LeoMat 20 марта 2011 в 12:34

Регулярные выражения, пособие для нови

Автор оригинала: A. M. Kuchling

Python

Перевод

Регулярные выражения (РВ) это, по существу, крошечный язык программирования, встро помощи модуля re. Используя его, вы указывается правила для множества возможных ст множество может содержать английские фразы, или адреса электронной почты, или TeX помощью РВ вы можете задавать вопросы, такие как «Соответствует ли эта строка шабло нибудь с этой строкой?». Вы можете также использовать регулярные выражения, чтобы и части различными способами.

Шаблоны регулярных выражений компилируются в серии байт-кода, которые затем испо написанным на C. Для продвинутого использования может быть важно уделять внимани данное регулярное выражение, и писать его так, чтобы получался байт-код, который раб рассматривается в этом документе, так как она требует от вас хорошего понимания внут

Язык регулярных выражений относительно мал и ограничен, поэтому не все возможные сделать с помощью регулярных выражений. Также существуют задачи, которые *можно* с выражений, но выражения оказываются слишком сложными. В этих случаях может быть пусть он будет работать медленнее, чем разработанное регулярное выражение, но будет

Простые шаблоны

Мы начнем с изучения простейших регулярных выражений. Поскольку регулярные выражения используются для работы со строками, мы начнем с наиболее распространенной задачи — соответствия символов.

За подробным объяснением технической стороны регулярных выражений (детерминированных и недетерминированных конечных автоматов) вы можете обратиться к практически любому учебнику по написанию компиляторов.

Соответствие символов

Большинство букв и символов соответствуют сами себе. Например, регулярное выражение `test` будет в точности соответствовать строке `test` (Вы можете включить режим без учета регистра, что позволит этому регулярному выражению также соответствовать `Test` или `TEST`, но об этом позже).

Из этого правила есть исключения; некоторые символы это специальные *метасимволы*, и сами себе не соответствуют. Вместо этого они указывают, что должна быть найдена некоторая необычная вещь, или влияют на другие части регулярного выражения, повторяя или изменяя их значение. Большая часть этого пособия посвящена обсуждению различных метасимволов и тому, что они делают.

Вот полный список метасимволов; их значения будут обсуждаться в остальной части этого HOWTO.

```
. ^ $ * + ? { [ ] \ | ( )
```

Первые метасимволы, что мы рассмотрим это `[` и `]`. Они используются для определения класса символов, являющегося набором символов, с которыми вы ищите совпадение. Символы могут быть перечислены по отдельности, или в виде некоторого диапазона символов, обозначенного первым и последним символом, разделенных знаком `'-'`. Например, `[abc]` будет соответствовать любому из символов `a`, `b` или `c`; это то же самое, что выражение `[a-c]`, использующее диапазон для

задания того же множества символов. Если вы хотите сопоставить только строчные буквы, РВ будет иметь вид `[a-z]`.

Метасимволы не активны внутри классов. Например, `[akm$]` будет соответствовать любому из символов `'a', 'k', 'm'` или `'$'`. Знак `'$'` это обычно метасимвол (как видно из списка символов выше), но внутри класса символов он лишается своей особой природы.

Для того, чтобы находить соответствие символам вне этого класса, в начале класса добавляется символ `'^'`. Например, выражение `[^5]` соответствует любому символу, кроме '5'.

Пожалуй, наиболее важным является метасимвол обратной косой черты `\`. Как и в строковых литералах Python, за бэкслешем могут следовать различные символы, обозначающие разные специальные последовательности. Он также используется для экранирования метасимволов, чтобы их можно было использовать в шаблонах; например, если нужно найти соответствие `[` или `\`, для того чтобы лишить их своей особой роли метасимволов, перед ним нужно поставить обратную косую черту: `\[` или `\\`.

Некоторые из специальных последовательностей, начинающихся с `'\'` представляют предопределенные наборы символов, часто бывающие полезными, такие как набор цифр, набор букв, или множества всего, что не является пробелами, символами табуляции и т. д. (whitespace). Следующие предопределенные последовательности являются их подмножеством. Полный список последовательностей и расширенных определений классов для Юникод-строк смотрите в последней части Regular Expression Syntax.

`\d`
Соответствует любой цифре; эквивалент класса `[0-9]`.
`\D`
Соответствует любому нечисловому символу; эквивалент класса `[^0-9]`.
`\s`
Соответствует любому символу whitespace; эквивалент `[ \t\n\r\f\v]`.
`\S`
Соответствует любому не-whitespace символу; эквивалент `[^ \t\n\r\f\v]`.
`\w`
Соответствует любой букве или цифре; эквивалент `[a-zA-Z0-9_]`.
`\W`
Наоборот; эквивалент `[^a-zA-Z0-9_]`.

Эти последовательности могут быть включены в класс символов. Например, [\s,.] является характер класс, который будет соответствовать любому whitespace-символу или запятой или точке.

Последний метасимвол в этом разделе это `'.'`. Он соответствует всем символам, кроме символа новой строки, но есть альтернативный режим (`re.DOTALL`), где это множество будет включать и его. `'.'` часто используется там, где вы хотите сопоставить «любой символ».

Повторяющиеся вещи

Возможность сопоставлять различные наборы символов это первое, что регулярные выражения могут сделать и что не всегда можно сделать строковыми методами. Однако, если бы это было единственной дополнительной возможностью, они бы не были так интересны. Другая возможность заключается в том, что вы можете указать какое число раз должна повторяться часть регулярного выражения.

Первый метасимвол для повторения это `*`. Он указывает, что предыдущий символ может быть сопоставлен ноль и более раз, вместо одного сравнения.

Например, `ca*t` будет соответствовать `ct` (0 символов `a`), cat (1 символ `a`), caaat (3 символа `a`), и так далее. Движок регулярных выражений имеет различные внутренние ограничения вытекающие из размера `int` типа для C, что не позволяет проводить ему сопоставление более 2 миллиардов символов 'a'. (Надеюсь, вам это не понадобится).

Повторения, такие как `*` называют *жадными* (greedy); движок будет пытаться повторить его столько раз, сколько это возможно. Если следующие части шаблона не соответствуют, движок вернется назад и попытается попробовать снова с несколькими

повторами символа.

Пошаговое рассмотрение какого-нибудь примера сделает объяснение более ясным. Давайте рассмотрим выражение `a[bcd]*b`. Оно соответствует букве `'a'`, нулю или более символов из класса `[bcd]`, и наконец, заключительной букве `'b'`. Теперь представим себе сопоставление этого регулярного выражения строке `abcbd`. Вот как происходит сравнение поэтапно:

1. `a` — 'a' соответствует регулярному выражению
2. `abcbd` — движок сопоставляет `[bcd]*` на как можно большем числе символов, то есть до конца строки (поскольку все символы соответствуют классу в скобках [])
3. Провал — движок пытается сопоставить последний символ в регулярном выражении — букву `b`, но текущая позиция уже в конце строки, где нет никаких символов, так что он терпит неудачу.
4. `abcb` — вернулись назад, уменьшили на один символ сопоставление с `[bcd]*`
5. Провал — пытаемся снова найти `b`, но в конце только `d`
6. `abc` — снова возвращаемся назад, теперь `[bcd]*` это только bc
7. `abcb` — снова ищем последний символ регулярного выражения — b. Теперь он действительно находится на нужной позиции и мы добиваемся успеха

Итак, был достигнут конец РВ и сопоставление с ним дало `abcb`. Этот пример показал, как движок сначала забирается так далеко, как может, и, если не находит соответствия, возвращается назад, снова и снова работая с остатком регулярного выражения. Он будет делать так до тех пор, пока не получит ноль совпадений для `[bcd]*`, и, если и тогда не получится совпадения, то заключит, что строка совсем не соответствует шаблону РВ.

Другой метасимвол повторения это `+`, повторяющий последовательность сравнения *один* или более раз. Обратите особое внимание на разницу между `*` и `+`. `*` требует соответствия необходимой части *ноль* или более раз, то есть повторяемое может и не присутствовать вовсе, а `+` требует, *по крайней мере одно* вхождение. Для аналогичного примера `ca+t` будет сопоставляться `cat` или, например, `caaat`, но никак не `ct`.

Есть еще два повторяющих спецификатора. Знак вопроса, `?`, проверяющий наличие совпадения ноль или один раз. Например, `home-?brew` соответствует как `homebrew`, так и `home-brew`.

Наиболее полный повторяющий спецификатор это `{m,n}`, где `m` и `n` — целые числа. Этот определитель означает, что здесь должно быть не менее `m` и не более `n` повторений. Например, `a/{1,3}b` соответствует `a/b`, `a//b` и `a///b`. Это не может быть `ab`, строка в которой нет слэшей или `a////b`, в которой их четыре.

Вы можете не задавать `m` или `n`, тогда для отсутствующего предполагается наиболее разумное значение. Опускание `m` означает, что нижний предел 0, опускание `n` предполагает верхним пределом бесконечность, но, как уже говорилось выше, последний ограничен памятью.

Читатели уже могли заметить, что все три остальных спецификатора могут быть выражены через последний. `{0,}` это то же, что `*`, {1,} эквивалентно `+`, и `{0,1}` может заменять знак `?`.

Использование регулярных выражений

Теперь, когда мы рассмотрели несколько простых регулярных выражений, как мы можем использовать их в Python? Модуль `re` предоставляет интерфейс для регулярных выражений, что позволяет компилировать регулярные выражения в объекты, а затем выполнять с ними сопоставления.

Компиляция регулярных выражений

Регулярные выражения компилируются в объекты шаблонов, имеющие методы для различных операций, таких как поиск вхождения шаблона или выполнение замены строки.

```
>>> import re
>>> p = re.compile('ab*')
```

```
>>> print p
<_sre.SRE_Pattern object at 0x...>
```

`re.compile()` также принимает необязательные аргументы, использующихся для включения различных особенностей и вариаций синтаксиса:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

Регулярное выражение передается `re.compile()` как строка. Регулярные выражения обрабатываются как строки, поскольку не являются частью языка Python, и нет никакого специального синтаксиса для их выражения. (Существуют приложения, которые вовсе не нуждаются в регулярных выражениях, так что нет необходимости забивать спецификацию языка, включая их.) Вместо этого имеется модуль `re`, представляющий собой обертку модуля на C, подобно модулям `socket` или `zlib`.

Передача регулярных выражений в виде строки позволяет Python быть проще, но имеет один недостаток, который является темой следующего раздела.

Бэкслеш бедствие (Или обратная косая чума :))

Как было отмечено ранее, в регулярных выражениях для того, чтобы обозначить специальную форму или позволить символам потерять их особую роль, используется символ бэкслеша (`'\'`). Это приводит к конфликту с использованием в строковых литералах Python такого же символа с той же целью.

Скажем, вы хотите написать регулярное выражение, соответствующее `\section`, которое надо найти в LaTeX-файле. Чтобы выяснить, что написать в коде программы, начнем со строки, которую необходимо сопоставить. Далее, вы должны избежать любых бэкслешей и других метасимволов, экранировав их обратной косой чертой, в результате чего в строке появляется часть `\\`. Тогда, результирующая строка, которая должна быть передана `re.compile ()` должен быть `\\section`. Однако, для того, чтобы выразить это как строковый литерал Python, оба бэкслеша должны быть экранированы *снова*, то есть `"\\\\section"`.

Одним словом, чтобы сопоставить бэкслеш, нужно писать в качестве строки регулярного выражения `'\\\\'`, потому что регулярное выражение должно быть `\\`, и каждая обратная косая черта должна быть переведена в обычную строку как `\\`.

Решение заключается в использовании для регулярных выражений «сырых» строк (raw string); в строковых литералах с префиксом `'r'` слэши никак не обрабатываются, так что `r"\n"` это строка из двух символов ('\' и 'n'), а "\n" — из одного символа новой строки. Поэтому регулярные выражения часто будут записываться с использованием сырых строк.

Regular String	**Raw string**
'ab*'	r'ab*'
'\\\\section'	r'\\section*'
'\\w+\\s+\\1'	r'\w+\s+\1'

Выполнение сопоставлений

После того, как у вас есть объект, представляющий скомпилированное регулярное выражение, что вы с ним будете делать? Объекты шаблонов имеют несколько методов и атрибутов. Только самые важные из них будут рассмотрены здесь; для получения полного списка ознакомьтесь с документацией `re`.

Метод/атрибут	Цель
match()	Определить, начинается ли совпадение регулярного выражения с начала строки
search()	Сканировать всю строку в поисках всех мест совпадений с регулярным выражением
findall()	Найти все подстроки совпадений с регулярным выражением и вернуть их в виде списка
finditer()	Найти все подстроки совпадений с регулярным выражением и вернуть их в виде итератора

Если не было найдено ни одного совпадения, то `match()` и `search()` возвращают `None`. Если поиск успешен, возвращается экземпляр `MatchObject`, содержащий информацию о совпадении: где оно начинается и заканчивается, подстрока соответствия, и так далее.

Вы можете узнать об этом, интерактивно поэкспериментировав с модулем `re`. Вы также можете взглянуть на `Tools/scripts/redemo.py`, демонстрационную программу, включенную в дистрибутив Python. Она позволяет вводить регулярные выражения и строки, и отображает, есть ли совпадение с регулярным выражением или нет. `redemo.py` может быть весьма полезна для отладки сложных регулярных выражений. Kodos Фила Шварца — еще один интерактивный инструмент для разработки и тестирования моделей РВ.

В этом пособии мы используем для примеров стандартный интерпретатор Python:

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

Теперь вы можете попробовать сравнить строки для регулярного выражения `[a-z]+`. Пустая строка ему не будет соответствовать, потому что + означает повторение «один или больше» раз. `match()` в этом случае должен вернуть `None`, что и видим:

```
>>> p.match("")
>>> print p.match("")
None
```

Теперь попробуем строку, которая должна совпасть с шаблоном: `'tempo'`. В этом случае `match()` вернет `MatchObject`, который вы можете разместить в какой-то переменной, чтобы использовать ее в дальнейшем:

```
>>> m = p.match('tempo')
>>> print m
<_sre.SRE_Match object at 0x...>
```

Теперь вы можете вызывать `MatchObject` для получения информации о соответствующих строках. Для `MatchObject` также имеется

несколько методов и атрибутов, наиболее важными из которых являются:

Метод/атрибут	**Цель**
group()	Вернуть строку, сошедшуюся с регулярным выражением
start()	Вернуть позицию начала совпадения
end()	Вернуть позицию конца совпадения
span()	Вернуть кортеж (start, end) позиций совпадения

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

Так как метод `match()` проверяет совпадения только с начала строки, `start()` всегда будет возвращать 0. Однако метод `search()` сканирует всю строку, так что для него начало не обязательно в нуле:

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message') ; print m
<_sre.SRE_Match object at 0x...>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

В реальных программах наиболее распространенный стиль это хранение `MatchObject` в переменной, а затем проверка по `None`. Обычно это выглядит следующим образом:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```

Два метода возвращают все совпадения для шаблона. `findall()` возвращает список совпавших подстрок:

```
>>> p = re.compile('\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

Метод `findall()` должен создать полный список, прежде чем он может быть возвращен в качестве результата. Метод `finditer()` возвращает последовательность экземпляров `MatchObject` в качестве итератора.

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x401833ac>
>>> for match in iterator:
...     print match.span()
...
(0, 2)
(22, 24)
(29, 31)
```

Функции на уровне модуля

Вам не обязательно нужно создавать объекты шаблонов и вызывать их методы; модуль `re` также предоставляет функции верхнего уровня `match(), search(), findall(), sub()` и так далее. Эти функции принимают те же аргументы, что и для шаблонов, со строкой РВ в качестве первого аргумента и также возвращают `None` или `MatchObject`.

```
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<_sre.SRE_Match object at 0x...>
```

Эти функции просто создают для вас объект шаблона и вызывают соответствующий метод. Они также хранят объект в кэше, так что будущие вызовы с использованием того же регулярного выражения будут быстрее.

Должны вы использовать эти функции или шаблоны с методами? Это зависит от того, как часто будет использоваться регулярное выражение и от вашего личного стиля кодинга. Если регулярное выражение используется только в одном месте кода, то такие функции, вероятно, более удобны. Если программа содержит много регулярных выражений, или повторно использует одни и те же в нескольких местах, то будет целесообразно собрать все определения в одном месте, в разделе кода, который предварительно компилирует все регулярные выражения. В качестве примера из стандартной библиотеки, вот кусок из `xmllib.py`:

```
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

Сам я предпочитаю работать со скомпилированными объектами, даже для одноразового использования, но мало кто окажется таким же пуристом в этом, как я.

Флаги компиляции

Флаги компиляции позволяют изменять некоторые аспекты того, как работают регулярные выражения. Флаги доступны в модуле под двумя именами: длинным, таким как `IGNORECASE` и коротким, в однобуквенной форме, таким как `I`. Несколько флагов могут быть заданы в форме двоичного ИЛИ; например `re.I | re.M` устанавливает флаги I и M.

DOTALL, S
Сопоставление, такое же как `'.'`, то есть с любым символом, но при включении этого флага, в рассмотрение добавляется и символ новой строки.

IGNORECASE, I
Сопоставление без учета регистра; Например, `[A-Z]` будет также соответствовать и строчным буквам, так что `Spam` будет соответствовать `Spam, spam, spAM` и так далее.

LOCALE, L
Делает `\w, \W, \b, \B` зависящими от локализации. Например, если вы работаете с текстом на французском, и хотите написать `\w+` для того, чтобы находить слова, но `\w` ищет только символы из множества `[A-Za-z]` и не будет искать 'é' или 'ç'. Если система настроена правильно и выбран французский язык, 'é' также будет рассматриваться как буква.

MULTILINE, M
(Метасимволы `^` и `$` еще не были описаны; они будут представлены немного позже, в начале второй части этого пособия.

Обычно `^` ищет соответствие только в начале строки, а `$` только в конце непосредственно перед символом новой строки (если таковые имеются). Если этот флаг указан, `^` сравнение происходит во всех строках, то есть и в начале, и сразу же после каждого символа новой строки. Аналогично для `$`.

UNICODE, U
Делает `\w, \W, \b, \B, \d, \D, \s, \S` соответствующими таблице Unicode.

VERBOSE, X
Включает многословные (подробные) регулярные выражения, которые могут быть организованы более ясно и понятно. Если указан этот флаг, пробелы в строке регулярного выражения игнорируется, кроме случаев, когда они имеются в классе символов или им предшествует неэкранированный бэкслеш; это позволяет вам организовать регулярные выражения более ясным образом. Этот флаг также позволяет помещать в регулярные выражения комментарии, начинающиеся с `'#'`, которые будут игнорироваться движком.

Пример того, как PB становится существенно проще читать:

```
charref = re.compile(r"""
 &[#]                # Start of a numeric entity reference
 (
     0[0-7]+         # Octal form
   | [0-9]+          # Decimal form
   | x[0-9a-fA-F]+   # Hexadecimal form
 )
 ;                   # Trailing semicolon
""", re.VERBOSE)
```

Без verbose это выглядело бы так:

```
charref = re.compile("&#(0[0-7]+"
                     "|[0-9]+"
                     "|x[0-9a-fA-F]+);")
```

В приведенном выше примере была использована автоматическая конкатенация Python строковых литералов для разбивки РВ на более мелкие части, но, все равно, без пояснений это пример труднее понять, чем версию с помощью `re.VERBOSE`.

На этом месте мы пока завершим наше рассмотрение. Советую немного отдохнуть перед второй половиной, содержащей рассказ о других метасимволах, методах разбиения, поиска и замены строк и большое количество примеров использования регулярных выражений.

Продолжение

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

Хабы: Python