# PYTHON ДЛЯ СЕТЕВЫХ ИНЖЕНЕРОВ

# РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

# МОДУЛЬ RE

# 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

# МОДУЛЬ RE

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

# ОБЪЕКТ MATCH

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

- search
- match
- finditer возвращает итератор с объектами Match

# ОБЪЕКТ MATCH

Пример объекта Match:

```
In [1]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in vlan 10 is flapping bet

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'
```

# GROUP()

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 Gi0/15'
```

# GROUP()

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'
```

# GROUP()

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

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

IndexError: no such group
```

# 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')
```

# GROUP()

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 vlan 10 is flapping bet

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

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

# GROUP()

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 vlan 10 is flapping bet

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.

# GROUP()

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 vlan 10 is flapping bet

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'
```

# GROUP()

Но эти же группы доступны и по номеру:

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

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

# GROUPS()

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

```
In [63]: log = 'Jun  3 14:39:05.941: %SW_MATM-4-MACFLAP_NOTIF: Host f03a.b216.7ad7 in vlan 10 is flapping be

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')
```

# GROUPS()

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')
```

# GROUPS()

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.

# GROUPS()

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 be

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'
```

# START(), END()

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'
```

# START(), END()

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 be

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)
```

# SPAN()

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)
```

# SPAN()

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 be

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()`

# `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.

# re.search()

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
```

# `re.search()`

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.

# re.search()

Проверка регулярного выражения с одной строкой из log-файла:

```
In [1]: import re

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

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

# `re.search()`

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

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

# re.search()

В итоге, в группы попали такие части строки:

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

# re.search()

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))
```

# re.search()

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 SOFTWARE (fc1)
Technical Support: http://www.cisco.com/techsupport
Copyright (c) 1986-2014 by Cisco Systems, Inc.
Compiled Mon 03-Mar-14 22:53 by prod_rel_team

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

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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))

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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'}}
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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'))
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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'}}
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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'))
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

Пояснения ко второму варианту:

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

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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.

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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()`

# `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

# re.match()

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

```
In [2]: import re

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

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

# `re.match()`

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.

# re.match()

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)
   ...:
```

# re.match()

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')
```

# re.match()

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))
```

# re.match()

Результат:

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

# `re.finditer()`

# `re.finditer()`

Функция `finditer()`:

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

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

# re.finditer()

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
   ...: '''
```

# re.finditer()

Регулярное выражение для обработки вывода:

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

# re.finditer()

В переменной result находится итератор:

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

# re.finditer()

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 >
```

# re.finditer()

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')]
```

# re.finditer()

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')]
```

# re.finditer()

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))
```

# `re.finditer()`

*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

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'))
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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:
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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'}}
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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.

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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
```

# ОБРАБОТКА ВЫВОДА SHOW CDP NEIGHBORS DETAIL

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()`

# `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

# re.findall()

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
```

# re.findall()

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']
```

# `re.findall()`

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

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

# re.findall()

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.

# re.findall()

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.

# re.findall()

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))
```

# re.findall()

Результат:

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

# `re.compile()`

# `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.

# re.compile()

Для компиляции регулярного выражения используется функция re.compile:

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

# `re.compile()`

Она возвращает объект RegexObject:

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

# re.compile()

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']
```

# re.compile()

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

Пример использования метода search:

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

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

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

# re.compile()

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'
```

# re.compile()

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))
```

# re.compile()

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.

# ПАРАМЕТРЫ, КОТОРЫЕ ДОСТУПНЫ ТОЛЬКО ПРИ ИСПОЛЬЗОВАНИИ RE.COMPILE

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'
```

# ПАРАМЕТРЫ, КОТОРЫЕ ДОСТУПНЫ ТОЛЬКО ПРИ ИСПОЛЬЗОВАНИИ RE.COMPILE

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'
```

# ПАРАМЕТРЫ, КОТОРЫЕ ДОСТУПНЫ ТОЛЬКО ПРИ ИСПОЛЬЗОВАНИИ RE.COMPILE

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'
```

# ПАРАМЕТРЫ, КОТОРЫЕ ДОСТУПНЫ ТОЛЬКО ПРИ ИСПОЛЬЗОВАНИИ RE.COMPILE

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'
```

# ПАРАМЕТРЫ, КОТОРЫЕ ДОСТУПНЫ ТОЛЬКО ПРИ ИСПОЛЬЗОВАНИИ RE.COMPILE

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

# 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

# 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
    ...: '''
```

# RE.DOTALL

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'
```

# RE.DOTALL

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
```

# RE.DOTALL

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
```

# RE.DOTALL

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

# 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']
```

# RE.SPLIT

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']
```

# RE.SPLIT

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']
```

# RE.SPLIT

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']
```

# RE.SPLIT

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

# 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'
```

# RE.SUB

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
```

# RE.SUB

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.

# RE.SUB

Во втором регулярном выражении эти группы используются.
Для того, чтобы сослаться на группу, используется обратный слеш и номер группы.
Чтобы не пришлось экранировать обратный слеш, используется raw строка.

В итоге вместо номеров групп будут подставлены соответствующие подстроки.
Для примера, также изменен формат записи MAC-адреса.