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 пригодится. 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.

Введение

В Python для работы с регулярными выражениями используется модуль re. Соответственно, для начала работы с регулярными выражениями, надо его импортировать.

В первой половине этого раздела для всех примеров будет использоваться функция search. А в следующих подразделах будут рассматриваться остальные функции модуля re.

Синтаксис функции search такой:

```
match = re.search(regex, string)
```

У функции search два обязательных параметра:

- regex - регулярное выражение
- string - строка, в которой ищется совпадение

Если совпадение было найдено, функция вернет специальный объект Match. Если же совпадения не было, функция вернет None.

При этом, особенность функции search в том, что она ищет только первое совпадение. То есть, если в строке есть несколько подстрок, которые соответствуют регулярному выражению, search вернет только первое найденное совпадение.

Чтобы получить представление о регулярных выражениях, рассмотрим несколько примеров.

Самый простой пример регулярного выражения - подстрока:

```
In [1]: import re

In [2]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [3]: match = re.search('MTU', int_line)
```

В этом примере:

- сначала импортируется модуль re
- затем идет пример строки int_line
- и в 3 строке функции search передается выражение, которое надо искать и строка int_line в которой ищется совпадение

В данном случае, мы просто ищем есть ли подстрока 'MTU' в строке int_line.

Если она есть, в переменной match будет находиться специальный объект Match:

```
In [4]: print(match)
<_sre.SRE_Match object; span=(2, 5), match='MTU'>
```

У объекта Match есть несколько методов, которые позволяют получать разную информацию о полученном совпадении. Например, метод group показывает, что в строке совпало с описанным выражением.

В данном случае, это просто подстрока 'MTU':

```
In [5]: match.group()
Out[5]: 'MTU'
```

Если совпадения не было, в переменной match будет значение None:

```
In [6]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [7]: match = re.search('MU', int_line)

In [8]: print(match)
None
```

Полностью возможности регулярных выражений проявляются при использовании специальных символов. Например, символ `\d` означает цифру, а `+` означает повторение предыдущего символа один или более раз. Если их совместить `\d+`, получится выражение, которое означает одну или более цифр.

Используя это выражение, можно получить часть строки, в которой описана пропускная способность:

```
In [9]: int_line = '  MTU 1500 bytes, BW 10000 Kbit, DLY 1000 usec,'

In [10]: match = re.search('BW \d+', int_line)

In [11]: match.group()
Out[11]: 'BW 10000'
```

Особенно полезны регулярные выражения в получении определенных подстрок из строки. Например, необходимо получить VLAN, MAC и порты из вывода такого лог-сообщения:

```
In [12]: log2 = 'Oct  3 12:49:15.941: %SW_MATM-4-MACFLAP_NOTIF: Host f04d.a206.7fd6 in
 vlan 1 is flapping between port Gi0/5 and port Gi0/16'
```

Это можно сделать с помощью такого регулярного выражения:

```
In [13]: re.search('Host (\S+) in vlan (\d+) is flapping between port (\S+) and port (
\S+)', log2).groups()
Out[13]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Метод groups возвращает только те части исходной строки, которые попали в круглые скобки. Таким образом, заключив часть выражения в скобки, можно указать какие части строки надо запомнить.

Выражение `\d+` уже использовалось ранее - оно описывает одну или более цифр. А выражение `\S+` - описывает все символы, кроме whitespace (пробел, таб и другие).

В следующих подразделах мы разберемся со специальными символами, которые используются в регулярных выражениях.

> Если вы знаете, что означают специальные символы в регулярных выражениях, можно пропустить следующий подраздел и сразу переключиться на подраздел о модуле re.

Синтаксис регулярных выражений

Для работы с регулярными выражениями надо разобраться с тем какие специальные символы поддерживаются.

Предопределенные наборы символов:

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

Символы повторения:

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

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

> Выше перечислены не все специальные символы, которые поддерживает Python. Подробнее в документации

Наборы символов

В Python есть специальные обозначения для наборов символов:

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

> Это не все наборы символов, которые поддерживает Python. Подробнее смотрите в документации.

Наборы символов позволяют писать более короткие выражения, без необходимости перечислять все нужные символы.

Например, получим время из строки лог-файла:

```
In [1]: log = '*Jul  7 06:15:18.695: %LINEPROTO-5-UPDOWN: Line protocol on Interface E
thernet0/3, changed state to down'

In [2]: re.search('\d\d:\d\d:\d\d', log).group()
Out[2]: '06:15:18'
```

Выражение `\d\d:\d\d:\d\d` описывает 3 пары чисел разделенных двоеточиями.

Получение MAC-адреса из лог-сообщения:

```
In [3]: log2 = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [4]: re.search('\w\w\w\w\.\w\w\w\w\.\w\w\w\w', log2).group()
Out[4]: 'f03a.b216.7ad7'
```

Выражение `\w\w\w\w\.\w\w\w\w\.\w\w\w\w` описывает 12 букв или цифр, которые разделены на три группы по четыре символа точками.

Группы символов очень удобны, но пока что, приходится вручную указывать повторение символа. В следующем подразделе рассматриваются символы повторения, которые упростят описание выражений.

Символы повторения

- `regex+` - одно или более повторений предшествующего элемента
- `regex*` - ноль или более повторений предшествующего элемента
- `regex?` - ноль или одно повторение предшествующего элемента
- `regex{n}` - ровно n повторений предшествующего элемента
- `regex{n,m}` - от n до m повторений предшествующего элемента
- `regex{n, }` - n или более повторений предшествующего элемента

+

Плюс указывает, что предыдущее выражение может повторяться сколько угодно раз, но, как минимум, один раз.

Например, тут повторение относится к букве a:

```
In [1]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [2]: re.search('a+', line).group()
Out[2]: 'aa'
```

А в этом выражении, повторяется строка 'a1':

```
In [3]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [4]: re.search('(a1)+', line).group()
Out[4]: 'a1a1'
```

> В выражении `(a1)+`, скобки используются для того чтобы указать, что повторение относится к последовательности символов 'a1'.

IP-адрес можно описать выражением `\d+\.\d+\.\d+\.\d+`. Тут плюс используется чтобы указать, что цифр может быть несколько. А также встречается выражение `\.`.

Оно необходимо из-за того, что точка является специальным символом (она обозначает любой символ). И чтобы указать, что нас интересует именно точка, надо ее экранировать - поместить перед точкой обратный слеш.

Используя это выражение можно получить IP-адрес из строки sh_ip_int_br:

```
In [5]: sh_ip_int_br = 'Ethernet0/1    192.168.200.1   YES NVRAM  up         up'

In [6]: re.search('\d+\.\d+\.\d+\.\d+', sh_ip_int_br).group()
Out[6]: '192.168.200.1'
```

Еще один пример выражения: `\d+\s+\S+` - оно описывает строку, в которой идут цифры, пробел (whitespace), не whitespace символы, то есть все кроме пробела, таба и других whitespace символов. С его помощью можно получить VLAN и MAC-адрес из строки:

```
In [7]: line = '1500     aab1.a1a1.a5d3    FastEthernet0/1'

In [8]: re.search('\d+\s+\S+', line).group()
Out[8]: '1500     aab1.a1a1.a5d3'
```

*

Звездочка указывает, что предыдущее выражение может повторяться 0 или более раз.

Например, если звездочка стоит после символа, она означает повторение этого символа.

Выражение `ba*` означает b, а затем ноль или более повтрений a:

```
In [9]: line = '100     a011.baaa.a5d3    FastEthernet0/1'

In [10]: re.search('ba*', line).group()
Out[10]: 'baaa'
```

Если в строке line до подстроки baaa встретится b, то совпадением будет b:

```
In [11]: line = '100     ab11.baaa.a5d3    FastEthernet0/1'

In [12]: re.search('ba*', line).group()
Out[12]: 'b'
```

Допустим, необходимо написать регулярное выражение, которое описывает email'ы двух форматов: user@example.com и user.test@example.com. То есть в левой части адреса может быть или одно слово или два слова разделенные точкой.

Первый вариант на примере адреса без точки:

```
In [13]: email1 = 'user1@gmail.com'
```

Этот адрес можно описать таким выражением `\w+@\w+\.\w+`:

```
In [14]: re.search('\w+@\w+\.\w+', email1).group()
Out[14]: 'user1@gmail.com'
```

Но такое выражение не подходит email с точкой:

```
In [15]: email2 = 'user2.test@gmail.com'

In [16]: re.search('\w+@\w+\.\w+', email2).group()
Out[16]: 'test@gmail.com'
```

Регулярное выражение для адреса с точкой:

```
In [17]: re.search('\w+\.\w+@\w+\.\w+', email2).group()
Out[17]: 'user2.test@gmail.com'
```

Чтобы описать оба варианта адресов, надо указать, что точка в адресе опциональна:

```
'\w+\.*\w+@\w+\.\w+'
```

Такое регулярное выражение описывает оба варианта:

```
In [18]: email1 = 'user1@gmail.com'

In [19]: email2 = 'user2.test@gmail.com'

In [20]: re.search('\w+\.*\w+@\w+\.\w+', email1).group()
Out[20]: 'user1@gmail.com'

In [21]: re.search('\w+\.*\w+@\w+\.\w+', email2).group()
Out[21]: 'user2.test@gmail.com'
```

?

В последнем примере регулярное выражение указывает, что точка опциональна. Но, в то же время, указывает и то, что точка может появиться много раз.

В этой ситуации, логичней использовать знак вопроса. Он обозначает ноль или одно повторение предыдущего выражения или символа. Теперь регулярное выражение выглядит так `\w+\.?\w+@\w+\.\w+`:

```
In [22]: mail_log = ['Jun 18 14:10:35 client-ip=154.10.180.10 from=user1@gmail.com, si
ze=551',
     ...:             'Jun 18 14:11:05 client-ip=150.10.180.10 from=user2.test@gmail.c
om, size=768']

In [23]: for message in mail_log:
     ...:     match = re.search('\w+\.?\w+@\w+\.\w+', message)
     ...:     if match:
     ...:         print("Found email: ", match.group())
     ...:
Found email:  user1@gmail.com
Found email:  user2.test@gmail.com
```

{n}

С помощью фигурных скобок можно указать сколько раз должно повторяться предшествующее выражение.

Например, выражение `\w{4}\.\w{4}\.\w{4}` описывает 12 букв или цифр, которые разделены на три группы по четыре символа точками. Таким образом можно получить MAC-адрес:

```
In [24]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [25]: re.search('\w{4}\.\w{4}\.\w{4}', line).group()
Out[25]: 'aab1.a1a1.a5d3'
```

В фигурных скобках можно указывать и диапазон повторений. Например, попробуем получить все номера VLAN'ов из строки mac_table:

```
In [26]: mac_table = '''
    ...: sw1#sh mac address-table
    ...:           Mac Address Table
    ...: -------------------------------------------
    ...:
    ...: Vlan    Mac Address       Type        Ports
    ...: ----    -----------       --------    -----
    ...:  100    a1b2.ac10.7000    DYNAMIC     Gi0/1
    ...:  200    a0d4.cb20.7000    DYNAMIC     Gi0/2
    ...:  300    acb4.cd30.7000    DYNAMIC     Gi0/3
    ...: 1100    a2bb.ec40.7000    DYNAMIC     Gi0/4
    ...:  500    aa4b.c550.7000    DYNAMIC     Gi0/5
    ...: 1200    a1bb.1c60.7000    DYNAMIC     Gi0/6
    ...: 1300    aa0b.cc70.7000    DYNAMIC     Gi0/7
    ...: '''
```

Так так search ищет только первое совпадение, в выражение `\d{1,4}` попадет номер VLAN:

```
In [27]: for line in mac_table.split('\n'):
    ...:     match = re.search('\d{1,4}', line)
    ...:     if match:
    ...:         print('VLAN: ', match.group())
    ...:
VLAN:  1
VLAN:  100
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

Выражение `\d{1,4}` описывает от одной до четырех цифр.

Обратите внимание, что в выводе команды нет первого VLAN. Такой результат получился из-за того, что в имени коммутатора есть цифра и она совпала с выражением.

Чтобы исправить это достаточно дополнить выражение и указать, что после цифр должен идти хотя бы один пробел:

```
In [28]: for line in mac_table.split('\n'):
    ...:     match = re.search('\d{1,4} +', line)
    ...:     if match:
    ...:         print('VLAN: ', match.group())
    ...:
VLAN:  100
VLAN:  200
VLAN:  300
VLAN:  1100
VLAN:  500
VLAN:  1200
VLAN:  1300
```

Специальные символы

- `.` - любой символ, кроме символа новой строки
- `^` - начало строки
- `$` - конец строки
- `[abc]` - любой символ в скобках
- `[^abc]` - любой символ, кроме тех, что в скобках
- `a|b` - элемент a или b
- `(regex)` - выражение рассматривается как один элемент. Кроме того, подстрока, которая совпала с выражением, запоминается

.

Точка обозначает любой символ.

Чаще всего, точка используется с символами повторения `+` и `*`, чтобы указать, что между определенными выражениями могут находиться любые символы.

Например, с помощью выражения `Interface.+Port ID.+` можно описать строку с интерфейсами в выводе sh cdp neighbors detail:

```
In [1]: cdp = '''
   ...: SW1#show cdp neighbors detail
   ...: ------------------------
   ...: Device ID: SW2
   ...: Entry address(es):
   ...:   IP address: 10.1.1.2
   ...: Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
   ...: Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
   ...: Holdtime : 164 sec
   ...: '''

In [2]: re.search('Interface.+Port ID.+', cdp).group()
Out[2]: 'Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/
1'
```

В результат попала только одна строка, так как точка обозначает любой символ, кроме символа перевода строки. Кроме того, символы повторения `+` и `*` по умолчанию захватывают максимально длинную строку. Этот аспект рассматривается в подразделе "Жадность символов повторения".

^

Символ `^` означает начало строки. Выражению `^\d+` соответствует подстрока:

```
In [3]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [4]: re.search('^\d+', line).group()
Out[4]: '100'
```

Символы с начала строки и до решетки (включая решетку):

```
In [5]: prompt = 'SW1#show cdp neighbors detail'

In [6]: re.search('^.+#', prompt).group()
Out[6]: 'SW1#'
```

$

Символ `$` обозначает конец строки.

Выражение `\S+$` описывает любые символы, кроме whitespace в конце строки:

```
In [7]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [8]: re.search('\S+$', line).group()
Out[8]: 'FastEthernet0/1'
```

[]

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

```
In [9]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [10]: re.search('[Ff]ast', line).group()
Out[10]: 'Fast'

In [11]: re.search('[Ff]ast[Ee]thernet', line).group()
Out[11]: 'FastEthernet'
```

С помощью квадратных скобок можно указать какие символы могут встречаться на конкретной позиции. Например, выражение `^.+[>#]` описывает символы с начала строки и до решетки или знака больше (включая их). С помощью такого выражения можно получить имя устройства:

```
In [12]: commands = ['SW1#show cdp neighbors detail',
    ...:             'SW1>sh ip int br',
    ...:             'r1-london-core# sh ip route']
    ...:

In [13]: for line in commands:
    ...:     match = re.search('^.+[>#]', line)
    ...:     if match:
    ...:         print(match.group())
    ...:
SW1#
SW1>
r1-london-core#
```

В квадратных скобках можно указывать диапазоны символов. Например, таким образом можно указать, что нас интересует любая цифра от 0 до 9:

```
In [14]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [15]: re.search('[0-9]+', line).group()
Out[15]: '100'
```

Аналогичным образом можно указать буквы:

```
In [16]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [17]: re.search('[a-z]+', line).group()
Out[17]: 'aa'

In [18]: re.search('[A-Z]+', line).group()
Out[18]: 'F'
```

В квадратных скобках можно указывать несколько диапазонов:

```
In [19]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [20]: re.search('[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+', line).group()
Out[20]: 'aa12.35fe.a5d3'
```

Выражение `[a-f0-9]+\.[a-f0-9]+\.[a-f0-9]+` описывает три группы символов, разделенных точкой. Символами в каждой группе могут быть буквы a-f или цифры 0-9. Это выражение описывает MAC-адрес.

Еще одна особенность квадратных скобок - специальные символы внутри квадратных скобок теряют свое специальное значение и обозначают просто символ. Например, точка внутри квадратных скобок будет обозначать точку, а не любой символ.

Выражение `[a-f0-9]+[./][a-f0-9]+` описывает три группы символов:

1. буквы a-f или цифры от 0 до 9
2. точка или слеш
3. буквы a-f или цифры от 0 до 9

Для строки line совпадением будет такая подстрока:

```
In [21]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [22]: re.search('[a-f0-9]+[./][a-f0-9]+', line).group()
Out[22]: 'aa12.35fe'
```

Если после открывающейся квадратной скобки, указан символ `^`, совпадением будет любой символ, кроме указанных в скобках:

```
In [23]: line = 'FastEthernet0/0    15.0.15.1       YES manual up         up'

In [24]: re.search('[^a-zA-Z]+', line).group()
Out[24]: '0/0    15.0.15.1       '
```

В данном случае, выражение описывает все, кроме букв.

|

Вертикальная черта работает как 'или':

```
In [25]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [26]: re.search('Fast|0/1', line).group()
Out[26]: 'Fast'
```

Обратите внимание на то, как срабатывает `|` - Fast и 0/1 воспринимаются как целое выражение. То есть, в итоге выражение означает, что мы ищем Fast или 0/1, а не то, что мы ищем Fas, затем t или 0 и 0/1.

( )

Скобки используются для группировки выражений. Как и в математических выражениях, с помощью скобок можно указать к каким элементам применяется операция.

Например, выражение `[0-9]([a-f]|[0-9])[0-9]` описывает три символа: цифра, потом буква или цифра и цифра:

```
In [27]: line = "100     aa12.35fe.a5d3    FastEthernet0/1"

In [28]: re.search('[0-9]([a-f]|[0-9])[0-9]', line).group()
Out[28]: '100'
```

Скобки позволяют указывать какое выражение является одним целым. Это особенно полезно при использовании символов повторения:

```
In [29]: line = 'FastEthernet0/0    15.0.15.1       YES manual up         up'

In [30]: re.search('([0-9]+\.)+[0-9]+', line).group()
Out[30]: '15.0.15.1'
```

Скобки позволяют не только группировать выражения. Кроме того, строка, которые совпала с выражением в скобках, запоминается. Ее можно получить отдельно с помощью специальных методов groups и group(n). Это рассматривается в подразделе "Группировка выражений".

Жадность символов повторения

По умолчанию, символы повторения в регулярных выражениях жадные (greedy). Это значит, что результирующая подстрока, которая соответствует шаблону, будет наиболее длинной.

Пример жадного поведения:

```
In [1]: import re
In [2]: line = '<text line> some text>'
In [3]: match = re.search('<.*>', line)

In [4]: match.group()
Out[4]: '<text line> some text>'
```

То есть, в данном случае выражение захватило максимально возможный кусок символов, заключенный в <>.

Если нужно отключить жадность, достаточно добавить знак вопроса после символов повторения:

```
In [5]: line = '<text line> some text>'

In [6]: match = re.search('<.*?>', line)

In [7]: match.group()
Out[7]: '<text line>'
```

Зачастую жадность наоборот полезна. Например, без отключения жадности последнего плюса, выражение `\d+\s+\S+` описывает такую строку:

```
In [8]: line = '1500     aab1.a1a1.a5d3    FastEthernet0/1'

In [9]: re.search('\d+\s+\S+', line).group()
Out[9]: '1500     aab1.a1a1.a5d3'
```

Символ `\S` обозначает все, кроме whitespace. Поэтому с жадным символом повторения, выражение `\S+` описывает максимально длинную строку до первого whitespace символа. В данном случае, до первого пробела.

Но если отключить жадность, результат будет таким:

```
In [10]: re.search('\d+\s+\S+?', line).group()
Out[10]: '1500     a'
```

Группировка выражений

Группировка выражений указывает, что последовательность символов надо рассматривать как одно целое. Но это не единственное преимущество группировки.

Кроме этого, с помощью групп можно получать только определенную часть строки, которая была описана выражением. Это очень полезно в ситуациях, когда надо описать строку достаточно подробно, чтобы отобрать нужные строки, но, в то же время, из самой строки надо получить только определенное значение.

Например, из log-файла надо отобрать строки в которых встречается "%SW_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, которые соответствует выражению в группе.

Пример использования групп:

```
In [8]: line = "FastEthernet0/1            10.0.12.1       YES manual up
      up"
In [9]: match = re.search('(\S+)\s+([\w.]+)\s+.*', line)
```

В данном примере указаны две группы:

- первая группа - любые символы, кроме whitespaces
- вторая группа - любая буква или цифра (символ `\w`) или точка

> Вторую группу можно было описать так же, как и первую. Другой вариант сделан просто для примера

Теперь можно обращаться к группам по номеру. Группа 0 это строка, которая соответствует всему шаблону:

```
In [10]: match.group(0)
Out[10]: 'FastEthernet0/1            10.0.12.1       YES manual up
up'

In [11]: match.group(1)
Out[11]: 'FastEthernet0/1'

In [12]: match.group(2)
Out[12]: '10.0.12.1'
```

При необходимости, можно перечислить несколько номеров групп:

```
In [13]: match.group(1, 2)
Out[13]: ('FastEthernet0/1', '10.0.12.1')

In [14]: match.group(2, 1, 2)
Out[14]: ('10.0.12.1', 'FastEthernet0/1', '10.0.12.1')
```

Начиная с версии Python 3.6, к группам можно обращаться таким образом:

```
In [15]: match[0]
Out[15]: 'FastEthernet0/1            10.0.12.1       YES manual up
up'

In [16]: match[1]
Out[16]: 'FastEthernet0/1'

In [17]: match[2]
Out[17]: '10.0.12.1'
```

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

```
In [18]: match.groups()
Out[18]: ('FastEthernet0/1', '10.0.12.1')
```

Именованные группы

Когда выражение сложное, не очень удобно определять номер группы.
Плюс, при дополнении выражения, может получиться так, что порядок групп изменился.
И придется изменить и код, который ссылается на группы.

Именованные группы позволяют задавать группе имя.

Синтаксис именованной группы `(?P<name>regex)` :

```
In [19]: line = "FastEthernet0/1            10.0.12.1       YES manual up
       up"

In [20]: match = re.search('(?P<intf>\S+)\s+(?P<address>[\d.]+)\s+', line)
```

Теперь к этим группам можно обращаться по имени:

```
In [21]: match.group('intf')
Out[21]: 'FastEthernet0/1'

In [22]: match.group('address')
Out[22]: '10.0.12.1'
```

Также очень полезно то, что с помощью метода groupdict(), можно получить словарь, где ключи - имена групп, а значения - подстроки, которые им соответствуют:

```
In [23]: match.groupdict()
Out[23]: {'address': '10.0.12.1', 'intf': 'FastEthernet0/1'}
```

И, в таком случае, можно добавить группы в регулярное выражение и полагаться на их имя, а не на порядок:

```
In [24]:  match = re.search('(?P<intf>\S+)\s+(?P<address>[\d\.]+)\s+\w+\s+\w+\s+(?P<st
atus>up|down|administratively down)\s+(?P<protocol>up|down)', line)

In [25]: match.groupdict()
Out[25]:
{'address': '10.0.12.1',
 'intf': 'FastEthernet0/1',
 'protocol': 'up',
 'status': 'up'}
```

Разбор вывода команды show ip dhcp snooping с помощью именованных групп

Рассмотрим еще один пример использования именованных групп.
В этом примере, задача в том, чтобы получить из вывода команды show ip dhcp snooping binding поля: MAC-адрес, IP-адрес, VLAN и интерфейс.

В файле dhcp_snooping.txt находится вывод команды show ip dhcp snooping binding:

```
MacAddress          IpAddress        Lease(sec)  Type           VLAN  Interface
------------------  ---------------  ----------  -------------  ----  ----------------
----
00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10    FastEthernet0/1
00:04:A3:3E:5B:69   10.1.5.2         63951       dhcp-snooping   5     FastEthernet0/1
0
00:05:B3:7E:9B:60   10.1.5.4         63253       dhcp-snooping   5     FastEthernet0/9
00:09:BC:3F:A6:50   10.1.10.6        76260       dhcp-snooping   10    FastEthernet0/3
Total number of bindings: 4
```

Для начала, попробуем разобрать одну строку:

```
In [1]: line = '00:09:BB:3D:D6:58  10.1.10.2 86250   dhcp-snooping   10  FastEthernet0
/1'
```

В регулярном выражении, именованные группы используются для тех частей вывода, которые нужно запомнить:

```
In [2]: match = re.search('(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P<por
t>\S+)', line)
```

Комментарии к регулярному выражению:

- `(?P<mac>\S+) +` - в группу с именем 'mac' попадают любые символы, кроме whitespace. Получается, что выражение описывает последовательность любых символов, до пробела
- `(?P<ip>\S+) +` - тут аналогично, последовательность любых символов, кроме whitespace, до пробела. Имя группы 'ip'
- `(\d+) +` - числовая последовательность (одна или более цифр), а затем один или более пробелов
  - сюда попадет значение Lease
- `\S+ +` - последовательность любых символов, кроме whitespace
  - сюда попадает тип соответствия (в данном случае, все они dhcp-snooping)

- `(?P<vlan>\d+) +` - именованная группа 'vlan'. Сюда попадают только числовые последовательности, с одним или более символами
- `(?P<int>.\S+)` - именованная группа 'int'. Сюда попадают любые символы, кроме whitespace

В результате, метод groupdict вернет такой словарь:

```
In [3]: match.groupdict()
Out[3]:
{'int': 'FastEthernet0/1',
 'ip': '10.1.10.2',
 'mac': '00:09:BB:3D:D6:58',
 'vlan': '10'}
```

Так как регулярное выражение отработало как нужно, можно создавать скрипт.
В скрипте, перебираются все строки файла dhcp_snooping.txt и на стандартный поток вывода, выводится информация об устройствах.

Файл parse_dhcp_snooping.py:

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

#'00:09:BB:3D:D6:58   10.1.10.2        86250       dhcp-snooping   10    FastEthernet0
/1'
regex = re.compile('(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P<port>\S+)'
)
result = []

with open('dhcp_snooping.txt') as data:
    for line in data:
        match = regex.search(line)
        if match:
            result.append(match.groupdict())

print('К коммутатору подключено {} 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.

Но иногда, скобки нужны для указания части выражения, которое повторяется. И, при этом, не нужно запоминать выражение.

Например, надо получить MAC-адрес, VLAN и порты из такого лог-сообщения:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v
lan 10 is flapping between port Gi0/5 and port Gi0/15'
```

Регулярное выражение, которое описывает нужные подстроки:

```
In [2]: match = re.search('(([0-9a-fA-F]{4}\.){2}[0-9a-fA-F]{4}).+vlan (\d+).+port (\S
+).+port (\S+)', log)
```

Выражение состоит из таких частей:

- `(([0-9a-fA-F]{4}\.){2}[0-9a-fA-F]{4})` - сюда попадет MAC-адрес
  - `[0-9a-fA-F]{4}\.` - эта часть описывает 4 буквы или цифры и точку
  - `([0-9a-fA-F]{4}\.){2}` - тут скобки нужны чтобы указать, что 4 буквы или цифры и точка повторяются два раза
  - `[0-9a-fA-F]{4}` - затем 4 буквы или цифры
- `.+vlan (\d+)` - в группу попадет номер VLAN
- `.+port (\S+)` - первый интерфейс
- `.+port (\S+)` - второй интерфейс

Метод groups вернет такой результат:

```
In [3]: match.groups()
Out[3]: ('f03a.b216.7ad7', 'b216.', '10', 'Gi0/5', 'Gi0/15')
```

Второй элемент по сути лишний. Он попал в вывод из-за скобок в выражении `([0-9a-fA-F]{4}\.){2}`.

В этом случае нужно отключить захват в группе. Это делается добавлением `?:` после открывающейся скобки группы.

Теперь выражение выглядит так:

```
In [4]: match = re.search('((?:[0-9a-fA-F]{4}\.){2}[0-9a-fA-F]{4}).+vlan (\d+).+port (
\S+).+port (\S+)', log)
```

И, соответственно, группы:

```
In [5]: match.groups()
Out[5]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

Повторение захваченного результата

При работе с группами, можно использовать результат, который попал в группу, дальше в этом же выражении.

Например, в выводе sh ip bgp последний столбец описывает атрибут AS Path (через какие автономные системы прошел маршрут):

```
In [1]: bgp = '''
   ...: R9# sh ip bgp | be Network
   ...:     Network          Next Hop       Metric LocPrf Weight Path
   ...: *  192.168.66.0/24  192.168.79.7                      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.

Кроме того, в этом выражении перед строкой регулярного выражения стоит буква r. Это так называемая raw строка.

Тут удобней использовать ее, так как иначе надо будет экранировать обратный слеш, чтобы ссылка на группу сработала корректно:

```
match = re.search('(\d+) \\1', line)
```

> При использовании регулярных выражений, лучше всегда использовать raw строки.

Аналогичным образом можно описать строки, в которых один и тот же номер встречается три раза:

```
In [3]: for line in bgp.split('\n'):
   ...:     match = re.search(r'(\d+) \1 \1', line)
   ...:     if match:
   ...:         print(line)
   ...:
*  192.168.66.0/24  192.168.79.7                       0 500 500 500 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
```

Аналогичным образом можно ссылаться на результат, который попал в именованную группу:

```
In [129]: for line in bgp.split('\n'):
     ...:     match = re.search('(?P<as>\d+) (?P=as)', line)
     ...:     if match:
     ...:         print(line)
     ...:
*  192.168.66.0/24   192.168.79.7                       0 500 500 500 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
```

Модуль re

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

Основные функции модуля **re**:

- `match()` - ищет последовательность в начале строки
- `search()` - ищет первое совпадение с шаблоном
- `findall()` - ищет все совпадения с шаблоном. Выдает результирующие строки в виде списка
- `finditer()` - ищет все совпадения с шаблоном. Выдает итератор
- `compile()` - компилирует регулярное выражение. К этому объекту затем можно применять все перечисленные функции
- `fullmatch()` - вся строка должна соответствовать описанному регулярному выражению

Кроме функций для поиска совпадений, в модуле есть такие функции:

- `re.sub` - для замены в строках
- `re.split` - для разделения строки на части

Объект Match

В модуле re несколько функций возвращают объект Match, если было найдено совпадение:

- search
- match
- finditer возвращает итератор с объектами Match

В этом подразделе рассматриваются методы объекта Match.

Пример объекта Match:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v
lan 10 is flapping between port Gi0/5 and port Gi0/15'

In [2]: match = re.search('Host (\S+) in vlan (\d+) .* port (\S+) and port (\S+)', log
)

In [3]: match
Out[3]: <_sre.SRE_Match object; span=(47, 124), match='Host f03a.b216.7ad7 in vlan 10
is flapping betwee>
```

Вывод в 3 строке просто отображает информацию об объекте. Поэтому не стоит полагаться на то, что отображается в части match, так как отображаемая строка обрезается по фиксированному количеству знаков.

group()

Метод group возвращает подстроку, которая совпала с выражением или с выражением в группе.

Если метод вызывается без аргументов, отображается вся подстрока:

```
In [4]: match.group()
Out[4]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/15'
```

На самом деле в этом случае метод group вызывается с группой 0:

```
In [13]: match.group(0)
Out[13]: 'Host f03a.b216.7ad7 in vlan 10 is flapping between port Gi0/5 and port Gi0/1
5'
```

Другие номера, отображают только содержимое соответствующей группы:

```
In [14]: match.group(1)
Out[14]: 'f03a.b216.7ad7'

In [15]: match.group(2)
Out[15]: '10'

In [16]: match.group(3)
Out[16]: 'Gi0/5'

In [17]: match.group(4)
Out[17]: 'Gi0/15'
```

Если вызвать метод group с номер группы, который больше, чем количество существующих групп, возникнет ошибка:

```
In [18]: match.group(5)
---------------------------------------------------------------
IndexError                         Traceback (most recent call last)
<ipython-input-18-9df93fa7b44b> in <module>()
----> 1 match.group(5)

IndexError: no such group
```

Если вызвать метод с несколькими номерами групп, результатом будет кортеж со строками, которые соответствуют совпадениям:

```
In [19]: match.group(1, 2, 3)
Out[19]: ('f03a.b216.7ad7', '10', 'Gi0/5')
```

В группу может ничего не попасть, тогда ей будет соответствовать пустая строка:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v
lan 10 is flapping between port Gi0/5 and port Gi0/15'

In [34]: match = re.search('Host (\S+) in vlan (\D*)', log)

In [36]: match.group(2)
Out[36]: ''
```

Если группа описывает часть шаблона и совпадений было несколько, метод отобразит последнее совпадение:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v
lan 10 is flapping between port Gi0/5 and port Gi0/15'

In [44]: match = re.search('Host (\w{4}\.)+', log)

In [45]: match.group(1)
Out[46]: 'b216.'
```

Такой вывод получился из-за того, что выражение в скобках описывает 4 буквы или цифры и после этого стоит плюс. Соответственно, сначала с выражением в скобках совпала первая часть MAC-адреса, потом вторая. Но запоминается и возвращается только последнее выражение.

Если в выражении использовались именованные группы, методу group можно передать имя группы и получить соответствующую подстроку:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in v
lan 10 is flapping between port Gi0/5 and port Gi0/15'

In [55]: match = re.search('Host (?P<mac>\S+) '
    ...:                   'in vlan (?P<vlan>\d+) .* '
    ...:                   'port (?P<int1>\S+) '
    ...:                   'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [53]: match.group('mac')
Out[53]: 'f03a.b216.7ad7'

In [54]: match.group('int2')
Out[54]: 'Gi0/15'
```

Но эти же группы доступны и по номеру:

```
In [58]: match.group(3)
Out[58]: 'Gi0/5'

In [59]: match.group(4)
Out[59]: 'Gi0/15'
```

groups()

Метод groups() возвращает кортеж со строками. В котором строки - это те подстроки, которые попали в соответствующие группы:

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [64]: match = re.search('Host (\S+) '
    ...:                   'in vlan (\d+) .* '
    ...:                   'port (\S+) '
    ...:                   'and port (\S+)',
    ...:                   log)
    ...:

In [65]: match.groups()
Out[65]: ('f03a.b216.7ad7', '10', 'Gi0/5', 'Gi0/15')
```

У метода groups есть опциональный параметр - default. Он срабатывает в ситуации, когда все, что попадает в группу опционально.

Например, при такой строке, совпадение будет и в первой группе и во второй:

```
In [76]: line = '100     aab1.a1a1.a5d3    FastEthernet0/1'

In [77]: match = re.search('(\d+) +(\w+)?', line)

In [78]: match.groups()
Out[78]: ('100', 'aab1')
```

Если же в строке нет ничего после пробела, в группу ничего не попадет. Но совпадение будет, так как в регулярном выражении описано, что группа опциональна:

```
In [80]: line = '100     '

In [81]: match = re.search('(\d+) +(\w+)?', line)

In [82]: match.groups()
Out[82]: ('100', None)
```

Соответственно, для второй группы значением будет None.

Но если передать методу groups аргумент, он будет возвращаться вместо None:

```
In [83]: line = '100     '

In [84]: match = re.search('(\d+) +(\w+)?', line)

In [85]: match.groups(0)
Out[85]: ('100', 0)

In [86]: match.groups('No match')
Out[86]: ('100', 'No match')
```

groupdict()

Метод groupdict возвращает словарь, в котором ключи - имена групп, а значения - соответствующие строки:

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [88]: match = re.search('Host (?P<mac>\S+) '
    ...:                   'in vlan (?P<vlan>\d+) .* '
    ...:                   'port (?P<int1>\S+) '
    ...:                   'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [89]: match.groupdict()
Out[89]: {'int1': 'Gi0/5', 'int2': 'Gi0/15', 'mac': 'f03a.b216.7ad7', 'vlan': '10'}
```

start(), end()

Методы start и end возвращают индексы начала и конца совпадения с регулярным выражением.

Если методы вызываются без аргументов, они возвращают индексы для всего совпадения:

```
In [101]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [102]: match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)

In [103]: match.start()
Out[103]: 2

In [104]: match.end()
Out[104]: 42

In [105]: line[match.start():match.end()]
Out[105]: '10     aab1.a1a1.a5d3    FastEthernet0/1'
```

Методам можно передавать номер или имя группы. Тогда они возвращают индексы для этой группы:

```
In [108]: match.start(2)
Out[108]: 9

In [109]: match.end(2)
Out[109]: 23

In [110]: line[match.start(2):match.end(2)]
Out[110]: 'aab1.a1a1.a5d3'
```

Аналогично для именованных групп:

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [88]: match = re.search('Host (?P<mac>\S+) '
    ...:                   'in vlan (?P<vlan>\d+) .* '
    ...:                   'port (?P<int1>\S+) '
    ...:                   'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [9]: match.start('mac')
Out[9]: 52

In [10]: match.end('mac')
Out[10]: 66
```

span()

Метод span возвращает кортеж с индексом начала и конца подстроки. Он работает аналогично методам start, end, но возвращает пару чисел.

Без аргументов метод span возвращает индексы для всего совпадения:

```
In [112]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [113]: match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)

In [114]: match.span()
Out[114]: (2, 42)
```

Но ему также можно передать номер группы:

```
In [115]: line = '  10     aab1.a1a1.a5d3    FastEthernet0/1  '

In [116]: match = re.search('(\d+) +([0-9a-f.]+) +(\S+)', line)

In [117]: match.span(2)
Out[117]: (9, 23)
```

Аналогично для именованных групп:

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in
vlan 10 is flapping between port Gi0/5 and port Gi0/15'

In [88]: match = re.search('Host (?P<mac>\S+) '
    ...:                   'in vlan (?P<vlan>\d+) .* '
    ...:                   'port (?P<int1>\S+) '
    ...:                   'and port (?P<int2>\S+)',
    ...:                   log)
    ...:

In [14]: match.span('mac')
Out[14]: (52, 66)

In [15]: match.span('vlan')
Out[15]: (75, 77)
```

`re.search()`

Функция `search()`:

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

Функция search подходит в том случае, когда надо найти только одно совпадение в строке. Например, когда регулярное выражение описывает всю строку или часть строки.

Рассмотрим пример использования функции search в разборе лог-файла.

В файле log.txt находятся лог-сообщения с информацией о том, что один и тот же MAC слишком быстро переучивается то на одном, то на другом интерфейсе. Одна из причин таких сообщений - петля в сети.

Содержимое файла log.txt:

```
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/
16 and port Gi0/24
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/
16 and port Gi0/24
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/
24 and port Gi0/19
%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping between port Gi0/
24 and port Gi0/16
```

При этом, MAC-адрес может прыгать между несколькими портами. В таком случае очень важно знать с каких портов прилетает MAC. И, если это вызвано петлей, выключить все порты, кроме одного.

Попробуем вычислить между какими портами и в каком VLAN образовалась проблема.

Проверка регулярного выражения с одной строкой из log-файла:

```
In [1]: import re

In [2]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping be
tween port Gi0/16 and port Gi0/24'

In [3]: match = re.search('Host \S+ '
   ...:                   'in vlan (\d+) '
   ...:                   'is flapping between port '
   ...:                   '(\S+) and port (\S+)', log)
   ...:
```

Регулярное выражение разбито на части, для удобства чтения. В нем есть три группы:

- `(\d+)` - описывает номер VLAN
- `(\S+) and port (\S+)` - в это выражение попадают номера портов

В итоге, в группы попали такие части строки:

```
In [4]: match.groups()
Out[4]: ('10', 'Gi0/16', 'Gi0/24')
```

В итоговом скрипте файл log.txt обрабатывается построчно и из каждой строки собирается информация о портах. Так как порты могут дублироваться, сразу добавляем их в множество, чтобы получить подборку уникальных интерфейсов (файл parse_log_search.py):

```
import re

regex = ('Host \S+ '
         'in vlan (\d+) '
         'is flapping between port '
         '(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for line in f:
        match = re.search(regex, line)
        if match:
            vlan = match.group(1)
            ports.add(match.group(2))
            ports.add(match.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат выполнения скрипта такой:

```
$ python parse_log_search.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Обработка вывода show cdp neighbors detail

Попробуем получить параметры устройств из вывода sh cdp neighbors detail.

Пример вывода информации для одного соседа:

```
SW1#show cdp neighbors detail
-------------------------
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP
Interface: GigabitEthernet1/0/16,  Port ID (outgoing port): GigabitEthernet0/1
Holdtime : 164 sec

Version :
Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE S
OFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

advertisement version: 2
VTP Management Domain: ''
Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

Задача получить такие поля:

- имя соседа (Device ID: SW2)
- IP-адрес соседа (IP address: 10.1.1.2)
- платформу соседа (Platform: cisco WS-C2960-8TC-L)
- версию IOS (Cisco IOS Software, C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9, RELEASE SOFTWARE (fc1))

И для удобства, надо получить данные в виде словаря. Пример итогового словаря для коммутатора SW2:

```
{'SW2': {'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L',
         'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9'}}
```

Пример проверяется на файле sh_cdp_neighbors_sw1.txt.

Первый вариант решения (файл parse_sh_cdp_neighbors_detail_ver1.py):

```
import re
from pprint import pprint


def parse_cdp(filename):
    result = {}

    with open(filename) as f:
        for line in f:
            if line.startswith('Device ID'):
                neighbor = re.search('Device ID: (\S+)', line).group(1)
                result[neighbor] = {}
            elif line.startswith('  IP address'):
                ip = re.search('IP address: (\S+)', line).group(1)
                result[neighbor]['ip'] = ip
            elif line.startswith('Platform'):
                platform = re.search('Platform: (\S+ \S+),', line).group(1)
                result[neighbor]['platform'] = platform
            elif line.startswith('Cisco IOS Software'):
                ios = re.search('Cisco IOS Software, (.+), RELEASE', line).group(1)
                result[neighbor]['ios'] = ios

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Тут нужные строки отбираются с помощью метода строк startswith. И в строке, с помощью регулярного выражения получается требуемая часть строки.

В итоге все собирается в словарь.

Результат выглядит так:

```
$ python parse_sh_cdp_neighbors_detail_ver1.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

Все получилось как нужно. Но, с помощью регулярных выражений, эту задачу можно решить более компактно.

Вторая версия решения (файл parse_sh_cdp_neighbors_detail_ver2.py):

```
import re
from pprint import pprint


def parse_cdp(filename):
    regex = ('Device ID: (?P<device>\S+)'
             '|IP address: (?P<ip>\S+)'
             '|Platform: (?P<platform>\S+ \S+),'
             '|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open('sh_cdp_neighbors_sw1.txt') as f:
        for line in f:
            match = re.search(regex, line)
            if match:
                if match.lastgroup == 'device':
                    device = match.group(match.lastgroup)
                    result[device] = {}
                elif device:
                    result[device][match.lastgroup] = match.group(match.lastgroup)

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Пояснения к второму варианту:

- в регулярном выражении описаны все варианты строк через знак или `|`
- без проверки строки, ищется совпадение
- если совпадение найдено, проверяется метод lastgroup
  - метод lastgroup возвращает имя последней именованной группы в регулярном выражении, для которой было найдено совпадение
  - если было найдено совпадение для группы device, в переменную device записывается значение, которое попало в эту группу
  - иначе в словарь записывается соответствие 'имя группы': соответствующее значение

> У этого решения ограничение в том, что подразумевается, что в каждой строке может быть только одно совпадение. И в регулярных выражениях, которые записаны через знак `|`, может быть только одна группа. Это можно исправить, расширив решение.

Результат будет таким же:

```
$ python parse_sh_cdp_neighbors_detail_ver2.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

`re.match()`

Функция `match()`:

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

Функция match отличается от search тем, что match всегда ищет совпадение в начале строки. Например, если повторить пример, который использовался для функции search, но уже с match:

```
In [2]: import re

In [3]: log = '%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18.0156 in vlan 10 is flapping be
tween port Gi0/16 and port Gi0/24'

In [4]: match = re.match('Host \S+ '
   ...:                  'in vlan (\d+) '
   ...:                  'is flapping between port '
   ...:                  '(\S+) and port (\S+)', log)
   ...:
```

Результатом будет None:

```
In [6]: print(match)
None
```

Так получилось из-за того, что match ищет слово Host в начале строки. Но это сообщение находится в середине.

В данном случае, можно легко исправить выражение, чтобы функция match находила совпадение:

```
In [4]: match = re.match('\S+: Host \S+ '
   ...:                  'in vlan (\d+) '
   ...:                  'is flapping between port '
   ...:                  '(\S+) and port (\S+)', log)
   ...:
```

Перед словом Host добавлено выражение `\S+:`. Теперь совпадение будет найдено:

```
In [11]: print(match)
<_sre.SRE_Match object; span=(0, 104), match='%SW_MATM-4-MACFLAP_NOTIF: Host 01e2.4c18
.0156 in >

In [12]: match.groups()
Out[12]: ('10', 'Gi0/16', 'Gi0/24')
```

Пример аналогичен тому, который использовался в функции search, с небольшими изменениями (файл parse_log_match.py):

```
import re

regex = ('\S+: Host \S+ '
         'in vlan (\d+) '
         'is flapping between port '
         '(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for line in f:
        match = re.match(regex, line)
        if match:
            vlan = match.group(1)
            ports.add(match.group(2))
            ports.add(match.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат:

```
$ python parse_log_match.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

re.finditer()

Функция `finditer()`:

- используется для поиска всех не пересекающихся совпадений в шаблоне
- возвращает итератор с объектами Match

Функция finditer отлично подходит для обработки тех команд, вывод которых отображается столбцами. Например, sh ip int br, sh mac address-table и др. В этом случае, его можно применять ко всему выводу команды.

Пример вывода sh ip int br:

```
In [8]: sh_ip_int_br = '''
   ...: R1#show ip interface brief
   ...: Interface              IP-Address      OK? Method Status           Protocol
   ...: FastEthernet0/0        15.0.15.1       YES manual up               up
   ...: FastEthernet0/1        10.0.12.1       YES manual up               up
   ...: FastEthernet0/2        10.0.13.1       YES manual up               up
   ...: FastEthernet0/3        unassigned      YES unset  up               up
   ...: Loopback0              10.1.1.1        YES manual up               up
   ...: Loopback100            100.0.0.1       YES manual up               up
   ...: '''
```

Регулярное выражение для обработки вывода:

```
In [9]: result = re.finditer('(\S+) +'
   ...:                      '([\d.]+) +'
   ...:                      '\w+ +\w+ +'
   ...:                      '(up|down|administratively down) +'
   ...:                      '(up|down)',
   ...:                      sh_ip_int_br)
   ...:
```

В переменной result находится итератор:

```
In [12]: result
Out[12]: <callable_iterator at 0xb583f46c>
```

В итераторе находятся объекты Match:

```
In [16]: groups = []

In [18]: for match in result:
    ...:     print(match)
    ...:     groups.append(match.groups())
    ...:
<_sre.SRE_Match object; span=(103, 171), match='FastEthernet0/0       15.0.15.1
YES manual >
<_sre.SRE_Match object; span=(172, 240), match='FastEthernet0/1       10.0.12.1
YES manual >
<_sre.SRE_Match object; span=(241, 309), match='FastEthernet0/2       10.0.13.1
YES manual >
<_sre.SRE_Match object; span=(379, 447), match='Loopback0             10.1.1.1
YES manual >
<_sre.SRE_Match object; span=(448, 516), match='Loopback100           100.0.0.1
YES manual >
```

Теперь в списке groups находятся кортежи со строками, которые попали в группы:

```
In [19]: groups
Out[19]:
[('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]
```

Аналогичный результат можно получить с помощью генератора списков:

```
In [20]: regex = '(\S+) +([\d.]+) +\w+ +\w+ +(up|down|administratively down) +(up|down
)'

In [21]: result = [match.groups() for match in re.finditer(regex, sh_ip_int_br)]

In [22]: result
Out[22]:
[('FastEthernet0/0', '15.0.15.1', 'up', 'up'),
 ('FastEthernet0/1', '10.0.12.1', 'up', 'up'),
 ('FastEthernet0/2', '10.0.13.1', 'up', 'up'),
 ('Loopback0', '10.1.1.1', 'up', 'up'),
 ('Loopback100', '100.0.0.1', 'up', 'up')]
```

Теперь разберем тот же лог-файл, который использовался в подразделах search и match.

В этом случае, вывод можно не перебирать построчно, а передать все содержимое файла (файл parse_log_finditer.py):

```
import re

regex = ('Host \S+ '
         'in vlan (\d+) '
         'is flapping between port '
         '(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for m in re.finditer(regex, f.read()):
        vlan = m.group(1)
        ports.add(m.group(2))
        ports.add(m.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

> В реальной жизни log-файл может быть очень большим. В таком случае, его лучше обрабатывать построчно.

Вывод будет таким же:

```
$ python parse_log_finditer.py
Петля между портами Gi0/19, Gi0/24, Gi0/16 в VLAN 10
```

Обработка вывода show cdp neighbors detail

С помощью finditer можно обработать вывод sh cdp neighbors detail, так же, как и в подразделе re.search.

Скрипт почти полностью аналогичен варианту с re.search (файл parse_sh_cdp_neighbors_detail_finditer.py):

```
import re
from pprint import pprint


def parse_cdp(filename):
    regex = ('Device ID: (?P<device>\S+)'
             '|IP address: (?P<ip>\S+)'
             '|Platform: (?P<platform>\S+ \S+),'
             '|Cisco IOS Software, (?P<ios>.+), RELEASE')

    result = {}

    with open('sh_cdp_neighbors_sw1.txt') as f:
        match_iter = re.finditer(regex, f.read())
        for match in match_iter:
            if match.lastgroup == 'device':
                device = match.group(match.lastgroup)
                result[device] = {}
            elif device:
                result[device][match.lastgroup] = match.group(match.lastgroup)

    return result

pprint(parse_cdp('sh_cdp_neighbors_sw1.txt'))
```

Теперь совпадения ищутся во всем файле, а не в каждой строке отдельно:

```
    with open('sh_cdp_neighbors_sw1.txt') as f:
        match_iter = re.finditer(regex, f.read())
```

Затем перебираются совпадения:

```
    with open('sh_cdp_neighbors_sw1.txt') as f:
        match_iter = re.finditer(regex, f.read())
        for match in match_iter:
```

Остальное аналогично.

Результат будет таким:

```
$ python parse_sh_cdp_neighbors_detail_finditer.py
{'R1': {'ios': '3800 Software (C3825-ADVENTERPRISEK9-M), Version 12.4(24)T1',
        'ip': '10.1.1.1',
        'platform': 'Cisco 3825'},
 'R2': {'ios': '2900 Software (C3825-ADVENTERPRISEK9-M), Version 15.2(2)T1',
        'ip': '10.2.2.2',
        'platform': 'Cisco 2911'},
 'SW2': {'ios': 'C2960 Software (C2960-LANBASEK9-M), Version 12.2(55)SE9',
         'ip': '10.1.1.2',
         'platform': 'cisco WS-C2960-8TC-L'}}
```

Хотя результат аналогичный, с finditer больше возможностей, так как можно указывать не только то, что должно находиться в нужной строке, но и в строках вокруг.

Например, можно точнее указать какой именно IP-адрес надо взять:

```
Device ID: SW2
Entry address(es):
  IP address: 10.1.1.2
Platform: cisco WS-C2960-8TC-L,  Capabilities: Switch IGMP

...

Native VLAN: 1
Duplex: full
Management address(es):
  IP address: 10.1.1.2
```

Например, если нужно взять первый IP-адрес, можно так дополнить регулярное выражение:

```
regex = ('Device ID: (?P<device>\S+)'
         '|Entry address.*\n +IP address: (?P<ip>\S+)'
         '|Platform: (?P<platform>\S+ \S+),'
         '|Cisco IOS Software, (?P<ios>.+), RELEASE')
```

`re.findall()`

Функция `findall()`:

- используется для поиска всех непересекающихся совпадений в шаблоне
- возвращает:
  - список строк, которые описаны регулярным выражением, если в регулярном выражении нет групп
  - список строк, которые совпали с регулярным выражением в группе, если в регулярном выражении одна группа
  - список кортежей, в которых находятся строки, которые совпали с выражением в группе, если групп несколько

Рассмотрим работу findall на примере вывода команды sh mac address-table:

```
In [2]: mac_address_table = open('CAM_table.txt').read()

In [3]: print(mac_address_table)
sw1#sh mac address-table
          Mac Address Table
-------------------------------------------

Vlan    Mac Address       Type        Ports
----    -----------       --------    -----
 100    a1b2.ac10.7000    DYNAMIC     Gi0/1
 200    a0d4.cb20.7000    DYNAMIC     Gi0/2
 300    acb4.cd30.7000    DYNAMIC     Gi0/3
 100    a2bb.ec40.7000    DYNAMIC     Gi0/4
 500    aa4b.c550.7000    DYNAMIC     Gi0/5
 200    a1bb.1c60.7000    DYNAMIC     Gi0/6
 300    aa0b.cc70.7000    DYNAMIC     Gi0/7
```

Первый пример - регулярное выражение без групп. В этом случае, findall возвращает список строк, которые совпали с регулярным выражением.

Например, с помощью findall можно получить список строк с соответствиями vlan - mac - interface и избавиться от заголовка в выводе команды:

```
In [4]: re.findall('\d+ +\S+ +\w+ +\S+', mac_address_table)
Out[4]:
['100    a1b2.ac10.7000    DYNAMIC     Gi0/1',
 '200    a0d4.cb20.7000    DYNAMIC     Gi0/2',
 '300    acb4.cd30.7000    DYNAMIC     Gi0/3',
 '100    a2bb.ec40.7000    DYNAMIC     Gi0/4',
 '500    aa4b.c550.7000    DYNAMIC     Gi0/5',
 '200    a1bb.1c60.7000    DYNAMIC     Gi0/6',
 '300    aa0b.cc70.7000    DYNAMIC     Gi0/7']
```

Обратите внимание, что findall возвращает список строк, а не объект Match.

Но как только в регулярном выражении появляется группа, findall ведет себя по-другому.

Если в выражении используется одна группа, findall возвращает список строк, которые совпали с выражением в группе:

```
In [5]: re.findall('\d+ +(\S+) +\w+ +\S+', mac_address_table)
Out[5]:
['a1b2.ac10.7000',
 'a0d4.cb20.7000',
 'acb4.cd30.7000',
 'a2bb.ec40.7000',
 'aa4b.c550.7000',
 'a1bb.1c60.7000',
 'aa0b.cc70.7000']
```

При этом, findall ищет совпадение всей строки, но возвращает результат похожий на метод groups() в объекте Match.

Если же групп несколько, findall вернет список кортежей:

```
In [6]: re.findall('(\d+) +(\S+) +\w+ +(\S+)', mac_address_table)
Out[6]:
[('100', 'a1b2.ac10.7000', 'Gi0/1'),
 ('200', 'a0d4.cb20.7000', 'Gi0/2'),
 ('300', 'acb4.cd30.7000', 'Gi0/3'),
 ('100', 'a2bb.ec40.7000', 'Gi0/4'),
 ('500', 'aa4b.c550.7000', 'Gi0/5'),
 ('200', 'a1bb.1c60.7000', 'Gi0/6'),
 ('300', 'aa0b.cc70.7000', 'Gi0/7')]
```

Если такие особенности работы функции findall, мешают получить необходимый результат, то лучше использовать фукнцию finditer. Но иногда такое поведение подходит и удобно использовать.

Пример использования findall в разборе лог-файла (файл parse_log_findall.py):

```
import re

regex = ('Host \S+ '
         'in vlan (\d+) '
         'is flapping between port '
         '(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    result = re.findall(regex, f.read())
    for vlan, port1, port2 in result:
        ports.add(port1)
        ports.add(port2)

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Результат:

```
$ python parse_log_findall.py
Петля между портами Gi0/19, Gi0/16, Gi0/24 в VLAN 10
```

re.compile()

В Python есть возможность заранее скомпилировать регулярное выражение, а затем использовать его. Это особенно полезно в тех случаях, когда регулярное выражение много используется в скрипте.

Использование компилированного выражения может ускорить обработку и, как правило, такой вариант удобней использовать, так как в программе разделяется создание регулярного выражения и его использование. Кроме того, при использовании функции re.compile создается объект RegexObject у которого есть несколько дополнительных возможностей, которых нет в объекте MatchObject.

Для компиляции регулярного выражения используется функция re.compile:

```
In [52]: regex = re.compile('\d+ +\S+ +\w+ +\S+')
```

Она возвращает объект RegexObject:

```
In [53]: regex
Out[53]: re.compile(r'\d+ +\S+ +\w+ +\S+', re.UNICODE)
```

У объекта RegexObject доступны такие методы и атрибуты:

```
In [55]: [ method for method in dir(regex) if not method.startswith('_')]
Out[55]:
['findall',
 'finditer',
 'flags',
 'fullmatch',
 'groupindex',
 'groups',
 'match',
 'pattern',
 'scanner',
 'search',
 'split',
 'sub',
 'subn']
```

Обратите внимание, что у объекта Regex доступны методы search, match, finditer, findall. Это те же функции, которые доступны в модуле глобально, то теперь их надо применять к объекту.

Пример использования метода search:

```
In [67]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [68]: match = regex.search(line)
```

Теперь search надо вызывать как метод объекта regex. И передать как аргумент строку.

Результатом будет объект Match:

```
In [69]: match
Out[69]: <_sre.SRE_Match object; span=(1, 43), match='100    a1b2.ac10.7000    DYNAMIC
     Gi0/1'>

In [70]: match.group()
Out[70]: '100    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

Пример компиляции регулярного выражения и его использования на примере разбора лог-файла (файл parse_log_compile.py):

```
import re

regex = re.compile('Host \S+ '
                   'in vlan (\d+) '
                   'is flapping between port '
                   '(\S+) and port (\S+)')

ports = set()

with open('log.txt') as f:
    for m in regex.finditer(f.read()):
        vlan = m.group(1)
        ports.add(m.group(2))
        ports.add(m.group(3))

print('Петля между портами {} в VLAN {}'.format(', '.join(ports), vlan))
```

Это модифицированный пример с использованием finditer. Тут изменилось описание регулярного выражения:

```
regex = re.compile('Host \S+ '
                   'in vlan (\d+) '
                   'is flapping between port '
                   '(\S+) and port (\S+)')
```

И вызов finditer теперь выполняется как метод объекта regex:

```
    for m in regex.finditer(f.read()):
```

Параметры, которые доступны только при использовании re.compile

При использовании функции re.compile, в методах search, match, findall, finditer и fullmatch появляются дополнительные параметры:

- pos - позволяет указывать индекс в строке, с которого надо начать искать совпадение
- endpos - указывает до какого индекса надо выполнять поиск

Их использование аналогично выполнению среза строки.

Например, таким будет результат без указания параметров pos, endpos:

```
In [75]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [76]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [77]: match = regex.search(line)

In [78]: match.group()
Out[78]: '100    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

В этом случае, указывается начальная позиция поиска:

```
In [79]: match = regex.search(line, 2)

In [80]: match.group()
Out[80]: '00    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

Указание начальной позиции аналогично срезу строки:

```
In [81]: match = regex.search(line[2:])

In [82]: match.group()
Out[82]: '00    a1b2.ac10.7000    DYNAMIC     Gi0/1'
```

И последний пример, с указанием двух индексов:

```
In [90]: line = ' 100    a1b2.ac10.7000    DYNAMIC     Gi0/1'

In [91]: regex = re.compile(r'\d+ +\S+ +\w+ +\S+')

In [92]: match = regex.search(line, 2, 40)

In [93]: match.group()
Out[93]: '00    a1b2.ac10.7000    DYNAMIC     Gi'
```

И аналогичный срез строки:

```
In [94]: match = regex.search(line[2:40])

In [95]: match.group()
Out[95]: '00    a1b2.ac10.7000    DYNAMIC     Gi'
```

В методах match, findall, finditer и fullmatch параметры pos и endpos работают аналогично.

Флаги

При использовании функций или создании скомпилированного регулярного выражения, можно указывать дополнительные флаги, которые влияют на поведение регулярного выражения.

Модуль re поддерживает такие флаги (в скобках короткий вариант обозначения флага):

- re.ASCII (re.A)
- re.IGNORECASE (re.I)
- re.MULTILINE (re.M)
- re.DOTALL (re.S)
- re.VERBOSE (re.X)
- re.LOCALE (re.L)
- re.DEBUG

В этом подразделе, для примера, рассматривается флаг re.DOTALL. Информация об остальных флагах доступна в документации.

re.DOTALL

С помощью регулярных выражений можно работать и с многострочной строкой.

Например, из строки table надо получить только строки с соответствиями VLAN-MAC-interface:

```
In [11]: table = '''
    ...: sw1#sh mac address-table
    ...:           Mac Address Table
    ...: -------------------------------------------
    ...:
    ...: Vlan    Mac Address       Type        Ports
    ...: ----    -----------       --------    -----
    ...:  100    aabb.cc10.7000    DYNAMIC     Gi0/1
    ...:  200    aabb.cc20.7000    DYNAMIC     Gi0/2
    ...:  300    aabb.cc30.7000    DYNAMIC     Gi0/3
    ...:  100    aabb.cc40.7000    DYNAMIC     Gi0/4
    ...:  500    aabb.cc50.7000    DYNAMIC     Gi0/5
    ...:  200    aabb.cc60.7000    DYNAMIC     Gi0/6
    ...:  300    aabb.cc70.7000    DYNAMIC     Gi0/7
    ...: '''
```

Конечно, в этом случае можно разделить строку на части и работать с каждой строкой отдельно.
Но можно получить часть с MAC-адресами и без разделения.

В этом примере нужно вырезать часть вывода, которая содержит соответствия.

В этом выражении описана строка с MAC-адресом:

```
In [12]: m = re.search(' *\d+ +[a-f0-9.]+ +\w+ +\S+', table)
```

В результат попадет первая строка с MAC-адресом:

```
In [13]: m.group()
Out[13]: ' 100    aabb.cc80.7000    DYNAMIC     Gi0/1'
```

Учитывая то, что по умолчанию регулярные выражения жадные, можно получить все соответствия таким образом:

```
In [14]: m = re.search('( *\d+ +[a-f0-9.]+ +\w+ +\S+\n)+', table)

In [15]: print(m.group())
 100    aabb.cc10.7000    DYNAMIC     Gi0/1
 200    aabb.cc20.7000    DYNAMIC     Gi0/2
 300    aabb.cc30.7000    DYNAMIC     Gi0/3
 100    aabb.cc40.7000    DYNAMIC     Gi0/4
 500    aabb.cc50.7000    DYNAMIC     Gi0/5
 200    aabb.cc60.7000    DYNAMIC     Gi0/6
 300    aabb.cc70.7000    DYNAMIC     Gi0/7
```

Тут описана строка с MAC-адресом, перевод строки и указано, что это выражение должно повторяться, как минимум, один раз.

Получается, что в данном случае надо получить все строки, начиная с первого соответствия VLAN-MAC-интерфейс.

Это можно описать таким образом:

```
In [16]: m = re.search(' *\d+ +[a-f0-9.]+ +\w+ +\S+.*', table)

In [17]: print(m.group())
 100    aabb.cc10.7000    DYNAMIC     Gi0/1
```

Пока что, в результате только одна строка, так как по умолчанию точка не включает в себя перевод строки.
Но, если добавить специальный флаг, re.DOTALL, точка будет включать и перевод

строки и в результат попадут все соответствия:

```
In [18]: m = re.search(' *\d+ +[a-f0-9.]+ +\w+ +\S+.*', table, re.DOTALL)

In [19]: print(m.group())
 100    aabb.cc10.7000    DYNAMIC     Gi0/1
 200    aabb.cc20.7000    DYNAMIC     Gi0/2
 300    aabb.cc30.7000    DYNAMIC     Gi0/3
 100    aabb.cc40.7000    DYNAMIC     Gi0/4
 500    aabb.cc50.7000    DYNAMIC     Gi0/5
 200    aabb.cc60.7000    DYNAMIC     Gi0/6
 300    aabb.cc70.7000    DYNAMIC     Gi0/7
```

re.split

Функция split работает аналогично методу split в строках.
Но в функции re.split, можно использовать регулярные выражения, а значит разделять строку на части по более сложным условиям.

Например, строку ospf_route надо разбить на элементы, по пробелам (как в методе str.split):

```
In [1]: ospf_route = 'O     10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/
0'

In [2]: re.split(' +', 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('[ ,]+', 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('[ ,\[\]]+', 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('(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('(?:via|[ ,\[\]])+', ospf_route)
Out[6]: ['O', '10.0.24.0/24', '110/41', '10.0.13.3', '3d18h', 'FastEthernet0/0']
```

re.sub

Функция re.sub работает аналогично методу replace в строках.
Но в функции re.sub, можно использовать регулярные выражения, а значит делать замены по более сложным условиям.

Заменим запятые, квадратные скобки и слово via на пробел в строке ospf_route:

```
In [7]: ospf_route = 'O    10.0.24.0/24 [110/41] via 10.0.13.3, 3d18h, FastEthernet0/0'


In [8]: re.sub('(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(' *(\d+) +'
   ...:              '([a-f0-9]+)\.'
   ...:              '([a-f0-9]+)\.'
   ...:              '([a-f0-9]+) +\w+ +'
   ...:              '(\S+)',
   ...:              r'\1 \2:\3:\4 \5',
   ...:              mac_table))
   ...:

100 aabb:cc10:7000 Gi0/1
200 aabb:cc20:7000 Gi0/2
300 aabb:cc30:7000 Gi0/3
100 aabb:cc40:7000 Gi0/4
500 aabb:cc50:7000 Gi0/5
200 aabb:cc60:7000 Gi0/6
300 aabb:cc70:7000 Gi0/7
```

Регулярное выражение разделено на группы:

- `(\d+)` - первая группа. Сюда попадет номер VLAN
- `([a-f,0-9]+).([a-f,0-9]+).([a-f,0-9]+)` - три следующие группы (2, 3, 4) описывают MAC-адрес
- `(\S+)` - пятая группа. Описывает интерфейс.

Во втором регулярном выражении эти группы используются.
Для того чтобы сослаться на группу, .использует 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:

- Regular Expression HOWTO
- Python 3 Module of the Week. Модуль re

Сайты для проверки регулярных выражений:

- для Python - тут можно указывать и методы search, match, findall и флаги. Пример регулярного выражения. К сожалению, иногда не все выражения воспринимает.
- Еще один сайт для Python - не поддерживает методы, но хорошо работает и отработал те выражения, на которые ругнулся предыдущий сайт. Подходит для однострочного текста отлично. С многострочным надо учитывать, что в питоне будет другая ситуация. Пример регулярного выражения
- regex101

Общие руководства по использованию регулярных выражений:

- Множество примеров использования регулярных выражений от основ до более сложных тем
- Книга Mastering Regular Expressions

Помощь в изучении регулярных выражений:

- Визуализация регулярного выражения
- Regex Crossword

Задания

Все задания и вспомогательные файлы можно скачать в репозитории.

> Если в заданиях раздела есть задания с буквами (например, 5.2a), то можно выполнить сначала задания без букв, а затем с буквами. Задания с буквами, как правило, немного сложнее заданий без букв и развивают/усложняют идею в соответствующем задании без буквы.
>
> Например, если в разделе есть задания: 5.1, 5.2, 5.2a, 5.2b, 5.3, 5.3a. Сначала, можно выполнить задания 5.1, 5.2, 5.3. А затем 5.2a, 5.2b, 5.3a.
>
> Однако, если задания с буквами получается сделать сразу, можно делать их по порядку.

Задание 9.1

Создать скрипт, который будет ожидать два аргумента:

1. имя файла, в котором находится вывод команды show
2. регулярное выражение

В результате выполнения скрипта, на стандартный поток вывода должны быть выведены те строки из файла с выводом команды show, в которых было найдено совпадение с регулярным выражением.

Проверить работу скрипта на примере вывода команды sh ip int br (файл sh_ip_int_br.txt). Например, попробуйте вывести информацию только по интерфейсу FastEthernet0/1.

Пример работы скрипта:

```
$ python task_9_1.py sh_ip_int_br.txt "Fas"
FastEthernet0/0            15.0.15.1       YES manual up                    up
FastEthernet0/1            10.0.12.1       YES manual up                    up
FastEthernet0/2            10.0.13.1       YES manual up                    up
FastEthernet0/3            unassigned      YES unset  up                    down

$ python task_9_1.py sh_ip_int_br.txt "manual"
FastEthernet0/0            15.0.15.1       YES manual up                    up
FastEthernet0/1            10.0.12.1       YES manual up                    up
FastEthernet0/2            10.0.13.1       YES manual up                    up
Loopback0                  10.1.1.1        YES manual up                    up
Loopback100                100.0.0.1       YES manual up                    up

$ python task_9_1.py sh_ip_int_br.txt "up +up"
FastEthernet0/0            15.0.15.1       YES manual up                    up
FastEthernet0/1            10.0.12.1       YES manual up                    up
FastEthernet0/2            10.0.13.1       YES manual up                    up
Loopback0                  10.1.1.1        YES manual up                    up
Loopback100                100.0.0.1       YES manual up                    up
```

> В данном случае, скрипт будет работать как фильтр include в CLI Cisco. Вы можете попробовать использовать регулярные выражения для фильтрации вывода команд show.

Задание 9.1a

Напишите регулярное выражение, которое отобразит строки с интерфейсами 0/1 и 0/3 из вывода sh ip int br.

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

В файле задания нужно написать только регулярное выражение.

Задание 9.1b

Переделайте регулярное выражение из задания 9.1a таким образом, чтобы оно, по-прежнему, отображало строки с интерфейсами 0/1 и 0/3, но, при этом, в регулярном выражении было не более 7 символов (не считая кавычки вокруг регулярного выражения).

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

В файле задания нужно написать только регулярное выражение.

Задание 9.1c

Проверить работу скрипта из задания 9.1 и регулярного выражения из задания 9.1a или 9.1b на выводе sh ip int br из файла sh_ip_int_br_switch.txt.

Если, в результате выполнения скрипта, были выведены не только строки с интерфейсами 0/1 и 0/3, исправить регулярное выражение. В результате, должны выводиться только строки с интерфейсами 0/1 и 0/3.

В файле задания нужно написать только регулярное выражение.

Задание 9.2

Создать функцию return_match, которая ожидает два аргумента:

- имя файла, в котором находится вывод команды show
- регулярное выражение

Функция должна обрабатывать вывод команды show построчно и возвращать список подстрок, которые совпали с регулярным выражением (не всю строку, где было найдено совпадение, а только ту подстроку, которая совпала с выражением).

Проверить работу функции на примере вывода команды sh ip int br (файл sh_ip_int_br.txt). Вывести список всех IP-адресов из вывода команды.

Соответственно, регулярное выражение должно описывать подстроку с IP-адресом (то есть, совпадением должен быть IP-адрес).

Обратите внимание, что в данном случае, мы можем не проверять корректность IP-адреса, диапазоны адресов и так далее, так как мы обрабатываем вывод команды, а не ввод пользователя.

Задание 9.3

Создать функцию parse_cfg, которая ожидает как аргумент имя файла, в котором находится конфигурация устройства.

Функция должна обрабатывать конфигурацию и возвращать IP-адреса и маски, которые настроены на интерфейсах, в виде списка кортежей:

- первый элемент кортежа - IP-адрес
- второй элемент кортежа - маска

Например (взяты произвольные адреса): [('10.0.1.1', '255.255.255.0'), ('10.0.2.1', '255.255.255.0')]

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла config_r1.txt.

Обратите внимание, что в данном случае, мы можем не проверять корректность IP-адреса, диапазоны адресов и так далее, так как мы обрабатываем конфигурацию, а не ввод пользователя.

Задание 9.3a

Переделать функцию parse_cfg из задания 9.3 таким образом, чтобы она возвращала словарь:

- ключ: имя интерфейса
- значение: кортеж с двумя строками:
  - IP-адрес
  - маска

Например (взяты произвольные адреса):

```
{'FastEthernet0/1': ('10.0.1.1', '255.255.255.0'),
 'FastEthernet0/2': ('10.0.2.1', '255.255.255.0')}
```

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла config_r1.txt.

Задание 9.3b

Проверить работу функции parse_cfg из задания 9.3a на конфигурации config_r2.txt.

Обратите внимание, что на интерфейсе e0/1 назначены два IP-адреса:

```
interface Ethernet0/1
 ip address 10.255.2.2 255.255.255.0
 ip address 10.254.2.2 255.255.255.0 secondary
```

А в словаре, который возвращает функция parse_cfg, интерфейсу Ethernet0/1 соответствует только один из них (второй).

Переделайте функцию parse_cfg из задания 9.3a 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 9.4

Создать функцию parse_sh_ip_int_br, которая ожидает как аргумент имя файла, в котором находится вывод команды show

Функция должна обрабатывать вывод команды show ip int br и возвращать такие поля:

- Interface
- IP-Address
- Status
- Protocol

Информация должна возвращаться в виде списка кортежей: [('FastEthernet0/0', '10.0.1.1', 'up', 'up'), ('FastEthernet0/1', '10.0.2.1', 'up', 'up'), ('FastEthernet0/2', 'unassigned', 'up', 'up')]

Для получения такого результата, используйте регулярные выражения.

Проверить работу функции на примере файла sh_ip_int_br_2.txt.

Задание 9.4a

Создать функцию convert_to_dict, которая ожидает два аргумента:

- список с названиями полей
- список кортежей с результатами отработки функции parse_sh_ip_int_br из задания 9.4

Функция возвращает результат в виде списка словарей (порядок полей может быть другой): [{'interface': 'FastEthernet0/0', 'status': 'up', 'protocol': 'up', 'address': '10.0.1.1'}, {'interface': 'FastEthernet0/1', 'status': 'up', 'protocol': 'up', 'address': '10.0.2.1'}]

Проверить работу функции на примере файла sh_ip_int_br_2.txt:

- первый аргумент - список headers
- второй аргумент - результат, который возвращает функции parse_show из прошлого задания.

Функцию parse_sh_ip_int_br не нужно копировать. Надо импортировать или саму функцию, и использовать то же регулярное выражение, что и в задании 9.4, или импортировать результат выполнения функции parse_show.

```
headers = ['interface', 'address', 'status', 'protocol']
```