шаблон

Функция должна возвращать строку с конфигурацией, которая была сгенерирована.

Проверить работу функции на шаблоне templates/for.txt и данных из файла data_files/for.yml.

```
import yaml


# так должен выглядеть вызов функции
if __name__ == "__main__":
    data_file = "data_files/for.yml"
    template_file = "templates/for.txt"
    with open(data_file) as f:
        data = yaml.safe_load(f)
    print(generate_config(template_file, data))
```

Задание 20.2

Создать шаблон templates/cisco_router_base.txt. В шаблон templates/cisco_router_base.txt 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_config из задания 20.1. Не копируйте код функции generate_config.

В качестве данных, используйте информацию из файла data_files/router_info.yml

Задание 20.3

Создайте шаблон templates/ospf.txt на основе конфигурации OSPF в файле cisco_ospf.txt. Пример конфигурации дан, чтобы показать синтаксис.

Шаблон надо создавать вручную, скопировав части конфига в соответствующий шаблон.

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_config из задания 20.1. Не копируйте код функции generate_config.

В результате должна получиться конфигурация такого вида (команды в режиме router ospf не обязательно должны быть в таком порядке, главное чтобы они были в нужном режиме):

```
router ospf 10
 router-id 10.0.0.1
 auto-cost reference-bandwidth 20000
 network 10.255.0.1 0.0.0.0 area 0
 network 10.255.1.1 0.0.0.0 area 0
 network 10.255.2.1 0.0.0.0 area 0
 network 10.0.10.1 0.0.0.0 area 2
 network 10.0.20.1 0.0.0.0 area 2
 passive-interface Fa0/0.10
 passive-interface Fa0/0.20
interface Fa0/1
 ip ospf hello-interval 1
interface Fa0/1.100
 ip ospf hello-interval 1
interface Fa0/1.200
 ip ospf hello-interval 1
```

Задание 20.4

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, 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_config из задания 20.1. Не копируйте код функции generate_config.

Задание 20.5

Создать шаблоны templates/gre_ipsec_vpn_1.txt и templates/gre_ipsec_vpn_2.txt, которые генерируют конфигурацию IPsec over GRE между двумя маршрутизаторами.

Шаблон templates/gre_ipsec_vpn_1.txt создает конфигурацию для одной стороны туннеля, а templates/gre_ipsec_vpn_2.txt - для второй.

Примеры итоговой конфигурации, которая должна создаваться на основе шаблонов в файлах: cisco_vpn_1.txt и cisco_vpn_2.txt.

Создать функцию create_vpn_config, которая использует эти шаблоны для генерации конфигурации VPN на основе данных в словаре data.

Параметры функции:

- template1 - имя файла с шаблоном, который создает конфигурацию для одной строны туннеля
- template2 - имя файла с шаблоном, который создает конфигурацию для второй строны туннеля
- data_dict - словарь со значениями, которые надо подставить в шаблоны

Функция должна возвращать кортеж с двумя конфигурациямя (строки), которые получены на основе шаблонов.

Примеры конфигураций VPN, которые должна возвращать функция create_vpn_config в файлах cisco_vpn_1.txt и cisco_vpn_2.txt.

```
data = {
    'tun_num': 10,
    'wan_ip_1': '192.168.100.1',
    'wan_ip_2': '192.168.100.2',
    'tun_ip_1': '10.0.1.1 255.255.255.252',
    'tun_ip_2': '10.0.1.2 255.255.255.252'
}
```

Задание 20.5a

Создать функцию configure_vpn, которая использует шаблоны из задания 20.5 для настройки VPN на маршрутизаторах на основе данных в словаре data.

Параметры функции:

- src_device_params - словарь с параметрами подключения к устройству
- dst_device_params - словарь с параметрами подключения к устройству
- src_template - имя файла с шаблоном, который создает конфигурацию для одной строны туннеля
- dst_template - имя файла с шаблоном, который создает конфигурацию для второй строны туннеля
- vpn_data_dict - словарь со значениями, которые надо подставить в шаблоны

Функция должна настроить VPN на основе шаблонов и данных на каждом устройстве с помощью netmiko. Функция возвращает вывод с набором команд с двух марушртизаторов (вывод, которые возвращает метод netmiko send_config_set).

При этом, в словаре data не указан номер интерфейса Tunnel, который надо использовать. Номер надо определить самостоятельно на основе информации с оборудования. Если на маршрутизаторе нет интерфейсов Tunnel, взять номер 0, если есть взять ближайший свободный номер, но одинаковый для двух маршрутизаторов.

Например, если на маршрутизаторе src такие интерфейсы: Tunnel1, Tunnel4. А на маршрутизаторе dest такие: Tunnel2, Tunnel3, Tunnel8. Первый свободный номер одинаковый для двух маршрутизаторов будет 5. И надо будет настроить интерфейс Tunnel 5.

Примечание: Для этого задания тест проверяет работу функции на первых двух устройствах из файла devices.yaml. И проверяет, что в выводе есть команды настройки интерфейсов, но при этом не проверяет настроенные номера тунелей и другие команды. Они должны быть, но тест упрощен, чтобы было больше свободы выполнения.

```
data = {
    'tun_num': None,
    'wan_ip_1': '192.168.100.1',
    'wan_ip_2': '192.168.100.2',
    'tun_ip_1': '10.0.1.1 255.255.255.252',
    'tun_ip_2': '10.0.1.2 255.255.255.252'
}
```

21. Обработка вывода команд 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.

Начало работы с TextFSM

Для начала библиотеку надо установить:

```
pip install textfsm
```

Для использования TextFSM надо создать шаблон, по которому будет обрабатываться вывод команды.

Пример вывода команды traceroute:

```
r2#traceroute 90.0.0.9 source 33.0.0.2
traceroute 90.0.0.9 source 33.0.0.2
Type escape sequence to abort.
Tracing the route to 90.0.0.9
VRF info: (vrf in name/id, vrf out name/id)
  1 10.0.12.1 1 msec 0 msec 0 msec
  2 15.0.0.5  0 msec 5 msec 4 msec
  3 57.0.0.7  4 msec 1 msec 4 msec
  4 79.0.0.9  4 msec *  1 msec
```

Например, из вывода надо получить хопы, через которые прошел пакет.

В таком случае шаблон TextFSM будет выглядеть так (файл traceroute.template):

```
Value ID (\d+)
Value Hop (\d+(\.\d+){3})
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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
'''

with open('traceroute.template') as template:
    fsm = textfsm.TextFSM(template)
    result = fsm.ParseText(traceroute)

print(fsm.header)
print(result)
```

Результат выполнения скрипта:

```
$ python parse_traceroute.py
['ID', 'Hop']
[['1', '10.0.12.1'], ['2', '15.0.0.5'], ['3', '57.0.0.7'], ['4', '79.0.0.9']]
```

Строки, которые совпали с описанным шаблоном, возвращаются в виде списка списков. Каждый элемент - это список, который состоит из двух элементов: номера хопа и IP-адреса.

Разберемся с содержимым скрипта:

- traceroute - это переменная, которая содержит вывод команды traceroute
- `template = open('traceroute.template')` - содержимое файла с шаблоном TextFSM считывается в переменную template
- `fsm = textfsm.TextFSM(template)` - класс, который обрабатывает шаблон и создает из него объект в TextFSM
- `result = fsm.ParseText(traceroute)` - метод, который обрабатывает переданный вывод согласно шаблону и возвращает список списков, в котором каждый элемент - это обработанная строка
- В конце выводится заголовок: `print(fsm.header)`, который содержит имена переменных и результат обработки

С этим выводом можно работать дальше. Например, периодически выполнять команду traceroute и сравнивать, изменилось ли количество хопов и их порядок.

Для работы с TextFSM нужны вывод команды и шаблон:

- для разных команд нужны разные шаблоны
- TextFSM возвращает результат обработки в табличном виде (в виде списка списков)
- этот вывод легко преобразовать в csv формат или в список словарей

Синтаксис шаблонов TextFSM

В этом разделе описан синтаксис шаблонов на основе документации TextFSM. В следующем разделе показаны примеры использования синтаксиса. Поэтому, в принципе, можно перейти сразу к следующему разделу, а к этому возвращаться по необходимости, для тех ситуаций, для которых нет примера, и когда нужно перечитать, что означает какой-то параметр.

Шаблон TextFSM описывает, каким образом данные должны обрабатываться.

Любой шаблон состоит из двух частей:

- определения переменных
- эти переменные описывают, какие столбцы будут в табличном представлении
- определения состояний

Пример разбора команды traceroute:

```
# Определение переменных:
Value ID (\d+)
Value Hop (\d+(\.\d+){3})

# Секция с определением состояний всегда должна начинаться с состояния Start
Start
#     Переменные      действие
   ^  ${ID} ${Hop} -> Record
```

Определение переменных

В секции с переменными должны идти только определения переменных. Единственное исключение - в этом разделе могут быть комментарии.

В этом разделе не должно быть пустых строк. Для TextFSM пустая строка означает завершение секции определения переменных.

Формат описания переменных:

```
Value [option[,option...]] name regex
```

Синтаксис описания переменных (для каждой опции ниже мы рассмотрим примеры):

- `Value` - это ключевое слово, которое указывает, что создается переменная. Его обязательно нужно указывать
- option - опции, которые определяют, как работать с переменной. Если нужно указать несколько опций, они должны быть отделены запятой, без пробелов. Поддерживаются такие опции:
  - **Filldown** - значение, которое ранее совпало с регулярным выражением, запоминается до следующей обработки строки (если не было явно очищено или снова совпало регулярное выражение). Это значит, что последнее значение столбца, которое совпало с регулярным выражением, запоминается и используется в следующих строках, если в них не присутствовал этот столбец.
  - **Key** - определяет, что это поле содержит уникальный идентификатор строки
  - **Required** - строка, которая обрабатывается, будет записана только в том случае, если эта переменная присутствует.
  - **List** - значение это список, и каждое совпадение с регулярным выражением будет добавлять в список элемент. По умолчанию каждое следующее совпадение перезаписывает предыдущее.
  - **Fillup** - работает как Filldown, но заполняет пустые значение выше до тех пор, пока не найдет совпадение. Не совместимо с Required.

- name - имя переменной, которое будет использоваться как имя колонки. Зарезервированные имена не должны использоваться как имя переменной.
- regex - регулярное выражение, которое описывает переменную. Регулярное выражение должно быть в скобках.

Определение состояний

После определения переменных нужно описать состояния:

- каждое определение состояния должно быть отделено пустой строкой (как минимум, одной)
- первая строка - имя состояния
- затем идут строки, которые описывают правила. Правила должны начинаться с одного или двух пробелов 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 двух пробелов и символа ^. Символ ^ означает начало строки и всегда должен указываться явно
- regex - это регулярное выражение, в котором могут использоваться переменные
  - для указания переменной, может использоваться синтаксис `$ValueName` или `${ValueName}`(этот формат предпочтителен)
  - в правиле на место переменных подставляются регулярные выражения, которые они описывают
  - если нужно явно указать символ конца строки, используется значение $$

Действия в правилах

После регулярного выражения в правиле могут указываться действия:

- между регулярным выражением и действием должен быть символ ->
- действия могут состоять из трех частей в таком формате: **L.R S**
  - **L - Line Action** - действия, которые применяются к входящей строке
  - **R - Record Action** - действия, которые применяются к собранным значениям
  - **S - State Transition** - переход в другое состояние
- по умолчанию используется **Next.NoRecord**

Line Actions

Line Actions:

- **Next** - обработать строку, прочитать следующую и начать проверять её с начала состояния. Это действие используется по умолчанию, если не указано другое
- **Continue** - продолжить обработку правил, как будто совпадения не было, при этом значения присваиваются

Record Action

Record Action - опциональное действие, которое может быть указано после Line Action. Они должны быть разделены точкой. Типы действий:

- **NoRecord** - не выполнять ничего. Это действие по умолчанию, когда другое не указано
- **Record** - запомнить значения, которые совпали с правилом. Все переменные, кроме тех, где указана опция Filldown, обнуляются.
- **Clear** - обнулить все переменные, кроме тех, где указана опция Filldown.
- **Clearall** - обнулить все переменные.

Разделять действия точкой нужно только в том случае, если нужно указать и Line, и Record действия. Если нужно указать только одно из них, точку ставить не нужно.

State Transition

После действия может быть указано новое состояние:

- состояние должно быть одним из зарезервированных или определенных в шаблоне
- если входная строка совпала:
  - все действия выполняются,
  - считывается следующая строка,
  - затем текущее состояние меняется на новое, и обработка продолжается в новом состоянии.

Если в правиле используется действие **Continue**, то в нём нельзя использовать переход в другое состояние. Это правило нужно для того, чтобы в последовательности состояний не было петель.

Error Action

Специальное действие **Error** останавливает всю обработку строк, отбрасывает все строки, которые были собраны до сих пор, и возвращает исключение.

Синтаксис этого действия такой:

```
^regex -> Error [word|"string"]
```

Примеры использования TextFSM

В этом разделе рассматриваются примеры шаблонов и использования TextFSM.

Для обработки вывода команд по шаблону в разделе используется скрипт parse_output.py. Он не привязан к конкретному шаблону и выводу: шаблон и вывод команды будут передаваться как аргументы:

```
import sys
import textfsm
from tabulate import tabulate

template = sys.argv[1]
output_file = sys.argv[2]

with open(template) as f, open(output_file) as output:
    re_table = textfsm.TextFSM(f)
    header = re_table.header
    result = re_table.ParseText(output.read())
    print(result)
    print(tabulate(result, headers=header))
```

Пример запуска скрипта:

```
$ python parse_output.py template command_output
```

Примечание: Модуль tabulate используется для отображения данных в табличном виде (его нужно установить, если хотите использовать этот скрипт). Аналогичный вывод можно было сделать и с помощью форматирования строк, но с tabulate это сделать проще.

Обработка данных по шаблону всегда выполняется одинаково. Поэтому скрипт будет одинаковый, только шаблон и данные будут отличаться.

Начиная с простого примера, разберемся с тем, как использовать TextFSM.

show clock

Первый пример - разбор вывода команды sh clock (файл output/sh_clock.txt):

```
15:10:44.867 UTC Sun Nov 13 2016
```

Для начала в шаблоне надо определить переменные:

- в начале каждой строки должно быть ключевое слово Value
- каждая переменная определяет столбец в таблице
- следующее слово - название переменной
- после названия, в скобках - регулярное выражение, которое описывает значение переменной

Определение переменных выглядит так:

```
Value Time (..:..:..)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)
```

Подсказка по спецсимволам:

- . - любой символ
- + - одно или более повторений предыдущего символа
- \S - все символы, кроме whitespace
- \w - любая буква или цифра
- \d - любая цифра

После определения переменных должна идти пустая строка и состояние **Start**, а после, начиная с пробела и символа ^, идет правило (файл templates/sh_clock.template):

```
Value Time (..:..:..)
Value Timezone (\S+)
Value WeekDay (\w+)
Value Month (\w+)
Value MonthDay (\d+)
Value Year (\d+)

Start
  ^${Time}.* ${Timezone} ${WeekDay} ${Month} ${MonthDay} ${Year} -> Record
```

Так как в данном случае в выводе всего одна строка, можно не писать в шаблоне действие Record. Но лучше его использовать в ситуациях, когда надо записать значения, чтобы привыкать к этому синтаксису и не ошибиться, когда нужна обработка нескольких строк.

Когда TextFSM обрабатывает строки вывода, он подставляет вместо переменных их значения. В итоге правило будет выглядеть так:

```
^(..:..:..).* (\S+) (\w+) (\w+) (\d+) (\d+)
```

Когда это регулярное выражение применяется к выводу show clock, в каждой группе регулярного выражения будет находиться соответствующее значение:

- 1 группа: 15:10:44
- 2 группа: UTC
- 3 группа: Sun
- 4 группа: Nov
- 5 группа: 13
- 6 группа: 2016

В правиле, кроме явного действия Record, которое указывает, что запись надо поместить в финальную таблицу, по умолчанию также используется правило Next. Оно указывает, что надо перейти к следующей строке текста. Так как в выводе команды sh clock только одна строка, обработка завершается.

Результат отработки скрипта будет таким:

```
$ python parse_output.py templates/sh_clock.template output/sh_clock.txt
Time      Timezone    WeekDay    Month      MonthDay    Year
--------  ----------  ---------  -------  ----------  ------
15:10:44  UTC         Sun        Nov              13    2016
```

show 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 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␣
↪SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2

-------------------------
Device ID: R1
Entry address(es):
  IP address: 10.1.1.1
Platform: Cisco 3825,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/22,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1, RELEASE␣
↪SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2009 by Cisco Systems, Inc.
Compiled Fri 19-Jun-09 18:40 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Duplex: full
Management address(es):

-------------------------
Device ID: R2
Entry address(es):
  IP address: 10.2.2.2
Platform: Cisco 2911,  Capabilities: Router Switch IGMP
Interface: GigabitEthernet1/0/21,  Port ID (outgoing port): GigabitEthernet0/0
Holdtime : 156 sec

Version :
Cisco IOS Software, 2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1, RELEASE␣
↪SOFTWARE (fc3)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2009 by Cisco Systems, Inc.
Compiled Fri 19-Jun-09 18:40 by prod_rel_team

advertisement version: 2
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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 ␣
↪GigabitEthernet0/0  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            ␣
↪REMOTE_PORT         IOS_VERSION
------------  -----------  ----------  --------------------  ---------------------  ------
↪------------  -------------
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/16 ␣
↪GigabitEthernet0/1  12.2(55)SE9
              R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/22 ␣
↪GigabitEthernet0/0  12.4(24)T1
              R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/21 ␣
↪GigabitEthernet0/0  15.2(2)T1
```

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 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 получили такой 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            ␣
↪REMOTE_PORT         IOS_VERSION
------------  -----------  ----------  --------------------  ---------------------  ------
↪------------  -------------
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/16 ␣
↪GigabitEthernet0/1  12.2(55)SE9
SW1           R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/22 ␣
↪GigabitEthernet0/0  12.4(24)T1
SW1           R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/21 ␣
↪GigabitEthernet0/0  15.2(2)T1
SW1
```

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            ␣
↪REMOTE_PORT         IOS_VERSION
------------  -----------  ----------  --------------------  ---------------------  ------
↪------------  -------------
SW1           SW2          10.1.1.2    cisco WS-C2960-8TC-L  GigabitEthernet1/0/16 ␣
↪GigabitEthernet0/1  12.2(55)SE9
SW1           R1           10.1.1.1    Cisco 3825            GigabitEthernet1/0/22 ␣
↪GigabitEthernet0/0  12.4(24)T1
SW1           R2           10.2.2.2    Cisco 2911            GigabitEthernet1/0/21 ␣
↪GigabitEthernet0/0  15.2(2)T1
```

show ip route ospf

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
O        10.1.1.0/24 [110/20] via 10.0.12.2, 1w2d, Ethernet0/1
O        10.2.2.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
O        10.3.3.3/32 [110/11] via 10.0.12.2, 1w2d, Ethernet0/1
O        10.4.4.4/32 [110/11] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/11] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.5.5.5/32 [110/21] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/21] via 10.0.12.2, 1w2d, Ethernet0/1
                     [110/21] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.6.6.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
```

Для этого примера упрощаем задачу и считаем, что маршруты могут быть только OSPF и с обозначением только O (то есть, только внутризональные маршруты).

Первая версия шаблона выглядит так:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
```

Результат получился такой:

```
network      mask    distance    metric  nexthop
---------  ------  ----------  --------  ---------
10.1.1.0       24         110        20  10.0.12.2
10.2.2.0       24         110        20  10.0.13.3
10.3.3.3       32         110        11  10.0.12.2
10.4.4.4       32         110        11  10.0.13.3
10.5.5.5       32         110        21  10.0.13.3
10.6.6.0       24         110        20  10.0.13.3
```

Всё нормально, но потерялись варианты путей для маршрутов 10.4.4.4/32 и 10.5.5.5/32. Это логично, ведь нет правила, которое подошло бы для такой строки.

Добавляем в шаблон правило для строк с частичными записями:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
```

Теперь вывод выглядит так:

```
network    mask      distance    metric  nexthop
---------  ------  ----------  --------  ---------
10.1.1.0   24             110        20  10.0.12.2
10.2.2.0   24             110        20  10.0.13.3
10.3.3.3   32             110        11  10.0.12.2
10.4.4.4   32             110        11  10.0.13.3
                          110        11  10.0.14.4
10.5.5.5   32             110        21  10.0.13.3
                          110        21  10.0.12.2
                          110        21  10.0.14.4
10.6.6.0   24             110        20  10.0.13.3
```

В частичных записях нехватает сети и маски, но в предыдущих примерах мы уже рассматривали Filldown и, при желании, его можно применить тут, но для этого примера будет использоваться другая опция - List.

List

Воспользуемся опцией **List** для переменной nexthop:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value List nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
```

Теперь вывод получился таким:

```
network    mask      distance    metric  nexthop
---------  ------  ----------  --------  -------------
10.1.1.0   24             110        20  ['10.0.12.2']
10.2.2.0   24             110        20  ['10.0.13.3']
10.3.3.3   32             110        11  ['10.0.12.2']
10.4.4.4   32             110        11  ['10.0.13.3']
                          110        11  ['10.0.14.4']
10.5.5.5   32             110        21  ['10.0.13.3']
                          110        21  ['10.0.12.2']
                          110        21  ['10.0.14.4']
10.6.6.0   24             110        20  ['10.0.13.3']
```

Изменилось то, что в столбце nexthop отображается список, но пока с одним элементом. При использовании List - значение это список, и каждое совпадение с регулярным выражением будет добавлять в список элемент. По умолчанию каждое следующее совпадение перезаписывает предыдущее. Если, например, оставить действие Record только для полных строк:

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value List nexthop (\S+)

Start
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop}, -> Record
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop},
```

Результат будет таким:

```
network      mask    distance    metric  nexthop
---------  ------  ----------  --------  ---------------------------------------
10.1.1.0       24         110        20  ['10.0.12.2']
10.2.2.0       24         110        20  ['10.0.13.3']
10.3.3.3       32         110        11  ['10.0.12.2']
10.4.4.4       32         110        11  ['10.0.13.3']
10.5.5.5       32         110        21  ['10.0.14.4', '10.0.13.3']
10.6.6.0       24         110        20  ['10.0.12.2', '10.0.14.4', '10.0.13.3']
```

Сейчас результат не совсем правильный, адреса хопов записались не к тем маршрутам. Так получилось потому что запись выполняется на полном маршруте, затем хопы неполных маршрутов собираются в список (другие переменные перезаписываются) и когда встречается следующий полный маршрут, список записывается к нему.

```
O        10.4.4.4/32 [110/11] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/11] via 10.0.14.4, 1w2d, Ethernet0/3
```

(continues on next page)

(продолжение с предыдущей страницы)

```
O        10.5.5.5/32 [110/21] via 10.0.13.3, 1w2d, Ethernet0/2
                     [110/21] via 10.0.12.2, 1w2d, Ethernet0/1
                     [110/21] via 10.0.14.4, 1w2d, Ethernet0/3
O        10.6.6.0/24 [110/20] via 10.0.13.3, 1w2d, Ethernet0/2
```

Фактически неполные маршруты действительно надо записывать, когда встретился следующий полный, но при этом, надо записать их к соответствующему маршруту. Надо сделать следующее: как только встретился полный маршрут, надо записать предыдущие значения, а затем продолжить обрабатывать тот же полный маршрут для получения его информации. В TextFSM это можно сделать с помощью действий Continue.Record:

```
^O -> Continue.Record
```

В ней действие **Record** говорит, что надо записать текущее значение переменных. Так как в этом правиле нет переменных, записывается то, что было в предыдущих значениях.

Действие **Continue** говорит, что надо продолжить работать с текущей строкой так, как будто совпадения не было. За счет этого сработает следующая строка шаблона. Итоговый шаблон выглядит так (файл templates/sh_ip_route_ospf.template):

```
Value network (\S+)
Value mask (\d+)
Value distance (\d+)
Value metric (\d+)
Value List nexthop (\S+)

Start
  ^O -> Continue.Record
  ^O +${network}/${mask}\s\[${distance}/${metric}\]\svia\s${nexthop},
  ^\s+\[${distance}/${metric}\]\svia\s${nexthop},
```

В результате мы получим такой вывод:

```
network      mask    distance    metric  nexthop
---------  ------  ----------  --------  ---------------------------------------
10.1.1.0       24         110        20  ['10.0.12.2']
10.2.2.0       24         110        20  ['10.0.13.3']
10.3.3.3       32         110        11  ['10.0.12.2']
10.4.4.4       32         110        11  ['10.0.13.3', '10.0.14.4']
10.5.5.5       32         110        21  ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.6.6.0       24         110        20  ['10.0.13.3']
```

show etherchannel summary

TextFSM удобно использовать для разбора вывода, который отображается столбцами, или для обработки вывода, который находится в разных строках. Менее удобными получаются шаблоны, когда надо получить несколько однотипных элементов из одной строки.

Пример вывода команды show etherchannel summary (файл output/sh_etherchannel_summary.txt):

```
sw1# sh etherchannel summary
Flags:  D - down        P - bundled in port-channel
        I - stand-alone s - suspended
        H - Hot-standby (LACP only)
        R - Layer3      S - Layer2
        U - in use      f - failed to allocate aggregator

        M - not in use, minimum links not met
        u - unsuitable for bundling
        w - waiting to be aggregated
        d - default port


Number of channel-groups in use: 2
Number of aggregators:           2

Group  Port-channel  Protocol    Ports
------+-------------+-----------+-----------------------------------------------
1      Po1(SU)         LACP      Fa0/1(P)   Fa0/2(P)   Fa0/3(P)
3      Po3(SU)          -        Fa0/11(P)   Fa0/12(P)   Fa0/13(P)   Fa0/14(P)
```

В данном случае нужно получить:

- имя и номер port-channel. Например, Po1
- список всех портов в нём. Например, [„Fa0/1“, „Fa0/2“, „Fa0/3“]

Сложность тут в том, что порты находятся в одной строке, а в TextFSM нельзя указывать одну и ту же переменную несколько раз в строке. Но есть возможность несколько раз искать совпадение в строке.

Первая версия шаблона выглядит так:

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Record
```

В шаблоне две переменные:

- CHANNEL - имя и номер агрегированного порта

- MEMBERS - список портов, которые входят в агрегированный порт. Для этой переменной указан тип - List

Результат:

```
CHANNEL    MEMBERS
---------  ----------
Po1        ['Fa0/1']
Po3        ['Fa0/11']
```

Пока что в выводе только первый порт, а нужно, чтобы попали все порты. В данном случае надо продолжить обработку строки с портами после найденного совпадения. То есть, использовать действие Continue и описать следующее выражение.

Единственная строка, которая есть в шаблоне, описывает первый порт. Надо добавить строку, которая описывает следующий порт.

Следующая версия шаблона:

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)

Start
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Record
```

Вторая строка описывает такое же выражение, но переменная MEMBERS смещается на следующий порт.

Результат:

```
CHANNEL    MEMBERS
---------  --------------------
Po1        ['Fa0/1', 'Fa0/2']
Po3        ['Fa0/11', 'Fa0/12']
```

Аналогично надо дописать в шаблон строки, которые описывают третий и четвертый порт. Но, так как в выводе может быть переменное количество портов, надо перенести правило Record на отдельную строку, чтобы оно не было привязано к конкретному количеству портов в строке.

Если Record будет находиться, например, после строки, в которой описаны четыре порта, для ситуации, когда портов в строке меньше, запись не будет выполняться.

Итоговый шаблон (файл templates/sh_etherchannel_summary.txt):

```
Value CHANNEL (\S+)
Value List MEMBERS (\w+\d+\/\d+)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Start
  ^\d+.* -> Continue.Record
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +\S+ +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){2} +${MEMBERS}\( -> Continue
  ^\d+ +${CHANNEL}\(\S+ +[\w-]+ +[\w ]+ +(\S+ +){3} +${MEMBERS}\( -> Continue
```

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
---------  ------------------------------------------------------------
Po1        ['Fa0/1', 'Fa0/2', 'Fa0/3']
Po3        ['Fa0/11', 'Fa0/12', 'Fa0/13', 'Fa0/14', 'Fa0/15', 'Fa0/16']
```

На этом мы заканчиваем разбираться с шаблонами TextFSM.

Примеры шаблонов для Cisco и другого оборудования можно посмотреть в проекте ntc-ansible.

TextFSM CLI Table

Благодаря TextFSM можно обрабатывать вывод команд и получать структурированный результат. Однако, всё ещё надо вручную прописывать, каким шаблоном обрабатывать команды show, каждый раз, когда используется TextFSM.

Было бы намного удобней иметь какое-то соответствие между командой и шаблоном, чтобы можно было написать общий скрипт, который выполняет подключения к устройствам, отправляет команды, сам выбирает шаблон и парсит вывод в соответствии с шаблоном.

В TextFSM есть такая возможность. Для того, чтобы ею 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]]`. Запись будет преобразована в выражение `sh((ow)?)? ip int((erface)?)? br((ief)?)?`. Это значит, что TextFSM сможет определить, какой шаблон использовать, даже если команда набрана не полностью. Например, такие варианты команды сработают:

- sh ip int br
- show ip inter bri

Как использовать CLI table

Посмотрим, как пользоваться классом clitable и файлом index.

В каталоге templates такие шаблоны и файл index:

```
sh_cdp_n_det.template
sh_clock.template
sh_ip_int_br.template
sh_ip_route_ospf.template
index
```

Сначала попробуем поработать с CLI Table в ipython, чтобы посмотреть, какие возможности есть у этого класса, а затем посмотрим на финальный скрипт.

Для начала импортируем класс clitable:

```
In [1]: from textfsm import clitable
```

Предупреждение: В зависимости от версии textfsm, надо по-разному импортировать clitable:

- `import clitable` для версии <= 0.4.1
- `from textfsm import clitable` для версии >= 1.1.0

Посмотреть версию textfsm: `pip show textfsm`.

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]: with open('output/sh_ip_route_ospf.txt') as f:
   ...:     output_sh_ip_route_ospf = f.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_
↪table.size
cli_table.AddKeys          cli_table.ParseCmd         cli_table.index_file       cli_
↪table.sort
```

(continues on next page)

(продолжение с предыдущей страницы)

```
cli_table.Append            cli_table.ReadIndex        cli_table.next             cli_
↪table.superkey
cli_table.CsvToTable        cli_table.Remove           cli_table.raw              cli_
↪table.synchronised
cli_table.FormattedTable    cli_table.Reset            cli_table.row              cli_
↪table.table
cli_table.INDEX             cli_table.RowWith          cli_table.row_class        cli_
↪table.template_dir
cli_table.KeyValue          cli_table.extend           cli_table.row_index
cli_table.LabelValueTable   cli_table.header           cli_table.separator
```

Например, если вызвать `print cli_table`, получим такой вывод:

```
In [7]: print(cli_table)
Network, Mask, Distance, Metric, NextHop
10.0.24.0, /24, 110, 20, ['10.0.12.2']
10.0.34.0, /24, 110, 20, ['10.0.13.3']
10.2.2.2, /32, 110, 11, ['10.0.12.2']
10.3.3.3, /32, 110, 11, ['10.0.13.3']
10.4.4.4, /32, 110, 21, ['10.0.13.3', '10.0.12.2', '10.0.14.4']
10.5.35.0, /24, 110, 20, ['10.0.13.3']
```

Метод FormattedTable позволяет получить вывод в виде таблицы:

```
In [8]: print(cli_table.FormattedTable())
 Network    Mask  Distance  Metric  NextHop
=====================================================================
 10.0.24.0  /24   110       20      10.0.12.2
 10.0.34.0  /24   110       20      10.0.13.3
 10.2.2.2   /32   110       11      10.0.12.2
 10.3.3.3   /32   110       11      10.0.13.3
 10.4.4.4   /32   110       21      10.0.13.3, 10.0.12.2, 10.0.14.4
 10.5.35.0  /24   110       20      10.0.13.3
```

Такой вывод может пригодиться для отображения информации.

Чтобы получить из объекта cli_table структурированный вывод, например, список списков, надо обратиться к объекту таким образом:

```
In [9]: data_rows = [list(row) for row in cli_table]

In [11]: data_rows
Out[11]:
[['10.0.24.0', '/24', '110', '20', ['10.0.12.2']],
 ['10.0.34.0', '/24', '110', '20', ['10.0.13.3']],
 ['10.2.2.2', '/32', '110', '11', ['10.0.12.2']],
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 ['10.3.3.3', '/32', '110', '11', ['10.0.13.3']],
 ['10.4.4.4', '/32', '110', '21', ['10.0.13.3', '10.0.12.2', '10.0.14.4']],
 ['10.5.35.0', '/24', '110', '20', ['10.0.13.3']]]
```

Отдельно можно получить названия столбцов:

```
In [12]: header = list(cli_table.header)

In [14]: header
Out[14]: ['Network', 'Mask', 'Distance', 'Metric', 'NextHop']
```

Теперь вывод аналогичен тому, который был получен в прошлом разделе.

Соберем всё в один скрипт (файл textfsm_clitable.py):

```
from textfsm import clitable

output_sh_ip_route_ospf = open('output/sh_ip_route_ospf.txt').read()

cli_table = clitable.CliTable('index', 'templates')

attributes = {'Command': 'show ip route ospf', 'Vendor': 'Cisco'}

cli_table.ParseCmd(output_sh_ip_route_ospf, attributes)
print('CLI Table output:\n', cli_table)

print('Formatted Table:\n', cli_table.FormattedTable())

data_rows = [list(row) for row in cli_table]
header = list(cli_table.header)

print(header)
for row in data_rows:
    print(row)
```

В упражнениях к этому разделу будет задание, в котором надо объединить описанную процедуру в функцию, а также вариант с получением списка словарей.

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']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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 опциональным аргументам.

Дополнительные материалы

Документация:

- TextFSM

Статьи:

- Programmatic Access to CLI Devices with TextFSM. Jason Edelman (26.02.2015) - основы TextFSM и идеи о развитии, которые легли в основу модуля ntc-ansible
- Parse CLI outputs with TextFSM. Henry Ölsner (24.08.2015) - пример использования TextFSM для разбора большого файла с выводом sh inventory. Подробнее объясняется синтаксис TextFSM
- Creating Templates for TextFSM and ntc_show_command. Jason Edelman (27.08.2015) - подробнее рассматривается синтаксис TextFSM, и показаны примеры использования модуля ntc-ansible

  (обратите внимание, что синтаксис модуля уже немного изменился)
- TextFSM and Structured Data. Kirk Byers (22.10.2015) - вводная статья о TextFSM. Тут не описывается синтаксис, но дается общее представление о том, что такое TextFSM, и

пример его использования

Проекты, которые используют TextFSM:

- Модуль ntc-ansible

Шаблоны TextFSM (из модуля ntc-ansible):

- ntc-templates

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 21.1

Создать функцию parse_command_output. Параметры функции:

- template - имя файла, в котором находится шаблон TextFSM (templates/sh_ip_int_br.template)
- command_output - вывод соответствующей команды show (строка)

Функция должна возвращать список:

- первый элемент - это список с названиями столбцов
- остальные элементы это списки, в котором находятся результаты обработки вывода

Проверить работу функции на выводе команды output/sh_ip_int_br.txt и шаблоне templates/sh_ip_int_br.template.

```
from netmiko import ConnectHandler


# вызов функции должен выглядеть так
if __name__ == "__main__":
    r1_params = {
        "device_type": "cisco_ios",
        "host": "192.168.100.1",
        "username": "cisco",
        "password": "cisco",
        "secret": "cisco",
    }
    with ConnectHandler(**r1_params) as r1:
        r1.enable()
        output = r1.send_command("sh ip int br")
    result = parse_command_output("templates/sh_ip_int_br.template", output)
    print(result)
```

Задание 21.1a

Создать функцию parse_output_to_dict.

Параметры функции:

- template - имя файла, в котором находится шаблон TextFSM (templates/sh_ip_int_br.template)
- command_output - вывод соответствующей команды show (строка)

Функция должна возвращать список словарей:

- ключи - имена переменных в шаблоне TextFSM
- значения - части вывода, которые соответствуют переменным

Проверить работу функции на выводе команды output/sh_ip_int_br.txt и шаблоне templates/sh_ip_int_br.template.

Задание 21.2

Сделать шаблон TextFSM для обработки вывода sh ip dhcp snooping binding и записать его в файл templates/sh_ip_dhcp_snooping.template

Вывод команды находится в файле output/sh_ip_dhcp_snooping.txt.

Шаблон должен обрабатывать и возвращать значения таких столбцов:

- mac - такого вида 00:04:A3:3E:5B:69
- ip - такого вида 10.1.10.6
- vlan - 10
- intf - FastEthernet0/10

Проверить работу шаблона с помощью функции parse_command_output из задания 21.1.

Задание 21.3

Создать функцию parse_command_dynamic.

Параметры функции:

- command_output - вывод команды (строка)
- attributes_dict - словарь атрибутов, в котором находятся такие пары ключ-значение:
  - „Command“: команда
  - „Vendor“: вендор

- index_file - имя файла, где хранится соответствие между командами и шаблонами. Значение по умолчанию - «index»
- templ_path - каталог, где хранятся шаблоны. Значение по умолчанию - «templates»

Функция должна возвращать список словарей с результатами обработки вывода команды (как в задании 21.1a):

- ключи - имена переменных в шаблоне TextFSM
- значения - части вывода, которые соответствуют переменным

Проверить работу функции на примере вывода команды sh ip int br.

Задание 21.4

Создать функцию send_and_parse_show_command.

Параметры функции:

- device_dict - словарь с параметрами подключения к одному устройству
- command - команда, которую надо выполнить
- templates_path - путь к каталогу с шаблонами TextFSM
- index - имя индекс файла, значение по умолчанию «index»

Функция должна подключаться к одному устройству, отправлять команду show с помощью netmiko, а затем парсить вывод команды с помощью TextFSM.

Функция должна возвращать список словарей с результатами обработки вывода команды (как в задании 21.1a):

- ключи - имена переменных в шаблоне TextFSM
- значения - части вывода, которые соответствуют переменным

Проверить работу функции на примере вывода команды sh ip int br и устройствах из devices.yaml.

Задание 21.5

Создать функцию send_and_parse_command_parallel.

Функция send_and_parse_command_parallel должна запускать в параллельных потоках функцию send_and_parse_show_command из задания 21.4.

Параметры функции send_and_parse_command_parallel:

- devices - список словарей с параметрами подключения к устройствам
- command - команда

- templates_path - путь к каталогу с шаблонами TextFSM
- limit - максимальное количество параллельных потоков (по умолчанию 3)

Функция должна возвращать словарь:

- ключи - IP-адрес устройства с которого получен вывод
- значения - список словарей (вывод который возвращает функция send_and_parse_show_command)

Пример словаря:

```
{'192.168.100.1': [{'address': '192.168.100.1',
                    'intf': 'Ethernet0/0',
                    'protocol': 'up',
                    'status': 'up'},
                   {'address': '192.168.200.1',
                    'intf': 'Ethernet0/1',
                    'protocol': 'up',
                    'status': 'up'}],
 '192.168.100.2': [{'address': '192.168.100.2',
                    'intf': 'Ethernet0/0',
                    'protocol': 'up',
                    'status': 'up'},
                   {'address': '10.100.23.2',
                    'intf': 'Ethernet0/1',
                    'protocol': 'up',
                    'status': 'up'}]}
```

Проверить работу функции на примере вывода команды sh ip int br и устройствах из devices.yaml.

7

VI. Основы объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) - это методология программирования, в которой программа состоит из объектов, которые взаимодействуют между собой. Объекты создаются на основании класса, определенного в коде и, как правило, объединяют данные и действия, которые можно выполнять с данными, в одно целое.

Без использования ООП вполне можно писать код, но, как минимум, изучение основ ООП поможет лучше понимать, что такое объект, класс, метод, переменная объекта, а это те вещи, которые используются в Python постоянно. Кроме того, знание ООП пригодится в чтении чужого кода. Например, будет проще разобраться в коде netmiko.

Хотя ООП лежит в основе того как все устроено в Python, при написании кода не обязательно использовать объектно-ориентированный подход.

> Тут речь о том, что в Python не обязательно нужно создавать классы, чтобы что-то сделать.

22. Основы ООП

Основы ООП

- Класс (class) - элемент программы, который описывает какой-то тип данных. Класс описывает шаблон для создания объектов, как правило, указывает переменные этого объекта и действия, которые можно выполнять применимо к объекту.
- Экземпляр класса (instance) - объект, который является представителем класса.
- Метод (method) - функция, которая определена внутри класса и описывает какое-то действие, которое поддерживает класс
- Переменная экземпляра (instance variable, а иногда и instance attribute) - данные, которые относятся к объекту
- Переменная класса (class variable) - данные, которые относятся к классу и разделяются всеми экземплярами класса
- Атрибут экземпляра (instance attribute) - переменные и методы, которые относятся к объектам (экземплярам) созданным на основании класса. У каждого объекта есть своя копия атрибутов.

Пример из реальной жизни в стиле ООП:

- Проект дома - это класс
- Конкретный дом, который был построен по проекту - экземпляр класса
- Такие особенности как цвет дома, количество окон - переменные экземпляра, то есть конкретного дома
- Дом можно продать, перекрасить, отремонтировать - это методы

Например, при работе с netmiko первое, что нужно было сделать создать подключение:

```
from netmiko import ConnectHandler

device = {
    "device_type": "cisco_ios",
    "host": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}

ssh = ConnectHandler(**device)
```

Переменная ssh - это объект, который представляет реальное соединение с оборудованием. Благодаря функции type, можно выяснить экземпляром какого класса является объект ssh:

```
In [3]: type(ssh)
Out[3]: netmiko.cisco.cisco_ios.CiscoIosSSH
```

У ssh есть свои методы и переменные, которые зависят от состояния текущего объекта. Например, переменная экземпляра `ssh.host` доступна у каждого экземпляра класса `netmiko.cisco.cisco_ios.CiscoIosSSH` и возвращает IP-адрес или имя хоста, в зависимости от того что указывалось в словаре device:

```
In [4]: ssh.host
Out[4]: '192.168.100.1'
```

Метод send_command выполняет команду на оборудовании:

```
In [5]: ssh.send_command("sh clock")
Out[5]: '*10:08:50.654 UTC Tue Feb 2 2021'
```

Метод enable переходит в режим enable и при этом объект ssh сохраняет состояние: до и после перехода видно разное приглашение:

```
In [6]: ssh.find_prompt()
Out[6]: 'R1>'

In [7]: ssh.enable()
Out[7]: 'enable\r\nPassword: \r\nR1#'

In [8]: ssh.find_prompt()
Out[8]: 'R1#'
```

В этом примере показаны важные аспекты ООП: объединение данных и действия над данными, а также сохранение состояния.

До сих пор, при написании кода, данные и действия были разделены. Чаще всего, действия описаны в виде функций, а данные передаются как аргументы этим функциям. При создании класса, данные и действия объединяются. Конечно же, эти данные и действия связаны. То есть, методами класса становятся те действия, которые характерны именно для объекта такого типа, а не какие-то произвольные действия.

Например, в экзепляре класса str, все методы относятся к работе с этой строкой:

```
In [10]: s = 'string'

In [11]: s.upper()
Out[11]: 'STRING'

In [12]: s.center(20, '=')
Out[12]: '=======string======='
```

Выше, при обращении к атрибутам экземпляра (переменным и методам) используется такой синтаксис: `objectname.attribute`. Эта запись `s.lower()` означает: вызвать метод lower у объекта s. Обращение к методам и переменным выполняется одинаково, но для вызова метода, надо добавить скобки и передать все необходимые аргументы.

Всё описанное неоднократно использовалось в книге, но теперь мы разберемся с формальной терминологией.

Создание класса

Примечание: Обратите внимание, что тут основы поясняются с учетом того, что у читающего нет опыта работы с ООП. Некоторые примеры не очень правильны с точки зрения идеологии Python, но помогают лучше понять происходящее. В конце даются пояснения как это правильней делать.

Для создания классов в питоне используется ключевое слово `class`. Самый простой класс, который можно создать в Python:

```
In [1]: class Switch:
   ...:     pass
   ...:
```

Примечание: Имена классов: в Python принято писать имена классов в формате CamelCase.

Для создания экземпляра класса, надо вызвать класс:

```
In [2]: sw1 = Switch()

In [3]: print(sw1)
<__main__.Switch object at 0xb44963ac>
```

Используя точечную нотацию, можно получать значения переменных экземпляра, создавать новые переменные и присваивать новое значение существующим:

```
In [5]: sw1.hostname = 'sw1'

In [6]: sw1.model = 'Cisco 3850'
```

В другом экземпляре класса Switch, переменные могут быть другие:

```
In [7]: sw2 = Switch()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [8]: sw2.hostname = 'sw2'

In [9]: sw2.model = 'Cisco 3750'
```

Посмотреть значение переменных экземпляра можно используя ту же точечную нотацию:

```
In [10]: sw1.model
Out[10]: 'Cisco 3850'

In [11]: sw2.model
Out[11]: 'Cisco 3750'
```

Создание метода

Прежде чем мы начнем разбираться с методами класса, посмотрим пример функции, которая ожидает как аргумент экземпляр класса Switch и выводит информацию о нем, используя переменные экземпляра hostname и model:

```
In [1]: class Switch:
   ...:     pass
   ...:

In [2]: def info(sw_obj):
   ...:     print('Hostname: {}\nModel: {}'.format(sw_obj.hostname, sw_obj.model))
   ...:

In [3]: sw1 = Switch()

In [4]: sw1.hostname = 'sw1'

In [5]: sw1.model = 'Cisco 3850'

In [6]: info(sw1)
Hostname: sw1
Model: Cisco 3850
```

В функции info параметр `sw_obj` ожидает экземпляр класса `Switch`. Скорее всего, в этом примере нет ничего нового, ведь аналогичным образом ранее мы писали функции, которые ожидают строку, как аргумент, а затем вызывают какие-то методы у этой строки.

Этот пример поможет разобраться с методом info, который мы добавим в класс Switch.

Для добавления метода, необходимо создать функцию внутри класса:

```
In [15]: class Switch:
    ...:     def info(self):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
    ...:
```

Если присмотреться, метод info выглядит точно так же, как функция info, только вместо имени sw_obj, используется self. Почему тут используется странное имя self, мы разберемся позже, а пока посмотрим как вызвать метод info:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'

In [19]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

В примере выше сначала создается экземпляр класса Switch, затем в экземпляр добавляются переменные hostname и model, и только после этого вызывается метод info. Метод info выводит информацию про коммутатор, используя значения, которые хранятся в переменных экземпляра.

Вызов метода отличается, от вызова функции: мы не передаем ссылку на экземпляр класса Switch. Нам это не нужно, потому что мы вызываем метод у самого экземпляра. Еще один непонятный момент - зачем же мы тогда писали self.

Все дело в том, что Python преобразует такой вызов:

```
In [39]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

Вот в такой:

```
In [38]: Switch.info(sw1)
Hostname: sw1
Model: Cisco 3850
```

Во втором случае, в параметре self уже больше смысла, он действительно принимает ссылку на экземпляр и на основании этого выводит информацию.

С точки зрения использования объектов, удобней вызывать методы используя первый вариант синтаксиса, поэтому, практически всегда именно он и используется.

Примечание: При вызове метода экземпляра класса, ссылка на экземпляр передается пер-

вым аргументом. При этом, экземпляр передается неявно, но параметр надо указывать явно.

Такое преобразование не является особенностью пользовательских классов и работает и для встроенных типов данных аналогично. Например, стандартный способ вызова метода append в списке, выглядит так:

```
In [4]: a = [1,2,3]

In [5]: a.append(5)

In [6]: a
Out[6]: [1, 2, 3, 5]
```

При этом, то же самое можно сделать и используя второй вариант, вызова через класс:

```
In [7]: a = [1,2,3]

In [8]: list.append(a, 5)

In [9]: a
Out[9]: [1, 2, 3, 5]
```

Параметр self

Параметр self указывался выше в определении методов, а также при использовании переменных экземпляра в методе. Параметр self это ссылка на конкретный экземпляр класса. При этом, само имя self не является особенным, а лишь договоренностью. Вместо self можно использовать другое имя, но так делать не стоит.

Пример с использованием другого имени, вместо self:

```
In [15]: class Switch:
    ...:     def info(sw_object):
    ...:         print(f'Hostname: {sw_object.hostname}\nModel: {sw_object.model}')
    ...:
```

Работать все будет аналогично:

```
In [16]: sw1 = Switch()

In [17]: sw1.hostname = 'sw1'

In [18]: sw1.model = 'Cisco 3850'

In [19]: sw1.info()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Hostname: sw1
Model: Cisco 3850
```

Предупреждение: Хотя технически использовать другое имя можно, всегда используйте self.

Во всех «обычных» методах класса первым параметром всегда будет self. Кроме того, создание переменной экземпляра внутри класса также выполняется через self.

Пример класса Switch с новым методом generate_interfaces: метод generate_interfaces должен сгенерировать список с интерфейсами на основании указанного типа и количества и создать переменную в экземпляре класса. Для начала, вариант создания обычно переменной внутри метода:

```
In [5]: class Switch:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = [f"{intf_type}{number}" for number in range(1, number_of_
↪intf + 1)]
   ...:
```

В этом случае, в экземплярах класса не будет переменной interfaces:

```
In [6]: sw1 = Switch()

In [7]: sw1.generate_interfaces('Fa', 10)

In [8]: sw1.interfaces
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-e6b457e4e23e> in <module>()
----> 1 sw1.interfaces

AttributeError: 'Switch' object has no attribute 'interfaces'
```

Этой переменной нет, потому что она существует только внутри метода, а область видимости у метода такая же, как и у функции. Даже другие методы одного и того же класса, не видят переменные в других методах.

Чтобы список с интерфейсами был доступен как переменная в экземплярах, надо присвоить значение в self.interfaces:

```
In [9]: class Switch:
   ...:     def info(self):
   ...:         print(f"Hostname: {self.hostname}\nModel: {self.model}")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:
   ...:     def generate_interfaces(self, intf_type, number_of_intf):
   ...:         interfaces = [f"{intf_type}{number}" for number in range(1, number_of_
↪intf+1)]
   ...:         self.interfaces = interfaces
   ...:
```

Теперь, после вызова метода generate_interfaces, в экземпляре создается переменная interfaces:

```
In [10]: sw1 = Switch()

In [11]: sw1.generate_interfaces('Fa', 10)

In [12]: sw1.interfaces
Out[12]: ['Fa1', 'Fa2', 'Fa3', 'Fa4', 'Fa5', 'Fa6', 'Fa7', 'Fa8', 'Fa9', 'Fa10']
```

Метод __init__

Для корректной работы метода info, необходимо чтобы у экземпляра были переменные hostname и model. Если этих переменных нет, возникнет ошибка:

```
In [15]: class Switch:
    ...:     def info(self):
    ...:         print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
    ...:

In [59]: sw2 = Switch()

In [60]: sw2.info()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-60-5a006dd8aae1> in <module>()
----> 1 sw2.info()

<ipython-input-57-30b05739380d> in info(self)
      1 class Switch:
      2     def info(self):
----> 3         print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))

AttributeError: 'Switch' object has no attribute 'hostname'
```

Практически всегда, при создании объекта, у него есть какие-то начальные данные. Например, чтобы создать подключение к оборудование с помощью netmiko, надо передать параметры подключения.

В Python эти начальные данные про объект указываются в методе `__init__`. Метод `__init__` выполняется после того как Python создал новый экземпляр и, при этом, методу `__init__` передаются аргументы с которыми был создан экземпляр:

```
In [32]: class Switch:
    ...:     def __init__(self, hostname, model):
    ...:         self.hostname = hostname
    ...:         self.model = model
    ...:
    ...:     def info(self):
    ...:         print(f'Hostname: {self.hostname}\nModel: {self.model}')
    ...:
```

Обратите внимание на то, что у каждого экземпляра, который создан из этого класса, будут созданы переменные: `self.model` и `self.hostname`.

Теперь, при создании экземпляра класса Switch, обязательно надо указать hostname и model:

```
In [33]: sw1 = Switch('sw1', 'Cisco 3850')
```

И, соответственно, метод info отрабатывает без ошибок:

```
In [36]: sw1.info()
Hostname: sw1
Model: Cisco 3850
```

Примечание: Метод `__init__` иногда называют конструктором класса, хотя технически в Python сначала выполняется метод `__new__`, а затем `__init__`. В большинстве случаев, метод `__new__` использовать не нужно.

Важной особенностью метода `__init__` является то, что он не должен ничего возвращать. Python сгенерирует исключение, если попытаться это сделать.

Пример класса

Пример класса, который описывает сеть:

```
class Network:
    def __init__(self, network):
        self.network = network
        self.hosts = tuple(str(ip) for ip in ipaddress.ip_network(network).hosts())
        self.allocated = []

    def allocate(self, ip):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if ip in self.hosts:
            if ip not in self.allocated:
                self.allocated.append(ip)
            else:
                raise ValueError(f"IP-адрес {ip} уже находится в allocated")
        else:
            raise ValueError(f"IP-адрес {ip} не входит в сеть {self.network}")
```

Использование класса:

```
In [2]: net1 = Network("10.1.1.0/29")

In [3]: net1.allocate("10.1.1.1")

In [4]: net1.allocate("10.1.1.2")

In [5]: net1.allocated
Out[5]: ['10.1.1.1', '10.1.1.2']

In [6]: net1.allocate("10.1.1.100")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-9a4157e02c78> in <module>
----> 1 net1.allocate("10.1.1.100")

<ipython-input-1-c5255d37a7fd> in allocate(self, ip)
     12                 raise ValueError(f"IP-адрес {ip} уже находится в allocated")
     13         else:
---> 14             raise ValueError(f"IP-адрес {ip} не входит в сеть {self.network}")
     15

ValueError: IP-адрес 10.1.1.100 не входит в сеть 10.1.1.0/29
```

Область видимости

У каждого метода в классе своя локальная область видимости. Это значит, что один метод класса не видит переменные другого метода класса. Для того чтобы переменные были доступны, надо присваивать их экземпляру через `self.name`. По сути метод - это функция привязанная к объекту. Поэтому все нюансы, которые касаются функций, относятся и к методам.

Переменные экземпляра доступны в другом методе, потому что каждому методу первым аргументом передается сам экзепляр. В примере ниже, в методе `__init__` переменные hostname и model присваиваются экземпляру, а затем в info используются, за счет того, что экземпляр передается первым аргументом:

```
class Switch:
    def __init__(self, hostname, model):
        self.hostname = hostname
        self.model = model

    def info(self):
        print('Hostname: {}\nModel: {}'.format(self.hostname, self.model))
```

Переменные класса

Помимо переменных экземпляра, существуют также переменные класса. Они создаются, при указании переменных внутри самого класса, не метода:

```
class Network:
    all_allocated_ip = []

    def __init__(self, network):
        self.network = network
        self.hosts = tuple(
            str(ip) for ip in ipaddress.ip_network(network).hosts()
        )
        self.allocated = []

    def allocate(self, ip):
        if ip in self.hosts:
            if ip not in self.allocated:
                self.allocated.append(ip)
                type(self).all_allocated_ip.append(ip)
            else:
                raise ValueError(f"IP-адрес {ip} уже находится в allocated")
        else:
            raise ValueError(f"IP-адрес {ip} не входит в сеть {self.network}")
```

К переменным класса можно обращаться по-разному:

- `self.all_allocated_ip`
- `Network.all_allocated_ip`
- `type(self).all_allocated_ip`

Вариант `self.all_allocated_ip` позволяет обратиться к значению переменной класса или добавить элемент, если переменная класса изменяемый тип данных. Минус этого варианта в том, что если в методе написать `self.all_allocated_ip = ...`, вместо изменения переменной класса, будет создана переменная экземпляра.

Вариант `Network.all_allocated_ip` будет работать корректно, но небольшой минус этого

варианта в том, что имя класса прописано вручную. Вместо него можно использовать третий вариант `type(self).all_allocated_ip`, так как `type(self)` возвращает класс.

Теперь у класса есть переменная all_allocated_ip в которую записываются все IP-адреса, которые выделены в сетях:

```
In [3]: net1 = Network("10.1.1.0/29")

In [4]: net1.allocate("10.1.1.1")
   ...: net1.allocate("10.1.1.2")
   ...: net1.allocate("10.1.1.3")
   ...:

In [5]: net1.allocated
Out[5]: ['10.1.1.1', '10.1.1.2', '10.1.1.3']

In [6]: net2 = Network("10.2.2.0/29")

In [7]: net2.allocate("10.2.2.1")
   ...: net2.allocate("10.2.2.2")
   ...:

In [9]: net2.allocated
Out[9]: ['10.2.2.1', '10.2.2.2']

In [10]: Network.all_allocated_ip
Out[10]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']
```

Переменная доступна не только через класс, но и через экземпляры:

```
In [40]: Network.all_allocated_ip
Out[40]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']

In [41]: net1.all_allocated_ip
Out[41]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']

In [42]: net2.all_allocated_ip
Out[42]: ['10.1.1.1', '10.1.1.2', '10.1.1.3', '10.2.2.1', '10.2.2.2']
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 22.1

Создать класс Topology, который представляет топологию сети.

При создании экземпляра класса, как аргумент передается словарь, который описывает топологию. Словарь может содержать «дублирующиеся» соединения. Тут «дублирующиеся» соединения, это ситуация когда в словаре есть два соединения:

```
("R1", "Eth0/0"): ("SW1", "Eth0/1")
("SW1", "Eth0/1"): ("R1", "Eth0/0")
```

Задача оставить только один из этих линков в итоговом словаре, не важно какой.

В каждом экземпляре должна быть создана переменная topology, в которой содержится словарь топологии, но уже без «дублей». Переменная topology должна содержать словарь без «дублей» сразу после создания экземпляра.

Пример создания экземпляра класса:

```
In [2]: top = Topology(topology_example)
```

После этого, должна быть доступна переменная topology:

```
In [3]: top.topology
Out[3]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

```
topology_example = {('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
                    ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
                    ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
                    ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
                    ('R3', 'Eth0/2'): ('R5', 'Eth0/0'),
                    ('SW1', 'Eth0/1'): ('R1', 'Eth0/0'),
                    ('SW1', 'Eth0/2'): ('R2', 'Eth0/0'),
                    ('SW1', 'Eth0/3'): ('R3', 'Eth0/0')}
```

Задание 22.1a

Скопировать класс Topology из задания 22.1 и изменить его.

Перенести функциональность удаления «дублей» в метод _normalize. При этом метод __init__ должен выглядеть таким образом:

```
class Topology:
    def __init__(self, topology_dict):
        self.topology = self._normalize(topology_dict)
```

Задание 22.1b

Изменить класс Topology из задания 22.1a или 22.1.

Добавить метод delete_link, который удаляет указанное соединение. Метод должен удалять и «обратное» соединение, если оно есть (ниже пример).

Если такого соединения нет, выводится сообщение «Такого соединения нет».

Создание топологии

```
In [7]: t = Topology(topology_example)

In [8]: t.topology
Out[8]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Удаление линка:

```
In [9]: t.delete_link(('R3', 'Eth0/1'), ('R4', 'Eth0/0'))

In [10]: t.topology
Out[10]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Удаление «обратного» линка: в словаре есть запись `('R3', 'Eth0/2'): ('R5', 'Eth0/0')`, но вызов delete_link с указанием ключа и значения в обратном порядке, должно удалять соединение:

```
In [11]: t.delete_link(('R5', 'Eth0/0'), ('R3', 'Eth0/2'))

In [12]: t.topology
Out[12]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3')}
```

Если такого соединения нет, выводится сообщение:

```
In [13]: t.delete_link(('R5', 'Eth0/0'), ('R3', 'Eth0/2'))
Такого соединения нет
```

Задание 22.1c

Изменить класс Topology из задания 22.1b.

Добавить метод delete_node, который удаляет все соединения с указаным устройством. Если такого устройства нет, выводится сообщение «Такого устройства нет».

Создание топологии

```
In [1]: t = Topology(topology_example)

In [2]: t.topology
Out[2]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Удаление устройства:

```
In [3]: t.delete_node('SW1')

In [4]: t.topology
Out[4]:
{('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Если такого устройства нет, выводится сообщение:

```
In [5]: t.delete_node('SW1')
Такого устройства нет
```

Задание 22.1d

Изменить класс Topology из задания 22.1c

Добавить метод add_link, который добавляет указанное соединение, если его еще нет в топологии. Если соединение существует, вывести сообщение «Такое соединение существует». Если одна из сторон есть в топологии, вывести сообщение «Соединение с одним из портов существует».

Пример создания топологии и добавления соединений

```
In [7]: t = Topology(topology_example)

In [8]: t.topology
Out[8]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [9]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/0'))

In [10]: t.topology
Out[10]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 ('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [11]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/0'))
Такое соединение существует

In [12]: t.add_link(('R1', 'Eth0/4'), ('R7', 'Eth0/5'))
Соединение с одним из портов существует
```

Задание 22.2

Создать класс CiscoTelnet, который подключается по Telnet к оборудованию Cisco.

При создании экземпляра класса, должно создаваться подключение Telnet, а также переход в режим enable. Класс должен использовать модуль telnetlib для подключения по Telnet.

У класса CiscoTelnet, кроме __init__, должно быть, как минимум, два метода:

- _write_line - принимает как аргумент строку и отправляет на оборудование строку преобразованную в байты и добавляет перевод строки в конце. Метод _write_line должен использоваться внутри класса.
- send_show_command - принимает как аргумент команду show и возвращает вывод полученный с обрудования

Параметры метода __init__:

- ip - IP-адрес
- username - имя пользователя
- password - пароль
- secret - пароль enable

Пример создания экземпляра класса:

```
In [2]: from task_22_2 import CiscoTelnet

In [3]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   ...:

In [4]: r1 = CiscoTelnet(**r1_params)

In [5]: r1.send_show_command('sh ip int br')
Out[5]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
↪           Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up
↪           up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up
↪           up      \r\nEthernet0/2                190.16.200.1    YES NVRAM  up
↪           up      \r\nEthernet0/3                192.168.130.1   YES NVRAM  up
↪           up      \r\nEthernet0/3.100            10.100.0.1      YES NVRAM  up
↪           up      \r\nEthernet0/3.200            10.200.0.1      YES NVRAM  up
↪           up      \r\nEthernet0/3.300            10.30.0.1       YES NVRAM  up
↪           up      \r\nLoopback0                  10.1.1.1        YES NVRAM  up
↪           up      \r\nLoopback55                 5.5.5.5         YES manual up
↪           up      \r\nR1#'
```

Примечание: Подсказка: Метод _write_line нужен для того чтобы можно было сократить строку: `self.telnet.write(line.encode("ascii") + b"\n")` до такой: `self._write_line(line)`. Он не должен делать ничего другого.

Задание 22.2a

Скопировать класс CiscoTelnet из задания 22.2 и изменить метод send_show_command добавив три параметра:

- parse - контролирует то, будет возвращаться обычный вывод команды или список словарей, полученные после обработки с помощью TextFSM. При parse=True должен возвращаться список словарей, а parse=False обычный вывод. Значение по умолчанию - True.
- templates - путь к каталогу с шаблонами. Значение по умолчанию - «templates»
- index - имя файла, где хранится соответствие между командами и шаблонами. Значение по умолчанию - «index»

Пример создания экземпляра класса:

```
In [1]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [2]: from task_22_2a import CiscoTelnet
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [3]: r1 = CiscoTelnet(**r1_params)
```

Использование метода send_show_command:

```
In [4]: r1.send_show_command("sh ip int br", parse=True)
Out[4]:
[{'intf': 'Ethernet0/0',
  'address': '192.168.100.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/1',
  'address': '192.168.200.1',
  'status': 'up',
  'protocol': 'up'},
 {'intf': 'Ethernet0/2',
  'address': '192.168.130.1',
  'status': 'up',
  'protocol': 'up'}]

In [5]: r1.send_show_command("sh ip int br", parse=False)
Out[5]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up
up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up...'
```

Задание 22.2b

Скопировать класс CiscoTelnet из задания 22.2a и добавить метод send_config_commands.

Метод send_config_commands должен уметь отправлять одну команду конфигурационного режима или список команд. Метод дожен возвращать вывод аналогичный методу send_config_set у netmiko (пример вывода ниже).

Пример создания экземпляра класса:

```
In [1]: from task_22_2b import CiscoTelnet

In [2]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [3]: r1 = CiscoTelnet(**r1_params)
```

Использование метода send_config_commands:

```
In [5]: r1.send_config_commands('logging 10.1.1.1')
Out[5]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
↪nR1(config)#logging 10.1.1.1\r\nR1(config)#end\r\nR1#'

In [6]: r1.send_config_commands(['interface loop55', 'ip address 5.5.5.5 255.255.255.255
↪'])
Out[6]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
↪nR1(config)#interface loop55\r\nR1(config-if)#ip address 5.5.5.5 255.255.255.255\r\
↪nR1(config-if)#end\r\nR1#'
```

Задание 22.2c

Скопировать класс CiscoTelnet из задания 22.2b и изменить метод send_config_commands добавив проверку команд на ошибки.

У метода send_config_commands должен быть дополнительный параметр strict:

- strict=True значит, что при обнаружении ошибки, необходимо сгенерировать исключение ValueError (значение по умолчанию)
- strict=False значит, что при обнаружении ошибки, надо только вывести на стандартный поток вывода сообщене об ошибке

Метод дожен возвращать вывод аналогичный методу send_config_set у netmiko (пример вывода ниже). Текст исключения и ошибки в примере ниже.

Пример создания экземпляра класса:

```
In [1]: from task_22_2c import CiscoTelnet

In [2]: r1_params = {
   ...:     'ip': '192.168.100.1',
   ...:     'username': 'cisco',
   ...:     'password': 'cisco',
   ...:     'secret': 'cisco'}

In [3]: r1 = CiscoTelnet(**r1_params)

In [4]: commands_with_errors = ['logging 0255.255.1', 'logging', 'a']
In [5]: correct_commands = ['logging buffered 20010', 'ip http server']
In [6]: commands = commands_with_errors+correct_commands
```

Использование метода send_config_commands:

```
In [7]: print(r1.send_config_commands(commands, strict=False))
При выполнении команды "logging 0255.255.1" на устройстве 192.168.100.1 возникла ошибка ->
↪ Invalid input detected at '^' marker.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
При выполнении команды "logging" на устройстве 192.168.100.1 возникла ошибка ->␣
↪Incomplete command.
При выполнении команды "a" на устройстве 192.168.100.1 возникла ошибка -> Ambiguous␣
↪command:  "a"
conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#logging 0255.255.1
                   ^
% Invalid input detected at '^' marker.

R1(config)#logging
% Incomplete command.

R1(config)#a
% Ambiguous command:  "a"
R1(config)#logging buffered 20010
R1(config)#ip http server
R1(config)#end
R1#

In [8]: print(r1.send_config_commands(commands, strict=True))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-0abc1ed8602e> in <module>
----> 1 print(r1.send_config_commands(commands, strict=True))

...

ValueError: При выполнении команды "logging 0255.255.1" на устройстве 192.168.100.1␣
↪возникла ошибка -> Invalid input detected at '^' marker.
```

23. Специальные методы

Специальные методы в Python - это методы, которые отвечают за «стандартные» возможности объектов и вызываются автоматически при использовани этих возможностей. Например, выражение `a + b`, где a и b это числа, преобразуется в такой вызов `a.__add__(b)`, то есть, специальный метод __add__ отвечает за операцию сложения. Все специальные методы начинаются и заканчиваются двойным подчеркиванием, поэтому на английском их часто называют dunder методы, сокращенно от «double underscore».

Примечание: Специальные методы часто называют волшебными (magic) методами.

Специальные методы отвечают за такие возможности как работа в менеджерах контекста, создание итераторов и итерируемых объектов, операции сложения, умножения и другие. Добавляя специальные методы в объекты, которые созданы пользователем, мы делаем их похожими на встроенные объекты.

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.

Одно подчеркивание перед именем

Одно подчеркивание перед именем метода указывает, что метод является внутренней особенностью реализации и его не стоит использовать напрямую.

Например, класс CiscoSSH использует paramiko для подключения к оборудованию:

```
import time
import paramiko


class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        self.client = paramiko.SSHClient()
        self.client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        self.client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            allow_agent=False)

        self.ssh = self.client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

После создания экземпляра класса, доступен не только метод send_show_command, но и атрибуты client и ssh (3 строка это подсказки по tab в ipython):

```
In [2]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [3]: r1.
            client
            send_show_command()
            ssh
```

Если же необходимо указать, что client и ssh являются внутренними атрибутами, которые нужны для работы класса, но не предназначены для пользователя, надо поставить нижнее подчеркивание перед именем:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        self._client = paramiko.SSHClient()
        self._client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        self._client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = self._client.invoke_shell()
        self._ssh.send('enable\n')
        self._ssh.send(enable + '\n')
        if disable_paging:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(1000)

    def send_show_command(self, command):
        self._ssh.send(command + '\n')
        time.sleep(2)
        result = self._ssh.recv(5000).decode('ascii')
        return result
```

Примечание: Часто такие методы и атрибуты называются приватными, но это не значит, что методы и переменные недоступны пользователю.

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
    ...:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [17]: dir(CiscoSwitch)
Out[17]:
['_CiscoSwitch__configure', '_CiscoSwitch__quantity', '_Switch__configure', '_Switch__
↪quantity', ...]
```

Два подчеркивания перед и после имени

Таким образом обозначаются специальные переменные и методы.

Например, в модуле Python есть такие специальные переменные:

- `__name__` - эта переменная равна строке `__main__`, когда скрипт запускается напрямую, и равна имени модуля, когда импортируется
- `__file__` - эта переменная равна имени скрипта, который был запущен напрямую, и 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
/home/vagrant/repos/tests/example2.py
```

Кроме того, таким образом в Python обозначаются специальные методы. Эти методы вызываются при использовании функций и операторов Python и позволяют реализовать определенный функционал.

Как правило, такие методы не нужно вызывать напрямую. Но, например, при создании своего класса может понадобиться описать такой метод, чтобы объект поддерживал какие-то операции в Python.

Например, для того, чтобы можно было получить длину объекта, он должен поддерживать метод `__len__`.

Методы __str__, __repr__

Специальные методы __str__ и __repr__ отвечают за строковое представления объекта. При этом используются они в разных местах.

Рассмотрим пример класса IPAddress, который отвечает за представление IPv4 адреса:

```
In [1]: class IPAddress:
   ...:     def __init__(self, ip):
   ...:         self.ip = ip
   ...:
```

После создания экземпляров класса, у них есть строковое представление по умолчанию, которое выглядит так (этот же вывод отображается при использовании print):

```
In [2]: ip1 = IPAddress('10.1.1.1')

In [3]: ip2 = IPAddress('10.2.2.2')

In [4]: str(ip1)
Out[4]: '<__main__.IPAddress object at 0xb4e4e76c>'

In [5]: str(ip2)
Out[5]: '<__main__.IPAddress object at 0xb1bd376c>'
```

К сожалению, это представление не очень информативно. И было бы лучше, если бы отображалась информация о том, какой именно адрес представляет этот экземпляр. За отображение информации при применении функции str, отвечает специальный метод __str__ - как аргумент метод ожидает только экземпляр и должен возвращать строку

```
In [6]: class IPAddress:
   ...:     def __init__(self, ip):
   ...:         self.ip = ip
   ...:
   ...:     def __str__(self):
   ...:         return f"IPAddress: {self.ip}"
   ...:

In [7]: ip1 = IPAddress('10.1.1.1')

In [8]: ip2 = IPAddress('10.2.2.2')

In [9]: str(ip1)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[9]: 'IPAddress: 10.1.1.1'

In [10]: str(ip2)
Out[10]: 'IPAddress: 10.2.2.2'
```

Второе строковое представление, которое используется в объектах Python, отображается при использовании функции repr, а также при добавлении объектов в контейнеры типа списков:

```
In [11]: ip_addresses = [ip1, ip2]

In [12]: ip_addresses
Out[12]: [<__main__.IPAddress at 0xb4e40c8c>, <__main__.IPAddress at 0xb1bc46ac>]

In [13]: repr(ip1)
Out[13]: '<__main__.IPAddress object at 0xb4e40c8c>'
```

За это отображение отвечает метод __repr__, он тоже должен возвращать строку, но при этом принято, чтобы метод возвращал строку, скопировав которую, можно получить экземпляр класса:

```
In [14]: class IPAddress:
    ...:     def __init__(self, ip):
    ...:         self.ip = ip
    ...:
    ...:     def __str__(self):
    ...:         return f"IPAddress: {self.ip}"
    ...:
    ...:     def __repr__(self):
    ...:         return f"IPAddress('{self.ip}')"
    ...:

In [15]: ip1 = IPAddress('10.1.1.1')

In [16]: ip2 = IPAddress('10.2.2.2')

In [17]: ip_addresses = [ip1, ip2]

In [18]: ip_addresses
Out[18]: [IPAddress('10.1.1.1'), IPAddress('10.2.2.2')]

In [19]: repr(ip1)
Out[19]: "IPAddress('10.1.1.1')"
```

Поддержка арифметических операторов

За поддержку арифметических операций также отвечают специальные методы, например, за операцию сложения отвечает метод __add__:

```
__add__(self, other)
```

Добавим к классу IPAddress поддержку суммирования с числами, но чтобы не усложнять реализацию метода, воспользуемся возможностями модуля ipaddress

```
In [1]: import ipaddress

In [2]: ipaddress1 = ipaddress.ip_address('10.1.1.1')

In [3]: int(ipaddress1)
Out[3]: 167837953

In [4]: ipaddress.ip_address(167837953)
Out[4]: IPv4Address('10.1.1.1')
```

Класс IPAddress с методом __add__:

```
In [5]: class IPAddress:
   ...:     def __init__(self, ip):
   ...:         self.ip = ip
   ...:
   ...:     def __str__(self):
   ...:         return f"IPAddress: {self.ip}"
   ...:
   ...:     def __repr__(self):
   ...:         return f"IPAddress('{self.ip}')"
   ...:
   ...:     def __add__(self, other):
   ...:         ip_int = int(ipaddress.ip_address(self.ip))
   ...:         sum_ip_str = str(ipaddress.ip_address(ip_int + other))
   ...:         return IPAddress(sum_ip_str)
   ...:
```

Переменная ip_int ссылается на значение исходного адреса в десятичном формате. а sum_ip_str это строка с IP-адресом полученным в результате сложения двух чисел. Как правило, желательно чтобы операция суммирования возвращала экземпляр того же класса, поэтому в последней строке метода создается экземпляр класса IPAddress и ему как аргумент передается строка с итоговым адресом.

Теперь экземпляры класса IPAddress должны поддерживать операцию сложения с числом. В результате мы получаем новый экземпляр класса IPAddress.

```
In [6]: ip1 = IPAddress('10.1.1.1')

In [7]: ip1 + 5
Out[7]: IPAddress('10.1.1.6')
```

Так как внутри метода используется модуль ipaddress, а он поддерживает создание IP-адреса только из десятичного числа, надо ограничить метод на работу только с данными типа int. Если же второй элемент был объектом другого типа, надо сгенерировать исключение. Исключение и сообщение об ошибке возьмем из аналогичной ошибки функции ipaddress.ip_address:

```
In [8]: a1 = ipaddress.ip_address('10.1.1.1')

In [9]: a1 + 4
Out[9]: IPv4Address('10.1.1.5')

In [10]: a1 + 4.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-a0a045adedc5> in <module>
----> 1 a1 + 4.0

TypeError: unsupported operand type(s) for +: 'IPv4Address' and 'float'
```

Теперь класс IPAddress выглядит так:

```
In [11]: class IPAddress:
    ...:     def __init__(self, ip):
    ...:         self.ip = ip
    ...:
    ...:     def __str__(self):
    ...:         return f"IPAddress: {self.ip}"
    ...:
    ...:     def __repr__(self):
    ...:         return f"IPAddress('{self.ip}')"
    ...:
    ...:     def __add__(self, other):
    ...:         if not isinstance(other, int):
    ...:             raise TypeError(f"unsupported operand type(s) for +:"
    ...:                             f" 'IPAddress' and '{type(other).__name__}'")
    ...:
    ...:         ip_int = int(ipaddress.ip_address(self.ip))
    ...:         sum_ip_str = str(ipaddress.ip_address(ip_int + other))
    ...:         return IPAddress(sum_ip_str)
    ...:
```

Если второй операнд не является экзепляром класса int, генерируется исключение TypeError. В исключении выводится информация, что суммирование не поддерживается между экзем-

плярами класса IPAddress и экземпляром класса операнда. Имя класса получено из самого класса, после обращения к type: `type(other).__name__`.

Проверка суммирования с десятичным числом и генерации ошибки:

```
In [12]: ip1 = IPAddress('10.1.1.1')

In [13]: ip1 + 5
Out[13]: IPAddress('10.1.1.6')

In [14]: ip1 + 5.0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-5e619f8dc37a> in <module>
----> 1 ip1 + 5.0

<ipython-input-11-77b43bc64757> in __add__(self, other)
     11     def __add__(self, other):
     12         if not isinstance(other, int):
---> 13             raise TypeError(f"unsupported operand type(s) for +:"
     14                             f" 'IPAddress' and '{type(other).__name__}'")
     15

TypeError: unsupported operand type(s) for +: 'IPAddress' and 'float'

In [15]: ip1 + '1'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-c5ce818f55d8> in <module>
----> 1 ip1 + '1'

<ipython-input-11-77b43bc64757> in __add__(self, other)
     11     def __add__(self, other):
     12         if not isinstance(other, int):
---> 13             raise TypeError(f"unsupported operand type(s) for +:"
     14                             f" 'IPAddress' and '{type(other).__name__}'")
     15

TypeError: unsupported operand type(s) for +: 'IPAddress' and 'str'
```

См.также:

Руководство по специальным методам (англ) Numeric magic methods

Протоколы

Специальные методы отвечают не только за поддержку операций типа сложение, сравнение, но и за поддержку протоколов. Протокол - это набор методов, которые должны быть реализованы в объекте, чтобы он поддерживал определенное поведение. Например, в Python есть такие протоколы: итерации, менеджер контекста, контейнеры и другие. После создания в объекте определенных методов, объект будет вести себя как встроенный и использовать интерфейс понятный всем, кто пишет на Python.

Примечание: Таблица с абстрактных классов в которой описаны какие методы должны присутствовать у объекта, чтобы он поддерживал определенный протокол

Протокол итерации

Итерируемый объект (iterable) - это объект, который способен возвращать элементы по одному. Для Python это любой объект у которого есть метод `__iter__` или метод `__getitem__`. Если у объекта есть метод __iter__, итерируемый объект превращается в итератор вызовом `iter(name)`, где name - имя итерируемого объекта. Если метода __iter__ нет, Python перебирает элементы используя __getitem__.

```
class Items:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]


In [2]: iterable_1 = Items([1, 2, 3, 4])

In [3]: iterable_1[0]
Вызываю __getitem__
Out[3]: 1

In [4]: for i in iterable_1:
   ...:     print('>>>>', i)
   ...:
Вызываю __getitem__
>>>> 1
Вызываю __getitem__
>>>> 2
Вызываю __getitem__
```

(continues on next page)

(продолжение с предыдущей страницы)

```
>>>> 3
Вызываю __getitem__
>>>> 4
Вызываю __getitem__

In [5]: list(map(str, iterable_1))
Вызываю __getitem__
Вызываю __getitem__
Вызываю __getitem__
Вызываю __getitem__
Вызываю __getitem__
Out[5]: ['1', '2', '3', '4']
```

Если у объекта есть метод __iter__ (который обязан возвращать итератор), при переборе значений используется он:

```
class Items:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]

    def __iter__(self):
        print('Вызываю __iter__')
        return iter(self.items)


In [12]: iterable_1 = Items([1, 2, 3, 4])

In [13]: for i in iterable_1:
     ...:     print('>>>>', i)
     ...:
Вызываю __iter__
>>>> 1
>>>> 2
>>>> 3
>>>> 4

In [14]: list(map(str, iterable_1))
Вызываю __iter__
Out[14]: ['1', '2', '3', '4']
```

В Python за получение итератора отвечает функция iter():

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

In [2]: iter(lista)
Out[2]: <list_iterator at 0xb4ede28c>
```

Функция iter отработает на любом объекте, у которого есть метод __iter__ или метод __getitem__. Метод __iter__ возвращает итератор. Если этого метода нет, функция iter() проверяет, нет ли метода __getitem__ - метода, который позволяет получать элементы по индексу. Если метод __getitem__ есть, элементы будут перебираться по индексу (начиная с 0).

Итератор (iterator) - это объект, который возвращает свои элементы по одному за раз. С точки зрения Python - это любой объект, у которого есть метод __next__. Этот метод возвращает следующий элемент, если он есть, или возвращает исключение StopIteration, когда элементы закончились. Кроме того, итератор запоминает, на каком объекте он остановился в последнюю итерацию. Также у каждого итератора присутствует метод __iter__ - то есть, любой итератор является итерируемым объектом. Этот метод возвращает сам итератор.

Пример создания итератора из списка:

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

In [4]: i = iter(lista)
```

Теперь можно использовать функцию next(), которая вызывает метод __next__, чтобы взять следующий элемент:

```
In [5]: next(i)
Out[5]: 1

In [6]: next(i)
Out[6]: 2

In [7]: next(i)
Out[7]: 3

In [8]: next(i)
---------------------------------------------------------------
StopIteration              Traceback (most recent call last)
<ipython-input-8-bed2471d02c1> in <module>()
----> 1 next(i)

StopIteration:
```

После того, как элементы закончились, возвращается исключение StopIteration. Для того, чтобы итератор снова начал возвращать элементы, его надо заново создать. Аналогичные действия выполяются, когда цикл for проходится по списку:

```
In [9]: for item in lista:
   ...:     print(item)
   ...:
1
2
3
```

Когда мы перебираем элементы списка, к списку сначала применяется функция iter(), чтобы создать итератор, а затем вызывается его метод __next__ до тех пор, пока не возникнет исключение StopIteration.

Пример функции my_for, которая работает с любым итерируемым объектом и имитирует работу встроенной функции for:

```
def my_for(iterable):
    if getattr(iterable, "__iter__", None):
        print('Есть __iter__')
        iterator = iter(iterable)
        while True:
            try:
                print(next(iterator))
            except StopIteration:
                break
    elif getattr(iterable, "__getitem__", None):
        print('Нет __iter__, но есть __getitem__')
        index = 0
        while True:
            try:
                print(iterable[index])
                index += 1
            except IndexError:
                break
```

Проверка работы функции на объекте у которого есть метод __iter__:

```
In [18]: my_for([1, 2, 3, 4])
Есть __iter__
1
2
3
4
```

Проверка работы функции на объекте у которого нет метода __iter__, но есть __getitem__:

```
class Items:
    def __init__(self, items):
        self.items = items
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def __getitem__(self, index):
        print('Вызываю __getitem__')
        return self.items[index]


In [20]: iterable_1 = Items([1,2,3,4,5])

In [21]: my_for(iterable_1)
Нет __iter__, но есть __getitem__
Вызываю __getitem__
1
Вызываю __getitem__
2
Вызываю __getitem__
3
Вызываю __getitem__
4
Вызываю __getitem__
5
Вызываю __getitem__
```

Создание итератора

Пример класса Network:

```
In [10]: import ipaddress
    ...:
    ...: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
```

Пример создания экземпляра класса Network:

```
In [14]: net1 = Network('10.1.1.192/30')

In [15]: net1
Out[15]: <__main__.Network at 0xb3124a6c>

In [16]: net1.addresses
Out[16]: ['10.1.1.193', '10.1.1.194']
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [17]: net1.network
Out[17]: '10.1.1.192/30'
```

Создаем итератор из класса Network:

```
In [12]: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
    ...:         self._index = 0
    ...:
    ...:     def __iter__(self):
    ...:         print('Вызываю __iter__')
    ...:         return self
    ...:
    ...:     def __next__(self):
    ...:         print('Вызываю __next__')
    ...:         if self._index < len(self.addresses):
    ...:             current_address = self.addresses[self._index]
    ...:             self._index += 1
    ...:             return current_address
    ...:         else:
    ...:             raise StopIteration
    ...:
```

Метод __iter__ в итераторе должен возвращать сам объект, поэтому в методе указано `return self`, а метод __next__ возвращает элементы по одному и генерирует исключение StopIteration, когда элементы закончились.

```
In [14]: net1 = Network('10.1.1.192/30')

In [15]: for ip in net1:
    ...:     print(ip)
    ...:
Вызываю __iter__
Вызываю __next__
10.1.1.193
Вызываю __next__
10.1.1.194
Вызываю __next__
```

Чаще всего, итератор это одноразовый объект и перебрав элементы, мы уже не можем это сделать второй раз:

```
In [16]: for ip in net1:
    ...:     print(ip)
    ...:
Вызываю __iter__
Вызываю __next__
```

Создание итерируемого объекта

Очень часто классу достаточно быть итерируемым объектом и не обязательно быть итератором. Если объект будет итерируемым, его можно использовать в цикле for, функциях map, filter, sorted, enumerate и других. Также, как правило, объект проще сделать итерируемым, чем итератором.

Для того чтобы класс Network создавал итерируемые объекты, надо чтобы в классе был метод __iter__ (__next__ не нужен) и чтобы метод возвращал итератор. Так как в данном случае, Network перебирает адреса, которые находятся в списке self.addresses, самый просто вариант возвращать итератор, это вернуть `iter(self.addresses)`:

```
In [17]: class Network:
    ...:     def __init__(self, network):
    ...:         self.network = network
    ...:         subnet = ipaddress.ip_network(self.network)
    ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
    ...:
    ...:     def __iter__(self):
    ...:         return iter(self.addresses)
    ...:
```

Теперь все экземпляры класса Network будут итерируемыми объектами:

```
In [18]: net1 = Network('10.1.1.192/30')

In [19]: for ip in net1:
    ...:     print(ip)
    ...:
10.1.1.193
10.1.1.194
```

Протокол последовательности

В самом базовом варианте, протокол последовательности (sequence) включает два метода: __len__ и __getitem__. В более полном варианте также методы: __contains__, __iter__, __reversed__, index и count. Если последовательность изменяема, добавляются еще несколько методов.

Добавим методы __len__ и __getitem__ к классу Network:

```
In [1]: class Network:
   ...:     def __init__(self, network):
   ...:         self.network = network
   ...:         subnet = ipaddress.ip_network(self.network)
   ...:         self.addresses = [str(ip) for ip in subnet.hosts()]
   ...:
   ...:     def __iter__(self):
   ...:         return iter(self.addresses)
   ...:
   ...:     def __len__(self):
   ...:         return len(self.addresses)
   ...:
   ...:     def __getitem__(self, index):
   ...:         return self.addresses[index]
   ...:
```

Метод __len__ вызывается функцией len:

```
In [2]: net1 = Network('10.1.1.192/30')

In [3]: len(net1)
Out[3]: 2
```

А метод __getitem__ при обращении по индексу таким образом:

```
In [4]: net1[0]
Out[4]: '10.1.1.193'

In [5]: net1[1]
Out[5]: '10.1.1.194'

In [6]: net1[-1]
Out[6]: '10.1.1.194'
```

Метод __getitem__ отвечает не только обращение по индексу, но и за срезы:

```
In [7]: net1 = Network('10.1.1.192/28')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [8]: net1[0]
Out[8]: '10.1.1.193'

In [9]: net1[3:7]
Out[9]: ['10.1.1.196', '10.1.1.197', '10.1.1.198', '10.1.1.199']

In [10]: net1[3:]
Out[10]:
['10.1.1.196',
 '10.1.1.197',
 '10.1.1.198',
 '10.1.1.199',
 '10.1.1.200',
 '10.1.1.201',
 '10.1.1.202',
 '10.1.1.203',
 '10.1.1.204',
 '10.1.1.205',
 '10.1.1.206']
```

Так как в данном случае, внутри метода __getitem__ используется список, ошибки отрабатывают корректно автоматически:

```
In [11]: net1[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-09ca84e34cb6> in <module>
----> 1 net1[100]

<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
---> 14         return self.addresses[index]
     15

IndexError: list index out of range

In [12]: net1['a']
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-facd90673864> in <module>
----> 1 net1['a']

<ipython-input-2-bc213b4a03ca> in __getitem__(self, index)
     12
     13     def __getitem__(self, index):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---> 14         return self.addresses[index]
     15

TypeError: list indices must be integers or slices, not str
```

Реализация остальных методов протокола последовательности вынесена в задания раздела:

- __contains__ - этот метод отвечает за проверку наличия элемента в последовательности `'10.1.1.198' in net1`. Если в объекте не определен этот метод, наличие элемента проверяется перебором элементов с помощью __iter__, а если и его нет перевором индексов с __getitem__.
- __reversed__ - используется встроенной функцией reversed. Этот метод как правило, лучше не создавать и полагаться на то, что функция reversed при отсутствии метода __reversed__ будет использовать методы __len__ и __getitem__.
- index - возвращает индекс первого элемента, значение которого равно указаному. Работает полностью аналогично методу index в списках и кортежах.
- count - возвращает количество значений. Работает полностью аналогично методу count в списках и кортежах.

Менеджер контекста

Менеджер контекста позволяет выполнять указанные действия в начале и в конце блока with. За работу менеджера контекста отвечают два метода:

- `__enter__(self)` - указывает, что надо сделать в начале блока with. Значение, которое возвращает метод, присваивается переменной после as.
- `__exit__(self, exc_type, exc_value, traceback)` - указывает, что надо сделать в конце блока with или при его прерывании. Если внутри блока возникло исключение, exc_type, exc_value, traceback будут содержать информацию об исключении, если исключения не было, они будут равны None.

Примеры использования менеджера контекста:

- открытие/закрытие файла
- открытие/закрытие сессии SSH/Telnet
- работа с транзакциями в БД

Класс CiscoSSH использует paramiko для подключения к оборудованию:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        client = paramiko.SSHClient()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self.ssh = client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

Пример использования класса:

```
In [9]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [10]: r1.send_show_command('sh clock')
Out[10]: 'sh clock\r\n*12:58:47.523 UTC Sun Jul 28 2019\r\nR1#'

In [11]: r1.send_show_command('sh ip int br')
Out[11]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status
↪            Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up
↪            up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up
↪            up      \r\nEthernet0/2                19.1.1.1        YES NVRAM  up
↪            up      \r\nEthernet0/3                192.168.230.1   YES NVRAM  up
↪            up      \r\nLoopback0                  4.4.4.4         YES NVRAM  up
↪            up      \r\nLoopback90                 90.1.1.1        YES manual up
↪            up      \r\nR1#'
```

Для того чтобы класс поддерживал работу в менеджере контекста, надо добавить методы __enter__ и __exit__:

```
class CiscoSSH:
    def __init__(self, ip, username, password, enable, disable_paging=True):
        print('Метод __init__')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self.ssh = client.invoke_shell()
        self.ssh.send('enable\n')
        self.ssh.send(enable + '\n')
        if disable_paging:
            self.ssh.send('terminal length 0\n')
        time.sleep(1)
        self.ssh.recv(1000)

    def __enter__(self):
        print('Метод __enter__')
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print('Метод __exit__')
        self.ssh.close()

    def send_show_command(self, command):
        self.ssh.send(command + '\n')
        time.sleep(2)
        result = self.ssh.recv(5000).decode('ascii')
        return result
```

Пример использования класса в менеджере контекста:

```
In [14]: with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:
Метод __init__
Метод __enter__
sh clock
*13:05:50.677 UTC Sun Jul 28 2019
R1#
Метод __exit__
```

Даже если внутри блока возникнет исключение, метод __exit__ выполняется:

```
In [18]: with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
    ...:     result = r1.send_show_command('sh clock')
    ...:     result / 2
    ...:
Метод __init__
Метод __enter__
Метод __exit__
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-b9ff1fa74be2> in <module>
      1 with CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco') as r1:
      2     result = r1.send_show_command('sh clock')
----> 3     result / 2
      4

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Дополнительные материалы

- Special method names в документации Python
- A Guide to Python's Magic Methods
- **`Раздел "Объектно-ориентированное программирование" из книги "A Byte of Python"<https://wombat.org.ua/AByteOfPython/object_oriented_programming.html>`__**
- Dive Into Python 3

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

Предупреждение: Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 23.1

В этом задании необходимо создать класс IPAddress.

При создании экземпляра класса, как аргумент передается IP-адрес и маска, а также должна выполняться проверка корректности адреса и маски:

Адрес считается корректно заданным, если он:

- состоит из 4 чисел разделенных точкой
- каждое число в диапазоне от 0 до 255

Маска считается корректной, если это число в диапазоне от 8 до 32 включительно

Если маска или адрес не прошли проверку, необходимо сгенерировать исключение ValueError с соответствующим текстом (вывод ниже).

Также, при создании класса, должны быть созданы две переменных экземпляра: ip и mask, в которых содержатся адрес и маска, соответственно.

Пример создания экземпляра класса:

```
In [1]: ip = IPAddress('10.1.1.1/24')

Атрибуты ip и mask
In [2]: ip1 = IPAddress('10.1.1.1/24')

In [3]: ip1.ip
Out[3]: '10.1.1.1'

In [4]: ip1.mask
Out[4]: 24
```

Проверка корректности адреса (traceback сокращен)

```
In [5]: ip1 = IPAddress('10.1.1/24')
---------------------------------------------------------------------------
...
ValueError: Incorrect IPv4 address
```

Проверка корректности маски (traceback сокращен)

```
In [6]: ip1 = IPAddress('10.1.1.1/240')
---------------------------------------------------------------------------
...
ValueError: Incorrect mask
```

Задание 23.1a

Скопировать и изменить класс IPAddress из задания 23.1.

Добавить два строковых представления для экземпляров класса IPAddress. Как дожны выглядеть строковые представления, надо определить из вывода ниже:

Создание экземпляра

```
In [5]: ip1 = IPAddress('10.1.1.1/24')

In [6]: str(ip1)
Out[6]: 'IP address 10.1.1.1/24'

In [7]: print(ip1)
IP address 10.1.1.1/24

In [8]: ip1
Out[8]: IPAddress('10.1.1.1/24')

In [9]: ip_list = []

In [10]: ip_list.append(ip1)

In [11]: ip_list
Out[11]: [IPAddress('10.1.1.1/24')]

In [12]: print(ip_list)
[IPAddress('10.1.1.1/24')]
```

Задание 23.2

Скопировать класс CiscoTelnet из любого задания 22.2x и добавить классу поддержку работы в менеджере контекста. При выходе из блока менеджера контекста должно закрываться соединение.

Пример работы:

```
In [14]: r1_params = {
    ...:     'ip': '192.168.100.1',
    ...:     'username': 'cisco',
    ...:     'password': 'cisco',
    ...:     'secret': 'cisco'}

In [15]: from task_23_2 import CiscoTelnet

In [16]: with CiscoTelnet(**r1_params) as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:
sh clock
*19:17:20.244 UTC Sat Apr 6 2019
R1#

In [17]: with CiscoTelnet(**r1_params) as r1:
    ...:     print(r1.send_show_command('sh clock'))
    ...:     raise ValueError('Возникла ошибка')
    ...:
sh clock
*19:17:38.828 UTC Sat Apr 6 2019
R1#
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-17-f3141be7c129> in <module>
      1 with CiscoTelnet(**r1_params) as r1:
      2     print(r1.send_show_command('sh clock'))
----> 3     raise ValueError('Возникла ошибка')
      4

ValueError: Возникла ошибка
```

Задание 23.3

Скопировать и изменить класс Topology из задания 22.1x.

Добавить метод, который позволит выполнять сложение двух экземпляров класса Topology. В результате сложения должен возвращаться новый экземпляр класса Topology.

Создание двух топологий:

```
In [1]: t1 = Topology(topology_example)

In [2]: t1.topology
Out[2]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [3]: topology_example2 = {('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
                             ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}

In [4]: t2 = Topology(topology_example2)

In [5]: t2.topology
Out[5]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Суммирование топологий:

```
In [6]: t3 = t1+t2

In [7]: t3.topology
Out[7]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
 ('R1', 'Eth0/4'): ('R7', 'Eth0/0'),
 ('R1', 'Eth0/6'): ('R9', 'Eth0/0'),
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}
```

Проверка, что исходные топологии не изменились

```
In [9]: t1.topology
Out[9]:
{('R1', 'Eth0/0'): ('SW1', 'Eth0/1'),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
 ('R2', 'Eth0/0'): ('SW1', 'Eth0/2'),
 ('R2', 'Eth0/1'): ('SW2', 'Eth0/11'),
 ('R3', 'Eth0/0'): ('SW1', 'Eth0/3'),
 ('R3', 'Eth0/1'): ('R4', 'Eth0/0'),
 ('R3', 'Eth0/2'): ('R5', 'Eth0/0')}

In [10]: t2.topology
Out[10]: {('R1', 'Eth0/4'): ('R7', 'Eth0/0'), ('R1', 'Eth0/6'): ('R9', 'Eth0/0')}
```

Задание 23.3a

В этом задании надо сделать так, чтобы экземпляры класса Topology были итерируемыми объектами. Основу класса Topology можно взять из любого задания 22.1x или задания 23.3.

После создания экземпляра класса, экземпляр должен работать как итерируемый объект. На каждой итерации должен возвращаться кортеж, который описывает одно соединение.

Пример работы класса:

```
In [1]: top = Topology(topology_example)

In [2]: for link in top:
   ...:     print(link)
   ...:
(('R1', 'Eth0/0'), ('SW1', 'Eth0/1'))
(('R2', 'Eth0/0'), ('SW1', 'Eth0/2'))
(('R2', 'Eth0/1'), ('SW2', 'Eth0/11'))
(('R3', 'Eth0/0'), ('SW1', 'Eth0/3'))
(('R3', 'Eth0/1'), ('R4', 'Eth0/0'))
(('R3', 'Eth0/2'), ('R5', 'Eth0/0'))
```

Проверить работу класса.

24. Наследование

Основы наследования

Наследование позволяет создавать новые классы на основе существующих. Различают дочерний и родительские классы: дочерний класс наследует родительский. При наследовании, дочерний класс наследует все методы и атрибуты родительского класса.

Пример класса ConnectSSH, который выполняет подключение по SSH с помощью paramiko:

```
import paramiko
import time


class ConnectSSH:
    def __init__(self, ip, username, password):
        self.ip = ip
        self.username = username
        self.password = password
        self._MAX_READ = 10000

        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        client.connect(
            hostname=ip,
            username=username,
            password=password,
            look_for_keys=False,
            allow_agent=False)

        self._ssh = client.invoke_shell()
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self._ssh.close()

    def close(self):
        self._ssh.close()

    def send_show_command(self, command):
        self._ssh.send(command + '\n')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        time.sleep(2)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def send_config_commands(self, commands):
        if isinstance(commands, str):
            commands = [commands]
        for command in commands:
            self._ssh.send(command + '\n')
            time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result
```

Этот класс будет использоваться как основа для классов, которые отвечают за подключение к устройствам разных вендоров. Например, класс CiscoSSH будет отвечать за подключение к устройствам Cisco будет наследовать класс ConnectSSH.

Синтаксис наследования:

```
class CiscoSSH(ConnectSSH):
    pass
```

После этого в классе CiscoSSH доступны все методы и атрибуты класса ConnectSSH:

```
In [3]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco')

In [4]: r1.ip
Out[4]: '192.168.100.1'

In [5]: r1._MAX_READ
Out[5]: 10000

In [6]: r1.send_show_command('sh ip int br')
Out[6]: 'sh ip int br\r\nInterface                  IP-Address      OK? Method Status    ␣
↪           Protocol\r\nEthernet0/0                192.168.100.1   YES NVRAM  up        ␣
↪           up      \r\nEthernet0/1                192.168.200.1   YES NVRAM  up        ␣
↪           up      \r\nEthernet0/2                19.1.1.1        YES NVRAM  up        ␣
↪           up      \r\nEthernet0/3                192.168.230.1   YES NVRAM  up        ␣
↪           up      \r\nLoopback0                  4.4.4.4         YES NVRAM  up        ␣
↪           up      \r\nLoopback33                 3.3.3.3         YES manual up        ␣
↪           up      \r\nLoopback90                 90.1.1.1        YES manual up        ␣
↪           up      \r\nR1#'



In [7]: r1.send_show_command('enable')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Out[7]: 'enable\r\nPassword: '

In [8]: r1.send_show_command('cisco')
Out[8]: '\r\nR1#'

In [9]: r1.send_config_commands(['conf t', 'int loopback 33',
   ...:                          'ip address 3.3.3.3 255.255.255.255', 'end'])
Out[9]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
↪nR1(config)#int loopback 33\r\nR1(config-if)#ip address 3.3.3.3 255.255.255.255\r\
↪nR1(config-if)#end\r\nR1#'
```

После наследования всех методов родительского класса, дочерний класс может:

- оставить их без изменения
- полностью переписать их
- дополнить метод
- добавить свои методы

В классе CiscoSSH надо создать метод __init__ и добавить к нему параметры:

- enable_password - пароль enable
- disable_paging - отвечает за включение/отключение постраничного вывода команд

Метод __init__ можно создать полностью с нуля, однако базовая логика подключения по SSH будет одинаковая в ConnectSSH и CiscoSSH, поэтому лучше добавить необходимые параметры, а для подключения, вызвать метод __init__ у класса ConnectSSH. Есть несколько вариантов вызова родительского метода, например, все эти варианты вызовут метод send_show_command родительского класса из дочернего класса CiscoSSH:

```
command_result = ConnectSSH.send_show_command(self, command)
command_result = super(CiscoSSH, self).send_show_command(command)
command_result = super().send_show_command(command)
```

Первый вариант `ConnectSSH.send_show_command` явно указывает имя родительского класса - это самый понятный вариант для восприятия, однако его минус в том, что при смене имени родительского класса, имя надо будет менять во всех местах, где вызывались методы родительского класса. Также у этого варианта есть минусы, при использовании множественного наследования. Второй и третий вариант по сути равнозначны, но третий короче, поэтому мы будем использовать его.

Класс CiscoSSH с методом __init__:

```
class CiscoSSH(ConnectSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)
```

Метод __init__ в классе CiscoSSH добавил параметры enable_password и disable_paging, и использует их соответственно для перехода в режим enable и отключения постраничного вывода. Пример подключения:

```
In [10]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [11]: r1.send_show_command('sh clock')
Out[11]: 'sh clock\r\n*11:30:50.280 UTC Mon Aug 5 2019\r\nR1#'
```

Теперь при подключении также выполняется переход в режим enable и по умолчанию отключен paging, так что можно попробовать выполнить длинную команду, например sh run.

Еще один метод, который стоит доработать - метод send_config_commands: так как класс CiscoSSH предназначен для работы с Cisco, можно добавить в него переход в конфигурационный режим перед командами и выход после.

```
class CiscoSSH(ConnectSSH):
    def __init__(self, ip, username, password, enable_password,
                 disable_paging=True):
        super().__init__(ip, username, password)
        self._ssh.send('enable\n')
        self._ssh.send(enable_password + '\n')
        if disable_paging:
            self._ssh.send('terminal length 0\n')
        time.sleep(1)
        self._ssh.recv(self._MAX_READ)

    def config_mode(self):
        self._ssh.send('conf t\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result

    def exit_config_mode(self):
        self._ssh.send('end\n')
        time.sleep(0.5)
        result = self._ssh.recv(self._MAX_READ).decode('ascii')
        return result
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def send_config_commands(self, commands):
        result = self.config_mode()
        result += super().send_config_commands(commands)
        result += self.exit_config_mode()
        return result
```

Пример использования метода send_config_commands:

```
In [12]: r1 = CiscoSSH('192.168.100.1', 'cisco', 'cisco', 'cisco')

In [13]: r1.send_config_commands(['interface loopback 33',
    ...:                          'ip address 3.3.3.3 255.255.255.255'])
Out[13]: 'conf t\r\nEnter configuration commands, one per line.  End with CNTL/Z.\r\
↪nR1(config)#interface loopback 33\r\nR1(config-if)#ip address 3.3.3.3 255.255.255.255\r\
↪nR1(config-if)#end\r\nR1#'
```

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

> **Предупреждение:** Начиная с раздела «4. Типы данных в Python» для проверки заданий есть автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами. Проверка заданий выполняется с помощью утилиты pyneng. Подробнее *о том как работать с утилитой pyneng*.

Задание 24.1

Создать класс CiscoSSH, который наследует класс BaseSSH из файла base_connect_class.py.

Создать метод __init__ в классе CiscoSSH таким образом, чтобы после подключения по SSH выполнялся переход в режим enable.

Для этого в методе __init__ должен сначала вызываться метод __init__ класса ConnectSSH, а затем выполняться переход в режим enable.

```
In [2]: from task_24_1 import CiscoSSH

In [3]: r1 = CiscoSSH(**device_params)

In [4]: r1.send_show_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status                ␣
↪Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/1                192.168.200.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/2                190.16.200.1    YES NVRAM  up                    ␣
↪up      \nEthernet0/3                192.168.230.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/3.100            10.100.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.200            10.200.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.300            10.30.0.1       YES NVRAM  up                    ␣
↪up      '
```

Задание 24.1a

Скопировать и дополнить класс CiscoSSH из задания 24.1.

Перед подключением по SSH необходимо проверить если ли в словаре с параметрами подключения такие параметры: username, password, secret. Если какого-то параметра нет, запросить значение у пользователя, а затем выполнять подключение. Если все параметры есть, выполнить подключение.

```
In [1]: from task_24_1a import CiscoSSH

In [2]: device_params = {
   ...:         'device_type': 'cisco_ios',
   ...:         'host': '192.168.100.1',
   ...: }

In [3]: r1 = CiscoSSH(**device_params)
Введите имя пользователя: cisco
Введите пароль: cisco
Введите пароль для режима enable: cisco

In [4]: r1.send_show_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status                ␣
↪Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/1                192.168.200.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/2                190.16.200.1    YES NVRAM  up                    ␣
↪up      \nEthernet0/3                192.168.230.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/3.100            10.100.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.200            10.200.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.300            10.30.0.1       YES NVRAM  up                    ␣
↪up      '
```

Задание 24.2

Создать класс MyNetmiko, который наследует класс CiscoIosSSH из netmiko.

Переписать метод __init__ в классе MyNetmiko таким образом, чтобы после подключения по SSH выполнялся переход в режим enable.

Для этого в методе __init__ должен сначала вызываться метод __init__ класса CiscoIosBase, а затем выполнялся переход в режим enable.

Проверить, что в классе MyNetmiko доступны методы send_command и send_config_set (они наследуются автоматически, это только для проверки).

```
In [2]: from task_24_2 import MyNetmiko
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status                ␣
↪Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/1                192.168.200.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/2                190.16.200.1    YES NVRAM  up                    ␣
↪up      \nEthernet0/3                192.168.230.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/3.100            10.100.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.200            10.200.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.300            10.30.0.1       YES NVRAM  up                    ␣
↪up      '
```

Импорт класса CiscoIosSSH:

```
from netmiko.cisco.cisco_ios import CiscoIosSSH


device_params = {
    "device_type": "cisco_ios",
    "ip": "192.168.100.1",
    "username": "cisco",
    "password": "cisco",
    "secret": "cisco",
}
```

Задание 24.2a

Скопировать и дополнить класс MyNetmiko из задания 24.2.

Добавить метод _check_error_in_command, который выполняет проверку на такие ошибки:

- Invalid input detected
- Incomplete command
- Ambiguous command

Метод ожидает как аргумент команду и вывод команды. Если в выводе не обнаружена ошибка, метод ничего не возвращает. Если в выводе найдена ошибка, метод генерирует исключение ErrorInCommand с сообщением о том какая ошибка была обнаружена, на каком устройстве и в какой команде.

Переписать метод send_command netmiko, добавив в него проверку на ошибки.

```
In [2]: from task_24_2a import MyNetmiko
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_command('sh ip int br')
Out[4]: 'Interface                  IP-Address      OK? Method Status                ␣
↪Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/1                192.168.200.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/2                190.16.200.1    YES NVRAM  up                    ␣
↪up      \nEthernet0/3                192.168.230.1   YES NVRAM  up                    ␣
↪up      \nEthernet0/3.100            10.100.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.200            10.200.0.1      YES NVRAM  up                    ␣
↪up      \nEthernet0/3.300            10.30.0.1       YES NVRAM  up                    ␣
↪up      '

In [5]: r1.send_command('sh ip br')
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-2-1c60b31812fd> in <module>()
----> 1 r1.send_command('sh ip br')
...
ErrorInCommand: При выполнении команды "sh ip br" на устройстве 192.168.100.1 возникла␣
↪ошибка "Invalid input detected at '^' marker."
```

Исключение ErrorInCommand:

```
class ErrorInCommand(Exception):
    """
    Исключение генерируется, если при выполнении команды на оборудовании, возникла ошибка.
    """
```

Задание 24.2b

Скопировать класс MyNetmiko из задания 24.2a.

Дополнить функционал метода send_config_set netmiko и добавить в него проверку на ошибки с помощью метода _check_error_in_command.

Метод send_config_set должен отправлять команды по одной и проверять каждую на ошибки. Если при выполнении команд не обнаружены ошибки, метод send_config_set возвращает вывод команд.

```
In [2]: from task_24_2b import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_config_set('lo')
```

(continues on next page)

(продолжение с предыдущей страницы)

```
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-2-8e491f78b235> in <module>()
----> 1 r1.send_config_set('lo')
...
ErrorInCommand: При выполнении команды "lo" на устройстве 192.168.100.1 возникла ошибка
↪"Incomplete command."
```

Задание 24.2c

Скопировать класс MyNetmiko из задания 24.2b. Проверить, что метод send_command кроме команду, принимает еще и дополнительные аргументы, например, strip_command.

Если возникает ошибка, переделать метод таким образом, чтобы он принимал любые аргументы, которые поддерживает netmiko.

```
In [2]: from task_24_2c import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [4]: r1.send_command('sh ip int br', strip_command=False)
Out[4]: 'sh ip int br\nInterface                  IP-Address      OK? Method Status      ␣
↪         Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up            ␣
↪       up      \nEthernet0/1                192.168.200.1   YES NVRAM  up              ␣
↪     up      \nEthernet0/2                190.16.200.1    YES NVRAM  up                ␣
↪   up      \nEthernet0/3                192.168.230.1   YES NVRAM  up                  ␣
↪ up      \nEthernet0/3.100            10.100.0.1      YES NVRAM  up                   ␣
↪up      \nEthernet0/3.200            10.200.0.1      YES NVRAM  up                  ␣
↪up      \nEthernet0/3.300            10.30.0.1       YES NVRAM  up                  ␣
↪up      '

In [5]: r1.send_command('sh ip int br', strip_command=True)
Out[5]: 'Interface                  IP-Address      OK? Method Status                ␣
↪Protocol\nEthernet0/0                192.168.100.1   YES NVRAM  up                  ␣
↪up      \nEthernet0/1                192.168.200.1   YES NVRAM  up                  ␣
↪up      \nEthernet0/2                190.16.200.1    YES NVRAM  up                  ␣
↪up      \nEthernet0/3                192.168.230.1   YES NVRAM  up                  ␣
↪up      \nEthernet0/3.100            10.100.0.1      YES NVRAM  up                  ␣
↪up      \nEthernet0/3.200            10.200.0.1      YES NVRAM  up                  ␣
↪up      \nEthernet0/3.300            10.30.0.1       YES NVRAM  up                  ␣
↪up      '
```

Задание 24.2d

Скопировать класс MyNetmiko из задания 24.2c или задания 24.2b.

Добавить параметр ignore_errors в метод send_config_set. Если передано истинное значение, не надо выполнять проверку на ошибки и метод должен работать точно так же как метод send_config_set в netmiko. Если значение ложное, ошибки должны проверяться.

По умолчанию ошибки должны игнорироваться.

```
In [2]: from task_24_2d import MyNetmiko

In [3]: r1 = MyNetmiko(**device_params)

In [6]: r1.send_config_set('lo')
Out[6]: 'config term\nEnter configuration commands, one per line.  End with CNTL/Z.\
↪nR1(config)#lo\n% Incomplete command.\n\nR1(config)#end\nR1#'

In [7]: r1.send_config_set('lo', ignore_errors=True)
Out[7]: 'config term\nEnter configuration commands, one per line.  End with CNTL/Z.\
↪nR1(config)#lo\n% Incomplete command.\n\nR1(config)#end\nR1#'

In [8]: r1.send_config_set('lo', ignore_errors=False)
---------------------------------------------------------------------------
ErrorInCommand                            Traceback (most recent call last)
<ipython-input-8-704f2e8d1886> in <module>()
----> 1 r1.send_config_set('lo', ignore_errors=False)


...
ErrorInCommand: При выполнении команды "lo" на устройстве 192.168.100.1 возникла ошибка
↪"Incomplete command."
```

8

VII. Работа с базами данных

25. Работа с базами данных

Использование баз данных - это еще один способ хранения информации. Базы данных полезны не только в хранении информации. Используя СУБД, можно делать срезы информации по различным параметрам.

База данных (БД) - это данные, которые хранятся в соответствии с определенной схемой. В этой схеме каким-то образом описаны соотношения между данными.

Язык БД (лингвистические средства) - используется для описания структуры БД, управления данными (добавление, изменение, удаление, получение), управления правами доступа к БД и ее объектам, управления транзакциями.

Система управления базами данных (СУБД) - это программные средства, которые дают возможность управлять БД. СУБД должны поддерживать соответствующий язык (языки) для управления БД.

SQL

SQL (structured query language) - используется для описания структуры БД, управления данными (добавление, изменение, удаление, получение), управления правами доступа к БД и ее объектам, управления транзакциями.

Язык SQL подразделяется на такие категории:

- DDL (Data Definition Language) - язык описания данных
- DML (Data Manipulation Language) - язык манипулирования данными
- DCL (Data Control Language) - язык определения доступа к данным
- TCL (Transaction Control Language) - язык управления транзакциями

В каждой категории есть свои операторы (перечислены не все операторы):

- DDL
  - CREATE - создание новой таблицы, СУБД, схемы
  - ALTER - изменение существующей таблицы, колонки
  - DROP - удаление существующих объектов из СУБД
- DML
  - SELECT - выбор данных
  - INSERT - добавление новых данных
  - UPDATE - обновление существующих данных
  - DELETE - удаление данных

- DCL
  - GRANT - предоставление пользователям разрешения на чтение/запись определенных объектов в СУБД
  - REVOKE - отзыв ранее предоставленных разрешений
- TCL
  - COMMIT - применение транзакции
  - ROLLBACK - 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. SQLite часто используется как встроенная СУБД в приложениях.

Примечание: Слово SQL-сервер здесь не используем, потому что как таковой сервер там не нужен — весь функционал, который встраивается в SQL-сервер, реализован внутри библиотеки (и, соответственно, внутри программы, которая её использует).

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.

Примечание: Если вы используете Linux или Mac OS, то, скорее всего, sqlite3 установлен. Если вы используете Windows, то можно скачать sqlite3 тут.

Для того, чтобы создать БД (или открыть уже созданную), надо просто вызвать sqlite3 таким образом:

```
$ sqlite3 testDB.db
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite>
```

Внутри sqlite3 можно выполнять команды SQL или так называемые метакоманды (или dot-команды).

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
```

litecli

У стандартного CLI-интерфейса SQLite есть несколько недостатков:

- нет автодополнения команд
- нет подсказок
- не всегда отображается все содержимое столбца

Все эти недостатки исправлены в litecli. Поэтому лучше использовать его.

Установка litecli:

```
$ pip install litecli
```

Открыть базу данных в litecli:

```
$ litecli example.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
example.db>
```

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.

Сначала подключимся к базе данных или создадим ее с помощью litecli:

```
$ litecli new_db.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
new_db.db>
```

Создадим таблицу switch, в которой хранится информация о коммутаторах:

```
new_db.db> create table switch (mac text not NULL primary key, hostname text, model text,␣
↪location text);
Query OK, 0 rows affected
Time: 0.010s
```

В данном примере мы описали таблицу switch: определили, какие поля будут в таблице, и значения какого типа будут в них находиться.

Кроме того, поле mac является первичным ключом. Это автоматически значит, что:

- поле должно быть уникальным
- в нём не может находиться значение NULL (в SQLite это надо задавать явно)

В этом примере это вполне логично, так как MAC-адрес должен быть уникальным.

На данный момент записей в таблице нет, есть только ее определение. Просмотреть определение можно такой командой:

```
new_db.db> .schema switch
+-----------------------------------------------------------------------------------------
↪------+
| sql                                                                                     ␣
↪      |
+-----------------------------------------------------------------------------------------
↪------+
| CREATE TABLE switch (mac text not NULL primary key, hostname text, model text, location␣
↪text) |
+-----------------------------------------------------------------------------------------
↪------+
Time: 0.037s
```

DROP

Оператор DROP удаляет таблицу вместе со схемой и всеми данными.

Удалить таблицу можно так:

```
new_db.db> DROP table switch;
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 0 rows affected
Time: 0.009s
```

INSERT

Оператор INSERT используется для добавления данных в таблицу.

Примечание: Если таблица была удалена на предыдущем шаге, надо ее создать:

```
new_db.db> create table switch (mac text not NULL primary key, hostname text, model text,␣
↪location text);
Query OK, 0 rows affected
Time: 0.010s
```

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 указываются значения для всех полей, добавить запись можно таким образом (порядок полей должен соблюдаться):

```
new_db.db> INSERT into switch values ('0010.A1AA.C1CC', 'sw1', 'Cisco 3750', 'London,␣
↪Green Str');
Query OK, 1 row affected
Time: 0.008s
```

Если нужно указать не все поля или указать их в произвольном порядке, используется такая запись:

```
new_db.db> INSERT into switch (mac, model, location, hostname) values ('0020.A2AA.C2CC',
↪'Cisco 3850', 'London, Green Str', 'sw2');
Query OK, 1 row affected
Time: 0.009s
```

SELECT

Оператор SELECT позволяет запрашивать информацию в таблице.

Например:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+
| mac            | hostname | model      | location           |
+----------------+----------+------------+--------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
+----------------+----------+------------+--------------------+
2 rows in set
Time: 0.033s
```

SELECT * означает, что нужно вывести все поля таблицы. Следом указывается, из какой таблицы запрашиваются данные: `from switch`.

Таким образом можно указывать конкретные столбцы, которые нужно вывести и в каком порядке:

```
new_db.db> SELECT hostname, mac, model from switch;
+----------+----------------+------------+
| hostname | mac            | model      |
+----------+----------------+------------+
| sw1      | 0010.A1AA.C1CC | Cisco 3750 |
| sw2      | 0020.A2AA.C2CC | Cisco 3850 |
+----------+----------------+------------+
2 rows in set
Time: 0.033s
```

WHERE

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:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
+----------------+----------+------------+-------------------+
2 rows in set
Time: 0.033s
```

Чтобы в таблице было больше записей, надо создать еще несколько строк. В litecli есть команда source, 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:

```
new_db.db> source add_rows_to_testdb.txt
Query OK, 1 row affected
Time: 0.023s

Query OK, 1 row affected
Time: 0.002s

Query OK, 1 row affected
Time: 0.003s

Query OK, 1 row affected
Time: 0.002s

Query OK, 1 row affected
Time: 0.002s
```

Теперь таблица switch выглядит так:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str |
+----------------+----------+------------+-------------------+
7 rows in set
Time: 0.040s
```

С помощью оператора WHERE можно показать только те коммутаторы, модель которых 3850:

```
new_db.db> SELECT * from switch WHERE model = 'Cisco 3850';
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str |
+----------------+----------+------------+-------------------+
3 rows in set
Time: 0.033s
```

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:

```
new_db.db> SELECT * from switch WHERE model = 'Cisco 3750';
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str |
+----------------+----------+------------+-------------------+
2 rows in set
Time: 0.037s
```

Если вместе с оператором WHERE использовать оператор LIKE:

```
new_db.db> SELECT * from switch WHERE model LIKE '%3750';
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str |
+----------------+----------+------------+-------------------+
3 rows in set
Time: 0.040s
```

ALTER

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:

```
new_db.db> ALTER table switch ADD COLUMN mngmt_ip text;
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 0 rows affected
Time: 0.009s

new_db.db> ALTER table switch ADD COLUMN mngmt_vid integer;
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 0 rows affected
Time: 0.010s
```

Теперь таблица выглядит так (новые поля установлены в значение NULL):

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+----------+-----------+
| mac            | hostname | model      | location          | mngmt_ip | mngmt_vid |
+----------------+----------+------------+-------------------+----------+-----------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | <null>   | <null>    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | <null>   | <null>    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>   | <null>    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>   | <null>    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>   | <null>    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>   | <null>    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>   | <null>    |
+----------------+----------+------------+-------------------+----------+-----------+
7 rows in set
Time: 0.034s
```

UPDATE

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:

```
new_db.db> UPDATE switch set mngmt_ip = '10.255.1.1' WHERE hostname = 'sw1';
Query OK, 1 row affected
Time: 0.009s
```

Теперь таблица выглядит так:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | <null>    |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>     | <null>    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>     | <null>    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>     | <null>    |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.035s
```

Аналогичным образом можно изменить и номер VLAN:

```
new_db.db> UPDATE switch set mngmt_vid = 255 WHERE hostname = 'sw1';
Query OK, 1 row affected
Time: 0.009s

new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>     | <null>    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>     | <null>    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>     | <null>    |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.037s
```

Можно изменить несколько полей за раз:

```
new_db.db> UPDATE switch set mngmt_ip = '10.255.1.2', mngmt_vid = 255 WHERE hostname =
↪'sw2'
Query OK, 1 row affected
Time: 0.009s

new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+--------------------+------------+-----------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | <null>     | <null>    |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | <null>     | <null>    |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | <null>     | <null>    |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | <null>     | <null>    |
+----------------+----------+------------+--------------------+------------+-----------+
7 rows in set
Time: 0.033s
```

Чтобы не заполнять поля mngmt_ip и mngmt_vid вручную, заполним остальное из файла update_fields_in_testdb.txt (команда `source update_fields_in_testdb.txt`):

```
UPDATE switch set mngmt_ip = '10.255.1.3', mngmt_vid = 255 WHERE hostname = 'sw3';
UPDATE switch set mngmt_ip = '10.255.1.4', mngmt_vid = 255 WHERE hostname = 'sw4';
UPDATE switch set mngmt_ip = '10.255.1.5', mngmt_vid = 255 WHERE hostname = 'sw5';
UPDATE switch set mngmt_ip = '10.255.1.6', mngmt_vid = 255 WHERE hostname = 'sw6';
UPDATE switch set mngmt_ip = '10.255.1.7', mngmt_vid = 255 WHERE hostname = 'sw7';
```

После загрузки команд таблица выглядит так:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+------------+-----------+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+--------------------+------------+-----------+
| 0010.A1AA.C1CC | sw1      | Cisco 3750 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | 10.255.1.3 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
+----------------+----------+------------+--------------------+------------+-----------+
7 rows in set
Time: 0.038s
```

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:

```
new_db.db> UPDATE switch set model = 'Cisco 3850', mac = '0010.D1DD.E1EE' WHERE hostname␣
↪= 'sw1';
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Query OK, 1 row affected
Time: 0.009s
```

Результат будет таким:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3750 | London, Green Str | 10.255.1.3 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.049s
```

REPLACE

Оператор REPLACE используется для добавления или замены данных в таблице.

Примечание: 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

Пример нарушения правила уникальности:

```
new_db.db> INSERT INTO switch VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850', 'London,␣
↪Green Str', '10.255.1.3', 255);
UNIQUE constraint failed: switch.mac
```

У выражения REPLACE есть два вида:

```
new_db.db> INSERT OR REPLACE INTO switch VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850',
↪'London, Green Str', '10.255.1.3', 255);
Query OK, 1 row affected
Time: 0.010s
```

Или более короткий вариант:

```
new_db.db> REPLACE INTO switch VALUES ('0030.A3AA.C1CC', 'sw3', 'Cisco 3850', 'London,␣
↪Green Str', '10.255.1.3', 255);
Query OK, 1 row affected
Time: 0.009s
```

Результатом любой из этих команд будет замена модели коммутатора sw3:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+------------+-----------+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+--------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
+----------------+----------+------------+--------------------+------------+-----------+
```

В данном случае MAC-адрес в новой записи совпадает с уже существующей, поэтому происходит замена.

Примечание: 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:

```
new_db.db> REPLACE INTO switch VALUES ('0080.A8AA.C8CC', 'sw8', 'Cisco 3850', 'London,␣
↪Green Str', '10.255.1.8', 255);
Query OK, 1 row affected
Time: 0.009s

new_db.db> SELECT * from switch;
+----------------+----------+------------+--------------------+------------+-----------+
| mac            | hostname | model      | location           | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+--------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
| 0080.A8AA.C8CC | sw8      | Cisco 3850 | London, Green Str | 10.255.1.8 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
8 rows in set
Time: 0.034s
```

DELETE

Оператор DELETE используется для удаления записей. Как правило, он используется вместе с оператором WHERE.

Например, таблица switch выглядит так:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
| 0080.A8AA.C8CC | sw8      | Cisco 3850 | London, Green Str | 10.255.1.8 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
8 rows in set
Time: 0.033s
```

Удаление информации про коммутатор sw8 выполняется таким образом:

```
new_db.db> DELETE from switch where hostname = 'sw8';
You're about to run a destructive command.
Do you want to proceed? (y/n): y
Your call!
Query OK, 1 row affected
Time: 0.008s
```

Теперь в таблице нет строки с коммутатором sw8:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.039s
```

ORDER BY

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:

```
new_db.db> SELECT * from switch;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.039s
```

С помощью оператора ORDER BY можно вывести записи в таблице switch, отсортировав их по имени коммутаторов:

```
new_db.db> SELECT * from switch ORDER BY hostname ASC;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 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       |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.034s
```

По умолчанию сортировка выполняется по возрастанию, поэтому в запросе можно было не указывать параметр ASC:

```
new_db.db> SELECT * from switch ORDER BY hostname;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.034s
```

Сортировка по IP-адресу по убыванию:

```
SELECT * from switch ORDER BY mngmt_ip DESC;
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
7 rows in set
Time: 0.034s
```

AND

Оператор AND позволяет группировать несколько условий:

```
new_db.db> select * from switch where model = 'Cisco 3850' and mngmt_ip LIKE '10.255.%';
+----------------+----------+------------+-------------------+------------+-----------+
| 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       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
5 rows in set
Time: 0.034s
```

OR

Оператор OR:

```
new_db.db> select * from switch where model LIKE '%3750' or model LIKE '%3850';
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0060.A6AA.C4CC | sw6      | C3750      | London, Green Str | 10.255.1.6 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
6 rows in set
Time: 0.046s
```

IN

Оператор IN:

```
new_db.db> select * from switch where model in ('Cisco 3750', 'C3750');
+----------------+----------+-------+-------------------+------------+-----------+
| mac            | hostname | model | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+-------+-------------------+------------+-----------+
| 0060.A6AA.C4CC | sw6      | C3750 | London, Green Str | 10.255.1.6 | 255       |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
+----------------+----------+-------+------------------+------------+-----------+
1 row in set
Time: 0.034s
```

NOT

Оператор NOT:

```
new_db.db> select * from switch where model not in ('Cisco 3750', 'C3750');
+----------------+----------+------------+-------------------+------------+-----------+
| mac            | hostname | model      | location          | mngmt_ip   | mngmt_vid |
+----------------+----------+------------+-------------------+------------+-----------+
| 0010.D1DD.E1EE | sw1      | Cisco 3850 | London, Green Str | 10.255.1.1 | 255       |
| 0020.A2AA.C2CC | sw2      | Cisco 3850 | London, Green Str | 10.255.1.2 | 255       |
| 0040.A4AA.C2CC | sw4      | Cisco 3850 | London, Green Str | 10.255.1.4 | 255       |
| 0050.A5AA.C3CC | sw5      | Cisco 3850 | London, Green Str | 10.255.1.5 | 255       |
| 0070.A7AA.C5CC | sw7      | Cisco 3650 | London, Green Str | 10.255.1.7 | 255       |
| 0030.A3AA.C1CC | sw3      | Cisco 3850 | London, Green Str | 10.255.1.3 | 255       |
+----------------+----------+------------+-------------------+------------+-----------+
6 rows in set
Time: 0.037s
```

Модуль sqlite3

Для работы с SQLite в Python используется модуль sqlite3.

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')
```

После создания соединения надо создать объект Cursor - это основной способ работы с БД.

Создается курсор из соединения с БД:

```
connection = sqlite3.connect('dhcp_snooping.db')
cursor = connection.cursor()
```

Выполнение команд SQL

Для выполнения команд SQL в модуле есть несколько методов:

- execute - метод для выполнения одного выражения SQL
- executemany - метод позволяет выполнить одно выражение SQL для последовательности параметров (или для итератора)
- executescript - метод позволяет выполнить несколько выражений SQL за один раз

Метод execute

Метод execute позволяет выполнить одну команду SQL.

Сначала надо создать соединение и курсор:

```
In [1]: import sqlite3

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

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

Создание таблицы switch с помощью метода execute:

```
In [4]: cursor.execute("create table switch (mac text not NULL primary key, hostname text,
↪ model text, location text)")
Out[4]: <sqlite3.Cursor at 0x1085be880>
```

Выражения SQL могут быть параметризированы - вместо данных можно подставлять специальные значения. За счет этого можно использовать одну и ту же команду SQL для передачи разных данных.

Например, таблицу switch нужно заполнить данными из списка data:

```
In [5]: data = [
   ...: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
   ...: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
   ...: ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
   ...: ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

Для этого можно использовать запрос вида:

```
In [6]: query = "INSERT into switch values (?, ?, ?, ?)"
```

Знаки вопроса в команде используются для подстановки данных, которые будут передаваться методу execute.

Теперь можно передать данные таким образом:

```
In [7]: for row in data:
   ...:     cursor.execute(query, row)
   ...:
```

Второй аргумент, который передается методу execute, должен быть кортежем. Если нужно передать кортеж с одним элементом, используется запись `(value, )`.

Чтобы изменения были применены, нужно выполнить commit (обратите внимание, что метод commit вызывается у соединения):

```
In [8]: connection.commit()
```

Теперь при запросе из командной строки sqlite3, можно увидеть эти строки в таблице switch:

```
$ litecli sw_inventory.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
sw_inventory.db> SELECT * from switch;
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0000.AAAA.CCCC | sw1      | Cisco 3750 | London, Green Str |
| 0000.BBBB.CCCC | sw2      | Cisco 3780 | London, Green Str |
| 0000.AAAA.DDDD | sw3      | Cisco 2960 | London, Green Str |
| 0011.AAAA.CCCC | sw4      | Cisco 3750 | London, Green Str |
+----------------+----------+------------+-------------------+
4 rows in set
Time: 0.039s
sw_inventory.db>
```

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:

```
$ litecli sw_inventory.db
Version: 1.0.0
Mail: https://groups.google.com/forum/#!forum/litecli-users
Github: https://github.com/dbcli/litecli
sw_inventory.db> SELECT * from switch;
+----------------+----------+------------+-------------------+
| mac            | hostname | model      | location          |
+----------------+----------+------------+-------------------+
| 0000.AAAA.CCCC | sw1      | Cisco 3750 | London, Green Str |
| 0000.BBBB.CCCC | sw2      | Cisco 3780 | London, Green Str |
| 0000.AAAA.DDDD | sw3      | Cisco 2960 | London, Green Str |
| 0011.AAAA.CCCC | sw4      | Cisco 3750 | London, Green Str |
| 0000.1111.0001 | sw5      | Cisco 3750 | London, Green Str |
| 0000.1111.0002 | sw6      | Cisco 3750 | London, Green Str |
| 0000.1111.0003 | sw7      | Cisco 3750 | London, Green Str |
| 0000.1111.0004 | sw8      | Cisco 3750 | London, Green Str |
+----------------+----------+------------+-------------------+
8 rows in set
Time: 0.034s
```

Метод executemany подставил соответствующие кортежи в команду SQL, и все данные добавились в таблицу.

Метод executescript

Метод executescript позволяет выполнить несколько выражений SQL за один раз.

Особенно удобно использовать этот метод при создании таблиц:

```
In [13]: connection = sqlite3.connect('new_db.db')

In [14]: cursor = connection.cursor()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
In [15]: cursor.executescript('''
    ...:     create table switches(
    ...:         hostname     text not NULL primary key,
    ...:         location     text
    ...:     );
    ...:
    ...:     create table dhcp(
    ...:         mac          text not NULL primary key,
    ...:         ip           text,
    ...:         vlan         text,
    ...:         interface    text,
    ...:         switch       text not null references switches(hostname)
    ...:     );
    ...: ''')
Out[15]: <sqlite3.Cursor at 0x10efd67a0>
```

Получение результатов запроса

Для получения результатов запроса в sqlite3 есть несколько способов:

- использование методов `fetch` - в зависимости от метода возвращаются одна, несколько или все строки
- использование курсора как итератора - возвращается итератор

Метод fetchone

Метод fetchone возвращает одну строку данных.

Пример получения информации из базы данных sw_inventory.db:

```
In [16]: import sqlite3

In [17]: connection = sqlite3.connect('sw_inventory.db')

In [18]: cursor = connection.cursor()

In [19]: cursor.execute('select * from switch')
Out[19]: <sqlite3.Cursor at 0x104eda810>

In [20]: cursor.fetchone()
Out[20]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
```

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

Если повторно вызвать метод, он вернет следующую строку:

```
In [21]: print(cursor.fetchone())
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
```

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

За счет этого метод можно использовать в цикле, например, так:

```
In [22]: cursor.execute('select * from switch')
Out[22]: <sqlite3.Cursor at 0x104eda810>

In [23]: while True:
   ...:     next_row = cursor.fetchone()
   ...:     if next_row:
   ...:         print(next_row)
   ...:     else:
   ...:         break
   ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

Метод fetchmany

Метод fetchmany возвращает список строк данных.

Синтаксис метода:

```
cursor.fetchmany([size=cursor.arraysize])
```

С помощью параметра size можно указывать, какое количество строк возвращается. По умолчанию параметр size равен значению cursor.arraysize:

```
In [24]: print(cursor.arraysize)
1
```

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

```
In [25]: cursor.execute('select * from switch')
Out[25]: <sqlite3.Cursor at 0x104eda810>

In [26]: from pprint import pprint

In [27]: while True:
    ...:     three_rows = cursor.fetchmany(3)
    ...:     if three_rows:
    ...:         pprint(three_rows)
    ...:     else:
    ...:         break
    ...:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')]
[('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')]
[('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Метод выдает нужное количество строк, а если строк осталось меньше, чем параметр size, то оставшиеся строки.

Метод fetchall

Метод fetchall возвращает все строки в виде списка:

```
In [28]: cursor.execute('select * from switch')
Out[28]: <sqlite3.Cursor at 0x104eda810>

In [29]: cursor.fetchall()
Out[29]:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Важный аспект работы метода - он возвращает все оставшиеся строки.

То есть, если до метода fetchall использовался, например, метод fetchone, то метод fetchall вернет оставшиеся строки запроса:

```
In [30]: cursor.execute('select * from switch')
Out[30]: <sqlite3.Cursor at 0x104eda810>

In [31]: cursor.fetchone()
Out[31]: ('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')

In [32]: cursor.fetchone()
Out[32]: ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')

In [33]: cursor.fetchall()
Out[33]:
[('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str'),
 ('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Метод fetchmany в этом аспекте работает аналогично.

Cursor как итератор

Если нужно построчно обрабатывать результирующие строки, лучше использовать курсор как итератор. При этом не нужно использовать методы fetch.

При использовании методов execute возвращается курсор. А, так как курсор можно использовать как итератор, можно использовать его, например, в цикле for:

```
In [34]: result = cursor.execute('select * from switch')

In [35]: for row in result:
    ...:     print(row)
    ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

Аналогичный вариант отработает и без присваивания переменной:

```
In [36]: for row in cursor.execute('select * from switch'):
    ...:     print(row)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ...:
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
('0000.1111.0001', 'sw5', 'Cisco 3750', 'London, Green Str')
('0000.1111.0002', 'sw6', 'Cisco 3750', 'London, Green Str')
('0000.1111.0003', 'sw7', 'Cisco 3750', 'London, Green Str')
('0000.1111.0004', 'sw8', 'Cisco 3750', 'London, Green Str')
```

Использование модуля sqlite3 без явного создания курсора

Методы execute доступны и в объекте Connection, и в объекте Cursor, а методы fetch доступны только в объекте Cursor.

При использовании методов execute с объектом Connection курсор возвращается как результат выполнения метода execute. Его можно использовать как итератор и получать данные без методов fetch. За счет этого при работе с модулем sqlite3 можно не создавать курсор.

Пример итогового скрипта (файл create_sw_inventory_ver1.py):

```
import sqlite3


data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


con = sqlite3.connect('sw_inventory2.db')

con.execute('''create table switch
            (mac text not NULL primary key, hostname text, model text, location text)'''
            )

query = 'INSERT into switch values (?, ?, ?, ?)'
con.executemany(query, data)
con.commit()

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

con.close()
```

Результат выполнения будет таким:

```
$ python create_sw_inventory_ver1.py
('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str')
('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str')
('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str')
('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')
```

Обработка исключений

Посмотрим на пример использования метода execute при возникновении ошибки.

В таблице switch поле mac должно быть уникальным. И, если попытаться записать пересекающийся MAC-адрес, возникнет ошибка:

```
In [37]: con = sqlite3.connect('sw_inventory2.db')

In [38]: query = "INSERT into switch values ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960',
↪'London, Green Str')"

In [39]: con.execute(query)
---------------------------------------------------------------
IntegrityError             Traceback (most recent call last)
<ipython-input-56-ad34d83a8a84> in <module>()
----> 1 con.execute(query)

IntegrityError: UNIQUE constraint failed: switch.mac
```

Соответственно, можно перехватить исключение:

```
In [40]: try:
    ...:     con.execute(query)
    ...: except sqlite3.IntegrityError as e:
    ...:     print("Error 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.

При этом:

- при возникновении исключения, транзакция автоматически откатывается
- если исключения не было, автоматически выполняется commit

Пример использования соединения с базой данных как менеджера контекстов (create_sw_inventory_ver2.py):

```
import sqlite3


data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


con = sqlite3.connect('sw_inventory3.db')
con.execute('''create table switch
               (mac text not NULL primary key, hostname text, model text, location text)''
↪'
            )

try:
    with con:
        query = 'INSERT into switch values (?, ?, ?, ?)'
        con.executemany(query, data)

except sqlite3.IntegrityError as e:
    print('Возникла ошибка: ', e)

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

con.close()
```

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

Для проверки этого функционала надо записать в таблицу данные, в которых MAC-адрес

повторяется. Но прежде, чтобы не повторять части кода, лучше разнести код в файле create_sw_inventory_ver2.py по функциям (файл create_sw_inventory_ver2_functions.py):

```
from pprint import pprint
import sqlite3


data = [('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
        ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
        ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
        ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]


def create_connection(db_name):
    '''
    Функция создает соединение с БД db_name
    и возвращает его
    '''
    connection = sqlite3.connect(db_name)
    return connection


def write_data_to_db(connection, query, data):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить
     * data - данные, которые надо передать в виде списка кортежей

    Функция пытается записать все данные из списка data.
    Если данные удалось записать успешно, изменения сохраняются в БД
    и функция возвращает True.
    Если в процессе записи возникла ошибка, транзакция откатывается
    и функция возвращает False.
    '''
    try:
        with connection:
            connection.executemany(query, data)
    except sqlite3.IntegrityError as e:
        print('Возникла ошибка: ', e)
        return False
    else:
        print('Запись данных прошла успешно')
        return True


def get_all_from_db(connection, query):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить

    Функция возвращает данные полученные из БД.
    '''
    result = [row for row in connection.execute(query)]
    return result


if __name__ == '__main__':
    con = create_connection('sw_inventory3.db')

    print('Создание таблицы...')
    schema = '''create table switch
                (mac text primary key, hostname text, model text, location text)'''
    con.execute(schema)

    query_insert = 'INSERT into switch values (?, ?, ?, ?)'
    query_get_all = 'SELECT * from switch'

    print('Запись данных в БД:')
    pprint(data)
    write_data_to_db(con, query_insert, data)
    print('\nПроверка содержимого БД')
    pprint(get_all_from_db(con, query_get_all))

    con.close()
```

Результат выполнения скрипта выглядит так:

```
$ python create_sw_inventory_ver2_functions.py
Создание таблицы...
Запись данных в БД:
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
Запись данных прошла успешно

Проверка содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

Теперь проверим, как функция write_data_to_db отработает при наличии одинаковых MAC-адресов в данных.

В файле create_sw_inventory_ver3.py используются функции из файла create_sw_inventory_ver2_functions.py и подразумевается, что скрипт будет запускаться после записи предыдущих данных:

```
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC-адрес sw7 совпадает с MAC-адресом коммутатора sw3 в списке data
data2 = [('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
         ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
         ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
         ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]

con = dbf.create_connection('sw_inventory3.db')

query_insert = "INSERT into switch values (?, ?, ?, ?)"
query_get_all = "SELECT * from switch"

print("\nПроверка текущего содержимого БД")
pprint(dbf.get_all_from_db(con, query_get_all))

print('-' * 60)
print("Попытка записать данные с повторяющимся MAC-адресом:")
pprint(data2)
dbf.write_data_to_db(con, query_insert, data2)
print("\nПроверка содержимого БД")
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

В списке data2 у коммутатора sw7 MAC-адрес совпадает с уже существующим в БД коммутатором sw3.

Результат выполнения скрипта:

```
$ python create_sw_inventory_ver3.py

Проверка текущего содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
------------------------------------------------------------
Попытка записать данные с повторяющимся MAC-адресом:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
[('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
Error occurred:  UNIQUE constraint failed: switch.mac

Проверка содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
```

Обратите внимание, что содержимое таблицы switch до и после добавления информации одинаково. Это значит, что не записалась ни одна строка из списка data2.

Так получилось из-за того, что используется метод executemany, и в пределах одной транзакции мы пытаемся записать все 4 строки. Если возникает ошибка с одной из них - откатываются все изменения.

Иногда это именно то поведение, которое нужно. Если же надо, чтобы игнорировались только строки с ошибками, надо использовать метод execute и записывать каждую строку отдельно.

В файле create_sw_inventory_ver4.py создана функция write_rows_to_db, которая уже по очереди пишет данные и, если возникла ошибка, то только изменения для конкретных данных откатываются:

```
from pprint import pprint
import sqlite3
import create_sw_inventory_ver2_functions as dbf

#MAC-адрес sw7 совпадает с MAC-адресом коммутатора sw3 в списке data
data2 = [('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
         ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
         ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
         ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]


def write_rows_to_db(connection, query, data, verbose=False):
    '''
    Функция ожидает аргументы:
     * connection - соединение с БД
     * query - запрос, который нужно выполнить
     * data - данные, которые надо передать в виде списка кортежей

    Функция пытается записать поочереди кортежи из списка data.
    Если кортеж удалось записать успешно, изменения сохраняются в БД.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    Если в процессе записи кортежа возникла ошибка, транзакция откатывается.

    Флаг verbose контролирует то, будут ли выведены сообщения об удачной
    или неудачной записи кортежа.
    '''
    for row in data:
        try:
            with connection:
                connection.execute(query, row)
        except sqlite3.IntegrityError as e:
            if verbose:
                print("При записи данных '{}' возникла ошибка".format(
                    ', '.join(row), e))
        else:
            if verbose:
                print("Запись данных '{}' прошла успешно".format(
                    ', '.join(row)))


con = dbf.create_connection('sw_inventory3.db')

query_insert = 'INSERT into switch values (?, ?, ?, ?)'
query_get_all = 'SELECT * from switch'

print('\nПроверка текущего содержимого БД')
pprint(dbf.get_all_from_db(con, query_get_all))

print('-' * 60)
print('Попытка записать данные с повторяющимся MAC-адресом:')
pprint(data2)
write_rows_to_db(con, query_insert, data2, verbose=True)
print('\nПроверка содержимого БД')
pprint(dbf.get_all_from_db(con, query_get_all))

con.close()
```

Теперь результат выполнения будет таким (пропущен только sw7):

```
$ python create_sw_inventory_ver4.py

Проверка текущего содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str')]
------------------------------------------------------------
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Попытка записать данные с повторяющимся MAC-адресом:
[('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw7', 'Cisco 2960', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
Запись данных "0055.AAAA.CCCC, sw5, Cisco 3750, London, Green Str" прошла успешно
Запись данных "0066.BBBB.CCCC, sw6, Cisco 3780, London, Green Str" прошла успешно
При записи данных "0000.AAAA.DDDD, sw7, Cisco 2960, London, Green Str" возникла ошибка
Запись данных "0088.AAAA.CCCC, sw8, Cisco 3750, London, Green Str" прошла успешно

Проверка содержимого БД
[('0000.AAAA.CCCC', 'sw1', 'Cisco 3750', 'London, Green Str'),
 ('0000.BBBB.CCCC', 'sw2', 'Cisco 3780', 'London, Green Str'),
 ('0000.AAAA.DDDD', 'sw3', 'Cisco 2960', 'London, Green Str'),
 ('0011.AAAA.CCCC', 'sw4', 'Cisco 3750', 'London, Green Str'),
 ('0055.AAAA.CCCC', 'sw5', 'Cisco 3750', 'London, Green Str'),
 ('0066.BBBB.CCCC', 'sw6', 'Cisco 3780', 'London, Green Str'),
 ('0088.AAAA.CCCC', 'sw8', 'Cisco 3750', 'London, Green Str')]
```

Пример использования SQLite

В 15 разделе был пример разбора вывода команды show ip dhcp snooping binding. На выходе мы получили информацию о параметрах подключенных устройств (interface, IP, MAC, VLAN).

В таком варианте можно посмотреть только все подключенные устройства к коммутатору. Если же нужно узнать на основании одного из параметров другие, то в таком виде это не очень удобно.

Например, если нужно по IP-адресу получить информацию о том, к какому интерфейсу подключен компьютер, какой у него MAC-адрес и в каком он VLAN, то по выводу скрипта это сделать не очень просто и, главное, не очень удобно.

Запишем информацию, полученную из вывода sh ip dhcp snooping binding в SQLite. Это позволит делать запросы по любому параметру и получать недостающие. Для этого примера достаточно создать одну таблицу, где будет храниться информация.

Определение таблицы прописано в отдельном файле dhcp_snooping_schema.sql и выглядит так:

```
create table if not exists dhcp (
    mac          text not NULL primary key,
    ip           text,
    vlan         text,
    interface    text
);
```

Для всех полей определен тип данных «текст».

MAC-адрес является первичным ключом нашей таблицы, что вполне логично, так как MAC-адрес должен быть уникальным.

Кроме того, используется выражение `create table if not exists` - SQLite создаст таблицу только в том случае, если она не существует.

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

```
import sqlite3

conn = sqlite3.connect('dhcp_snooping.db')

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

conn.close()
```

Комментарии к файлу:

- при выполнении строки `conn = sqlite3.connect('dhcp_snooping.db')`:
  - создается файл dhcp_snooping.db, если его нет
  - создается объект Connection
- в БД создается таблица (если ее не было) на основании команд, которые указаны в файле dhcp_snooping_schema.sql:
  - открывается файл dhcp_snooping_schema.sql
  - `schema = f.read()` - весь файл считывается в одну строку
  - `conn.executescript(schema)` - метод executescript позволяет выполнять команды SQL, которые прописаны в файле

Выполнение скрипта:

```
$ python create_sqlite_ver1.py
Creating schema...
Done
```

В результате должен быть создан файл БД и таблица dhcp.

Проверить, что таблица создалась, можно с помощью утилиты sqlite3, которая позволяет выполнять запросы прямо в командной строке.

Список созданных таблиц выводится таким образом:

```
$ sqlite3 dhcp_snooping.db "SELECT name FROM sqlite_master WHERE type='table'"
dhcp
```

Теперь нужно записать информацию из вывода команды sh ip dhcp snooping binding в таблицу (файл dhcp_snooping.txt):

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  --------------------
00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10    FastEthernet0/1
00:04:A3:3E:5B:69   10.1.5.2         63951       dhcp-snooping   5     FastEthernet0/10
00:05:B3:7E:9B:60   10.1.5.4         63253       dhcp-snooping   5     FastEthernet0/9
00:09:BC:3F:A6:50   10.1.10.6        76260       dhcp-snooping   10    FastEthernet0/3
Total number of bindings: 4
```

Во второй версии скрипта сначала вывод в файле dhcp_snooping.txt обрабатывается регулярными выражениями, а затем записи добавляются в БД (файл create_sqlite_ver2.py):

```
import sqlite3
import re

regex = re.compile(r'(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

conn = sqlite3.connect('dhcp_snooping.db')

print('Creating schema...')
with open('dhcp_snooping_schema.sql', 'r') as f:
    schema = f.read()
    conn.executescript(schema)
print('Done')

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = '''insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)'''
            conn.execute(query, row)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    except sqlite3.IntegrityError as e:
        print('Error occured: ', e)

conn.close()
```

Примечание: Пока что файл БД каждый раз надо удалять, так как скрипт пытается его создать при каждом запуске.

Комментарии к скрипту:

- в регулярном выражении, которое проходится по выводу команды sh ip dhcp snooping binding, используются не именованные группы, как в примере раздела Регулярные выражения, а нумерованные
  - группы созданы только для тех элементов, которые нас интересуют
- result - это список, в котором хранится результат обработки вывода команды
  - но теперь тут не словари, а кортежи с результатами
  - это нужно для того, чтобы их можно было сразу передавать на запись в БД
- Перебираем в полученном списке кортежей элементы
- В этом скрипте используется еще один вариант записи в БД
  - строка query описывает запрос. Но вместо значений указываются знаки вопроса. Такой вариант записи запроса позволяет динамически подставлять значение полей
  - затем методу execute передается строка запроса и кортеж row, где находятся значения

Выполняем скрипт:

```
$ python create_sqlite_ver2.py
Creating schema...
Done
Inserting DHCP Snooping data
```

Проверим, что данные записались:

```
$ sqlite3 dhcp_snooping.db "select * from dhcp"
-- Loading resources from /home/vagrant/.sqliterc

mac                ip          vlan        interface
-----------------  ----------  ----------  ---------------
00:09:BB:3D:D6:58  10.1.10.2   10          FastEthernet0/1
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
00:05:B3:7E:9B:60  10.1.5.4    5           FastEthernet0/9
00:09:BC:3F:A6:50  10.1.10.6   10          FastEthernet0/3
```

Теперь попробуем запросить по определенному параметру:

```
$ sqlite3 dhcp_snooping.db "select * from dhcp where ip = '10.1.5.2'"
-- Loading resources from /home/vagrant/.sqliterc

mac                ip          vlan        interface
-----------------  ----------  ----------  ----------------
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/10
```

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

Переделаем скрипт таким образом, чтобы в нём была проверка на наличие файла dhcp_snooping.db. Если файл БД есть, то не надо создавать таблицу, считаем, что она уже создана.

Файл create_sqlite_ver3.py:

```
import os
import sqlite3
import re

data_filename = 'dhcp_snooping.txt'
db_filename = 'dhcp_snooping.db'
schema_filename = 'dhcp_snooping_schema.sql'

regex = re.compile(r'(\S+) +(\S+) +\d+ +\S+ +(\d+) +(\S+)')

result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groups())

db_exists = os.path.exists(db_filename)

conn = sqlite3.connect(db_filename)

if not db_exists:
    print('Creating schema...')
    with open(schema_filename, 'r') as f:
        schema = f.read()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    conn.executescript(schema)
    print('Done')
else:
    print('Database exists, assume dhcp table does, too.')

print('Inserting DHCP Snooping data')

for row in result:
    try:
        with conn:
            query = '''insert into dhcp (mac, ip, vlan, interface)
                       values (?, ?, ?, ?)'''
            conn.execute(query, row)
    except sqlite3.IntegrityError as e:
        print('Error occured: ', e)

conn.close()
```

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.

Примечание: Разделение процесса создания таблицы и заполнения ее данными вынесено в задания к разделу.

Если файла нет (предварительно его удалить):

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

Проверим. В случае, если файл уже есть, но данные не записаны:

```
$ rm dhcp_snooping.db

$ python create_sqlite_ver1.py
Creating schema...
Done
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
```

Если есть и БД и данные:

```
$ python create_sqlite_ver3.py
Database exists, assume dhcp table does, too.
Inserting DHCP Snooping data
Error occurred:  UNIQUE constraint failed: dhcp.mac
Error occurred:  UNIQUE constraint failed: dhcp.mac
Error occurred:  UNIQUE constraint failed: dhcp.mac
Error occurred:  UNIQUE constraint failed: dhcp.mac
```

Теперь делаем отдельный скрипт, который занимается отправкой запросов в БД и выводом результатов. Он должен:

- ожидать от пользователя ввода параметров:
  - имя параметра
  - значение параметра
- делать нормальный вывод данных по запросу

Файл get_data_ver1.py:

```
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

key, value = sys.argv[1:]
keys = ['mac', 'ip', 'vlan', 'interface']
keys.remove(key)

conn = sqlite3.connect(db_filename)

#Позволяет далее обращаться к данным в колонках, по имени колонки
conn.row_factory = sqlite3.Row

print('\nDetailed information for host(s) with', key, value)
print('-' * 40)

query = 'select * from dhcp where {} = ?'.format(key)
result = conn.execute(query, (value, ))

for row in result:
    for k in keys:
        print('{:12}: {}'.format(k, row[k]))
    print('-' * 40)
```

Комментарии к скрипту:

- из аргументов, которые передали скрипту, считываются параметры key, value

  - из списка keys удаляется выбранный ключ. Таким образом, в списке остаются только те параметры, которые нужно вывести

- подключаемся к БД
  - `conn.row_factory = sqlite3.Row` - позволяет далее обращаться к данным в колонках по имени колонки
- из БД выбираются те строки, в которых ключ равен указанному значению
  - в SQL значения можно подставлять через знак вопроса, но нельзя подставлять имя столбца. Поэтому имя столбца подставляется через форматирование строк, а значение - штатным средством SQL.
  - Обратите внимание на `(value,)` - таким образом передается кортеж с одним элементом
- Полученная информация выводится на стандартный поток вывода: * перебираем полученные результаты и выводим только те поля, названия

  которых находятся в списке keys

Проверим работу скрипта.

Показать параметры хоста с IP 10.1.10.2:

```
$ python get_data_ver1.py ip 10.1.10.2

Detailed information for host(s) with ip 10.1.10.2
----------------------------------------
mac         : 00:09:BB:3D:D6:58
vlan        : 10
interface   : FastEthernet0/1
----------------------------------------
```

Показать хосты в VLAN 10:

```
$ python get_data_ver1.py vlan 10

Detailed information for host(s) with vlan 10
----------------------------------------
mac         : 00:09:BB:3D:D6:58
ip          : 10.1.10.2
interface   : FastEthernet0/1
----------------------------------------
mac         : 00:07:BC:3F:A6:50
ip          : 10.1.10.6
interface   : FastEthernet0/3
----------------------------------------
```

Вторая версия скрипта для получения данных с небольшими улучшениями:

- Вместо форматирования строк используется словарь, в котором описаны запросы, соответствующие каждому ключу.
- Выполняется проверка ключа, который был выбран
- Для получения заголовков всех столбцов, который соответствуют запросу, используется метод keys()

Файл get_data_ver2.py:

```
import sqlite3
import sys

db_filename = 'dhcp_snooping.db'

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

key, value = sys.argv[1:]
keys = query_dict.keys()

if not key in keys:
    print('Enter key from {}'.format(', '.join(keys)))
else:
    conn = sqlite3.connect(db_filename)
    conn.row_factory = sqlite3.Row

    print('\nDetailed information for host(s) with', key, value)
    print('-' * 40)

    query = query_dict[key]
    result = conn.execute(query, (value, ))

    for row in result:
        for row_name in row.keys():
            print('{:12}: {}'.format(row_name, row[row_name]))
        print('-' * 40)
```

В этом скрипте есть несколько недостатков:

- не проверяется количество аргументов, которые передаются скрипту
- хотелось бы собирать информацию с разных коммутаторов. А для этого надо добавить поле, которое указывает, на каком коммутаторе была найдена запись

Кроме того, многое нужно доработать в скрипте, который создает БД и записывает данные.

Все доработки будут выполняться в заданиях этого раздела.

Дополнительные материалы

Документация:

- SQLite Tutorial - подробное описание SQLite
- Документация модуля sqlite3
- sqlite3 на сайте PyMOTW

Статьи:

- A thorough guide to SQLite database operations in Python

Задания

Предупреждение: Для заданий 25 раздела нет тестов!

Задание 25.1

Для заданий 25 раздела нет тестов!

Необходимо создать два скрипта:

1. create_db.py
2. add_data.py

Код в скриптах должен быть разбит на функции. Какие именно функции и как разделить код, надо решить самостоятельно. Часть кода может быть глобальной.

1. create_db.py - в этот скрипт 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
- имя файла бд - dhcp_snooping.db

В БД должно быть две таблицы (схема описана в файле dhcp_snooping_schema.sql):

- switches - в ней находятся данные о коммутаторах
- dhcp - тут хранится информация полученная из вывода sh ip dhcp snooping binding

Пример выполнения скрипта, когда файла dhcp_snooping.db нет:

```
$ python create_db.py
Создаю базу данных...
```

После создания файла:

```
$ python create_db.py
База данных существует
```

2. add_data.py - с помощью этого скрипта, выполняется добавление данных в БД. Скрипт должен добавлять данные из вывода sh ip dhcp snooping binding и информацию о коммутаторах

Соответственно, в файле add_data.py должны быть 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

Пример выполнения скрипта, когда база данных еще не создана:

```
$ python add_data.py
База данных не существует. Перед добавлением данных, ее надо создать
```

Пример выполнения скрипта первый раз, после создания базы данных:

```
$ python add_data.py
Добавляю данные в таблицу switches...
Добавляю данные в таблицу dhcp...
```

Пример выполнения скрипта, после того как данные были добавлены в таблицу (порядок добавления данных может быть произвольным, но сообщения должны выводиться аналогично выводу ниже):

```
$ python add_data.py
Добавляю данные в таблицу switches...
При добавлении данных: ('sw1', 'London, 21 New Globe Walk') Возникла ошибка: UNIQUE␣
↪constraint failed: switches.hostname
При добавлении данных: ('sw2', 'London, 21 New Globe Walk') Возникла ошибка: UNIQUE␣
↪constraint failed: switches.hostname
При добавлении данных: ('sw3', 'London, 21 New Globe Walk') Возникла ошибка: UNIQUE␣
↪constraint failed: switches.hostname
Добавляю данные в таблицу dhcp...
При добавлении данных: ('00:09:BB:3D:D6:58', '10.1.10.2', '10', 'FastEthernet0/1', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:04:A3:3E:5B:69', '10.1.5.2', '5', 'FastEthernet0/10', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:05:B3:7E:9B:60', '10.1.5.4', '5', 'FastEthernet0/9', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:07:BC:3F:A6:50', '10.1.10.6', '10', 'FastEthernet0/3', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7',
↪'sw1') Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:E9:BC:3F:A6:50', '100.1.1.6', '3', 'FastEthernet0/20', 'sw3')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:E9:22:11:A6:50', '100.1.1.7', '3', 'FastEthernet0/21', 'sw3')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:A9:BB:3D:D6:58', '10.1.10.20', '10', 'FastEthernet0/7', 'sw2
↪') Возникла ошибка: UNIQUE constraint failed: dhcp.mac
```

(continues on next page)

(продолжение с предыдущей страницы)

```
При добавлении данных: ('00:B4:A3:3E:5B:69', '10.1.5.20', '5', 'FastEthernet0/5', 'sw2')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:C5:B3:7E:9B:60', '10.1.5.40', '5', 'FastEthernet0/9', 'sw2')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:A9:BC:3F:A6:50', '10.1.10.60', '20', 'FastEthernet0/2', 'sw2
↪') Возникла ошибка: UNIQUE constraint failed: dhcp.mac
```

Оба скрипта вызываются без аргументов.

Задание 25.2

Для заданий 25 раздела нет тестов!

В этом задании необходимо 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.

Скрипту могут передаваться аргументы и, в зависимости от аргументов, надо выводить разную информацию. Если скрипт вызван:

- без аргументов, вывести всё содержимое таблицы dhcp
- с двумя аргументами, вывести информацию из таблицы dhcp, которая соответствует полю и значению
- с любым другим количеством аргументов, вывести сообщение, что скрипт поддерживает только два или ноль аргументов

Файл БД можно скопировать из задания 25.1.

Примеры вывода для разного количества и значений аргументов:

```
$ 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  192.168.100.100   1  FastEthernet0/7   sw1
00:E9:BC:3F:A6:50  100.1.1.6         3  FastEthernet0/20  sw3
00:E9:22:11:A6:50  100.1.1.7         3  FastEthernet0/21  sw3
00:A9:BB:3D:D6:58  10.1.10.20       10  FastEthernet0/7   sw2
00:B4:A3:3E:5B:69  10.1.5.20         5  FastEthernet0/5   sw2
00:C5:B3:7E:9B:60  10.1.5.40         5  FastEthernet0/9   sw2
00:A9:BC:3F:A6:50  10.1.10.60       20  FastEthernet0/2   sw2
-----------------  ---------------  --  ----------------  ---
```

(continues on next page)

(продолжение с предыдущей страницы)

```
$ python get_data.py vlan 10

Информация об устройствах с такими параметрами: vlan 10
-----------------  ----------  --  ---------------  ---
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1  sw1
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/3  sw1
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7  sw2
-----------------  ----------  --  ---------------  ---

$ python get_data.py ip 10.1.10.2

Информация об устройствах с такими параметрами: ip 10.1.10.2
-----------------  ---------  --  ---------------  ---
00:09:BB:3D:D6:58  10.1.10.2  10  FastEthernet0/1  sw1
-----------------  ---------  --  ---------------  ---

$ python get_data.py vln 10
Данный параметр не поддерживается.
Допустимые значения параметров: mac, ip, vlan, interface, switch

$ python get_data.py ip vlan 10
Пожалуйста, введите два или ноль аргументов
```

Задание 25.3

Для заданий 25 раздела нет тестов!

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

Скопируйте скрипт add_data.py из задания 25.1 и попробуйте выполнить его повторно, на существующей БД. Должен быть такой вывод:

```
$ python add_data.py
Добавляю данные в таблицу switches...
При добавлении данных: ('sw1', 'London, 21 New Globe Walk') Возникла ошибка: UNIQUE␣
↪constraint failed: switches.hostname
При добавлении данных: ('sw2', 'London, 21 New Globe Walk') Возникла ошибка: UNIQUE␣
↪constraint failed: switches.hostname
При добавлении данных: ('sw3', 'London, 21 New Globe Walk') Возникла ошибка: UNIQUE␣
↪constraint failed: switches.hostname
Добавляю данные в таблицу dhcp...
При добавлении данных: ('00:09:BB:3D:D6:58', '10.1.10.2', '10', 'FastEthernet0/1', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
```

(continues on next page)

(продолжение с предыдущей страницы)

```
При добавлении данных: ('00:04:A3:3E:5B:69', '10.1.5.2', '5', 'FastEthernet0/10', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:05:B3:7E:9B:60', '10.1.5.4', '5', 'FastEthernet0/9', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:07:BC:3F:A6:50', '10.1.10.6', '10', 'FastEthernet0/3', 'sw1')␣
↪Возникла ошибка: UNIQUE constraint failed: dhcp.mac
При добавлении данных: ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7',
↪'sw1') Возникла ошибка: UNIQUE constraint failed: dhcp.mac
... (вывод сокращен)
```

При создании схемы БД, было явно указано, что поле MAC-адрес, должно быть уникальным. Поэтому, при добавлении записи с таким же MAC-адресом, возникает исключение (ошибка). В задании 25.1 исключение обрабатывается и выводится сообщение на стандартный поток вывода.

В этом задании считается, что информация периодически считывается с коммутаторов и записывается в файлы. После этого, информацию из файлов надо перенести в базу данных. При этом, в новых данных могут быть изменения: MAC пропал, MAC перешел на другой порт/vlan, появился новый MAC и тп.

В этом задании в таблице dhcp надо создать новое поле active, которое будет указывать является ли запись актуальной. Новая схема БД находится в файле dhcp_snooping_schema.sql

Поле active должно принимать такие значения:

- 0 - означает False. 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.

Например, в таблице dhcp такие записи:

```
mac                ip          vlan        interface         switch      active
-----------------  ----------  ----------  ----------------  ----------  ----------
00:09:BB:3D:D6:58  10.1.10.2   10          FastEthernet0/1   sw1         1
00:04:A3:3E:5B:69  10.1.5.2    5           FastEthernet0/10  sw1         1
00:05:B3:7E:9B:60  10.1.5.4    5           FastEthernet0/9   sw1         1
00:07:BC:3F:A6:50  10.1.10.6   10          FastEthernet0/3   sw1         1
00:09:BC:3F:A6:50  192.168.10  1           FastEthernet0/7   sw1         1
```

И надо добавить такую информацию из файла:

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  --------------------
00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10    FastEthernet0/1
00:04:A3:3E:5B:69   10.1.15.2        63951       dhcp-snooping   15    FastEthernet0/15
00:05:B3:7E:9B:60   10.1.5.4         63253       dhcp-snooping   5     FastEthernet0/9
00:07:BC:3F:A6:50   10.1.10.6        76260       dhcp-snooping   10    FastEthernet0/5
```

После добавления данных таблица должна выглядеть так:

```
mac                 ip               vlan        interface         switch      active
------------------  ---------------  ----------  ---------------   ----------  ----------
00:09:BC:3F:A6:50   192.168.100.100  1           FastEthernet0/7   sw1         0
00:09:BB:3D:D6:58   10.1.10.2        10          FastEthernet0/1   sw1         1
00:04:A3:3E:5B:69   10.1.15.2        15          FastEthernet0/15  sw1         1
00:05:B3:7E:9B:60   10.1.5.4         5           FastEthernet0/9   sw1         1
00:07:BC:3F:A6:50   10.1.10.6        10          FastEthernet0/5   sw1         1
```

Новая информация должна перезаписывать предыдущую:

- MAC 00:04:A3:3E:5B:69 перешел на другой порт и попал в другой интерфейс и получил другой адрес
- MAC 00:07:BC:3F:A6:50 перешел на другой порт

Если какого-то MAC-адреса нет в новом файле, его надо оставить в бд со значением active = 0: MAC-адреса 00:09:BC:3F:A6:50 нет в новой информации (выключили комп).

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, сначала добавьте в бд информацию из файлов sw*_dhcp_snooping.txt, а потом добавьте информацию из файлов new_data/sw*_dhcp_snooping.txt

Данные должны выглядеть так (порядок строк может быть любым)

```
------------------  ---------------  --  ----------------  ---  -
00:09:BC:3F:A6:50   192.168.100.100   1  FastEthernet0/7   sw1  0
00:C5:B3:7E:9B:60   10.1.5.40         5  FastEthernet0/9   sw2  0
00:09:BB:3D:D6:58   10.1.10.2        10  FastEthernet0/1   sw1  1
00:04:A3:3E:5B:69   10.1.15.2        15  FastEthernet0/15  sw1  1
00:05:B3:7E:9B:60   10.1.5.4          5  FastEthernet0/9   sw1  1
00:07:BC:3F:A6:50   10.1.10.6        10  FastEthernet0/5   sw1  1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
00:E9:BC:3F:A6:50  100.1.1.6         3  FastEthernet0/20  sw3  1
00:E9:22:11:A6:50  100.1.1.7         3  FastEthernet0/21  sw3  1
00:A9:BB:3D:D6:58  10.1.10.20       10  FastEthernet0/7   sw2  1
00:B4:A3:3E:5B:69  10.1.5.20         5  FastEthernet0/5   sw2  1
00:A9:BC:3F:A6:50  10.1.10.65       20  FastEthernet0/2   sw2  1
00:A9:33:44:A6:50  10.1.10.77       10  FastEthernet0/4   sw2  1
-----------------  ---------------  --  ----------------  ---  -
```

Задание 25.4

Для заданий 25 раздела нет тестов!

Скопировать файл get_data из задания 25.2. Добавить в скрипт поддержку столбца active, который мы добавили в задании 25.3.

Теперь, при запросе информации, сначала должны отображаться активные записи, а затем, неактивные. Если неактивных записей нет, не отображать заголовок «Неактивные записи».

Примеры выполнения итогового скрипта

```
$ python get_data.py
В таблице dhcp такие записи:

Активные записи:

-----------------  ----------  --  ----------------  ---  -
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1   sw1  1
00:04:A3:3E:5B:69  10.1.15.2   15  FastEthernet0/15  sw1  1
00:05:B3:7E:9B:60  10.1.5.4     5  FastEthernet0/9   sw1  1
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/5   sw1  1
00:E9:BC:3F:A6:50  100.1.1.6    3  FastEthernet0/20  sw3  1
00:E9:22:11:A6:50  100.1.1.7    3  FastEthernet0/21  sw3  1
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7   sw2  1
00:B4:A3:3E:5B:69  10.1.5.20    5  FastEthernet0/5   sw2  1
00:A9:BC:3F:A6:50  10.1.10.65  20  FastEthernet0/2   sw2  1
00:A9:33:44:A6:50  10.1.10.77  10  FastEthernet0/4   sw2  1
-----------------  ----------  --  ----------------  ---  -

Неактивные записи:

-----------------  ---------------  -  ---------------  ---  -
00:09:BC:3F:A6:50  192.168.100.100  1  FastEthernet0/7  sw1  0
00:C5:B3:7E:9B:60  10.1.5.40        5  FastEthernet0/9  sw2  0
-----------------  ---------------  -  ---------------  ---  -
```

(continues on next page)

(продолжение с предыдущей страницы)

```
$ python get_data.py vlan 5

Информация об устройствах с такими параметрами: vlan 5

Активные записи:

-----------------  ---------  -  ---------------  ---  -
00:05:B3:7E:9B:60  10.1.5.4   5  FastEthernet0/9  sw1  1
00:B4:A3:3E:5B:69  10.1.5.20  5  FastEthernet0/5  sw2  1
-----------------  ---------  -  ---------------  ---  -

Неактивные записи:

-----------------  ---------  -  ---------------  ---  -
00:C5:B3:7E:9B:60  10.1.5.40  5  FastEthernet0/9  sw2  0
-----------------  ---------  -  ---------------  ---  -


$ python get_data.py vlan 10

Информация об устройствах с такими параметрами: vlan 10

Активные записи:

-----------------  ----------  --  ---------------  ---  -
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1  sw1  1
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/5  sw1  1
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7  sw2  1
00:A9:33:44:A6:50  10.1.10.77  10  FastEthernet0/4  sw2  1
-----------------  ----------  --  ---------------  ---  -
```

Задание 25.5

Для заданий 25 раздела нет тестов!

После выполнения заданий 25.1 - 25.5 в БД остается информация о неактивных записях. И, если какой-то MAC-адрес не появлялся в новых записях, запись с ним, может остаться в БД навсегда.

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

Получить строку со временем и датой, в указанном формате, можно с помощью функции datetime в запросе SQL. Синтаксис использования такой:

```
sqlite> insert into dhcp (mac, ip, vlan, interface, switch, active, last_active)
   ...> values ('00:09:BC:3F:A6:50', '192.168.100.100', '1', 'FastEthernet0/7', 'sw1', '0
↪', datetime('now'));
```

То есть вместо значения, которое записывается в базу данных, надо указать datetime(„now“).

После этой команды в базе данных появится такая запись:

```
mac                ip               vlan   interface        switch   active   last_active
-----------------  ---------------  -----  ---------------  -------  -------  ------------
↪-------
00:09:BC:3F:A6:50  192.168.100.100  1      FastEthernet0/7  sw1      0        2019-03-08␣
↪11:26:56
```

Задание 25.5a

Для заданий 25 раздела нет тестов!

После выполнения задания 25.5, 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 некоторых записях и поставить время 7 или более дней.

В файле задания описан пример работы с объектами модуля datetime. Показано как получить дату 7 дней назад. С этой датой надо будет сравнивать время last_active.

Обратите внимание, что строки с датой, которые пишутся в БД, можно сравнивать между собой.

```
from datetime import timedelta, datetime

now = datetime.today().replace(microsecond=0)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
week_ago = now - timedelta(days=7)

#print(now)
#print(week_ago)
#print(now > week_ago)
#print(str(now) > str(week_ago))
```

Задание 25.6

Для заданий 25 раздела нет тестов!

В этом задании выложен файл parse_dhcp_snooping.py. В файле parse_dhcp_snooping.py нельзя ничего менять.

В файле созданы несколько функций и описаны аргументы командной строки, которые принимает файл.

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.

Для того, чтобы было проще начать, попробуйте создать необходимые функции в файле parse_dhcp_snooping_functions.py и 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. В примерах показывается вариант, когда в базе данных есть поля active и last_active, но можно также использовать вариант без этих полей.

```
$ 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          имя БД
  -k {mac,ip,vlan,interface,switch}
                        параметр для поиска записей
  -v VALUE              значение параметра
  -a                    показать все содержимое БД


$ python parse_dhcp_snooping.py add -h
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                  filename [filename ...]

positional arguments:
  filename      файл(ы), которые надо добавить

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  имя БД
  -s            если флаг установлен, добавлять данные коммутаторов, иначе -
                DHCP записи


$ python parse_dhcp_snooping.py add -h
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                  filename [filename ...]

positional arguments:
  filename      файл(ы), которые надо добавить

optional arguments:
  -h, --help    show this help message and exit
  --db DB_FILE  имя БД
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  -s            если флаг установлен, добавлять данные коммутаторов, иначе
                добавлять DHCP записи


$ 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          имя БД
  -k {mac,ip,vlan,interface,switch}
                        параметр для поиска записей
  -v VALUE              значение параметра
  -a                    показать все содержимое БД


$ python parse_dhcp_snooping.py create_db
Создаю БД dhcp_snooping.db со схемой dhcp_snooping_schema.sql
Создаю базу данных...


$ python parse_dhcp_snooping.py add sw[1-3]_dhcp_snooping.txt
Читаю информацию из файлов
sw1_dhcp_snooping.txt, sw2_dhcp_snooping.txt, sw3_dhcp_snooping.txt

Добавляю данные по DHCP записях в dhcp_snooping.db


$ python parse_dhcp_snooping.py add -s switches.yml
Добавляю данные о коммутаторах

$ python parse_dhcp_snooping.py get
В таблице dhcp такие записи:

Активные записи:

-----------------  ---------------  --  ----------------  ---  -  -------------------
00:09:BB:3D:D6:58  10.1.10.2        10  FastEthernet0/1   sw1  1  2019-03-08 16:47:52
00:04:A3:3E:5B:69  10.1.5.2          5  FastEthernet0/10  sw1  1  2019-03-08 16:47:52
00:05:B3:7E:9B:60  10.1.5.4          5  FastEthernet0/9   sw1  1  2019-03-08 16:47:52
00:07:BC:3F:A6:50  10.1.10.6        10  FastEthernet0/3   sw1  1  2019-03-08 16:47:52
00:09:BC:3F:A6:50  192.168.100.100   1  FastEthernet0/7   sw1  1  2019-03-08 16:47:52
00:A9:BB:3D:D6:58  10.1.10.20       10  FastEthernet0/7   sw2  1  2019-03-08 16:47:52
```

(continues on next page)

(продолжение с предыдущей страницы)

```
00:B4:A3:3E:5B:69  10.1.5.20         5  FastEthernet0/5   sw2  1  2019-03-08 16:47:52
00:C5:B3:7E:9B:60  10.1.5.40         5  FastEthernet0/9   sw2  1  2019-03-08 16:47:52
00:A9:BC:3F:A6:50  10.1.10.60       20  FastEthernet0/2   sw2  1  2019-03-08 16:47:52
00:E9:BC:3F:A6:50  100.1.1.6         3  FastEthernet0/20  sw3  1  2019-03-08 16:47:52
-----------------  ---------------  --  ----------------  ---  -  -------------------


$ python parse_dhcp_snooping.py get -k vlan -v 10
Данные из БД: dhcp_snooping.db
Информация об устройствах с такими параметрами: vlan 10

Активные записи:

-----------------  ----------  --  ---------------  ---  -  -------------------
00:09:BB:3D:D6:58  10.1.10.2   10  FastEthernet0/1  sw1  1  2019-03-08 16:47:52
00:07:BC:3F:A6:50  10.1.10.6   10  FastEthernet0/3  sw1  1  2019-03-08 16:47:52
00:A9:BB:3D:D6:58  10.1.10.20  10  FastEthernet0/7  sw2  1  2019-03-08 16:47:52
-----------------  ----------  --  ---------------  ---  -  -------------------


$ python parse_dhcp_snooping.py get -k vln -v 10
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]
parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'vln' (choose from 'mac',
↪'ip', 'vlan', 'interface', 'switch')
```

9

VIII. Дополнительная информация

В этом разделе собрана информация, которая не вошла в основные разделы книги, но которая, тем не менее, может быть полезна.

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.

В книге рассматривается только argparse, но кроме него стоит обратить внимание на те модули, которые не входят в стандартную библиотеку Python. Например, click.

Примечание: Хорошая статья, которая сравнивает разные модули обработки аргументов командной строки (рассматриваются argparse, click и docopt).

Пример скрипта ping_function.py:

```
import subprocess
import argparse
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def ping_ip(ip_address, count):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    reply = subprocess.run(
        f"ping -c {count} -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding="utf-8",
    )
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stdout + reply.stderr


parser = argparse.ArgumentParser(description="Ping script")

parser.add_argument("-a", dest="ip", required=True)
parser.add_argument("-c", 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', dest="ip")`
  - аргумент, который передается после опции `-a`, сохранится в переменную `ip`
- `parser.add_argument('-c', 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 был указан, по умолчанию будет значение 2

Строка `args = parser.parse_args()` указывается после того, как определены все аргументы.

После её выполнения в переменной `args` содержатся все аргументы, которые были переданы скрипту. К ним можно обращаться, используя синтаксис `args.ip`.

Попробуем вызвать скрипт с разными аргументами. Если переданы оба аргумента:

```
$ python ping_function.py -a 8.8.8.8 -c 5
Namespace(count=5, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.673 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.902 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=48.696 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=48 time=50.040 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=48 time=48.831 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.673/49.228/50.040/0.610 ms

Namespace - это объект, который возвращает метод parse\_args()
```

Передаем только IP-адрес:

```
$ python ping_function.py -a 8.8.8.8
Namespace(count=2, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.563 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.616 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.563/49.090/49.616/0.526 ms
```

Вызов скрипта без аргументов:

```
$ python ping_function.py
Namespace(count=2, ip=None)
Traceback (most recent call last):
  File "ping_function.py", line 31, in <module>
    rc, message = ping_ip( args.ip, args.count )
  File "ping_function.py", line 16, in ping_ip
    stderr=temp)
  File "/usr/local/lib/python3.6/subprocess.py", line 336, in check_output
    **kwargs).stdout
  File "/usr/local/lib/python3.6/subprocess.py", line 403, in run
    with Popen(*popenargs, **kwargs) as process:
  File "/usr/local/lib/python3.6/subprocess.py", line 707, in __init__
    restore_signals, start_new_session)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  File "/usr/local/lib/python3.6/subprocess.py", line 1260, in _execute_child
    restore_signals, start_new_session, preexec_fn)
TypeError: expected str, bytes or os.PathLike object, not NoneType
```

Если бы функция была вызвана без аргументов, когда не используется argparse, возникла бы ошибка, что не все аргументы указаны.

Из-за argparse, фактически аргумент передается, только он равен None. Это видно в строке `Namespace(count=2, ip=None)`.

В таком скрипте IP-адрес необходимо указывать всегда. И в argparse можно указать, что аргумент является обязательным. Для этого надо изменить опцию `-a`: добавить в конце `required=True`:

```
parser.add_argument('-a', dest="ip", required=True)
```

Теперь, если вызвать скрипт без аргументов, вывод будет таким:

```
$ python ping_function.py
usage: ping_function.py [-h] -a IP [-c COUNT]
ping_function.py: error: the following arguments are required: -a
```

Теперь отображается понятное сообщение, что надо указать обязательный аргумент, и подсказка usage.

Также, благодаря argparse, доступен help:

```
$ python ping_function.py -h
usage: ping_function.py [-h] -a IP [-c COUNT]

Ping script

optional arguments:
  -h, --help  show this help message and exit
  -a IP
  -c COUNT
```

Обратите внимание, что в сообщении все опции находятся в секции `optional arguments`. argparse сам определяет, что указаны опции, так как они начинаются с `-` и в имени только одна буква.

Зададим IP-адрес как позиционный аргумент. Файл ping_function_ver2.py:

```
import subprocess
import argparse

def ping_ip(ip_address, count):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    reply = subprocess.run(
        f"ping -c {count} -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding="utf-8",
    )
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stdout + reply.stderr


parser = argparse.ArgumentParser(description="Ping script")

parser.add_argument("host", help="IP or name to ping")
parser.add_argument("-c", 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.

Примечание: Этот пример покажет больше возможностей argparse, но они этим не ограничиваются, поэтому, если вы будете использовать argparse, обязательно посмотрите документацию модуля или статью на PyMOTW.

Файл parse_dhcp_snooping.py:

```python
# -*- coding: utf-8 -*-
import argparse

# Default values:
DFLT_DB_NAME = 'dhcp_snooping.db'
DFLT_DB_SCHEMA = 'dhcp_snooping_schema.sql'


def create(args):
    print(f"Creating DB {args.name} with DB schema {args.schema}")


def add(args):
    if args.sw_true:
        print("Adding switch data to database")
    else:
        print(f"Reading info from file(s) \n{', '.join(args.filename)}")
        print(f"\nAdding data to db {args.db_file}")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def get(args):
    if args.key and args.value:
        print(f"Geting data from DB: {args.db_file}")
        print(f"Request data for host(s) with {args.key} {args.value}")
    elif args.key or args.value:
        print("Please give two or zero args\n")
        print(show_subparser_help('get'))
    else:
        print(f"Showing {args.db_file} content...")


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__':
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    args = parser.parse_args()
    if not vars(args):
        parser.print_usage()
    else:
        args.func(args)
```

Теперь создается не только парсер, как в прошлом примере, но и вложенные парсеры. Вложенные парсеры будут отображаться как команды. Фактически, они будут использоваться как обязательные аргументы.

С помощью вложенных парсеров создается иерархия аргументов и опций. Аргументы, которые добавлены во вложенный парсер, будут доступны как аргументы этого парсера. Например, в этой части создан вложенный парсер create_db, и к нему добавлена опция `-n`:

```
create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', dest='name', default=DFLT_DB_NAME,
                           help='db filename')
```

Синтаксис создания вложенных парсеров и добавления к ним аргументов одинаков:

```
create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
create_parser.set_defaults(func=create)
```

Метод `add_argument` добавляет аргумент. Тут синтаксис точно такой же, как и без использования вложенных парсеров.

В строке `create_parser.set_defaults(func=create)` указывается, что при вызове парсера create_parser будет вызвана функция create.

Функция create получает как аргумент все аргументы, которые были переданы. И внутри функции можно обращаться к нужным:

```
def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))
```

Если вызвать help для этого скрипта, вывод будет таким:

```
$ python parse_dhcp_snooping.py -h
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(продолжение с предыдущей страницы)

```
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')
```

Параметр 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. В таких случаях можно указывать choices.

Для этого парсера help выглядит так:

```
$ python parse_dhcp_snooping.py get -h
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]

optional arguments:
  -h, --help            show this help message and exit
  --db DB_FILE          db name
  -k {mac,ip,vlan,interface,switch}
                        host key (parameter) to search
  -v VALUE              value of key
  -a                    show db content
```

А если выбрать неправильный вариант:

```
$ python parse_dhcp_snooping.py get -k test
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                  [-k {mac,ip,vlan,interface,switch}]
                                  [-v VALUE] [-a]
parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'test' (choose from 'mac',
↪ 'ip', 'vlan', 'interface', 'switch')

В данном примере важно указать варианты на выбор, так как затем на
основании выбранного варианта генерируется SQL-запрос. И, благодаря
``choices``, нет возможности указать какой-то параметр, кроме
разрешенных.
```

Импорт парсера

В файле parse_dhcp_snooping.py последние две строки будут выполняться только в том случае, если скрипт был вызван как основной.

```
if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

А значит, если импортировать файл, эти строки не будут вызваны.

Попробуем импортировать парсер в другой файл (файл call_pds.py):

```
from parse_dhcp_snooping import parser

args = parser.parse_args()
args.func(args)
```

Вызов сообщения help:

```
$ python call_pds.py -h
usage: call_pds.py [-h] {create_db,add,get} ...

optional arguments:
  -h, --help           show this help message and exit

subcommands:
  valid subcommands

  {create_db,add,get}  description
    create_db          create new db
    add                add data to db
    get                get data from db
```

Вызов аргумента:

```
$ python call_pds.py add test.txt test2.txt
Reading info from file(s)
test.txt, test2.txt

Adding data to db dhcp_snooping.db
```

Всё работает без проблем.

Передача аргументов вручную

Последняя особенность argparse - возможность передавать аргументы вручную.

Аргументы можно передать как список при вызове метода `parse_args()` (файл call_pds2.py):

```
from parse_dhcp_snooping import parser, get

args = parser.parse_args('add test.txt test2.txt'.split())
args.func(args)

Необходимо использовать метод ``split()``, так как метод ``parse_args()``
ожидает список аргументов.
```

Результат будет таким же, 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
```

Форматирование строк с оператором %

Пример использования оператора %:

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

В старом синтаксисе форматирования строк используются такие обозначения:

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

Вывести данные столбцами одинаковой ширины по 15 символов с выравниванием по правой стороне:

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

In [4]: print("%15s %15s %15s" % (vlan, mac, intf))
            100  aabb.cc80.7000           Gi0/1
```

Выравнивание по левой стороне:

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

С помощью форматирования строк можно также влиять на отображение чисел.

Например, можно указать, сколько цифр после запятой выводить:

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

Примечание: У форматирования строк есть ещё много возможностей. Хорошие примеры и объяснения двух вариантов форматирования строк можно найти тут.

Соглашение об именах

В Python есть определенные соглашения об именовании объектов.

В целом, лучше придерживаться этих соглашений. Однако, если в определенной библиотеке или модуле используются другие соглашения, то стоит придерживаться того стиля, который используется в них.

> В этом разделе описаны не все правила. Подробнее можно почитать в документе PEP8 на английском или на русском.

Имена переменных

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

Имена переменных обычно пишутся полностью большими или маленькими буквами. В пределах одного скрипта/модуля/пакета лучше придерживаться одного из вариантов.

Если переменные - константы для модуля, то лучше использовать имена, написанные заглавными буквами:

```
DB_NAME = 'dhcp_snooping.db'
TESTING = True
```

Для обычных переменных лучше использовать имена в нижнем регистре:

```
db_name = 'dhcp_snooping.db'
testing = True
```

Имена модулей и пакетов

Имена модулей и пакетов задаются маленькими буквами.

Модули могут использовать подчеркивания между словами для того, чтобы имена были более понятными. Для пакетов лучше выбирать короткие имена.

Имена функций

Имена функций задаются маленькими буквами, с подчеркиваниями между словами.

```
def ignore_command(command, ignore):

    ignore_command = False

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

Имена классов

Имена классов задаются словами с заглавными буквами, без пробелов.

```
class CiscoSwitch:

    def __init__(self, name, vendor='cisco', model='3750'):
        self.name = name
        self.vendor = vendor
        self.model = model
```

Подчеркивание в именах

В Python подчеркивание в начале или в конце имени указывает на специальные имена. Чаще всего это всего лишь договоренность, но иногда это действительно влияет на поведение объекта.

Подчеркивание как имя

В Python одно подчеркивание используется для обозначения того, что данные просто выбрасываются.

Например, если из строки line надо получить MAC-адрес, IP-адрес, VLAN и интерфейс и отбросить остальные поля, можно использовать такой вариант:

```
In [1]: line = '00:09:BB:3D:D6:58  10.1.10.2 86250   dhcp-snooping   10  FastEthernet0/1'

In [2]: mac, ip, _, _, vlan, intf = line.split()

In [3]: print(mac, ip, vlan, intf)
00:09:BB:3D:D6:58 10.1.10.2 10 FastEthernet0/1
```

Такая запись говорит о том, что нам не нужны третий и четвертый элементы.

Можно сделать так:

```
In [4]: mac, ip, lease, entry_type, vlan, intf = line.split()
```

Но тогда может быть непонятно, почему переменные lease и entry_type не используются дальше. Если понятней использовать имена, то лучше назвать переменные именами вроде ignored.

Аналогичный прием может использоваться, когда переменная цикла не нужна:

```
In [5]: [0 for _ in range(10)]
Out[5]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Подчеркивание в интерпретаторе

В интерпретаторе python и ipython подчеркивание используется для получения результата последнего выражения

```
In [6]: [0 for _ in range(10)]
Out[6]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [7]: _
Out[7]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

In [8]: a = _

In [9]: a
Out[9]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Одно подчеркивание

Одно подчеркивание перед именем

Одно подчеркивание перед именем указывает, что имя используется как внутреннее.

Например, если одно подчеркивание указано в имени функции или метода, это означает, что этот объект является внутренней особенностью реализации и не стоит его использовать напрямую.

Но, кроме того, при импорте вида `from module import *` не будут импортироваться объекты, которые начинаются с подчеркивания.

Например, в файле example.py такие переменные и функции:

```
db_name = 'dhcp_snooping.db'
_path = '/home/nata/pyneng/'


def func1(arg):
    print arg


def _func2(arg):
    print arg
```

Если импортировать все объекты из модуля, то те, которые начинаются с подчеркивания, не будут импортированы:

```
In [7]: from example import *

In [8]: db_name
Out[8]: 'dhcp_snooping.db'

In [9]: _path
...
NameError: name '_path' is not defined

In [10]: func1(1)
1

In [11]: _func2(1)
...
NameError: name '_func2' is not defined
```

Одно подчеркивание после имени

Одно подчеркивание после имени используется в том случае, когда имя объекта или параметра пересекается со встроенными именами.

Пример:

```
In [12]: line = '00:09:BB:3D:D6:58  10.1.10.2 86250   dhcp-snooping   10  FastEthernet0/1'

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]:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
['_CiscoSwitch__configure', '_CiscoSwitch__quantity', '_Switch__configure', '_Switch__
↪quantity', ...]
```

Два подчеркивания перед и после имени

Таким образом обозначаются специальные переменные и методы.

Например, в модуле Python есть такие специальные переменные:

- `__name__` - эта переменная равна строке `__main__`, когда скрипт запускается напрямую, и равна имени модуля, когда импортируется
- `__file__` - эта переменная равна имени скрипта, который был запущен напрямую, и 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
/home/vagrant/repos/tests/example2.py
```

Кроме того, таким образом в Python обозначаются специальные методы. Эти методы вызываются при использовании функций и операторов Python и позволяют реализовать определенный функционал.

Как правило, такие методы не нужно вызывать напрямую. Но, например, при создании своего класса может понадобиться описать такой метод, чтобы объект поддерживал какие-то операции в Python.

Например, для того, чтобы можно было получить длину объекта, он должен поддерживать метод `__len__`.

Ещё один специальный метод `__str__` вызывается, когда используется оператор print или вызывается функция str(). Если необходимо, чтобы при этом отображение было в определенном виде, надо создать этот метод в классе:

```
In [10]: class Switch(object):
    ...:
    ...:     def set_name(self, name):
    ...:         self.name = name
    ...:
    ...:     def __configure(self):
    ...:         pass
    ...:
    ...:     def __str__(self):
    ...:         return 'Switch {}'.format(self.name)
    ...:

In [11]: sw1 = Switch()

In [12]: sw1.set_name('sw1')

In [13]: print sw1
Switch sw1

In [14]: str(sw1)
Out[14]: 'Switch sw1'
```

Таких специальных методов в Python очень много. Несколько полезных ссылок, где можно почитать про конкретный метод:

- документация
- Dive Into Python 3

Отличия Python 2.7 и Python 3.6

Unicode

В Python 2.7 было два типа строк: str и unicode:

```
In [1]: line = 'test'

In [2]: line2 = u'тест'
```

В Python 3 строка - это тип str, но, кроме этого, в Python 3 появился тип bytes:

```
In [3]: line = 'тест'

In [4]: line.encode('utf-8')
Out[4]: b'\xd1\x82\xd0\xb5\xd1\x81\xd1\x82'

In [5]: byte_str = b'test'
```

Функция print

В Python 2.7 print был оператором:

```
In [6]: print 1, 'test'
1 test
```

В Python 3 print - функция:

```
In [7]: print(1, 'test')
1 test
```

В Python 2.7 можно брать аргументы в скобки, но от этого print не становится функцией и, кроме того, print возвращает другой результат (кортеж):

```
In [8]: print(1, 'test')
(1, 'test')
```

В Python 3, использование синтаксиса Python 2.7 приведет к ошибке:

```
In [9]: print 1, 'test'
  File "<ipython-input-2-328abb6b105d>", line 1
    print 1, 'test'
          ^
SyntaxError: Missing parentheses in call to 'print'
```

input вместо raw_input

В Python 2.7 для получения информации от пользователя в виде строки использовалась функция raw_input:

```
In [10]: number = raw_input('Number: ')
Number: 55

In [11]: number
Out[11]: '55'
```

В Python 3 используется input:

```
In [12]: number = input('Number: ')
Number: 55

In [13]: number
Out[13]: '55'
```

range вместо xrange

В Python 2.7 были две функции

- range - возвращает список
- xrange - возвращает итератор

Пример range и xrange в Python 2.7:

```
In [14]: range(5)
Out[14]: [0, 1, 2, 3, 4]

In [15]: xrange(5)
Out[15]: xrange(5)

In [16]: list(xrange(5))
Out[16]: [0, 1, 2, 3, 4]
```

В Python 3 есть только функция range, и она возвращает итератор:

```
In [17]: range(5)
Out[17]: range(0, 5)

In [18]: list(range(5))
Out[18]: [0, 1, 2, 3, 4]
```

Методы словарей

Несколько изменений произошло в методах словарей.

dict.keys(), values(), items()

Методы keys(), values(), items() в Python 3 возвращают «views» вместо списков. Особенность view заключается в том, что они меняются вместе с изменением словаря. И фактически они лишь дают способ посмотреть на соответствующие объекты, но не создают их копию.

В Python 3 нет методов:

- viewitems, viewkeys, viewvalues
- iteritems, iterkeys, itervalues

Для сравнения, методы словаря в Python 2.7:

```
In [19]: d = {1:100, 2:200, 3:300}

In [20]: d.
    d.clear      d.get        d.iteritems  d.keys       d.setdefault d.viewitems
    d.copy       d.has_key    d.iterkeys   d.pop        d.update     d.viewkeys
    d.fromkeys   d.items      d.itervalues d.popitem    d.values     d.viewvalues
```

И в Python 3:

```
In [21]: d = {1:100, 2:200, 3:300}

In [22]: d.
           clear()      get()        pop()        update()
           copy()       items()      popitem()    values()
           fromkeys()   keys()       setdefault()
```

Распаковка переменных

В Python 3 появилась возможность использовать * при распаковке переменных:

```
In [23]: a, *b, c = [1,2,3,4,5]

In [24]: a
Out[24]: 1

In [25]: b
Out[25]: [2, 3, 4]

In [26]: c
Out[26]: 5
```

В Python 2.7 этот синтаксис не поддерживается:

```
In [27]: a, *b, c = [1, 2, 3, 4, 5]
  File "<ipython-input-10-e3f57143ffb4>", line 1
    a, *b, c = [1, 2, 3, 4, 5]
       ^
SyntaxError: invalid syntax
```

Итератор вместо списка

В Python 2.7 map, filter и zip возвращали список:

```
In [28]: map(str, [1, 2, 3, 4, 5])
Out[28]: ['1', '2', '3', '4', '5']

In [29]: filter(lambda x: x > 3, [1, 2, 3, 4, 5])
Out[29]: [4, 5]

In [30]: zip([1, 2, 3], [100, 200, 300])
Out[30]: [(1, 100), (2, 200), (3, 300)]
```

В Python 3 они возвращают итератор:

```
In [31]: map(str, [1, 2, 3, 4, 5])
Out[31]: <map at 0xb4ee3fec>

In [32]: filter(lambda x: x > 3, [1, 2, 3, 4, 5])
Out[32]: <filter at 0xb448c68c>

In [33]: zip([1, 2, 3], [100, 200, 300])
Out[33]: <zip at 0xb4efc1ec>
```

subprocess.run

В версии Python 3.5 в модуле subprocess появилась новая функция - run. Она предоставляет более удобный интерфейс для работы с модулем и получения вывода команд.

Соответственно, вместо функций call и check_output используется функция run, но функции call и check_output остались.

Jinja2

В модуле Jinja2 больше не нужно использовать такой код, так как кодировка по умолчанию и так utf-8:

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

В самих шаблонах, как и в Python, изменились методы словарей. Тут, аналогично, вместо iteritems надо использовать items.

Модули pexpect, telnetlib, paramiko

Модули pexpect, telnetlib, paramiko отправляют и получают байты, поэтому надо делать encode/decode соответственно.

В netmiko эта конвертация выполняется автоматически.

Мелочи

- Название модуля Queue сменилось на queue
- С версии Python 3.6 объект csv.DictReader возвращает OrderedDict вместо обычного словаря.

Дополнительная информация

Ниже приведены ссылки на ресурсы с информацией об изменениях в Python 3.

Документация:

- What's New In Python 3.0
- Should I use Python 2 or Python 3 for my development activity?

Статьи:

- The key differences between Python 2.7.x and Python 3.x with examples
- Supporting Python 3: An in-depth guide

Проверка заданий с помощью утилиты pyneng

Начиная с раздела «4. Типы данных в Python» для проверки заданий используются автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами.

Помимо перечисленных выше положительных моментов, в тестах также можно посмотреть какой итоговый результат нужен: прояснить структуру данных и мелочи, которые могут влиять на результат.

Для запуска тестов используется `pyneng.py` - скрипт, который находится в репозитории заданий.

Где решать задания

Задания надо выполнять в подготовленных файлах. Например, в разделе 04_data_structures есть задание 4.3. Чтобы выполнить его надо открыть файл exercises/04_data_structures/task_4_3.py и выполнять задание прямо в этом файле после описания задания.

Это важно потому что тесты привязаны к тому, что задания выполняются в определенных файлах и в определенной структуре каталогов. Кроме того, что задания надо делать в подготовленных файлах, обязательно скопировать себе весь каталог exercises (а еще лучше весь репозиторий pyneng-examples-exercises), так как тесты зависят от файлов в каталоге exercises, не только от файлов в каталоге конкретных заданий.

Установка скрипта pyneng

Для начала, его надо установить, чтобы не надо было каждый раз писать `python pyneng.py`.

Для установки скрипта, в репозитории должны находиться файлы pyneng.py и setup.py. Если репозиторий создавался после 28 января 2021 из шаблона pyneng-examples-exercises, эти файлы находится в корне репозитория. Иначе, их надо скопировать в свой репозиторий самостоятельно.

Надо перейти в свой репозиторий, например (пишите имя своего репозитория):

```
cd my_repo/
```

Затем внутри репозитория дать команду

```
pip install .
```

Это установит модуль и даст возможность вызывать его в любом каталоге по слову pyneng.

Скрипт pyneng

Этапы работы с заданиями:

1. Выполнение заданий
2. Проверка, что задание отрабатывает как нужно `python task_4_2.py` или запуск скрипта в редакторе/IDE
3. Проверка заданий тестами `pyneng 1-5`
4. Если тесты проходят, смотрим варианты решения `pyneng 1-5 -a`

Примечание: Второй шаг очень важен, потому что на этом этапе намного проще найти ошибки в синтаксисе и подобные проблемы с работой скрипта, чем при запуске кода через тест на 3 этапе.

Скрипт упрощает запуск тестов, так как не надо указывать никакие параметры, по умолчанию вывод настроен на подробный и запускается с плагином pytest-clarity, который улучшает diff при отличиях в решении и правильном решении. Также скрываются некоторые вещи, например, warning которые показывает pytest, чтобы не отвлекать от задачи.

Тесты по-прежнему можно *запускать с помощью pytest*, если вы уже к нему привыкли или ранее использовали. Скрипт pyneng всего лишь обертка вокруг запуска pytest.

Вторая часть работы скрипта - копирование вариантов решения заданий. Эта часть сделана для удобства, чтобы не надо было искать ответы и задумана так, что сначала задание должно пройти тест и только после этого `pyneng -a` отработает и покажет ответы (скопирует их в текущий каталог). Для копирования ответов, скрипт клонирует репозиторий ответов в домашний каталог пользователя, копирует нужные ответы и удаляет репозиторий ответов.

Проверка заданий тестами

После выполнения задания, его надо проверить с помощью тестов. Для запуска тестов, надо вызвать pyneng в каталоге заданий. Например, если вы делаете 4 раздел заданий, надо находиться в каталоге exercises/04_data_structures/ и запустить pyneng одним из способов, в зависимости от того какие задания на проверять.

Запуск проверки всех заданий текущего раздела:

```
pyneng
```

Запуск тестов для задания 4.1:

```
pyneng 1
```

Запуск тестов для заданий 4.1, 4.2, 4.3:

```
pyneng 1-3
```

Если есть задания с буквами, например, в 7 разделе, можно запускать так, чтобы запустить проверку для заданий 7.2a, 7.2b (надо находиться в каталоге 07_files):

```
pyneng 2a-b
```

или так, чтобы запустить все задания 7.2x с буквами и без:

```
pyneng 2*
```

Получение ответов

Если задания проходят тесты, можно посмотреть варианты решения заданий.

Для этого к предыдущим вариантам команды надо добавить -a. Такой вызов значит запустить тесты для заданий 1 и 2 и скопировать ответы, если тесты прошли:

```
pyneng 1-2 -a
```

Тогда для указанных заданий запустятся тесты и для тех заданий из них, которые прошли тесты, скопируются ответы в файлы answer_task_x.py в текущем каталоге.

Вывод pyneng

Warning

В конце вывода теста часто написано «1 warning». Это можно игнорировать, предупреждения в основном связаны с работой каких-то модулей и скрыты чтобы не отвлекать от заданий.

Тесты прошли успешно

Тесты не прошли

Когда какие-то тесты не прошли, в выводе показываются отличия между тем как должен выглядеть вывод и какой вывод был получен.

Отличия показываются как Left и Right, к сожалению тут нет такого что зеленым выделен правильный вариант, а красным неправильный, надо смотреть по ситуации. Каждый раз при выводе отличий, перед ними есть строка вида:

```
assert correct_stdout in out.strip()
```

В этом случае Left это правильный вывод, right вывод задания:

или так:

```
return_value == correct_return_value
```

В этом случае Right это правильный вывод, Left вывод задания:

Проверка заданий с помощью pytest

Предупреждение: Для проверки заданий тестами появилась новая утилита *pyneng*. Она упрощает работу с тестами.

Начиная с раздела «4. Типы данных в Python» для проверки заданий используются автоматические тесты. Они помогают проверить все ли соответствует поставленной задаче, а также дают обратный отклик по тому, что не соответствует задаче. Как правило, после первого периода адаптации к тестам, становится проще делать задания с тестами.

Помимо перечисленных выше положительных моментов, в тестах также можно посмотреть какой итоговый результат нужен: прояснить структуру данных и мелочи, которые могут влиять на результат.

Для запуска тестов используется pytest - фреймворк для написания тестов.

Перед запуском тестов надо установить такие модули:

```
pip install pytest-clarity pyyaml
```

Для корректной работы тестов, надо скопировать себе не только каталог заданий одного раздела, а весь каталог exercises из репозитория с заданиями. А лучше создать себе копию репозитория с заданиями, как написано тут.

Примечание: Запись лекции по использованию pytest для проверки заданий

Основы pytest

Для начала надо установить pytest и pyyaml:

```
pip install pytest
pip install pyyaml
```

Хотя на курсе не надо будет писать тесты, чтобы их понимать, стоит посмотреть на пример теста. Например, есть следующий код с функцией check_ip:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        return True
    except ValueError as err:
        return False


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

Функция check_ip проверяет является ли аргумент, который ей передали, IP-адресом. Пример вызова функции с разными аргументами:

```
In [1]: import ipaddress
   ...:
   ...:
   ...: def check_ip(ip):
   ...:     try:
   ...:         ipaddress.ip_address(ip)
   ...:         return True
   ...:     except ValueError as err:
   ...:         return False
   ...:

In [2]: check_ip('10.1.1.1')
Out[2]: True

In [3]: check_ip('10.1.')
Out[3]: False

In [4]: check_ip('a.a.a.a')
Out[4]: False

In [5]: check_ip('500.1.1.1')
Out[5]: False
```

Теперь необходимо написать тест для функции check_ip. Тест должен проверять, что при передаче корректного адреса, функция возвращает True, а при передаче неправильного аргумента - False.

Чтобы упростить задачу, тест можно написать в том же файле. В pytest, тестом может быть обычная функция, с именем, которое начинается на **test_**. Внутри функции надо написать условия, которые проверяются. В pytest это делается с помощью assert.

assert

assert ничего не делает, если выражение, которое написано после него истинное и генерирует исключение, если выражение ложное:

```
In [6]: assert 5 > 1

In [7]: a = 4

In [8]: assert a in [1,2,3,4]

In [9]: assert a not in [1,2,3,4]
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-9-1956288e2d8e> in <module>
----> 1 assert a not in [1,2,3,4]

AssertionError:

In [10]: assert 5 < 1
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-10-b224d03aab2f> in <module>
----> 1 assert 5 < 1

AssertionError:
```

После assert и выражения можно писать сообщение. Если сообщение есть, оно выводится в исключении:

```
In [11]: assert a not in [1,2,3,4], "а нет в списке"
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-11-7a8f87272a54> in <module>
----> 1 assert a not in [1,2,3,4], "а нет в списке"

AssertionError: а нет в списке
```

Пример теста

pytest использует assert, чтобы указать какие условия должны выполняться, чтобы тест считался пройденным.

В pytest тест можно написать как обычную функцию, но имя функции должно начинаться с **test_**. Ниже написан тест test_check_ip, который проверяет работу функции check_ip, передав ей два значения: правильный адрес и неправильный, а также после каждой проверки написано сообщение:

```
import ipaddress


def check_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


def test_check_ip():
    assert check_ip('10.1.1.1') == True, 'При правильном IP, функция должна возвращать␣
↪True'
    assert check_ip('500.1.1.1') == False, 'Если адрес неправильный, функция должна␣
↪возвращать False'


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

Код записан в файл check_ip_functions.py. Теперь надо разобраться как вызывать тесты. Самый простой вариант, написать слово pytest. В этом случае, pytest автоматически обнаружит тесты в текущем каталоге. Однако, у pytest есть определенные правила, не только по названию функцию, но и по названию файлов с тестами - имена файлов также должны начинаться на **test_**. Если правила соблюдаются, pytest автоматически найдет тесты, если нет - надо указать файл с тестами.

В случае с примером выше, надо будет вызвать такую команду:

```
$ pytest check_ip_functions.py
========================= test session starts =========================
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item
```

(continues on next page)

(продолжение с предыдущей страницы)

```
check_ip_functions.py .                                              [100%]

====================== 1 passed in 0.02 seconds =======================
```

По умолчанию, если тесты проходят, каждый тест (функция test_check_ip) отмечается точкой. Так как в данном случае тест только один - функция test_check_ip, после имени check_ip_functions.py стоит точка, а также ниже написано, что 1 тест прошел.

Теперь, допустим, что функция работает неправильно и всегда возвращает False (напишите return False в самом начале функции). В этом случае, выполнение теста будет выглядеть так:

```
$ pytest check_ip_functions.py
========================= test session starts ==========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

check_ip_functions.py F                                              [100%]

============================== FAILURES ================================
_____________________________ test_check_ip ______________________________

    def test_check_ip():
>       assert check_ip('10.1.1.1') == True, 'При правильном IP, функция должна␣
↪возвращать True'
E       AssertionError: При правильном IP, функция должна возвращать True
E       assert False == True
E        +  where False = check_ip('10.1.1.1')

check_ip_functions.py:14: AssertionError
====================== 1 failed in 0.06 seconds =======================
```

Если тест не проходит, pytest выводит более подробную информацию и показывает в каком месте что-то пошло не так. В данном случае, при выполении строки `assert check_ip('10.1.1.1') == True`, выражение не дало истинный результат, поэтому было сгенерировано исключение.

Ниже, pytest показывает, что именно он сравнивал: `assert False == True` и уточняет, что False - это `check_ip('10.1.1.1')`. Посмотрев на вывод, можно заподозрить, что с функцией check_ip что-то не так, так как она возвращает False на правильном адресе.

Чаще всего, тесты пишутся в отдельных файлах. Для данного примера тест всего один, но он все равно вынесен в отдельный файл.

Файл test_check_ip_function.py:

```
from check_ip_functions import check_ip


def test_check_ip():
    assert check_ip('10.1.1.1') == True, 'При правильном IP, функция должна возвращать␣
↪True'
    assert check_ip('500.1.1.1') == False, 'Если адрес неправильный, функция должна␣
↪возвращать False'
```

Файл check_ip_functions.py:

```
import ipaddress


def check_ip(ip):
    #return False
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError as err:
        return False


if __name__ == "__main__":
    result = check_ip('10.1.1.1')
    print('Function result:', result)
```

В таком случае, тест можно запустить не указывая файл:

```
$ pytest
================= test session starts ========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/general/pyneng.github.io/code_examples/pytest
collected 1 item

test_check_ip_function.py .                                [100%]

================= 1 passed in 0.02 seconds ====================
```

Особенности использования pytest для проверки заданий

На курсе pytest используется, в первую очередь, для самопроверки заданий. Однако, эта проверка не является опциональной - задание считается сделанным, когда оно соблюдает все указанные пункты и проходит тесты. Со своей стороны, я тоже сначала проверяю задания автоматическими тестами, а затем уже смотрю код, пишу комментарии, если нужно и показываю вариант решения.

Поначалу тесты требуют усилий, но через пару разделов, они будут помогать в решении заданий.

> **Предупреждение:** Тесты, которые написаны для заданий курса, не являются эталоном или best practice написания тестов. Тесты написаны с максимальным упором на понятность и многие вещи принято делать по-другому.

При решении заданий, особенно, когда есть сомнения по поводу итогового формата данных, которые должны быть получены, лучше посмтреть в тест. Например, если задание task_9_1.py, то соответствующий тест будет в файле test_task_9_1.py.

Пример теста test_task_9_1.py:

```
import pytest
import task_9_1
import sys
sys.path.append('..')

from common_functions import check_function_exists, check_function_params


# Проверяет создана ли функция generate_access_config в задании task_9_1
def test_function_created():
    check_function_exists(task_9_1, 'generate_access_config')

# Проверяет параметры функции
def test_function_params():
    check_function_params(function=task_9_1.generate_access_config,
                          param_count=2, param_names=['intf_vlan_mapping', 'access_
↪template'])


def test_function_return_value():
    access_vlans_mapping = {
        'FastEthernet0/12': 10,
        'FastEthernet0/14': 11,
        'FastEthernet0/16': 17
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    }
    template_access_mode = [
        'switchport mode access', 'switchport access vlan',
        'switchport nonegotiate', 'spanning-tree portfast',
        'spanning-tree bpduguard enable'
    ]
    correct_return_value = ['interface FastEthernet0/12',
                            'switchport mode access',
                            'switchport access vlan 10',
                            'switchport nonegotiate',
                            'spanning-tree portfast',
                            'spanning-tree bpduguard enable',
                            'interface FastEthernet0/14',
                            'switchport mode access',
                            'switchport access vlan 11',
                            'switchport nonegotiate',
                            'spanning-tree portfast',
                            'spanning-tree bpduguard enable',
                            'interface FastEthernet0/16',
                            'switchport mode access',
                            'switchport access vlan 17',
                            'switchport nonegotiate',
                            'spanning-tree portfast',
                            'spanning-tree bpduguard enable']

    return_value = task_9_1.generate_access_config(access_vlans_mapping, template_access_
↪mode)
    assert return_value != None, "Функция ничего не возвращает"
    assert type(return_value) == list, "Функция должна возвращать список"
    assert return_value == correct_return_value, "Функция возвращает неправильное значение
↪"
```

Обратите внимание на переменную correct_return_value - в этой переменной содержится итоговый список, который должна возвращать функция generate_access_config. Поэтому, если, к примеру, по мере выполнения задания, возник вопрос надо ли добавлять пробелы перед командами или перевод строки в конце, можно посмотреть в тесте, что именно требуется в результате. А также сверить свой вывод, с выводом в переменной correct_return_value.

Как запускать тесты для проверки заданий

Самое главное, это откуда надо запускать тесты: все тесты надо запускать из каталога с заданиями раздела. Например, в разделе 09_functions, такая структура каталога с заданиями:

```
[~/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions]
vagrant: [master|✓]
$ tree
.

├── config_r1.txt
├── config_sw1.txt
├── config_sw2.txt
├── conftest.py
├── task_9_1a.py
├── task_9_1.py
├── task_9_2a.py
├── task_9_2.py
├── task_9_3a.py
├── task_9_3.py
├── task_9_4.py
├── test_task_9_1a.py
├── test_task_9_1.py
├── test_task_9_2a.py
├── test_task_9_2.py
├── test_task_9_3a.py
├── test_task_9_3.py
└── test_task_9_4.py
```

Запускать тесты, в этом случае, надо из каталога 09_functions:

```
[~/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions]
vagrant: [master|✓]
$ pytest test_task_9_1.py
========================= test session starts =========================
platform linux -- Python 3.7.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions
collected 3 items

test_task_9_1.py ...                                          [100%]
...
```

conftest.py

К тестами относится и файл conftest.py - это специальный файл, в котором можно писать функции (а точнее фикстуры) общие для раных тестов. Например, в этот файл вынесены функции, которые подключаются по SSH/Telnet к оборудованию.

pytest.ini

Это конфигурационный файл pytest. В нем можно настроить аргументы вызова pytest. Например, если вы хотите всегда вызывать pytest с -vv, надо написать в pytest.ini:

```
[pytest]
addopts = -vv
```

В подготовленном файле pytest.ini находится такая строка:

```
addopts = -vv --no-hints
```

Это параметр, который нужен модулю pytest-clarity, он описывается ниже.

Полезные команды

Запуск одного теста:

```
$ pytest test_task_9_1.py
```

Запуск одного теста с более подробным выводом информации (показывает diff между данными в тесте и тем, что получено из функции):

```
$ pytest test_task_9_1.py -vv
```

Запуск всех тестов одного раздела:

```
[~/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions]
vagrant: [master|✓]
$ pytest
======================= test session starts ========================
platform linux -- Python 3.6.3, pytest-4.6.2, py-1.5.2, pluggy-0.12.0
rootdir: /home/vagrant/repos/pyneng-7/pyneng-online-may-aug-2019/exercises/09_functions
collected 21 items

test_task_9_1.py ..F                                     [ 14%]
test_task_9_1a.py FFF                                    [ 28%]
test_task_9_2.py FFF                                     [ 42%]
test_task_9_2a.py FFF                                    [ 57%]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_task_9_3.py FFF                                    [ 71%]
test_task_9_3a.py FFF                                   [ 85%]
test_task_9_4.py FFF                                    [100%]

...
```

Запуск всех тестов одного раздела с отображением сообщений об ошибках в одну строку:

```
$ pytest --tb=line
```

pytest-clarity

Плагин pytest-clarity улучшает отображение отличий необходимого результата с решением задания.

Установка:

```
pip install pytest-clarity
```

Плагин pytest-clarity отрабатывает только в том случае, когда тест вызывается с флагом `-vv`. Также по умолчанию у него довольно объемный вывод, поэтому лучше вызывать его с аргументом `--no-hints` (эта опция прописана в подготовленном репозитории в файле pytest.ini):

```
$ pytest test_task_9_3.py -vv --no-hints
====================== test session starts ========================

test_task_9_3.py::test_function_created PASSED               [ 33%]
test_task_9_3.py::test_function_params PASSED                [ 66%]
test_task_9_3.py::test_function_return_value FAILED          [100%]

======================== FAILURES =================================
__________ test_function_return_value ______________________________

...
        access, trunk = return_value
>       assert (
            return_value == correct_return_value
        ), "Функция возвращает неправильное значение"
E       AssertionError: Функция возвращает неправильное значение
E       assert left == right failed.
E         Showing unified diff (L=left, R=right):
E
E          L ({'FastEthernet0/0': '10',
E          R ({'FastEthernet0/0': 10,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
E          L   'FastEthernet0/2': '20',
E          R   'FastEthernet0/2': 20,
E          L   'FastEthernet1/0': '20',
E          R   'FastEthernet1/0': 20,
E          L   'FastEthernet1/1': '30'},
E          R   'FastEthernet1/1': 30},
E            {'FastEthernet0/1': [100, 200],
E             'FastEthernet0/3': [100, 300, 400, 500, 600],
E             'FastEthernet1/2': [400, 500, 600]})

test_task_9_3.py:59: AssertionError
```

Так как агументы `-vv` и `--no-hints` надо постоянно передавать, можно записать их в pytest.ini:

```
[pytest]
addopts = -vv --no-hints
```

10

Продолжение обучения

Как правило, информацию тяжело усвоить с первого раза. Особенно, новую информацию.

Если делать практические задания и пометки, в ходе изучения, то усвоится намного больше информации, чем, если просто читать книгу. Но, скорее всего, в каком-то виде, надо будет читать о той же информации несколько раз.

Книга дает лишь основы Python и поэтому надо обязательно продолжать учиться и повторять уже пройденные темы и изучать новое. И тут есть множество вариантов:

- автоматизировать что-то в работе
- изучать дальше Python для автоматизации работы с сетью
- изучать Python без привязки к сетевому оборудованию

> Тут ресурсы перечислены выборочно, с учетом того, что вы уже прочитали книгу. Но, кроме этого, я сделала подборку ресурсов в которой можно найти и другие материалы.

Написание скриптов для автоматизации рабочих процессов

Скорее всего, после прочтения книги, появятся идеи, что можно автоматизировать на работе. Это отличный вариант, так как на реальной задаче всегда проще учиться и изучать новое. Но лучше не ограничиваться только рабочими задачами и изучать Python дальше.

Python позволяет делать достаточно многое обладая только базовыми знаниями. Поэтому не всегда рабочие задачи позволят принципиально повысить уровень знаний или подтолкнуть к этому, но зная Python лучше, те же задачи можно решать, как правило, намного проще. Поэтому лучше не останавливаться и учиться дальше.

Ниже описаны ресурсы с привязкой к сетевому оборудованию и в целом по Python. В зависимости от того, по каким материалам вы лучше учитесь, можно выбрать книги или видео курсы из списка

Python для автоматизации работы с сетевым оборудованием

Книги:

- Network Programmability and Automation: Skills for the Next-Generation Network Engineer
- Mastering Python Networking (Eric Chou) - отчасти перекликается с тем, что рассматривалось в этой книге, но в ней есть и много новых тем. Плюс, рассматриваются примеры не только на оборудовании Cisco, но Juniper и Arista.

Блоги - позволят быть в курсе новостей в этой сфере:

- Kirk Byers
- Jason Edelman
- Matt Oswalt
- Michael Kashin
- Henry Ölsner
- Mat Wood

У Packet Pushers достаточно часто выходят подкасты об автоматизации:

- Show 176 – Intro To Python & Automation For Network Engineers
- Show 198 – Kirk Byers On Network Automation With Python & Ansible
- Show 270: Design & Build 9: Automation With Python And Netmiko
- Show 332: Don’t Believe The Programming Hype
- Show 333: Automation & Orchestration In Networking
- PQ Show 99: Netmiko & NAPALM For Network Automation

Проекты:

- CiscoConfParse - библиотека, которая парсит конфигурации типа Cisco IOS. С ее помощью можно: проверять существующие конфигурации маршрутизаторов/коммутаторов, получать определенную часть конфигурации, изменять конфигурацию
- NAPALM - NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) это библиотека, которая позволяет работать с сетевым оборудованием разных вендоров, используя унифицированный API

- NOC Project - NOC is the scalable, high-performance and open-source OSS system for ISP, service and content providers
- Requests - библиотека для работы с HTTP
- SaltStack - аналог Ansible
- Scapy - сетевая утилита, которая позволяет манипулировать сетевыми пакетами
- StackStorm - StackStorm is event-driven automation commonly used for auto-remediation, security responses, facilitated troubleshooting, complex deployments, and more
- netdev
- Nornir
- eNMS

Python без привязки к сетевому оборудованию

Книги

Основы:

- Think Python - хорошая книга по основам Python. В книге есть задания.
- Python Crash Course: A Hands-On, Project-Based Introduction to Programming - книга по основам Python. Половина книги посвящена «стандартному» описанию основ Python, а во второй половине эти основы используются для проектов. В книге есть задания.
- Automate the Boring Stuff with Python. На русском - в этой книге можно найти много идей по автоматизации ежедневной работы. Тут рассматриваются такие темы: работа с файлами PDF, Excel, Word, отправка писем, работа с картинками, работа в веб

Среднего/продвинутого уровня:

- Python Tricks - отличный вариант для 2-3 книги по Python. В книге описываются различные аспекты Python и то как правильно использовать. Книга достаточно новая (конец 2017 года), в ней рассматривается Python 3.
- Effective Python: 59 Specific Ways to Write Better Python (Effective Software Development Series) - книга полезных советов как лучше писать код. В конце 2019 года планируется выход второго издания книги.
- Dive Into Python 3 - коротко рассматриваются основы Python, а затем более продвинутые темы: closure, генераторы, тесты и так далее. Книга 2009 года, но рассматривается Python 3 и 99% тем остались без изменений.
- Problem Solving with Algorithms and Data Structures using Python - отличная книга по структурам данных и алгоритмам. Много примеров и домашних заданий. На русском

- Fluent Python - отличная книга по более продвинутым темам. Даже те темы, которые устарели в текущей версии Python (asyncio) стоит прочитать ради прекрасного объяснения темы.
- Python Cookbook - отличная книга рецептов. Рассматривается огромное количество сценариев с решениями и пояснением.

Курсы

- MITx - 6.00.1x Introduction to Computer Science and Programming Using Python - очень хороший курс по Python. Отличный вариант для продолжения обучения после книги. В нём вы и повторите пройденный материал по основам Python, но под другим углом и узнаете много нового. В курсе много практических заданий и он достаточно интенсивный.
- Курсы от Talk Python

Сайты с задачами

- Bites of Py
- HackerRank - на этом сайте задачи разбиты по областям: алгоритмы, регулярные выражения, базы данных и другие. Но есть и базовые задачи
- CheckIO - online game for Python and JavaScript coders

Подкасты

Подкасты позволят в целом расширить кругозор и получить представление о разных проектах, модулях и библиотеках Python:

- Talk Python To Me
- Best Python Podcasts

Документация

- Официальная документация Python
- Python Module of the Week
- Tiny-Python-3.6-Notebook - Отличная шпаргалка по Python 3.6

11

Скачать PDF/Epub

- Epub
- PDF