LeoMat 20 марта 2011 в 19:46

Регулярные выражения, пособие для нови

Автор оригинала: A. M. Kuchling

Python

Перевод

В первой половине этого пособия мы раскрыли лишь малую часть возможностей регуля
половине мы рассмотрим некоторые новые метасимволы, то, как использовать группы д
текста, разбивать строки, находить и замещать части текста. В конце немного поговорим

Больше метасимволов

Есть некоторые метасимволы, которые мы еще не изучили. Большинство из них будут ра

Некоторые из оставшихся метасимволов являются *утверждениями нулевого размера*. Он
строке, они вообще не охватывают никаких символов, возможен просто успех или неуда
том, что текущая позиция находится на границе (*boundary*) слова, при этом сам символ \
что утверждения нулевого размера никогда не должны повторяться, потому что, если он
они, очевидно, будут соответствовать этому месту бесконечное число раз.

|
Соответствует оператору ИЛИ. Если A и B являются регулярными выражениями, то `A|B` б
которая соответствует A или B. Метасимвол | имеет очень низкий приоритет для того, чт
когда вы чередуете несколько символов строки. Crow|Servo будет искать соответствие либо `Crow`, либо `Servo`, а не `Cro('w' или 'S')ervo`.

^
Ищет соответствие только в начале строки. Если включен флаг `MULTILINE`, как говорилось в прошлой части, то происходит сравнение и для каждой части после символа новой строки.

Например, если вы хотите найти только те строки, у которых в начале имеется `From`, то в регулярном выражении записывается `^From`:

```
>>> print re.search('^From', 'From Here to Eternity')
<_sre.SRE_Match object at 0x...>
>>> print re.search('^From', 'Reciting From Memory')
None
```

$
То же, что ^, но в конце строки, которая определяется либо, собственно по концу строки как таковому, либо по символу новой строки.

```
>>> print re.search('}$', '{block}')
<_sre.SRE_Match object at 0x...>
>>> print re.search('}$', '{block} ')
```

```
None
>>> print re.search('}$', '{block}\n')
<_sre.SRE_Match object at 0x...>
```

`\A`
Совпадение только в начале строки, то есть тоже, что `^`, но не зависит от флага `MULTILINE`

`\Z`
Совпадение только в конце строки, то есть тоже, что `$`, но не зависит от флага `MULTILINE`

`\b`
Граница слова. Слово определяется как последовательность символов чисел и/или букв, так что границы слова представляют пробелы или любые символы, не относящиеся к перечисленным.

Следующий пример ищет слово `class` только когда это отдельное слово. Если оно содержится внутри другого слова, соответствия не находится:

```
>>> p = re.compile(r'\bclass\b')
>>> print p.search('no class at all')
<_sre.SRE_Match object at 0x...>
>>> print p.search('the declassified algorithm')
None
>>> print p.search('one subclass is')
None
```

Есть две тонкости, которые вы должны помнить при использовании этой специальной последовательности. Во-первых, это одно из худших столкновений между строковыми литералами Python и последовательностями регулярных выражений: в строковых литералах Python, `\b` это символ backspace, ASCII значение 8. Если не использовать «сырые» строки, Python будет конвертировать `\b` в backspace, и ваше регулярное выражение будет не таким, как задумано:

```
>>> p = re.compile('\bclass\b')
>>> print p.search('no class at all')
None
>>> print p.search('\b' + 'class' + '\b')
<_sre.SRE_Match object at 0x...>
```

Во-вторых, внутри класса символов нельзя использовать данное сочетание, потому как сочетание `\b` для совместимости со строковыми литералами Python представляет символ backspace.

`\B`
Противоположное предыдущему сочетание, соответствующая текущей позиции не на границе слова.

Группировка

Часто необходимо получить больше информации, чем просто узнать, находит ли РВ соответствие или нет. Регулярные выражения часто используются для разрезания строк написанием регулярных выражений, разделенных на несколько подгрупп, которые соответствуют различным компонентам запроса. Например, в стандарте RFC-822 в заголовке имеются различные поля,

разделенные двоеточием:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

Это может быть обработано написанием регулярного выражения, которое соответствует всей строке заголовка, и в нем есть одна группа, которая соответствует имени заголовка, и другая группа, которая соответствует значению заголовка.

Группы обозначаются метасимволами в виде круглых скобок `'(', ')'`. `'('` и `')'` имеют такой же смысл, как в математических выражениях; они группируют вместе выражения, содержащиеся в них, и вы можете повторять содержание группы повторяющими квалификаторами, такими как `*, +, ?` и `{m, n}`. Например, `(ab)*` будет соответствовать нулю или более повторений `ab`.

```
>>> p = re.compile('(ab)*')
>>> print p.match('abababababab').span()
(0, 10)
```

Группы, определяемые скобками, также захватывают начальные и конечные индексы совпадающего текста; это может быть получено передачей аргумента `group(), start(), end()` и `span()`. Группы нумеруются, начина с 0. Группа 0 имеется всегда, это само регулярное выражение целиком, так что методы `MatchObject` всегда содержат 0 как аргумент по умолчанию:

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Подгруппы нумеруются слева направо, от 1 и далее. Группы могут быть вложенными; для того чтобы определить число вложений, просто подсчитываем слева направо символы открывающей скобки:

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` может принять одновременно несколько номеров групп в одном запросе, и будет возвращен кортеж, содержащий значения для соответствующих групп:

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

Метод `groups()` возвращает кортеж строк для всех подгрупп, начиная с 1-ой:

```
>>> m.groups()
('abc', 'b')
```

Обратные ссылки в шаблоне позволяют вам указать, что содержание ранее захваченной группы также должно быть найдено в текущей позиции строки. Например, `\1` соответствует тому, что содержание группы 1 в точности повторяется в текущей позиции.

Например, следующее РВ обнаруживает в строке дважды подряд повторяющиеся слова:

```
>>> p = re.compile(r'(\b\w+)\s+\1')
>>> p.search('Paris in the the spring').group()
'the the'
```

Обратные ссылки, такие, как эта, не часто бывают полезны для поиска по строке, но вы скоро узнаете, что они очень полезны при выполнении замены строки.

Группы с захватом содержимого и именованные группы

Регулярные выражения могут использовать множество групп, как для захвата необходимой подстроки, так и для группировки и структурирования самих РВ. В сложных регулярных выражениях становится трудно отслеживать номера групп. Есть две особенности, которые помогают справиться с этой проблемой. Обе из них используют общий синтаксис для расширения регулярных выражений, который мы поэтому и рассмотрим сначала.

В Perl 5 было добавлено несколько дополнительных функций для стандартных регулярных выражений, и модуль `re` поддерживает большинство из них. Было бы сложно выбрать новые односимвольные метасимволы или новые последовательности с бэкслешем, для того чтобы представить новые особенности так, чтобы регулярные выражения Perl без путаницы отличались от стандартных регулярных выражений. Если выбрать в качестве нового метасимвола, например, `&`, то старые регулярные выражения принимали бы его как обычный символ и нельзя было бы экранировать его `\&` или `[&]`.

Решение, выбранное разработчиками Perl состояло в том, чтобы использовать в качестве расширения синтаксиса `(?...)`. Знак вопроса после скобки в случае обычного РВ это синтаксическая ошибка, поскольку `?` нечего повторять, так что это не приводит к каким-либо проблемам в совместимости. Символы сразу после `?` показывают какое расширение используется, так `(?=foo)` это одно (положительное утверждение о предпросмотре), а `(?:foo)` это что-то другое (группа без захвата содержимого, включающая подвыражение `foo`).

К расширенному синтаксису Perl в Python добавляется собственное расширение. Если первый символ после знака вопроса `P`, то это означает что используется расширение, специфичное для Python. В настоящее время существуют два таких расширения: `(?P<some_name>...`)определяет именованную группу, а `(?P=some_name)` служит для нее обратной ссылкой. Если в будущих версиях Perl 5 добавятся аналогичные возможности, использующие другой синтаксис, модуль `re` будет изменен для поддержки нового синтаксиса, сохраняя при этом для совместимости Python-синтаксис.

Иногда вам нужно использовать группу для сбора части регулярного выражения, но вы не заинтересованы в извлечении содержимого группы. Вы можете сделать это, используя группу без захвата содержимого: `(?:...)`, где вы можете заменить `...` любым другим регулярным выражением:

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

За исключением того, что вы не получаете содержимого того, с чем совпала группа, эти группы ведут себя также, как и обычные; вы можете в них поместить что угодно, повторить с помощью соответствующего символа, такого как `*`, и вставить их в другие группы (собирающие данные или нет).

Более важной особенностью являются именованные группы: вместо ссылки на них по номерам, на эти группы можно ссылаться по имени.

Синтаксис именованных групп это одно из специфичных Python-расширений: `(?P<some_name>...)`. Именованные группы ведут себя в точности как обычные, но вдобавок к этому ассоциируются с каким-то именем. Методы `MatchObject`, которые использовались для обычных групп принимают как числа, ссылающиеся на номер группы, так и строки, содержащие имя необходимой группы. То есть именованные группы все также принимают и числа, так что вы можете получить информацию о группе двумя способами:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Именованные группы удобны тем, что они позволяют использовать вместо цифр легко запоминающиеся имена. Вот пример регулярного выражения из модуля `imaplib`:

```
InternalDate = re.compile(r'INTERNALDATE "'
    r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
    r'(?P<year>[0-9][0-9][0-9][0-9])'
    r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
    r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
    r'"')
```

Синтаксис обратных ссылок в регулярном выражение типа `(...)\1` ссылается на номер группы. Более естественно было бы использовать вместо номеров имена групп. Другое Python расширение: `(?P=name)` показывает, что содержимое названной группы снова должно быть сопоставлено в текущей позиции. Наше прежнее регулярное выражение для поиска дублирующихся слов, `(\b\w+)\s+\1` может быть также записано как `(?P<doble_word>\b\w+)\s+(?P=doble_word)`:

```
>>> p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)')
>>> p.search('Paris in the the spring').group()
'the the'
```

Опережающие проверки

Проверки доступны в позитивной и негативной (ретроспективной) форме, и выглядят так:

`(?=...)`
Положительная проверка. Соответствует случаю, когда содержащееся выражение, представленное здесь как `...`, соответствует текущей позиции. Но, после того как содержащееся выражение было опробовано, сравнивающий движок не продвигается далее; остаток шаблона сравнивается далее справа от того места, где начинается утверждение.

`(?!...)`
Отрицательное проверка соответствует случаю, когда содержащиеся выражение не соответствует текущей позиции строки.

Для конкретики, рассмотрим случай, в котором полезен предпросмотр. Рассмотрим простой шаблон для сравнения имени файла и разбиения его на части: само имя и расширение, отделенные друг от друга точкой.

Шаблон для такого сравнения довольно прост:

`.*[.].*$`

Отметим, что символ `.` требует специальных скобок, поскольку сама точка является метасимволом, что видно в этом же выражении. Также отметим заключительный $; он добавлен для того, чтобы обеспечить включение в расширение всего остатка строки.

Теперь, рассмотрим проблему немного шире; что, если вы хотите сравнить имена всех файлов, у которых расширение не `bat`? Несколько неверных попыток:

`.*[.][^b].*$`
Первая попытка состоит в том, чтобы исключить `bat` требованием, чтобы первый символ расширение был не `b`. Это неправильно, потому что шаблон также исключит `foo.bar`.

`.*[.]([^b]..|.[^a].|..[^t])$`
Выражение получится еще неряшливее, когда вы решите подправить первое решение отдельным заданием нужных символов: первый символ расширения должен быть не b; второй — не a; третий — не t. Это позволит включить `foo.bar` и отклонить `autoexec.bat`, но требует расширения из трех букв и не будет работать с двухсимвольными расширениями имен файлов, наприме `sendmail.cf`. Тогда нам придется снова усложнить шаблон, чтобы решить эту проблему:

`.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$`
В третьей попытке, вторая и третья буквы для того, чтобы позволить сравнение расширений, короче чем три символа, сделаны необязательными.

Шаблон теперь действительно готов, его сложно читать и понимать. Еще хуже, если изменится проблема и вам нужно будет исключить и bat, и exe расширения, шаблон станет еще более сложным и запутанным.

Негативная опережающая проверка решает все эти затруднения:

`.*[.](?!bat$).*$`
Отрицательный предпросмотр означает: если выражение `bat` не соответствует данной позиции, сравнить остаток шаблона; если обнаружено соответствие `bat$`, то весь шаблон нам не подходит. Заключающий выражение знак `$` нужен для того, чтобы было разрешено и такое выражение, как `sample.batch`.

Исключить другое расширение теперь тоже легко; просто добавляем его как альтернативное в том же утверждении. Следующий шаблон исключает имена файлов, которые заканчиваются расширением `bat` или `exe`:

```
.*[.](?!bat$|exe$).*$
```

Изменение строк

До этого момента, мы просто выполняли поиск по статической строке. Регулярные выражения также часто используется для изменения строк различными способами, используя следующие методы шаблонов:

Метод/атрибут	Цель
split()	Разбить строку в список там, где есть совпадение РВ
sub()	Найти все подстроки совпадений с РВ и заменить их другой строкой
subn()	Делает то же, что и sub(), но возвращает новую строку и число замещений

Разбиение строк

Метод шаблона `split()` разбивает строку на части там, где есть совпадение РВ, возвращая список частей. Это похоже на строковый метод `split()`, но обеспечивает всеобщность в разделителях, по которым происходит разбиение; обычный `split()` обеспечивает разбиение только по whitespace-символам или фиксированной строке. Как и следовало ожидать, существует и модульная функция `re.split()`.

.split(string[, maxsplit=0])
Разбивает строку по совпадениям с регулярным выражением. Если в РВ имеются захватывающие скобки, то их содержание также будет возвращаться как часть полученного списка. Если `maxsplit` не ноль, выполняется не более, чем `maxsplit` разбиений, остаток строки будет возвращен как последний элемент списка.

В следующем примере разделителем является любая последовательность небуквенно-цифровых символов.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Иногда вы заинтересованы не только в том, какой текст был между разделителями, но также должны знать, какой разделитель использовался. Если в РВ есть захватывающие скобки, то и эти значения также возвращаются, как часть списка. Сравните:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

Функция модуля `re.split()` в качестве первого аргумента забирает РВ, а в остальном ведет себя также:

```
>>> re.split('[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

Поиск и замена

Другая распространенная задача — найти все совпадения с шаблоном и заменить их другой строкой. Метод `sub()` берет в качестве аргументов значение замещающей части (которая может быть как строкой, так и функцией) и строку, которая должна быть обработана.

.sub(replacement, string[, count=0])
Возвращает строку, получающуюся при замене. Если шаблон не найден строка возвращается неизменной.

Добавочный аргумент `count` это максимальное число замещаемых совпадений.

Простой пример использования метода `sub()`. Названия цветов замещаются словом `colour`:

```
>>> p = re.compile( '(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

Метод subn() делает то же самое, но возвращает кортеж, содержащий новую строку и число произведенных замен:

```
>>> p = re.compile( '(blue|white|red)')
>>> p.subn( 'colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn( 'colour', 'no colours at all')
('no colours at all', 0)
```

Пустые совпадения заменяются только тогда, когда они не смежны с предыдущим совпадением:

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b-d-'
```

Если заместителем служит строка, то в ней поддерживаются символы экранирования. Так, `\n` это одиночный символ новой строки, `\r` это возврат каретки и так далее. Обратные ссылки, такие как `\6` замещаются подстрокой, совпавшей с соответствующей группой в РВ. Это позволяет вам включать части оригинального текста в результат замены строки.

Пример соответствует слову `section` в части строки, предшествующей части в фигурных скобках `{, }`, и заменяет `section` на `subsection`:

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First} section{second}')
'subsection{First} subsection{second}'
```

Также имеется возможность ссылаться на именованные группы. Для этого используется последовательность `\g<...>`, где в качестве `...` может выступать номер или имя группы. `\g<2>` это эквивалент `\2`, но он не двусмыслен в таких заместителях, как `\g<2>0`. (`\20` будет интерпретироваться как ссылка на группу 20, а не как вторую группу с последующим литералом '0'.) Следующие операции эквиваленты, но используют три различных способа:

```
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

Заместитель также может быть и функцией, дающем вам больше возможностей управления. Если это так, функция вызывается для каждого случая неперекрытия шаблона. При каждом вызове функции передается в качестве аргумента `MatchObject`.

В следующем примере функция замены переводит десятичные числа в шестнадцатеричные:

```
>>> def hexrepl( match ):
...     "Return the hex string for a decimal number"
...     value = int( match.group() )
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

Общие проблемы

Регулярные выражения это мощный инструмент для некоторых применений, но в некоторых отношениях их поведение не интуитивно, а иногда они не ведут себя так, как вы ожидаете от них. В этом разделе будет указано на некоторые из наиболее

распространенных ошибок.

Использование строковых методов

Иногда использование модуля `re` это ошибка. Если вы ищете фиксированную строку или одиночный символ, и вам не нужно использовать никаких особенностей `re`, то и вся мощь регулярных выражения для этого не требуется. У строк есть несколько методов для операций с фиксированными строками и они обычно гораздо быстрее, потому что оптимизированы для этой цели.

Представим, нужно заменить одну фиксированную строку другой, например, заменить слово `word` словом `deed`. Здесь, конечно, подойдет функция `re.sub()`, но рассмотрим строковый метод `replace()`. Отметим, что `replace()` будет также заменять `word` внутри слов, меняя `swordfish` на `sdeedfish`, но и простецкое регулярное выражение будет делать это то же. (Чтобы избежать выполнения замещения на части слов, шаблон должен содержать `\bword\b`).

Другая распространенная задача это удаление одиночного символа из строки или замена его другим символом. Вы можете сделать это чем-то вроде `re.sub('\n', ' ', S)`, но метод translate() справится с обеими задачами и сделает это быстрее, чем любое регулярное выражение.

Короче говоря, прежде чем использовать модуль `re`, посмотрите, может ли проблема быть решена более быстрыми и простыми методами строк.

match() в сравнении с search()

Функция `match()` ищет соответствие РВ в начале строки, тогда как `search()` ищет соответствие во всей строке. Важно иметь в виду это различие:

```
>>> print re.match('super', 'superstition').span()
(0, 5)
>>> print re.match('super', 'insuperable')
None
>>> print re.search('super', 'superstition').span()
(0, 5)
>>> print re.search('super', 'insuperable').span()
(2, 7)
```

У вас может возникнуть соблазн всегда пользоваться `re.match()`, просто добавляя перед вашим регулярным выражением `.*`. Сопротивляйтесь этому искушению, и вместо этого используйте `re.search()`. Компилятор регулярных выражений производит небольшой анализ РВ для того, чтобы ускорить процесс поиска соответствия. Один из видов такого анализа заключается в определении того, что должно быть первым символом совпадения, например, совпадение с шаблоном, начинающимся с `Crow`, должно начинаться с `'C'`. Этот анализ приводит к тому, что движок быстро пробегает строку в поиске начального символа, и начинает полное сравнение только, если найден символ 'C'.

Добавление `.*` сводит на нет эту оптимизацию, требуя сканирования до конца строки, а затем возвращения, чтобы сравнить остаток регулярного выражения. Используйте вместо этого `re.search()`.

Жадный против нежадного

При повторении в РВ, таком как `a*`, результирующее действие съедает настолько большую часть шаблона, насколько это возможно. На этом часто обжигаются те, кто хочет найти пару симметричных определителей, таких как угловые скобки `<>`, окружающие HTML-теги. Наивный подход к шаблону сопоставления тега HTML не будет работать из-за «жадного» характера `.*`:

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print re.match('<.*>', s).span()
(0, 32)
>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>
```

РВ сопоставляет `'<'` в первом теге — html, и `.*` забирает остаток строки. В итоге сопоставление простирается от открывающей склюки `'<'` тега `html` до закрывающей скобки `'>'` закрывающего тега `/title`, что, конечно, не есть то, чего мы хотели.

В таком случае, решение заключается в использовании нежадных определителей `*?, +?, ??` или `{m,n}?`, которые сопоставляют так мало текста, как это возможно. В примере выше, будет выбран первый символ `'>'` после `'<'`, и только, если не получится, движок будет продолжать попытки найти символ `'>'` на следующей позиции, в зависимости от того, как длинно имя тега. Это дает нужный результат:

```
>>> print re.match('<.*?>', s).group()
<html>
```

(Заметим, что парсинг HTML или XML с использованием регулярных выражений является делом болезненным. Наспех сделанный шаблон может справиться с какими-то вещами, но рушится при изменении кода страницы. А хорошо продуманный шаблон может оказаться *слишком* сложным для того, чтобы пытаться его модифицировать. Для таких задач лучше использовать модули HTML или XML парсеров.)

Использование re.VERBOSE

К настоящему моменты вы, вероятно, заметили, что РВ обладают очень компактной записью, но иногда они не очень читабельны. РВ умеренной сложности могут представлять собой длинные последовательности слэшей, круглых скобок, метасимволов, что делает их трудными для чтения и понимания.

Для таких РВ может быть полезным указание флага VERBOSE при компиляции регулярного выражения потому что это позволяет форматировать регулярное выражение более ясным образом.

Флаг VERBOSE имеет несколько особенностей. Пробелы в РВ, которые не находятся внутри класса символов игнорируется. Это означает, что выражения, такие как `dog | cat` эквивалентны менее читабельным беспробельным `dog|cat`, но [a b] будет по-прежнему соответствовать символам `'a', 'b'` или пробелу. Кроме того, вы можете также помещать внутрь РВ комментарии, длящиеся с символа # до следующей строки. Форматирование будет более аккуратным при использовании тройных кавычек:

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

Это гораздо проще читается, чем:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

В заключение

Документация модуля re
Хабраблог «Регулярные выражения»
Regexp editor

Теги: python, re, regular expressions, regexp, howto, регулярные выражения

Хабы: Python