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

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

МОДУЛЬ RE

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

SEARCH()

Функция search():

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

SEARCH()

Поиск подстроки в строке:

```
In [1]: import re

In [2]: line = '00:09:BB:3D:D6:58   10.1.10.2    86250   dhcp-snooping   10    FastEthernet0/1'

In [3]: print(re.search('dhcp', line))
<_sre.SRE_Match object; span=(41, 45), match='dhcp'>

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

SEARCH()

Из объекта Match можно получить несколько вариантов полезной информации.

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

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

In [6]: match.span()
Out[6]: (41, 45)

In [7]: line[41:45]
Out[7]: 'dhcp'
```

SEARCH()

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

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

SEARCH()

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

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

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

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

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

FINDALL()

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

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

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

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

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

FINDITER()

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

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

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

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

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

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

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

FINDITER()

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

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

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

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

COMPILE()

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

Пример компиляции регулярного выражения и его использования:

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

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

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

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

Полностью возможности регулярных выражений проявляются при использовании специальных символов.

Специальные символы:

- . - любой символ, кроме символа новой строки (опция m позволяет включить и символ новой строки)
- ^ - начало строки
- $ - конец строки
- [abc] - любой символ в скобках
- [^abc] - любой символ, кроме тех, что в скобках
- a|b - элемент a или b
- (regex) - выражение рассматривается как один элемент. Текст, который совпал с выражением, запоминается

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

Повторение:

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

Предопределенные наборы символов:

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

Посмотрим на примеры использования специальных символов:

```
In [1]: import re

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

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

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

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

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

Выражению ^.a соответствует подстрока 'Fa':

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

Символ $ обозначает конец строки:

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

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

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

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

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

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

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

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

Вертикальная черта работает как 'или':

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

ЖАДНОСТЬ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

ЖАДНОСТЬ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

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

Пример жадного поведения:

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

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

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

ЖАДНОСТЬ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

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>'
```

ГРУППИРОВКА ВЫРАЖЕНИЙ

НУМЕРОВАННЫЕ ГРУППЫ

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

Группа определяется помещением выражения в круглые скобки ().

Внутри выражения, группы нумеруются слева направо, начиная с 1. Затем к группам можно обращаться по номерам и получать текст, которые соответствует выражению в группе.

НУМЕРОВАННЫЕ ГРУППЫ

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. Эта группа не жадная
- вторая группа - любая буква или цифра (символ \w) или точка. Эта группа не жадная

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                    up'

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

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

НУМЕРОВАННЫЕ ГРУППЫ

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

```
In [13]: match[0]
Out[13]: 'FastEthernet0/1            10.0.12.1       YES manual up                    up'

In [14]: match[1]
Out[14]: 'FastEthernet0/1'

In [15]: match[2]
Out[15]: '10.0.12.1'
```

НУМЕРОВАННЫЕ ГРУППЫ

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

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

ИМЕНОВАННЫЕ ГРУППЫ

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

Именованные группы позволяют задавать группе имя.

ИМЕНОВАННЫЕ ГРУППЫ

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

```
In [14]: line = "FastEthernet0/1            10.0.12.1       YES manual up                    up"

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

Теперь к этим группам можно обращаться по имени:

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

In [16]: match.group('address')
Out[16]: '10.0.12.1'
```

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 [17]: match.groupdict()
Out[17]: {'address': '10.0.12.1', 'intf': 'FastEthernet0/1'}
```

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

```
In [18]: match = re.search('(?P<intf>\S+?)\s+(?P<address>[\w.]+?)\s+(?P<status>up|down|administratively down

In [19]: match.groupdict()
Out[19]: {'address': 'manual', 'intf': 'YES', '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

В этом примере, задача в том, чтобы получить из вывода команды show ip dhcp snooping binding поля: MAC-адрес, IP-адрес, VLAN и интерфейс.

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

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

Для начала, попробуем разобрать одну строку:

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

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

```
In [2]: match = re.search(r'(?P<mac>.+?) +(?P<ip>.*?) +(\d+) +([\w-]+) +(?P<vlan>\d+) +(?P<int>.*$)', line)
```

Комментарии к регулярному выражению (что попадет в группу):

- (?P<mac>.+?) + - любые символы, до пробела
- (?P<ip>.*?) + - любые символы, до пробела
- (\d+) + - одна или более цифр
- ([\w-]+) + - буквы или -, в количестве одного или более
- (?P<vlan>\d+) + - одна или более цифр
- (?P<int>.*$) - любые символы, которые находятся в конце строки

Обратите внимание, что для первых двух групп элементов отключена жадность.

Для остальных жадность можно не отключать, так как в них более четко указаны какие именно символы должны быть.

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

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

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

print("К коммутатору подключено {} 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
```

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

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.

```
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

Если указать то же выражение с помощью круглых скобок, в итоговый список попадут и разделители.

```
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.

```
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.

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

RE.SUB

```
In [10]: print(re.sub(' *(\d+) +([a-f0-9]+)\.([a-f0-9]+)\.([a-f0-9]+) +\w+ +(\S+)', r'\1 \2:\3:\4 \5', mac_t

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.DOTALL

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

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

```
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
```