

# **PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ**

# РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

# 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

# СИНТАКСИС РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

# ФУНКЦИЯ SEARCH

# ФУНКЦИЯ SEARCH

Синтаксис функции search:

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

## ФУНКЦИЯ SEARCH

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`.

## ФУНКЦИЯ `SEARCH`

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.



# ФУНКЦИЯ SEARCH

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 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'>
```

## ФУНКЦИЯ SEARCH

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'
```

## ФУНКЦИЯ SEARCH

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
```

## ФУНКЦИЯ SEARCH

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.

## ФУНКЦИЯ SEARCH

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'
```

## ФУНКЦИЯ SEARCH

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 be
```

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')
```

## ФУНКЦИЯ SEARCH

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).

# СИНТАКСИС РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ



# СИНТАКСИС РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

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

# СИНТАКСИС РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Символы повторения:

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

# СИНТАКСИС РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

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  $b$
- $(\text{regex})$  - выражение рассматривается как один элемент.  
Кроме того, подстрока, которая совпала с выражением, запоминается

# НАБОРЫ СИМВОЛОВ

# НАБОРЫ СИМВОЛОВ

В Python есть специальные обозначения для наборов символов:

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

# НАБОРЫ СИМВОЛОВ

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 st  
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.

# НАБОРЫ СИМВОЛОВ

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 be  
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.

# СИМВОЛЫ ПОВТОРЕНИЯ



# СИМВОЛЫ ПОВТОРЕНИЯ

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

+

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

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

+

Используя это выражение, можно получить IP-адрес из строки sh\_ip\_int\_br:

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

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

```
Out[6]: '192.168.200.1'
```

+

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

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

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

```
Out[8]: '1500      aab1.a1a1.a5d3'
```

\*

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

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

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

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

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

```
Out[10]: 'baaa'
```

\*

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

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

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

```
Out[12]: 'b'
```



\*

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

Первый вариант на примере адреса без точки:

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

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

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

\*

Но такое выражение не подходит для email с точкой:

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

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

```
Out[16]: 'test@gmail.com'
```

\*

Регулярное выражение для адреса с точкой:

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

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

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

\*

Такое регулярное выражение описывает оба варианта:

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

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

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

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

?

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

?

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

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

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

## {N}

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

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

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

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

```
Out[25]: 'aab1.a1a1.a5d3'
```

# {N}

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
...: '''
```



# {N}

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
```

# {N}

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**

# 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'
```

^

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'
```

Λ

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 `<a target="_blank" href="[a-f]|[0-9]">0-9</a>[0-9]` описывает три символа: цифра, потом буква или цифра и цифра:

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

```
In <a target="_blank" href="[a-f]|[0-9]">28]: re.search('[0-9</a>[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]+\.[0-9]+\.[0-9]+)', line).group()
Out[30]: '15.0.15.1'
```

# **ЖАДНОСТЬ СИМВОЛОВ ПОВТОРЕНИЯ**

# 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 <>.

# ЖАДНОСТЬ СИМВОЛОВ ПОВТОРЕНИЯ

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>'
```



## ЖАДНОСТЬ СИМВОЛОВ ПОВТОРЕНИЯ

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'
```

## ЖАДНОСТЬ СИМВОЛОВ ПОВТОРЕНИЯ

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

Но если отключить жадность, результат будет таким:

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

# ГРУППИРОВКА ВЫРАЖЕНИЙ

# ГРУППИРОВКА ВЫРАЖЕНИЙ

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.

# ГРУППИРОВКА ВЫРАЖЕНИЙ

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"
In [9]: match = re.search('(\S+)\s+(\[\w.\]+\)\s+.*', line)
```

## НУМЕРОВАННЫЕ ГРУППЫ

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'

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

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

# НУМЕРОВАННЫЕ ГРУППЫ

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')
```



# НУМЕРОВАННЫЕ ГРУППЫ

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'

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

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

## НУМЕРОВАННЫЕ ГРУППЫ

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)
```

## ИМЕНОВАННЫЕ ГРУППЫ

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'
```

## ИМЕНОВАННЫЕ ГРУППЫ

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'}
```

## ИМЕНОВАННЫЕ ГРУППЫ

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

```
In [24]: match = re.search('(P<intf>\S+)\s+(P<address>[\d\.]+\s+\w+\s+\w+\s+(P<status>up|down|administrat

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**

# 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('(?P<mac>\S+) +(?P<ip>\S+) +\d+ +\S+ +(?P<vlan>\d+) +(?P<port>\S+)', line)
```

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'}  
}
```

## 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}: {}'.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**

## **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.

## ГРУППА БЕЗ ЗАХВАТА

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 bet
```

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)
```



# ГРУППА БЕЗ ЗАХВАТА

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

## ГРУППА БЕЗ ЗАХВАТА

Метод `groups` вернет такой результат:

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

## ГРУППА БЕЗ ЗАХВАТА

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.

## ГРУППА БЕЗ ЗАХВАТА

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**

## 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      0 700 700 700 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      0 800 800 i
...: ''
```

# ПОВТОРЕНИЕ ЗАХВАЧЕННОГО РЕЗУЛЬТАТА

Допустим, надо получить те префиксы, у которых в пути несколько раз повторяется один и тот же номер AS.

Это можно сделать с помощью ссылки на результат, который был захвачен группой. Например, такое выражение отображает все строки, в которых один и тот же номер повторяется хотя бы два раза:

```
In [2]: for line in bgp.split('\n'):
...:     match = re.search(r'(\d+) \1', line)
...:     if match:
...:         print(line)
...:
* 192.168.66.0/24 192.168.79.7 0 500 500 500 i
* 192.168.67.0/24 192.168.79.7 0 700 700 700 i
* 192.168.88.0/24 192.168.79.7 0 700 700 700 i
*> 192.168.89.8 0 800 800 i
```

## ПОВТОРЕНИЕ ЗАХВАЧЕННОГО РЕЗУЛЬТАТА

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.



## ПОВТОРЕНИЕ ЗАХВАЧЕННОГО РЕЗУЛЬТАТА

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.

## ПОВТОРЕНИЕ ЗАХВАЧЕННОГО РЕЗУЛЬТАТА

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 0 700 700 700 i
* 192.168.88.0/24 192.168.79.7 0 700 700 700 i
```

## ПОВТОРЕНИЕ ЗАХВАЧЕННОГО РЕЗУЛЬТАТА

Аналогичным образом можно сослаться 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 0 700 700 700 i
* 192.168.88.0/24 192.168.79.7 0 700 700 700 i
*> 192.168.89.8 0 800 800 i
```