

Регулярные выражения в Python

Регулярные выражения

Регулярное выражение это специальная последовательность из обычных и специальных символов. Эта последовательность задает шаблон, который позже используется для обработки подстрок.

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

re.search()

Попробуем найти подстроку, которая соответствует шаблону, в указанной строке. Пока что, мы не используем специальные символы и в роли шаблона у нас простая строка 'dhcp':

```
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 at 0x10edeb9f0>
```

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

Функция search():

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

re.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]: (49, 53)
```

```
In [7]: line[49:53]  
Out[7]: 'dhcp'
```

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

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

re.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)
```

re.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 мы использовали выше в функции `search()`.

re.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'
```

re.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
```


re.compile()

Кроме вышеперечисленных функций, 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 использование:

```
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
```

Функции для работы с регулярными выражениями

- `match()` - последовательность ищется 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

Специальные символы

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

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

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

Жадность регулярных выражений

По умолчанию, регулярное выражение, 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 же нам 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>'
```

Обратите внимание, что жадность касается именно символов повторения

Замена подстроки с помощью re

С помощью регулярных выражений можно выполнять замену в строках.

Простой пример замены:

```
In [24]: line = "cat's are great, cat's cat's cat's"
```

```
In [25]: regex = re.compile('(cat)')
```

```
In [26]: print regex.sub('dog', line)  
dog's are great, dog's dog's dog's
```

Split строк с помощью re

В ситуациях, когда разделитель, используя который, надо разбить строки на части, сложный и не получается разбить строку стандартными методами, можно использовать функцию `split()`.

В данном случае, разделитель - это два пробела или больше.

```
In [27]: line = 'Text\n in one line.\n\nSecond  
line\n\n\nAnother line\n\n\nThe End'
```

```
In [28]: re.split(r'\n{2,}', line)  
Out[28]: ['Text\n in one line.', 'Second line', 'Another  
line', 'The End']
```

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 к ним по номерам и получать текст, которые соответствует выражению в группе.

Пример использования групп:

```
In [8]: line = '<text line> some text <12343>'
In [9]: match = re.search('(<.*?>).* (<\d+>)', line)
```

В данном примере у нас указаны две группы:

- первая группа - это любые символы заключенные в <>. Эта группа не жадная
- вторая группа - числа, заключенные в <>

Нумерованные группы

```
In [8]: line = '<text line> some text <12343>'
In [9]: match = re.search('(<.*?>).* (<\d+>)', line)
```

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

```
In [10]: match.group(0)
Out[10]: '<text line> some text <12343>'
```

```
In [11]: match.group(1)
Out[11]: '<text line>'
```

```
In [12]: match.group(2)
Out[12]: '<12343>'
```

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

```
In [13]: match.groups()
Out[13]: ('<text line>', '<12343>')
```

Именованные группы

Когда выражение сложное, становится сложнее определять номер группы, плюс, при дополнении выражения, может получиться так, что порядок групп изменился. И тогда надо будет изменить и код, который ссылается на группы.

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

```
In [12]: line = '<text line> some text <12343>'
```

```
In [13]: re.search('(P<text><.*?>).* (P<number><\d+>)', line)
```

```
In [14]: match = re.search('(P<text><.*?>).* (P<number><\d+>)', line)
```

Именованные группы

```
In [12]: line = '<text line> some text <12343>'
```

```
In [13]: re.search('(?P<text><.*?>).* (?P<number><\d+>)', line)
```

```
In [14]: match = re.search('(?P<text><.*?>).* (?P<number><\d+>)', line)
```

Теперь к этим группам можно обращаться по имени:

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

```
Out[15]: '<text line>'
```

```
In [16]: match.group('number')
```

```
Out[16]: '<12343>'
```

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]: {'number': '<12343>', 'text': '<text line>'}
```

Пример использования регулярных выражений

Пример использования регулярных выражений

У нас есть файл dhcp_snooping.txt с выводом 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 [18]: 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 мы хотим позже использовать:

```
In [22]: match = re.search(r'(?P<mac>.+?) +(?P<ip>.*?) +(\d+) +
([\w-]+) +(?P<vlan>\d+) +(?P<int>.*$)', line)
```

Пример использования регулярных выражений

```
In [22]: match = re.search(r'(?P<mac>.+?) +(?P<ip>.*?) +(\d+) +([\w-]+) +(?P<vlan>\d+) +(?P<int>.*$)', line)
```

- `(?P<mac>.+?) +` - в группу с именем 'mac' попадают любые символы, но, так как выражение не жадное, а следом указано, что должны идти пробелы (один или более), то на выходе мы получаем последовательность символов, до пробела
- `(?P<ip>.*?) +` - тут аналогично, последовательность любых символов до пробела. Имя группы 'ip'
- `(\d+) +` - числовая последовательность (1 или более цифр), а затем один или более пробелов
 - сюда попадет значение Lease
- `([\w-]+) +` - буквы или '-', в количестве 1 или более
 - 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>.*$)'` - именованная группа 'int'. Сюда попадают любые символы, которые находятся в конце строки (в предыдущем выражении эта группа ограничена пробелом)

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.

Пример использования регулярных выражений

```
In [22]: match = re.search(r'(?P<mac>.+?) +(?P<ip>.*?) +  
(\d+) +([\w-]+) +(?P<vlan>\d+) +(?P<int>.*$)', line)
```

В результате мы получим такой словарь:

```
In [23]: match.groupdict()  
Out[23]:  
{ 'int': 'FastEthernet0/1',  
  'ip': '10.1.10.2',  
  'mac': '00:09:BB:3D:D6:58',  
  'vlan': '10' }
```

Пример использования регулярных выражений

Теперь переберем аналогично все строки файла dhcp_snooping.txt в скрипте parse_dhcp_snooping.py и выведем в нормальном виде информацию об устройствах.

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 %d устройства" % len(result)
```

```
for num, comp in enumerate(result, 1):
```

```
    print "Параметры устройства %s:" % num
```

```
    for key in comp:
```

```
        print "\t%s:\t%s" % (key, comp[key])
```


Пример использования регулярных выражений

`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

Python для сетевых инженеров

Автор курса: Наташа Самойленко
nataliya.samoylenko@gmail.com