Regular Expression HOWTO¶

Author:	A.M. Kuchling <amk@amk.ca>
Site:	https://docs.python.org/2/howto/regex.html
Traduzido:	Flávio Augusto Casacurta <flavio@casacurta.com>

Abstract

Este documento é um tutorial introdutório ao uso de expressões regulares em Python com o módulo de re. Ele fornece uma introdução mais suave do que a seção correspondente na Biblioteca de Referência.

Introdução¶

O módulo re foi adicionado no Python 1.5, e provê padrões de expressões regulares do “Perl-style”. As versões anteriores do Python vieram com o regexmodule, que forneceu os padrões “Emacs-style”. O modulo de regex foi completamente removido do Python 2,5.

Expressões regulares (REs chamados, ou expressões regulares, ou padrões de regex) são, essencialmente, uma pequena linguagem de programação altamente especializada incorporada dentro do Python e disponibilizados através do módulo de re. Utilizando esta pequena linguagem, você especifica as regras para o conjunto de seqüências de caracteres possíveis que você deseja corresponder, este conjunto pode conter frases em inglês, ou e-mail, ou comandos TeX, ou qualquer coisa que você queira. Você pode então fazer perguntas como "Será que esta seqüência corresponde ao padrão?", Ou "Existe uma correspondência para o padrão em qualquer lugar nesta string?". Você também pode usar REs para modificar uma string ou dividi-la para além de várias maneiras.

Padrões de expressões regulares são compiladas em uma série de bytecodes que são executadas por um mecanismo de correspondência escrita em C. Para uso avançado, pode ser necessário prestar muita atenção à forma como o mecanismo irá executar um RE dado, e escrever o RE em um certa maneira, a fim de produzir bytecode que é mais rápido. Otimização não é abordada neste documento, porque ele requer que você tenha um bom entendimento interno do mecanismo de correspondência.

A linguagem de expressão regular é relativamente pequena e restrita, por isso nem todas as tarefas possíveis de processamento de strings podem ser feitas usando expressões regulares. Há também as tarefas que podem ser feitas com expressões regulares, mas as expressões acabam por ser muito complicadas. Nestes casos, pode

ser melhor para você escrever um código Python para fazer o processamento, enquanto o código Python será mais lento do que uma expressão regular elaborada, mas também irá provavelmente ser mais compreensível.

Padrões Simples ¶

Vamos começar por aprender sobre as mais simples possíveis expressões regulares. Como as expressões regulares são usados para operar em strings, vamos começar com a tarefa mais comum: de correspondência caracteres.

Para uma explicação detalhada da ciência da computação subjacente a expressões regulares (autômatos finitos determinísticos e não-determinístico), você pode consultar a praticamente qualquer livro sobre a escrita de compiladores.

Caracteres Correspondentes¶

A maioria das letras e caracteres simplesmente correspondên-se. Por exemplo, o teste de expressão regular irá combinar com o teste de strings exatamente. (Você pode habilitar o modo de maiúsculas e minúsculas que deixaria na RE esta correspondência Test ou TEST, como tal, veremos sobre isso mais adiante.)

Há exceções a essa regra, alguns caracteres são metacaracteres especiais, e não se correspondem. Em vez disso, eles sinalizam que alguma coisa fora do normal deve ser correspondida, ou eles afetam outras partes da RE por repeti-las ou alterar o seu significado. Grande parte deste documento é dedicada à discussão metacaracteres vários e o que eles fazem.

Aqui está a lista completa dos metacaracteres; seus significados serão discutidos no resto deste documento.

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

Os primeiros metacaracteres que vamos olhar são `[` e `]`. Eles são usados para especificar uma classe de caracteres, que é um conjunto de caracteres que você deseja corresponder. Os caracteres podem ser listados individualmente, ou um intervalo de caracteres pode ser indicada dando dois caracteres e separando-os por um '-'. Por exemplo, [abc] irá corresponder a qualquer dos caracteres a, b, c ou, o que é o mesmo que [a-c], que usa um intervalo de expressar o mesmo conjunto de caracteres. Se você quiser corresponder apenas letras minúsculas, o RE seria [a-z].

Metacaracteres não são ativos dentro classes “`[ ]`”. Por exemplo, [akm$] irá corresponder a qualquer um dos caracteres 'a', 'k', 'm', ou '$'; '$' é geralmente um

metacaractere, mas dentro de uma classe de caracteres é despojado de sua natureza especial.

Você pode combinar os caracteres que não deverão listados dentro da classe, complementando o conjunto. Isto é indicado através da inclusão de um '^' como o **primeiro** caractere da classe; '^' fora de uma classe de caracteres simplesmente corresponder ao caracter '^'. Por exemplo, [^5] irá corresponder a qualquer caractere, exceto '5 '.

Talvez o metacaractere mais importante é a barra invertida, \. Como em literais string Python, a barra invertida pode ser seguido por vários caracteres para sinalizar várias sequências especiais. Também é usado para escapar de todos os metacaracteres assim você ainda pode combiná-los em padrões, por exemplo, se você precisa corresponder a um [ou \, você pode precedê-los com uma barra invertida para remover seu significado especial: \ [ou \ \.

Algumas das sequências especiais que começam com '\' representam conjuntos predefinidos de caracteres que muitas vezes são úteis, como o conjunto de dígitos, o conjunto de letras, ou o conjunto de tudo que não é branco. As seguintes pré-definidas seqüências especiais são um subconjunto das disponíveis. As classes são equivalentes para os padrões de strings de bytes. Para obter uma lista completa de seqüências e definições de classes expandidas para os padrões de seqüência de caracteres Unicode, consulte a última parte da sintaxe de Expressões Regulares.

\d corresponde a qualquer dígito decimal, que é equivalente à classe [0-9].
\D Coincide com qualquer caractere não-dígito, o que é equivalente à classe [^0-9].
\s Coincide com qualquer caractere espaço em branco, o que é equivalente à classe[\t\n\r\f\v].
\S Corresponde a qualquer caractere não-branco, o que é equivalente à classe [^\t\n\r\f\v].
\w Corresponde a qualquer caractere alfanumérico, o que é equivalente à classe [a-zA-Z0-9_].
\W Corresponde a qualquer caractere não-alfanumérico, o que é equivalente à classe [^a-zA-Z0-9_].

Estas sequências podem ser incluídos dentro de uma classe caracter. Por exemplo, [\s,.] É uma classe de caracteres que irá corresponder a qualquer caractere espaço em branco, ou ',' ou '.'.

O metacaractere final desta seção é o '**.**'. Ele encontra tudo, exceto um caractere de nova linha, e não há um modo alternativo (re.DOTALL), onde ele irá corresponder ainda uma nova linha. '**.**' é muitas vezes usado quando você quiser combinar "qualquer caractere".

Repetindo Coisas¶

Ser capaz de combinar conjuntos diferentes de caracteres é a primeira coisa que as expressões regulares podem fazer isso não é já possível com os métodos disponíveis em strings. No entanto, se essa fosse a única capacidade adicional de expressões regulares, elas não adiantariam muito. Outro recurso é que você pode especificar que partes do RE deve ser repetido de um certo número de vezes.

O primeiro metacharacter para repetir as coisas que vamos ver é o *. * Não coincide com o carácter literal *; em vez disso, especifica que o carácter anterior pode ser combinado zero ou mais vezes, em vez de apenas uma vez.

Por exemplo, `ca*t` irá corresponder a `ct` (0 `a` caracteres), `cat` (1 `a`), `caaat` (3 `a` caracteres), e assim por diante. O motor RE tem várias limitações internas decorrentes do tamanho do tipo `int` de C que irá impedi-lo de correspondência de mais de 2 bilhões de caracteres; você provavelmente não tem memória suficiente para a construção de uma cadeia tão grande, então você não deve se preocupar com esse limite.

Repetições, tais como * são gananciosas; ao repetir a RE, o motor de correspondência vai tentar repeti-lo tantas vezes quanto possível. Se porções posteriores do padrão não corresponderem, o motor de correspondência, em seguida, volta e tenta novamente com algumas repetições.

Um exemplo passo-a-passo irá tornar isso mais óbvio. Vamos considerar a expressão `a[bcd]*b`. Isto corresponde a letra `'a'`, zero ou mais cartas da classe `[bcd]` e, finalmente, termina com um `'b'`. Agora imagine combinar este RE contra o string `abcbd`.

Passo	Coincide	Explanação
1	`a`	O caracter `a` na RE correspondência.
2	`abcbd`	O motor coincide `[bcd]*`, indo tão longe quanto possível, que é o fim do string.
3	*Falha*	O motor tenta coincidir `b`, mas a corrente posição é o fim do string, então ele falha.
4	`abcb`	Volta, de modo que `[bcd]*` corresponde a um caracter menor.
5	*Falha*	Tenta `b` novamente, mas a corrente posição é a do último caracter, que é um `'d'`.
6	`abc`	Volta novamente, de modo que `[bcd]*` somente `bc` está coincidindo .
7	`abcb`	Tenta `b` novamente. Desta vez, o caractere na posição atual é `'b'`, por isso sucesso.

O fim da ER já foi alcançado, e foi correspondido `abcb`. Isso demonstra como o motor de correspondência vai tão longe quanto possível ao início, e se não for encontrada nenhuma correspondência fará então progressivamente para traz e repita o resto do RE novamente e novamente. Ele vai fazer a volta para traz até que não tenha conseguido nenhum resultado para [bcd] *, e se isso falhar posteriormente, o motor vai concluir que a string não coincide com o RE em tudo.

Outro metacaracter de repetição é o +, que corresponde a uma ou mais vezes. Preste muita atenção para a diferença entre * e +; * coincide zero ou mais vezes, assim que o que está sendo repetida pode não estar presente em tudo, enquanto + requer pelo menos uma ocorrência. Para usar um exemplo similar, `ca+t` vai corresponder a `cat` (1 `a`), `caaat` (3 `a`'s), mas não coincide com `ct`.

Há mais dois qualificadores de repetição. ? O carácter ponto de interrogação, corresponde uma vez ou zero vezes; você pode pensar nisso como marcação algo como sendo opcional. Por exemplo, `home-?brew` combina tanto com `homebrew` ou `home-brew`.

O qualificador de repetição mais complicado é o `{m,n}`, em que *m* e *n* são números inteiros decimais. Esta qualificação significa que deve haver pelo menos *m* repetições, e no máximo *n*. Por exemplo, `a/{1,3}b` irá corresponder a `a/b`, `a//b` e `a///b`. Não vai coincidir com `ab`, que não tem barras ou a um `a////b`, que tem quatro.

Você pode omitir `m` ou `n`; nesse caso, um valor razoável é assumido para o valor em falta. Omitindo `m` é interpretado como um limite inferior a 0, enquanto omitindo `n` resulta em um limite superior do infinito - na verdade, o limite superior é o limite de 2 bilhões mencionado anteriormente, mas que poderia muito bem ser infinito.

Os leitores de uma inclinação reducionista podem notar que os três outros qualificadores podem ser expressas utilizando esta notação. `{0,}` é o mesmo que `*`, `{1,}` é equivalente a `+`, e `{0,1}` é o mesmo que `?`. É melhor usar `*`, `+` ou `?` quando você pode, simplesmente porque eles são mais curtos e mais fácil de ler.

Usando expressões regulares ¶

Agora que nós vimos algumas expressões regulares simples, como podemos realmente usá-los em Python? O módulo **`re`** fornece uma interface para o mecanismo de expressão regular, o que lhe permite compilar REs em objetos e, em seguida, executar comparações com eles.

Compilando Expressões Regulares ¶

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

As expressões regulares são compiladas em objetos padrão, que têm métodos para várias operações, tais como a procura de partidas padrão ou realizar substituições de strings.

```
>>>
>>> import re
>>> p = re.compile('ab*')
>>> print p
<_sre.SRE_Pattern object at 0x...>
```

`re.compile()` também aceita *flags* opcionais como argumentos, utilizados para habilitar recursos especiais e variações de sintaxe diferentes. Nós vamos ver todas as configurações disponíveis mais tarde, mas por agora um único exemplo vai servir:

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

O RE é passado para **`re.compile()`** como uma string. REs são tratados como strings porque as expressões regulares não são parte do núcleo da linguagem Python, e nenhuma sintaxe especial foi criado para expressá-las. (Existem aplicações que não necessitam de REs em tudo, por isso não há necessidade de inchar a especificação da linguagem, incluindo-os.) Em vez disso, o módulo **`re`** é simplesmente um módulo de extensão C incluído no Python, assim como os módulos de **`socket`** ou **`zlib`**.

Colocando REs em strings mantém a linguagem Python mais simples, mas tem uma desvantagem que é o tema da próxima seção.

A praga da barra invertida ¶

Como afirmado anteriormente, expressões regulares usam o caractere de barra invertida ('\') para indicar formas especiais ou para permitir caracteres especiais para serem usados sem invocar o seu significado especial. Isso entra em conflito com o uso do mesmo caráter para o mesmo efeito nas cadeias de caracteres de Python.

Vamos dizer que você quer escrever um RE que coincide com o string `\section`, que pode ser encontrado em um arquivo LaTeX. Para descobrir o que escrever no código do programa, comece com a string pretendida a ser correspondido. Em seguida, você deve escapar qualquer barra invertida e outros metacaracteres precedendo-los com

uma barra invertida, resultando na string `\\section`. A string resultante que deve ser passada para **`re.compile()`** deve ser `\\section`. No entanto, para expressar isso como uma string literal Python, ambas as barras invertidas devem ser escapadas novamente.

CharactREs	Stage
`\section`	string de texto a ser correspondido
`\\section`	barra invertida escapada para **`re.compile()`**
`"\\\\section"`	barras invertidas escapadas para uma string literal

Em suma, para coincidir com uma barra invertida literal, tem de se escrever `'\\\\'` como a string de RE, porque a expressão regular deve ser `\\`, e cada barra invertida deve ser expressa como `\\` dentro de uma string Python normal literal. Em REs que apresentam barras invertidas repetidas vezes, isso leva a um monte de barras invertidas repetidas e faz as strings resultantes difíceis de entender.

A solução é usar a notação de string crua (raw) do Python para expressões regulares; barras invertidas não são tratados de qualquer maneira especial em uma string literal prefixado com `'r'`, então `r"\n"` é uma sequência de dois caracteres contendo `'\'` e `'n'`, enquanto `"\n"` é uma string de um único caractere contendo uma nova linha. As expressões regulares, muitas vezes, são escritas em código Python usando esta notação de string crua.

String Regular	String Crua
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

Executando Comparações¶

Uma vez que você tem um objeto que representa uma expressão regular compilado, o que você faz com ele? Objetos padrão têm vários métodos e atributos. Apenas os mais significativos serão vistos aqui; consulte a documentação **re** para uma lista completa.

Método/Atributo	Propósito
`match()`	Determina se a RE combina com o inicio da string.
`search()`	Varre toda a string, procurando qualquer local onde esta RE corresponde.
`findall()`	Encontra todas as substrings onde a RE combina, e retorna elas como uma lista.
`finditer()`	Encontra todas as substrings onde a RE combina, e retorna elas como um *iterator*.

`match()` e **`search()`** retornam **`None`** se nenhuma combinação pode ser encontrada. Se tiveram successo, uma instância de `MatchObject` é retornada, contendo informações sobre a combinação: onde ela começa e termina, o substring que ela combinou, entre outras.

Você pode aprender sobre isto interativamente experimentando com o **re**module. Se você tiver Tkinter disponível, você pode também dar uma olhada em `Tools/scripts/redemo.py`, um programa demonstração incluído na distribuição Python. Ele permite você entrar com REs e strings, e exibe se a RE combina ou falha. `redemo.py` pode ser bastante útil quando se tenta depuração de uma complicada RE. Phil Schwartz's Kodos é também uma ferramenta interativa para desenvolvimento e teste de padrões RE.

Este HOWTO usa o interpretador Python padrão para seus exemplos. Primeiro, rode o interpretador Python, importe o modulo **re**, e compile uma RE:

```
Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

Agora, você pode tentar combinar várias strings contra a RE [a-z]+. Uma string vazia não deveria combinar em nada, desde que + foi especificado 'uma ou mais repetições'.match() deve retornar None neste caso, o que fará com que o

interpretador não imprima nenhuma saída. Você pode imprimir explicitamente o resultado do match() para deixar isso claro.

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

Agora, vamos experimentá-lo em uma string que ele deve corresponder, como `tempo`. Neste caso, **`match()`** irá retornar um **`MatchObject`**, assim que você deve armazenar o resultado em uma variável para uso posterior.

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

Agora você pode consultar o **`MatchObject`** para obter informações sobre a string correspondente. Instâncias do **`MatchObject`** também tem vários métodos e atributos; os mais importantes são os seguintes:

Método/Atributo	Propósito
`group()`	Retorna a string combinada pela RE
`start()`	Retorna a posição inicial da string combinada
`end()`	Retorna a posição final da string combinada
`span()`	Retorna uma tupla contend as posições (inicial, final) da string combinada

Tentando estes métodos logo irá esclarecer o seu significado:

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

`group()` retorna a substring que foi combinada pela ER. **`start()`** e **`end()`** retornam os índices inicial e o final da substring combinada. **`span()`** retorna tanto o índices inicial e final em uma única tupla. Como o método **`match()`** somente verifica se o RE corresponde no início de uma string, **`start()`** será sempre zero. No entanto, o

método **`search()`** de padrões varre toda string, de modo que a substring combinada pode não iniciar em zero nesse caso.

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

Nos programas de reais, o estilo mais comum é armazenar o **`MatchObject`** em uma variável e, em seguida, verificar se foi **`None`**. Isso geralmente se parece com:

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

Dois métodos padrão retornam todos as combinações de um **`pattern.findall()`** retorna uma lista de strings correspondentes:

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

`findall()` tem de criar a lista inteira antes que ele possa ser devolvido como o resultado. O método **`finditer()`** retorna uma sequência de instâncias **`MatchObject`** como um *iterator*. [1]

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

Funções de nível de módulo ¶

Você não tem que criar um objeto padrão e chamar seus métodos; o módulo **re** também fornece funções de nível superior chamada **match()**, **search()**, **findall()**, **sub()**, e assim por diante. Estas funções recebem os mesmos argumentos que o método padrão correspondente, com a string RE adicionado como o primeiro argumento, e ainda retornam **None** ou uma instância **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...>
```

Sob o capô, estas funções simplesmente criam um objeto padrão para você e chamar o método apropriado sobre ela. Eles também armazenam o objeto compilado em um cache, futuras chamadas usando a mesma RE são mais rápidas.

Se você usar essas funções de nível de módulo, ou você deve obter o padrão e chamar seus métodos de si mesmo? Essa escolha depende da frequência com que a RE será usada, e no seu estilo de codificação pessoal. Se a RE será usada em apenas um ponto no código, em seguida, as funções do módulo são provavelmente mais convenientes. Se um programa contém uma grande quantidade de expressões regulares, ou reutiliza os mesmos em vários locais, então pode valer a pena para recolher todas as definições em um lugar, em uma seção de código que compila todos as REs antes do uso. Para dar um exemplo da biblioteca padrão, aqui está um extrato de `xmllib.py`:

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

Eu geralmente prefiro trabalhar com o objeto compilado, mesmo para usos de uma só vez, mas poucas pessoas vão ser um tanto purista sobre isso como eu sou.

Flags de Compilação¶

Flags de compilação permitem modificar alguns aspectos de como as expressões regulares funcionam. Flags estão disponíveis no módulo **re** sob dois nomes, um nome longo, tais como **IGNORECASE** e um curto, de uma letra forma como **I**. (Se você estiver familiarizado com os modificadores padrão do Perl, as formas de uma letra

usar as mesmas letras; o forma abreviada de **re.VERBOSE** é **re.X**, por exemplo) Várias flags podem ser especificados por bit a bit ou intercala-los por OU(|); `re.I | re.M` define as flags **I** e **M**, por exemplo.

Aqui está a tabela das flags disponíveis, seguida por uma explicação mais detalhada de cada uma.

Flag	Significado
DOTALL, **S**	Faz . corresponder a qualquer caractere, incluindo novas linhas
IGNORECASE, **I**	Faz correspondências de maiúsculas e minúsculas
LOCALE, **L**	Faz uma correspondência local
MULTILINE, **M**	correspondência multi-linha, afetando ^ e $
VERBOSE, **X**	Habilita REs detalhado, que podem ser organizados de forma mais clara e compreensível.
UNICODE, **U**	Faz vários escapes como \w, \b \s e \d dependente da base de dados de caracteres Unicode.

I
IGNORECASE

Executa a correspondência de maiúsculas e minúsculas; classe de caracteres e strings literais irá coincidir com letras ignorando a caixa. Por exemplo, `[A-Z]` irá corresponder letras minúsculas, também, e `Spam` irá corresponder a `Spam`, `spam`, ou `spAM`. Este “lowercasing” não leva o local atual em conta; ele vai também se definir a flag **LOCALE**.

L
LOCALE

Faz `\w`, `\W`, `\b`, e `\B`, dependendo da localidade atual.

Locale é uma característica da biblioteca C destina-se a ajudar na criação de programas que tenham em conta as diferenças linguísticas. Por exemplo, se você está processando texto francês, que você gostaria de ser capaz de escrever `\w+` para combinar palavras, mas `\w` corresponde apenas a classe de caracteres `[A-Za-z]`; ele não vai ter correspondência de `'é'` ou `'ç'`. Se o sistema estiver configurado corretamente e um local francesa é seleccionado, algumas funções C vão dizer ao programa que `'é'` também deve ser considerada uma letra. Definir o flag **LOCALE** quando compilar uma expressão regular fará com que o objeto compilado resultante para usar essas funções C para `\w`; isto é mais lento, mas também permite que `\w+` para combinar palavras em francês como seria de esperar.

M
MULTILINE

(^ E $ ainda não foram explicadas, eles vão ser introduzido na seção *More Metacharacters*.)

Normalmente `^` corresponde apenas no início da string e `$` corresponde apenas no final da string e imediatamente antes da nova linha (se houver), no final da string. Quando esta flag for especificada, `^` corresponde ao início da string e no início de cada linha dentro da string, imediatamente após cada mudança de linha. Da mesma forma, o `$` metacaracter correspondências quer no fim da string e no fim de cada linha (imediatamente antes do cada nova linha).

S
DOTALL
Faz o '. caractere especial corresponder a qualquer caractere em tudo, incluindo uma nova linha; sem esta flag, '.' irá corresponder a qualquer coisa, exceto uma nova linha.

U
UNICODE
Faz `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` e `\S` dependendo das propriedades dos caracteres Unicode do banco de dados.

X
VERBOSE
Esta flag permite que você escreva expressões regulares que são mais legíveis, concedendo-lhe mais flexibilidade em como você pode formatá-los. Quando esta flag foi especificada, o espaço em branco dentro da string RE é ignorado, exceto quando o espaço em branco está em uma classe de caracteres ou precedida por uma barra invertida sem escape; isto permite-lhe organizar e recuar o RE mais claramente. Esta flag também permite que você coloque observações num RE que será ignorado pelo motor; comentários são marcadas por um "#" que não é nem em uma classe de caracteres ou precedida por uma barra invertida sem escape.

Por exemplo, aqui está uma RE que usa **`re.VERBOSE`**; veja como é muito mais fácil de ler?

```
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)
```

Sem verbose definido, seria algo como isto:

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

No exemplo acima, a concatenação automática de strings literais em Python foram usados para quebrar o RE em pedaços menores, mas ainda é mais difícil de entender do que a versão usando **`re.VERBOSE`**.

Mais Poder nos Padrões ¶

Até agora, temos apenas cobrimos uma parte dos recursos de expressões regulares. Nesta seção, vamos cobrir alguns novos metacaracteres, e como usar grupos para recuperar partes do texto que foi combinado.

Mais Metacaracteres¶

Há alguns metacaracteres que nós ainda não vimos. A maioria deles serão vistos nesta seção.

Alguns dos metacaracteres restantes a serem discutidos são de tamanho zero. Eles não usam o motor para fazer avançar através da string; em vez disso, eles consomem sem caracteres em tudo, e simplesmente tem sucesso ou falha. Por exemplo, `\b` é uma afirmação de que a posição atual está localizada em um limite de palavra; a posição não é alterada pela `\b` em tudo. Isto significa que de tamanho de zero afirmações nunca deve ser repetido, porque se eles combinam uma vez em um determinado local, eles podem, obviamente, ser combinado um número infinito de vezes.

`|`
Alternância, ou o "or" operador. Se A e B são expressões regulares, `A|B` irá corresponder a qualquer string que corresponde `A` ou `B`. `|` tem muito baixa prioridade, a fim de fazê-lo funcionar razoavelmente quando você está alternando sequências de vários caracteres. `Crow|Servo` irá corresponder quer `Crow` ou `Servo`, não `Cro`, um `'w'` ou um `'S'`, and `ervo`.

Para coincidir com um literal `'|'`, use `\|`, ou coloque ele dentro de uma classe de caracteres, como em `[|]`.

`^`

Corresponde ao início das linhas. A menos que a flag MULTILINE tiver sido definida, isso só irá corresponder no início da string. No modo MULTILINE, isso também corresponde imediatamente após cada nova linha dentro da string.

Por exemplo, se você deseja corresponder a palavra apenas no início de uma linha, o RE usado é `^From`.

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

`$`
Corresponde ao fim de uma linha, que é definido como o fim da string, ou em qualquer local, seguido por um carácter de nova linha.

```
>>> 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...>
```

Para coincidir com um literal`'$'`, use `\$` ou coloque-o dentro de uma classe de caracteres, como em `[$]`.

`\A`
Corresponde apenas no início da string. Quando não estiver em modo **`MULTILINE`**, `\A` e `^` são efetivamente o mesmo. No modo **`MULTILINE`**, eles são diferentes: `\A` ainda corresponde apenas no início da string, mas `^` pode corresponder a qualquer localização dentro da string que segue um caractere de nova linha.

`\Z`
Corresponde apenas no final da string.

`\b`
Limite de palavra. Esta é uma afirmação de largura zero que corresponde apenas no início ou no final de uma palavra. Uma palavra é definida como uma sequência de caracteres alfanuméricos, de modo que o fim de uma palavra é indicado por espaços em branco ou um carácter não alfanumérico.

O exemplo seguinte corresponde `class` apenas quando é uma palavra completa; ele não irá corresponder quando for contido dentro de uma outra palavra.

```
>>>
```

```
>>> 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
```

Há duas sutilezas você deve se lembrar ao usar essa sequência especial. Em primeiro lugar, este é a pior colisão entre strings literais do Python e sequências de expressão regular. Nos strings literais do Python, `\b` é o caráter backspace, o valor ASCII 8. Se você não estiver usando strings cruas, então Python irá converter o `\b` em um backspace e seu RE não irá corresponder, como você espera dele. O exemplo a seguir é a mesma de nossa RE anterior, mas omite o 'r' na frente da string RE.

```
>>>
```

```
>>> 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...>
```

Segundo, dentro de uma classe de caracteres, onde não há nenhum uso para esta afirmação, `\b` representa o caractere backspace, para compatibilidade com strings literais do Python.

`\B`
Outra afirmação de largura zero, isto é o oposto de `\b`, correspondentes única quando a posição corrente não é de um limite de palavra.

Agrupamento ¶

Frequentemente é necessário obter mais informações do que apenas se a RE foi correspondido ou não. As expressões regulares são muitas vezes utilizados para dissecar strings escrevendo uma RE dividida em vários subgrupos que corresponda a diferentes componentes de interesse. Por exemplo, uma linha de cabeçalho RFC-822 é dividida em um nome e um valor de cabeçalho, separados por um ‘:’, como este:

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

Isto pode ser manipulado ao escrever uma expressão regular que coincide com uma linha inteira de cabeçalho, e tem um grupo que corresponde ao nome do cabeçalho, e um outro grupo, que corresponde ao valor do cabeçalho.

Os grupos são marcados pelos `'('`, `')'` metacaracteres. `'('` e `')'` têm muito o mesmo significado que eles fazem em expressões matemáticas; se agrupam as expressões contidas dentro deles, e você pode repetir o conteúdo de um grupo com um qualificador de repetição, como `*`, `+`, `?`, ou `{m,n}`. Por exemplo, `(ab)*` irá corresponder a zero ou mais repetições de `ab`.

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

Grupos indicados com `'('`, `')'` também capturam o índice inicial e final do texto que eles correspondem; esta pode ser recuperada por meio de um argumento para **`group()`**, **`start()`**, **`end()`**, e **`span()`**. Os grupos são numerados começando com 0. Grupo 0 está sempre presente; é todo o RE, por isso MatchObject métodos têm todos grupo 0 como seu argumento padrão. Mais tarde veremos como expressar grupos que não captam a extensão de texto que eles combinam.

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

Subgrupos são numeradas a partir da esquerda para a direita, a partir de 1 para cima. Os grupos podem ser aninhados; para determinar o número, basta contar os caracteres de abertura parêntese, indo da esquerda para a direita.

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

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

`group()` podem ser passados vários grupos de números de uma vez, caso em que ele irá retornar uma tupla contendo os valores correspondentes para esses grupos.

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

O método **`groups()`** retorna uma tupla contendo as strings de todos os subgrupos, de 1 até enquanto houverem.

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

Referências para trás em um padrão permitem que você especifique que o conteúdo de um grupo de captura anteriormente também devem ser encontrados na posição atual na sequência. Por exemplo, `\1` terá sucesso se o conteúdo exato do grupo 1 podem ser encontrados na posição atual, e não o contrário. Lembre-se que strings literais do Python também usam uma barra invertida seguida por números para permitir inclusão de caracteres arbitrários em uma string, por isso certifique-se de usar strings cruas ao incorporar referências anteriores em uma RE.

Por exemplo, a seguinte RE detecta palavras duplicadas em uma string.

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

Referências para trás como esta não são muito úteis para apenas pesquisar através de um string — há poucos formatos de texto que se repetem dados dessa forma — mas em breve você vai descobrir que eles são muito úteis ao realizar substituições de cadeia.

Não captura e Grupos Nomeados¶

REs elaborados podem usar muitos grupos, tanto para capturar substrings de interesse, e para agrupar e estruturar o próprio RE. Em REs complexas, torna-se difícil manter o controle dos números dos grupos. Existem duas características que ajudam

a este problema. Ambos usam uma sintaxe comum para extensões de expressão regular, por isso vamos olhar para isso em primeiro lugar.

Perl 5 acrescentou vários recursos adicionais para expressões regulares padrão, e o módulo **re** Python suporta a maioria deles. Teria sido difícil escolher novos metacharactres single-keystroke ou novas sequências especiais que começam com \ para representar os novos recursos sem fazer expressões regulares do Perl confusamente diferente de REs padrão. Se você escolheu `&` como um novo metacharacter, por exemplo, velhas expressões estariam assumindo que o & era um personagem regular e não teria que escapa-lo escrevendo `\&` ou `[&]`.

A solução escolhida pelos programadores Perl foi usar `(?...)` Como a sintaxe de extensão. `?` imediatamente após um parêntese era um erro de sintaxe porque o `?` não teria nada a repetir, de modo que este não introduzir quaisquer problemas de compatibilidade. Os caracteres imediatamente após a `?` indicam que a extensão está sendo usado, então `(?=foo)` é uma coisa (a afirmação lookahead positivo) e `(?:foo)` é outra coisa (um grupo de não captura contendo a subexpressão `foo`).

Python adiciona uma sintaxe de extensão de sintaxe de extensão do Perl. Se o primeiro caractere após o ponto de interrogação é um `P`, você sabe que é uma extensão que é específico para Python. Atualmente, existem duas tais extensões : `(?P<name>...)` define um grupo nomeado e `(?P=name)` é uma referência anterior a um grupo chamado. Se futuras versões de Perl 5 adicionar funcionalidades semelhantes utilizando uma sintaxe diferente, o módulo **re** vai ser alterado para suportar a nova sintaxe, preservando a sintaxe específica do Python por causa da compatibilidade.

Agora que nós vimos que a sintaxe geral de extensão, podemos voltar para as características que simplificam o trabalho com grupos em REs complexos. Como os grupos são numerados a partir da esquerda para a direita e uma expressão complexa pode usar muitos grupos, pode tornar-se difícil manter o controle da numeração correta. Modificando um RE tão complexo é irritante, também: inserir um novo grupo perto do início e você tem que alterar os números de tudo o que se segue.

Às vezes você vai querer usar um grupo para recolher uma parte de uma expressão regular, mas não estão interessados em recuperar o conteúdo do grupo. Você pode fazer este fato explícito usando um grupo de não-captura: (?: ...), Onde você pode substituir o ... com qualquer outra expressão regular.

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

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

Exceto pelo fato de que não é possível recuperar o conteúdo de que o grupo combinado, um grupo de não captura se comporta exatamente o mesmo que um grupo de captura; você pode colocar qualquer coisa dentro dele, repeti-la com um metacharacter de repetição, como *, e aninhá-lo dentro de outros grupos (captura ou não de captura). `(?:...)` é particularmente útil ao modificar um padrão existente, desde que você pode adicionar novos grupos, sem alterar a forma como todos os outros grupos estão contados. Refira-se que não há diferença de desempenho na busca entre captação e grupos de não captura; nem qualquer forma é mais rápido do que o outro.

Uma característica mais significativa dos grupos nomeados é: em vez de se referir a eles por números, os grupos podem ser referenciados por um nome.

A sintaxe de um grupo nomeado é uma das extensões específicas do Python : `:(?P<name>...)`. *name* é, obviamente, o nome do grupo. Os grupos nomeados também se comportam exatamente como grupos de captura, e, adicionalmente, associam um nome com um grupo. Os métodos MatchObject que lidam com grupos de captura todos aceitam tanto inteiros que se referem ao grupo em número ou em cadeias que contêm o nome do grupo desejado. Os grupos nomeados ainda recebem números, para que possa recuperar informações sobre um grupo de duas maneiras:

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

Os grupos nomeados são úteis porque eles permitem que você use nomes fácil de lembrar, em vez de ter que lembrar de números. Aqui está um exemplo RE a partir do módulo **`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'"')
```

É óbvio que é muito mais fácil para recuperar m.group ('zonem'), em vez de ter que se lembrar de grupo 9 recuperar.

A sintaxe para referências anteriores em uma expressão, tais como `(...)\1` refere-se ao número do grupo. Há, naturalmente, uma variante que usa o nome do grupo em vez do número. Isto é outra extensão Python: `(?P=name)` Indica que o conteúdo do grupo chamado *name* deve novamente ser correspondido no ponto atual. A expressão regular para encontrar palavras duplicadas, `(\b\w+)\s+\1` também pode ser escrita como `(?P<word>\b\w+)\s+(?P=word)`:

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

Lookahead Assertions¶

Outra afirmação de largura zero é a afirmação lookahead. Afirmações LookAhead estão disponíveis tanto na forma positiva e negativa, e parecido com isto:

`(?=...)`
Afirmação lookahead positivo. Isto bem-sucedido se a expressão regular contida, aqui representada por `...`, corresponde ao com sucesso no local atual, e não o contrário. Mas, uma vez que a expressão contida foi tentada, o motor de correspondência não avança em tudo; o resto do padrão é tentado a direita onde a afirmação começou.

`(?!...)`
Afirmação lookahead negativo. Isto é o oposto da afirmação positiva; bem-sucedida se a expressão contida não corresponde na posição atual na seqüência.

Para tornar isto concreto, vamos olhar para um caso em que um lookahead é útil. Considere-se um padrão simples para coincidir com um nome de arquivo e dividi-lo à parte, a um nome de base e uma extensão, separados por um `.`. Por exemplo, em `news.rc`,`news` é o nome base, e `rc` é a extensão do nome de arquivo.

O padrão para corresponder a isto é muito simples:

```
.*[.].*$
```

Observe que o `.` precisa ser tratado de forma especial, porque é um metacaractere; Eu colocá-lo dentro de uma classe de caracteres. Também note o trailer `$`; este é adicionado para garantir que todo o resto da cadeia devem ser incluídos na extensão. Esta expressão regular corresponde: `foo.bar`, `autoexec.bat`, `sendmail.cf` e `printers.conf`.

Agora, considere o que complica o problema um pouco; E se você deseja corresponder nomes de arquivos onde a extensão não é `bat`? Algumas tentativas incorretas:

```
.*[.][^b].*$
```

A primeira tentativa acima tenta excluir `bat`, exigindo que o primeiro caractere da extensão não é um `b`. Isso é errado, porque o padrão também não corresponde `foo.bar`.

```
.*[.]([^b]..|.[^a].|..[^t])$
```

A expressão fica mais confusa quando você tenta consertar a primeira solução, exigindo um dos seguintes casos ao jogo: o primeiro caractere da extensão não é `b`; o segundo personagem não é `a`; ou o terceiro personagem não é `t`. Esta aceita `foo.bar` e rejeita `autoexec.bat`, mas requer uma extensão de três letras e não aceitará um nome de arquivo com uma extensão de duas letras, como `sendmail.cf`. Vamos complicar o padrão novamente em um esforço para corrigi-lo.

```
.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$
```

Na terceira tentativa, os segundo e terceiro letras são todos feitos opcional, a fim de permitir que as extensões mais curtas do que três caracteres, tais como `sendmail.cf` correspondentes.

O padrão está ficando muito complicado agora, o que faz com que seja difícil de ler e compreender. Pior ainda, se o problema muda e você quiser excluir tanto `bat` e `exe` como extensões, o padrão iria ficar ainda mais complicado e confuso.

Um lookahead negativo corta toda esta confusão:

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

O lookahead negativo significa: se a expressão `bat` não corresponde, neste momento, tente o resto do padrão; se `bat$` faz correspondência, todo o padrão irá falhar. O trailer $ é necessária para garantir que algo como `sample.batch`, onde a extensão só começa com o `bat`, será permitido.

Excluindo uma outra extensão de arquivo agora é fácil; basta adicioná-lo como uma alternativa dentro da afirmação. O padrão a seguir exclui os nomes de arquivos que terminam em qualquer `bat` ou `exe`:

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

Modificando Strings¶

Até este ponto, nós simplesmente realizado pesquisas contra uma string estática. As expressões regulares também são comumente usadas para modificar strings de vários meios, usando os seguintes métodos padrão:

Método/Atributo	Propósito
`split()`	Divide a string em uma lista, dividindo-a onde quer que haja correspondência da RE
`sub()`	Pesquisa todos os substrings correspondentes a RE, e os substituí por uma string diferente
`subn()`	Faz a mesma coisa que **`sub()`**, mas retorna a nova string e o número de substituições

Splitting Strings¶

O método **`split()`** de um padrão divide uma string aparte onde quer que a RE corresponda, retornando uma lista das peças. É semelhante ao método **`split()`** de strings, mas oferece muito mais generalidade nos delimitadores que podem ser divididas por; **`split()`** só suporta a divisão de espaço em branco ou por uma string fixa. Como seria de esperar, há uma função de nível de módulo **`re.split()`** também.

`.split`(*string*[, *maxsplit=0*])
Divide a *string* pelas correspondências da expressão regular. Se parênteses de captura são utilizados na RE, em seguida, seu conteúdo também será retornado como parte da lista resultante. Se *maxsplit* é diferente de zero, na maioria das divisões *maxsplit* são executadas.

Você pode limitar o número de divisões feitas, passando um valor para *maxsplit*. Quando maxsplit é diferente de zero, na maioria das divisões *maxsplit* será feita, e o restante da string é retornado como o elemento final da lista. No exemplo a seguir, o delimitador é qualquer sequência de caracteres não alfanuméricos.

```
>>> 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().']
```

Às vezes, você não está apenas interessado no que o texto que está entre delimitadores, mas também precisa saber qual o delimitador foi usado. Se parênteses de captura são utilizados no RE, em seguida, os respectivos valores são também retornados como parte da lista. Compare as seguintes chamadas:

```
>>>
>>> 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', '.', '']
```

A função do nível de módulo **`re.split()`** adiciona o RE para ser utilizado como o primeiro argumento, mas de outra forma é a mesma.

```
>>>
>>> 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.']
```

Search e Replace¶

Outra tarefa comum é encontrar todas as combinações para um padrão e substituí-las por uma string diferente. O método **`sub()`** recebe um valor de substituição, que pode ser uma string ou uma função, e a string a ser processada.

`.sub`(*replacement*, *string*[, *count=0*])
Retorna a string obtida substituindo as ocorrências mais à esquerda não sobrepostas do RE em *string* pela substituição de *replacement*. Se o padrão não for encontrado, a *string* é retornada inalterada.

O argumento opcional *count* é o número máximo de ocorrências de padrão a ser substituído; *count* deve ser um número inteiro não negativo. O valor padrão de 0 significa para substituir todas as ocorrências.

Aqui está um exemplo simples do uso do método de **`sub()`**. Ele substitui nomes de cores pela palavra `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'
```

O método **`subn()`** faz o mesmo trabalho, mas retorna uma tupla que contém o novo valor de cadeia e o número de substituições que foram realizadas:

```
>>>
>>> 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)
```

Correspondências vazias somente são substituídas quando não estão adjacente a uma correspondência anterior.

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

Se a substituição é uma string, qualquer barra invertida escapa em que são processados. Isto é, `\n` é convertido a um único caractere de nova linha, `\r` é convertido em um retorno do carro, e assim por diante. Escapes desconhecidos, como `\j` são deixados sozinhos. Referências para trás, como `\6`, são substituídas com a substring correspondida pelo grupo correspondente no RE. Isso permite que você incorporar partes do texto original na cadeia de substituição resultante.

Este exemplo corresponde à palavra `section` seguida por uma string colocada entre `{`, `}`, e altera `section` para `subsection`:

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

Há também uma sintaxe para se referir a grupos nomeados como definido pela `(?P<name>...)` sintaxe. `\g<name>` usará a substring correspondida pelo grupo nomeado `name` e `\g<number>` utiliza o número do grupo correspondente. `.\g<2>` é, portanto, equivalente a `\2`, mas não é ambígua em uma cadeia de substituição, tais como `\g<2>0`. (`\20` seria interpretada como uma referência ao grupo de 20, e não uma referência ao grupo 2 seguido pelo caractere literal `'0'`). As seguintes

substituições são todas equivalentes, mas use todas as três variações do texto de substituição.

```
>>>
>>> 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}'
```

***Sub**stituição* também pode ser uma função, que lhe dá ainda mais controle. Se a substituição é uma função, a função é chamada para cada ocorrência não sobreposição de padrão. Em cada chamada, a função é passado um argumento MatchObject para a correspondência e pode usar esta informação para calcular a string de substituição desejada e devolvê-lo.

No exemplo a seguir, a função de substituição traduz decimais em hexadecimal:

```
>>>
>>> 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.'
```

Ao utilizar a função de nível de módulo re.sub (), o padrão é passado como o primeiro argumento. O padrão pode ser fornecida como um objecto ou como uma string; se você precisa especificar flags de expressões regulares, você deve usar um objeto padrão como o primeiro parâmetro, ou use modificadores incorporados na cadeia padrão, por exemplo, `sub("(?i)b+", "x", "bbbb BBBB")` retorna `'x x'`.

Problemas Comuns ¶

Expressões regulares são uma ferramenta poderosa para algumas aplicações, mas de certa forma o seu comportamento não é intuitivo e, às vezes eles não se comportam da maneira que você pode esperar que eles. Esta seção irá apontar algumas das armadilhas mais comuns.

Use String Methods¶

Às vezes usando o módulo **re** é um equívoco. Se você está combinando uma string fixa, ou uma única classe de caracter, e você não está usando nenhum recurso de **re** como a flag **IGNORECASE**, então pode não ser necessária toda a potência de expressões regulares. Strings tem vários métodos para executar operações com strings fixas e eles são geralmente muito mais rápido, porque a implementação é um único pequeno laço C que foi otimizado para o efeito, em vez do grande mecanismo de expressão regular mais generalizado,.

Um exemplo pode ser a substituição de uma string fixa única por outra; por exemplo, você pode substituir `word` por `deed`. `re.sub()` parece ser a função a ser usada para isso, mas considere o método **replace()**. Note que **replace()** também irá substituir `word` dentro de palavras, transformando `swordfish` em `sdeedfish`, mas a palavra RE ingênuo teria feito isso também. (Para evitar a realização da substituição de partes de palavras, o padrão teria que ser `\bword\b`, a fim de exigir que `word` tenha um limite de palavra em ambos os lados. Isso leva o trabalho para além da capacidade do **replace()**.)

Outra tarefa comum é apagar todas as ocorrências de um único caractere de uma string ou substituindo-o por outro único caracter. Você pode fazer isso com algo como `re.sub('\n', ' ', S)`, mas **translate()** é capaz de fazer ambas as tarefas e será mais rápido do que qualquer operação de expressão regular pode ser.

Em suma, antes de recorrer ao o módulo **re**, considere se o seu problema pode ser resolvido com um método string mais rápido e mais simples.

match() versus search()¶

A função **match()** somente verifica se o RE corresponde ao início da string, enquanto **search()** fará a varredura para frente através na string para uma combinação. É importante manter esta distinção em mente. Lembre-se, **match()** só irá relatar um casamento bem-sucedido, que começará no 0; Se a coincidência não iria começar do zero, **match()** não vai reportá-lo.

```
>>>
>>> print re.match('super', 'superstition').span()
```

```
(0, 5)
>>> print re.match('super', 'insuperable')
None
```

Por outro lado, **`search()`** fará a varredura para frente através da string, relatando a primeira correspondência que encontrar.

```
>>>
>>> print re.search('super', 'super.stition').span()
(0, 5)
>>> print re.search('super', 'insuperable').span()
(2, 7)
```

Às vezes você vai ser tentado a continuar usando **`re.match()`**, e apenas adicionar `.*` na frente de sua RE. Resista a essa tentação e use **`re.search()`** em vez disso. O compilador de expressão regular faz alguma análise das REs, a fim de acelerar o processo de procura de uma combinação. Tal análise descobre o que o primeiro caractere de uma string deve ser; por exemplo, um padrão começando com `Crow` deve coincidir com um começando com `'C'`. A análise permite que o motor de varredura rapidamente através da string de olhar para o caracter de partida, apenas a tentar a combinação completa se um `'C'` é encontrado.

Adicionando `.*` perde essa otimização, o que requer a digitalização para o fim da string e, em seguida, recuar para encontrar uma correspondência para o resto do RE. Use **`re.search()`** em vez disso.

Gulosos versus não Gulosos ¶

Ao repetir uma expressão regular, como em `a*`, a ação resultante é consumir tanto do padrão quanto possível. Este fato, muitas vezes fere quando você está tentando combinar um par de delimitadores equilibrado, como os colchetes que cercam uma tag HTML. O padrão ingênuo para combinar uma única tag HTML não funciona por causa da natureza gulosa de `.*`.

```
>>>
>>> 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>
```

O RE corresponde a `'<'` em `<html>`, e o `.*` Consome o resto da string. Há ainda mais à esquerda no RE, no entanto, e o `>` pode não corresponder, no final da string, de modo que o mecanismo de expressão regular tem que recuar caractere por caractere até encontrar uma correspondência para a `>`. A correspondência final se estende do `'<'` em `<html>` ao `'>'` em `</title>`, que não é o que você quer.

Neste caso, a solução é usar os qualificadores não-gulosos `*?`, `+?`,`??`, or `{m,n}?`, que corresponde o mínimo de texto possível. No exemplo acima, o `'>'` é tentado imediatamente após a primeira '<' combinação, e quando ele falhar, o motor avança um carácter de cada vez, e tenta novamente a ">" a cada passo. Isso produz apenas o resultado correto:

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

(Note que a análise de HTML ou XML com expressões regulares é dolorosa. Padrões sujos rápidos irão lidar com casos comuns, mas HTML e XML tem casos especiais que irão quebrar a expressão regular óbviamente; pelo tempo que você tenha escrito uma expressão regular que lida com todos os casos possíveis, os padrões será muito complicado. Use um módulo de HTML ou XML parser para tais tarefas.)

Usando re.VERBOSE¶

Até agora você já deve ter notado que as expressões regulares são uma notação muito compacta, mas elas não são extremamente legíveis. REs de complexidade moderada pode se tornar longas coleções de barras invertidas, parênteses e metacaracteres, tornando-as difíceis de ler e compreender.

Para essas REs, especificando a flag `re.VERBOSE` ao compilar a expressão regular pode ser útil, porque permite que você formate a expressão regular de forma mais clara.

A flag `re.VERBOSE` tem vários efeitos. Espaço em branco na expressão regular que não está dentro de uma classe de caracteres é ignorado. Isto significa que uma expressão como `dog | cat` é equivalente ao menos legível `dog|cat`, mas `[a b]` ainda vai coincidir com os caracteres `'a'`, `'b'`, ou um espaço. Além disso, você também pode colocar comentários dentro de um RE; observações se estendem a partir de uma # character para o final da linha. Quando usado em strings com aspas triplas, isso permite REs a ser formatado mais ordenadamente:

```
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)
```

Isto é muito mais legível do que:

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

Comentários ¶

As expressões regulares são um tópico complicado. Será que este documento ajudo-o a compreendê-las? Houve partes que foram pouco claras, ou problemas encontrados que não foram abordados aqui? Se assim for, por favor, envie sugestões de melhorias para o autor.

O livro mais completo sobre expressões regulares é quase certamente expressões regulares Mastering de Jeffrey Friedl, publicado pela O'Reilly. Infelizmente, concentra-se exclusivamente em Perl e sabores de expressões regulares do Java, e não contém qualquer material Python em tudo, por isso não vai ser útil como uma referência para a programação em Python. (A primeira edição coberta módulo regex agora removido do Python, que não vai ajudar muito.) Considere a verificar-se a partir de sua biblioteca.

N.T.

- Meu primeiro contato com expressões regulares foi com o livro Expressões Regulares - Guia de Consulta Rápida por Aurelio Marinho Jargas e Editora Novatec, ©2001, é uma abordagem divertida sobre expressões regulares embora não seja específico sobre Python, você aprende brincando com REs. E mesclando com este HOWTO dará a você uma boa base de conhecimentos sobre RE.
- Alguns bons recursos para testar suas RE s, são:
    - http://www.pyregex.com/
    - http://tools.lymas.com.br/regexp_br.php#

Footnotes

[1] Introduced in Python 2.2.2.