Expressões Regulares HOWTO

Release 3.8.18

Guido van Rossum
and the Python development team

fevereiro 08, 2024

Python Software Foundation
Email: docs@python.org

Sumário

Autor A.M. Kuchling <amk@amk.ca>

Resumo

Este documento é um tutorial introdutório sobre expressões regulares em Python com o módulo `re`. Ele provê uma introdução mais tranquila que a seção correspondente à documentação do módulo.

1 Introdução

Expressões regulares (chamadas REs, ou regexes ou padrões regex) são essencialmente uma mini linguagem de programação altamente especializada incluída dentro do Python e disponível através do módulo `re`. Usando esta pequena linguagem, você especifica as regras para o conjunto de strings possíveis que você quer combinar; esse conjunto pode conter sentenças em inglês, endereços de e-mail, ou comandos TeX ou qualquer coisa que você queira. Você poderá então perguntar coisas como "Essa string se enquadra dentro do padrão?" ou "Existe alguma parte da string que se enquadra nesse padrão?". Você também pode usar as REs para modificar uma string ou dividi-la de diversas formas.

Os padrões das expressões regulares são compilados em uma série de bytecodes que são então executadas por um mecanismo de combinação escrito em C. Para usos avançados, talvez seja necessário prestar atenção em como o mecanismo irá executar uma dada RE, e escrever a RE de forma que os bytecodes executem de forma mais rápida. Otimização é um tema que não será visto neste documento, porque ele requer que você tenha um bom entendimento dos mecanismos de combinação internos.

A linguagem de expressões regulares é relativamente pequena e restrita, então nem todo tipo de processamento de strings pode ser feito através dela. Existem tarefas que *podem* ser feitas com expressões regulares, mas as expressões acabam ficando muito complicadas. Nesses casos, é melhor escrever código Python para fazer o processamento; apesar do código Python ficar mais lento que uma expressão regular elaborada, ele provavelmente será mais facilmente compreensível.

2 Padrões Simples

Vamos começar aprendendo o expressão regular mais simples possível. Dado que expressões regulares são usado para operar com strings, nós começaremos com o tipo mais comum de tarefa: combinando caracteres.

Para uma explicação detalhada da ciência por trás das expressões regulares (determinística e não-determinística finita automata), você pode procurar na maioria dos livros sobre escrita de compiladores.

2.1 Combinando Caracteres

A maior parte das letras e caracteres irão simplesmente combinar a si mesmo. Por exemplo, a expressão regular `test` irá corresponder à string `test` exatamente. (Você pode ativar modo que é insensível a letras maiúsculas que permite essa RE corresponder também a `Test` ou `TEST`; mais sobre isso posteriormente.)

Existem exceções para esta regra; alguns caracteres são *metacaracteres* especiais, e não combinam com eles mesmos. Ao invés disso, eles sinalizam que alguma coisa fora do comum deve ser combinada ou eles afetam outra porção da RE por se repetirem ou modificando seu significado. A maior parte desse documento é dedicado em discutir vários metacaracteres e o que eles fazem.

Aqui está a lista completa de metacaracteres; seus significados serão discutidos ao longo deste documento.

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

O primeiro metacaractere que vamos estudos é o `[` e o `]`. Eles são usados para especificar uma classe de caracteres, que é um conjunto de caracteres que você deseja combinar. Caracteres podem ser listados individualmente ou um range de caracteres pode ser indicado dando dois caracteres e separando-os por um `'-'`. Por exemplo, `[abc]` irá encontrar qualquer caractere `a`, `b`, ou `c`; isso é o mesmo que escrever `[a-c]`, que usa um range para expressar o mesmo conjunto de caracteres, Se você deseja encontrar apenas letras minúsculas, sua RE seria `[a-z]`.

Metacaracteres não são ativos dentro de classes. Por exemplo, `[akm$]` irá combinar com qualquer dos caracteres `'a'`, `'k'`, `'m'`, or `'$'`; `'$'` é normalmente um metacaractere, mas dentro de uma classe de caracteres ele perde sua natureza especial.

Você pode combinar os caracteres não listados na classe *complementando* o conjunto. Isso é indicado pela inclusão de um `'^'` como o primeiro caractere da classe. Por exemplo, `[^5]` combinará a qualquer caractere, exceto `'5'`. Se o sinal de intercalação aparecer em outro lugar da classe de caracteres, ele não terá um significado especial. Por exemplo: `[5^]` corresponderá a um `'5'` ou a `'^'`.

Talvez o metacaractere mais importante é a contrabarra, `\`. Como as strings literais em Python, a barra invertida pode ser seguida por vários caracteres para sinalizar várias sequências especiais. Ela também é usada para *escapar* todos os metacaracteres, e assim, você poder combiná-los em padrões; por exemplo, se você precisa fazer correspondência 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 de caracteres predefinidos que são frequentemente úteis, como o conjunto de dígitos, o conjunto de letras ou o conjunto de qualquer coisa que não seja espaço em branco.

Vejamos um exemplo: `\w` corresponde a qualquer caractere alfanumérico. Se o padrão regex for expresso em bytes, isso é equivalente à classe `[a-zA-Z0-9_]`. Se o padrão regex for uma string, `\w` combinará todos os caracteres marcados como letras no banco de dados Unicode fornecido pelo módulo `unicodedata`. Você pode usar a definição mais restrita de `\w` em um padrão de string, fornecendo o sinalizador `re.ASCII` ao compilar a expressão regular.

A lista a seguir de sequências especiais não está completa. Para obter uma lista completa das sequências e definições de classe expandidas para padrões de Strings Unicode, veja a última parte de Sintaxe de Expressão Regular na referência da Biblioteca Padrão. Em geral, as versões Unicode correspondem a qualquer caractere que esteja na categoria apropriada do banco de dados Unicode.

`\d` corresponde a qualquer `dígito decimal`, que é equivalente à classe `[0-9]`.

`\D` corresponde a qualquer caractere `não-dígito`, o que é equivalente à classe `[^0-9]`.

`\s` corresponde a qualquer caractere `espaço-em-branco`, o que é equivalente à classe `[\t\n\r\f\v]`.

`\S` corresponde a qualquer caractere `não-espaço-branco`, o que é equivalente à classe `[^\t\n\r\f\v]`.

`\w` corresponde a qualquer caractere `alfanumérico`, o que é equivalente à classe `[azA-Z0-9_]`.

`\W` corresponde a qualquer caractere `não-alfanumérico`, o que é equivalente à classe `[^a-zA-Z0-9_]`.

Estas sequências podem ser incluídas dentro de uma classe caractere. Por exemplo, `[\s,.]` É uma classe caractere 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 existe um modo alternativo (`re.DOTALL`), onde ele irá corresponder até mesmo a um caractere de nova linha. `.` é frequentemente usado quando você quer corresponder com “qualquer caractere”.

2.2 Repetindo Coisas

Ser capaz de corresponder com variados conjuntos de caracteres é a primeira coisa que as expressões regulares podem fazer que ainda não é possível com os métodos disponíveis para strings. No entanto, se essa fosse a única capacidade adicional das expressões regulares, elas não seriam um avanço relevante. Outro recurso que você pode especificar é que partes do RE devem ser repetidas um certo número de vezes.

O primeiro metacaractere para repetir coisas que veremos é `*`. `*` não corresponde ao caractere literal `'*'`; em vez disso, ele especifica que o caractere anterior pode ser correspondido zero ou mais vezes, em vez de exatamente uma vez.

Por exemplo, `ca*t` vai corresponder `'ct'` (0 `'a'` caracteres), `'cat'` (1 `'a'`), `'caaat'` (3 `'a'` caracteres), e assim por diante.

Repetições, tais como `*` são gananciosas; ao repetir a RE, o motor de correspondência vai tentar repeti-la 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 fará isso mais óbvio. Vamos considerar a expressão "a[bcd]*b". Isto corresponde à letra "a", zero ou mais letras da classe "[bcd]" e, finalmente, termina com "b". Agora, imagine corresponder este RE com a string "abcbd".

Passo	Correspondência	Explanação
1	`a`	O caractere `a` na RE tem correspondência.
2	`abcbd`	O motor corresponde com [bcd]*, indo tão longe quanto possível, que é o fim do string.
3	*Failure*	O motor tenta corresponder com `b`, mas a posição corrente está no final da string, então ele falha.
4	`abcb`	Voltando, de modo que `[bcd]*` corresponde a um caractere a menos.
5	*Failure*	Tenta `b` novamente, mas a posição corrente é a do último caractere, que é um `d`.
6	`abc`	Voltando novamente, de modo que [bcd]* está correspondendo com `bc` somente.
6	`abcb`	Tenta `b` novamente. Desta vez, o caractere na posição corrente é `b`, por isso sucesso.

O final da RE foi atingido e correspondeu a `'abcb'`. Isso demonstra como o mecanismo de correspondência vai tão longe quanto pode no início e, se nenhuma correspondência for encontrada, ele fará o backup progressivamente e tentará novamente o restante da RE. Ele fará backup até que tenha tentado zero correspondências para `[bcd]*`, e se isso falhar subsequentemente, o mecanismo concluirá que a string não corresponde a RE de forma alguma.

Another repeating metacharacter is `+`, which matches one or more times. Pay careful attention to the difference between `*` and `+`; `*` matches *zero* or more times, so whatever's being repeated may not be present at all, while `+` requires at least *one* occurrence. To use a similar example, `ca+t` will match `'cat'` (1 `'a'`), `'caaat'` (3 `'a'`s), but won't match `'ct'`.

There are two more repeating qualifiers. The question mark character, `?`, matches either once or zero times; you can think of it as marking something as being optional. For example, `home-?brew` matches either `'homebrew'` or `'home-brew'`.

The most complicated repeated qualifier is `{m,n}`, where *m* and *n* are decimal integers. This qualifier means there must be at least *m* repetitions, and at most *n*. For example, `a/{1,3}b` will match `'a/b'`, `'a//b'`, and `'a///b'`. It won't match `'ab'`, which has no slashes, or `'a////b'`, which has four.

You can omit either *m* or *n*; in that case, a reasonable value is assumed for the missing value. Omitting *m* is interpreted as a lower limit of 0, while omitting *n* results in an upper bound of infinity.

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

3 Usando expressões regulares

Agora que nós vimos algumas expressões regulares simples, como nós realmente as usamos em Python? O módulo `re` fornece uma interface para o mecanismo de expressão regular, permitindo compilar REs em objetos e, em seguida, executar comparações com eles.

3.1 Compilando Expressões Regulares

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

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

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

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

A RE é passada para `re.compile()` como uma string. REs são tratadas como strings porque as expressões regulares não são parte do núcleo da linguagem Python, e nenhuma sintaxe especial foi criada para expressá-las. (Existem aplicações que não necessitam de REs nenhuma, por isso não há necessidade de inchar a especificação da linguagem, incluindo-as.) 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.

3.2 A praga da barra invertida

Como afirmado anteriormente, expressões regulares usam o caractere de barra invertida (`\`) para indicar formas especiais ou para permitir que caracteres especiais sejam usados sem invocar o seu significado especial. Isso entra em conflito com o uso pelo Python do mesmo caractere para o mesmo propósito nas strings literais.

Vamos dizer que você quer escrever uma RE que corresponde com a string `\section`, que pode ser encontrada em um arquivo LaTeX. Para descobrir o que escrever no código do programa, comece com a string que se deseja corresponder. Em seguida, você deve preceder qualquer barra invertida e outros metacaracteres com uma barra invertida, tendo como resultado a 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 precedidas com uma barra invertida novamente.

Caracteres	Etapa
`\section`	Text string to be matched
`\\section`	preceder com barra invertida para `re.compile()`
`"\\\\section"`	barras invertidas precedidas novamente para uma string literal

Em suma, para corresponder com uma barra invertida literal, tem de se escrever `'\\\\'` como a string da RE, porque a expressão regular deve ser `\\`, e cada barra invertida deve ser expressa como `\\` dentro de uma string literal Python normal. 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 tratadas de nenhuma forma especial em uma string literal se prefixada com `r`, então `r"\n"` é uma string 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 no código Python usando esta notação de string crua (raw).

In addition, special escape sequences that are valid in regular expressions, but not valid as Python string literals, now result in a DeprecationWarning and will eventually become a SyntaxError, which means the sequences will be invalid if raw string notation or escaping the backslashes isn't used.

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

3.3 Executando Comparações

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

Método/Atributo	Purpose
match()	Determina se a RE combina com o início da string.
search()	Varre toda a string, procurando qualquer local onde esta RE tem correspondência.
findall()	Encontra todas as substrings onde a RE corresponde, e as retorna como uma lista.
finditer()	Encontra todas as substrings onde a RE corresponde, e as retorna como um iterador.

match() and search() return None if no match can be found. If they're successful, a match object instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the re module. If you have tkinter available, you may also want to look at Tools/demo/redemo.py, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. redemo.py can be quite useful when trying to debug a complicated RE.

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

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```

Now, you can try matching various strings against the RE [a-z]+. An empty string shouldn't match at all, since + means 'one or more repetitions'. match() should return None in this case, which will cause the interpreter to print no output. You can explicitly print the result of match() to make this clear.

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

Now, let's try it on a string that it should match, such as tempo. In this case, match() will return a match object, so you should store the result in a variable for later use.

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

Now you can query the match object for information about the matching string. Match object instances also have several methods and attributes; the most important ones are:

Método/Atributo	Purpose
`group()`	Retorna a string que corresponde com a RE
`start()`	Retorna a posição inicial da string correspondente
`end()`	Retorna a posição final da string correspondente
`span()`	Retorna uma tupla contendo as posições (inicial, final) da string combinada

Experimentando estes métodos teremos seus significado esclarecidos:

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

`group()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match()` method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

Nos programas reais, o estilo mais comum é armazenar o objeto Match em uma variável e, em seguida, verificar se ela é `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')
```

Two pattern methods return all of the matches for a pattern. `findall()` returns a list of matching strings:

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

The `r` prefix, making the literal a raw string literal, is needed in this example because escape sequences in a normal "cooked" string literal that are not recognized by Python, as opposed to regular expressions, now result in a `DeprecationWarning` and will eventually become a `SyntaxError`. See *A praga da barra invertida*.

`findall()` has to create the entire list before it can be returned as the result. The `finditer()` method returns a sequence of match object instances as an iterator:

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

3.4 Funções de Nível de Módulo

You don't have to create a pattern object and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `findall()`, `sub()`, and so forth. These functions take the same arguments as the corresponding pattern method with the RE string added as the first argument, and still return either `None` or a match object instance.

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From '>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE won't need to parse the pattern again and again.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? If you're accessing a regex within a loop, pre-compiling it will save a few function calls. Outside of loops, there's not much difference thanks to the internal cache.

3.5 Sinalizadores de Compilação

Sinalizadores de compilação permitem modificar alguns aspectos de como as expressões regulares funcionam. Sinalizadores estão disponíveis no módulo `re` sob dois nomes, um nome longo, tal como `IGNORECASE` e um curto, na forma de uma letra, como `I`. (Se você estiver familiarizado com o padrão dos modificadores do Perl, o nome curto usa as mesmas letras; o forma abreviada de `re.VERBOSE` é `re.X`, por exemplo). Vários sinalizadores podem ser especificados aplicando um `OU` bit a bit nelas; `re.I | re.M` define os sinalizadores `I` e `M`, por exemplo.

Aqui está uma tabela dos sinalizadores disponíveis, seguida por uma explicação mais detalhada de cada um:

Flag	Significado
`ASCII`, `A`	Makes several escapes like `\w`, `\b`, `\s` and `\d` match only on ASCII characters with the respective property.
`DOTALL`, `S`	Make `.` match any character, including newlines.
`IGNORECASE`, `I`	Faz combinações sem diferenciar maiúsculo de minúsculo
`LOCALE`, `L`	Faz uma correspondência considerando a localidade.
`MULTILINE`, `M`	Correspondência multilinha, afetando `^` e `$`.
`VERBOSE`, `X` (for 'extended')	Habilita REs detalhadas, que podem ser organizadas de forma mais clara e compreensível.

I
IGNORECASE

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too. Full Unicode matching also works unless the `ASCII` flag is used to disable non-ASCII matches. When the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: 'İ' (U+0130, Latin capital letter I with dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). `Spam` will match `'Spam'`, `'spam'`, `'spAM'`, or `'ſpam'` (the latter is matched only in Unicode mode). This lowercasing doesn't take the current locale into account; it will if you also set the `LOCALE` flag.

L
LOCALE

Make `\w`, `\W`, `\b`, `\B` and case-insensitive matching dependent on the current locale instead of the Unicode database.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing encoded French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]` in bytes patterns; it won't match bytes

corresponding to é or ç. If your system is configured properly and a French locale is selected, certain C functions will tell the program that the byte corresponding to é should also be considered a letter. Setting the LOCALE flag when compiling a regular expression will cause the resulting compiled object to use these C functions for \w; this is slower, but also enables \w+ to match French words as you'd expect. The use of this flag is discouraged in Python 3 as the locale mechanism is very unreliable, it only handles one "culture" at a time, and it only works with 8-bit locales. Unicode matching is already enabled by default in Python 3 for Unicode (str) patterns, and it is able to handle different locales/languages.

M
MULTILINE

(^ e $ ainda não foram explicados, eles serão comentados na seção *Mais Metacaracteres*.)

Normalmente ^ corresponde apenas ao início da string e $ corresponde apenas ao final da string, e imediatamente antes da nova linha (se existir) no final da string. Quando este sinalizador é especificada, o ^ corresponde ao início da string e ao início de cada linha dentro da string, imediatamente após cada nova linha. Da mesma forma, o metacaractere $ corresponde tanto ao final da string e ao final de cada linha (imediatamente antes de cada nova linha).

S
DOTALL

Faz o caractere especial . corresponder com qualquer caractere que seja, incluindo o nova linha; sem este sinalizador, . irá corresponder a qualquer coisa, exceto o nova linha.

A
ASCII

Make \w, \W, \b, \B, \s and \S perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode patterns, and is ignored for byte patterns.

X
VERBOSE

Este sinalizador permite escrever expressões regulares mais legíveis, permitindo mais flexibilidade na maneira de formatá-la. Quando este sinalizador é especificado, o espaço em branco dentro da string RE é ignorado, exceto quando o espaço em branco está em uma classe de caracteres ou precedido por uma barra invertida não "escapada"; isto permite organizar e formatar a RE de maneira mais clara. Este sinalizador também permite que se coloque comentários dentro de uma RE que serão ignorados pelo mecanismo; os comentários são marcados por um "#" que não está nem em uma classe de caracteres nem precedido por uma barra invertida não "escapada". Por exemplo, aqui está uma RE que usa re.VERBOSE; veja, o quanto mais fácil de ler é ?

Por exemplo, aqui está uma RE que usa re.VERBOSE; veja, o quanto mais fácil de ler é?

```
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 o "verbose" definido, A RE iria se parecer 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 foi usada para quebrar a RE em partes menores, mas ainda é mais difícil de entender do que a versão que usa re.VERBOSE.

4 Mais Poder dos Padrões

Até agora, cobrimos apenas uma parte dos recursos das expressões regulares. Nesta seção, vamos abordar alguns metacaracteres novos, e como usar grupos para recuperar partes do texto que teve correspondência.

4.1 Mais Metacaracteres

Existem alguns metacaracteres que nós ainda não vimos. A maioria deles serão referenciados nesta seção.

Alguns dos metacaracteres restantes a serem discutidos são como uma afirmação de `largura zero` (zero-width assertions). Eles não fazem com que o mecanismo avance pela string; ao contrário, eles não consomem nenhum caractere, e simplesmente tem sucesso ou falha. Por exemplo, `\b` é uma afirmação de que a posição atual está localizada nas bordas de uma palavra; a posição não é alterada de nenhuma maneira por `\b`. Isto significa que afirmações de `largura zero` nunca devem ser repetidas, porque se elas combinam uma vez em um determinado local, elas podem, obviamente, combinar um número infinito de vezes.

`|` Alternation, or the "or" operator. If *A* and *B* are regular expressions, `A|B` will match any string that matches either *A* or *B*. `|` has very low precedence in order to make it work reasonably when you're alternating multi-character strings. `Crow|Servo` will match either `'Crow'` or `'Servo'`, not `'Cro'`, a `'w'` or an `'S'`, and `'ervo'`.

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

`^` Corresponde ao início de linha. A menos que o sinalizador `MULTILINE` tenha sido definido, isso só irá corresponder ao início da string. No modo `MULTILINE`, isso também corresponde imediatamente após cada nova linha de dentro da string.

Por exemplo, para ter correspondência com a palavra `From` apenas no início de uma linha, aRE a ser usada é `^From`.

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

To match a literal `'^'`, use `\^`.

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

```
>>> print(re.search('}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('}$', '{block} '))
None
>>> print(re.search('}$', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

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

`\A` Corresponde apenas com o início da string. Quando não estiver em modo `MULTILINE`, `\A` e `^` são efetivamente a mesma coisa. No modo `MULTILINE`, eles são diferentes: `\A` continua a corresponder apenas com o início da string, mas `^` pode corresponder com qualquer localização de dentro da string, que seja posterior a um caractere nova linha.

`\Z` Corresponde apenas ao final da string.

`\b` Borda de palavra. Esta é uma afirmação de `largura zero` que corresponde apenas ao início ou ao 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 caractere não alfanumérico.

O exemplo a seguir corresponde a `class` apenas quando é a palavra exata; 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'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

Há duas sutilezas você deve lembrar ao usar essa sequência especial. Em primeiro lugar, esta é a pior colisão entre strings literais do Python e sequências de expressão regular. Nas strings literais do Python, `\b` é o caractere backspace, o valor ASCII 8. Se você não estiver usando strings cruas (raw), então Python irá converter o `\b` em um backspace e sua RE não irá funcionar da maneira que você espera. O exemplo a seguir parece igual a 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'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

Além disso, 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`, correspondendo apenas quando a posição corrente não é de uma borda de palavra.

4.2 Agrupamento

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a `':'`, like this:

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

Isto pode ser gerenciado ao escrever uma expressão regular que corresponde 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 `)`. `(` e `)` têm muito do mesmo significado que eles têm em expressões matemáticas; eles 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`.

Grupos indicados com `(` e `)` também capturam o índice inicial e final do texto que eles correspondem; isso pode ser obtido por meio da passagem de um argumento para `group()`, `start()`, `end()`, e `span()`. Os grupos são numerados começando com 0. O grupo 0 está sempre presente; é toda a RE, logo, todos os métodos MatchObject têm o grupo 0 como seu argumento padrão. Mais tarde veremos como expressar grupos que não capturam a extensão de texto com a qual eles correspondem.

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

Groups indicated with `'('`, `')'` also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

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

Subgrupos são numerados a partir da esquerda para a direita, de forma crescente a partir de 1. Os grupos podem ser aninhados; para determinar o número, basta contar os caracteres de abertura de 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()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

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

The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

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

Referências anteriores em um padrão permitem que você especifique que o conteúdo de um grupo capturado anteriormente também deve ser encontrado na posição atual na sequência. Por exemplo, `\1` terá sucesso se o conteúdo exato do grupo 1 puder ser encontrado na posição atual, e falhar caso contrário. Lembre-se que as strings literais do Python também usam a barra invertida seguida por números para permitir a inclusão de caracteres arbitrários em uma string, por isso certifique-se de usar strings cruas (raw) 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\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Referências anteriores como esta não são, geralmente, muito úteis apenas para fazer pesquisa percorrendo uma string — existem alguns formatos de texto que repetem dados dessa forma — mas em breve você irá descobrir que elas são muito úteis para realizar substituições de strings.

4.3 Não captura e Grupos Nomeados

REs elaboradas podem usar muitos grupos, tanto para capturar substrings de interesse, quanto para agrupar e estruturar a própria RE. Em REs complexas, torna-se difícil manter o controle dos números dos grupos. Existem dois recursos que ajudam a lidar com esse problema. Ambos usam uma sintaxe comum para extensões de expressão regular, então vamos olhar para isso em primeiro lugar.

Perl 5 is well known for its powerful additions to standard regular expressions. For these new features the Perl developers couldn't choose new single-keystroke metacharacters or new special sequences beginning with `\` without making Perl's regular expressions confusingly different from standard REs. If they chose `&` as a new metacharacter, for example, old expressions would be assuming that `&` was a regular character and wouldn't have escaped it by writing `\&` or `[&]`.

A solução escolhida pelos desenvolvedores do Perl foi usar `(?...)` como uma sintaxe de extensão. Um `?` imediatamente após um parêntese era um erro de sintaxe porque o `?` não teria nada a repetir, de modo que isso não introduz quaisquer problemas de compatibilidade. Os caracteres imediatamente após um `?` indicam que a extensão está sendo usada, então `(?=foo)` é uma coisa (uma afirmação `lookahead` positiva) e `(?:foo)` é outra coisa (um grupo de não captura contendo a subexpressão `foo`).

Python supports several of Perl's extensions and adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a `P`, you know that it's an extension that's specific to Python.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs.

Sometimes you'll want to use a group to denote a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group: `(?:...)`, where you can replace the `...` with any other regular expression.

```
>>> 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 sobre o qual o grupo corresponde, um grupo de não captura se comporta exatamente da mesma forma que um grupo de captura; você pode colocar qualquer coisa dentro dele, repeti-lo com um metacaractere de repetição, como o '*', e aninhá-lo dentro de outros grupos (de captura ou não captura). `(?:...)` é particularmente útil para modificar um padrão existente, já que você pode adicionar novos grupos sem alterar a forma como todos os outros grupos estão numerados. Deve ser mencionado que não há diferença de desempenho na busca entre grupos de captura e grupos de não captura; uma forma não é mais rápida que outra.

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

The syntax for a named group is one of the Python-specific extensions: `(?P<name>...)`. *name* is, obviously, the name of the group. Named groups behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

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

Additionally, you can retrieve named groups as a dictionary with `groupdict()`:

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

Os grupos nomeados são úteis porque eles permitem que você use nomes de fácil lembrança, em vez de ter que lembrar de números. Aqui está um exemplo de RE usando o 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 obviamente muito mais fácil fazer referência a `m.group('zonem')`, do que ter que se lembrar de capturar o grupo 9.

The syntax for backreferences in an expression such as `(...)\1` refers to the number of the group. There's naturally a variant that uses the group name instead of the number. This is another Python extension: `(?P=name)` indicates that the contents of the group called *name* should again be matched at the current point. The regular expression for finding doubled words, `\b(\w+)\s+\1\b` can also be written as `\b(?P<word>\w+)\s+(?P=word)\b`:

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

4.4 Afirmação Lookahead

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

`(?=...)` Afirmação lookahead positiva. Retorna sucesso se a expressão regular informada, aqui representada por `...`, corresponde com o conteúdo da localização atual, e retorna falha caso contrário. Mas, uma vez que a expressão informada tenha sido testada, o mecanismo de correspondência não faz qualquer avanço; o resto do padrão é tentado no mesmo local de onde a afirmação foi iniciada.

`(?!...)` Afirmação lookahead negativa. É o oposto da afirmação positiva; será bem-sucedida se a expressão informada não corresponder com o conteúdo da posição atual na string.

Para tornar isto concreto, vamos olhar para um caso em que um lookahead é útil. Considere um padrão simples para corresponder com um nome de arquivo e divida-o em pedaços, um nome 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 com isso é muito simples:

`.*[.].*$`

Notice that the `.` needs to be treated specially because it's a metacharacter, so it's inside a character class to only match that specific character. Also notice the trailing `$`; this is added to ensure that all the rest of the string must be included in the extension. This regular expression matches `foo.bar` and `autoexec.bat` and `sendmail.cf` and `printers.conf`.

Agora, considere complicar um pouco o problema; e se você desejar corresponder com 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 a `foo.bar`.

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

A expressão fica mais confusa se você tentar remendar a primeira solução, exigindo que uma das seguintes situações corresponda: o primeiro caractere da extensão não é `b`; o segundo caractere não é `a`; ou o terceiro caractere não é `t`. Isso 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, tal como `sendmail.cf`. Nós iremos complicar o padrão novamente em um esforço para corrigi-lo.

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

Na terceira tentativa, a segunda e terceira letras são todas consideradas opcionais, a fim de permitir correspondência com as extensões mais curtas do que três caracteres, tais como `sendmail.cf`.

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

Um lookahead negativo elimina toda esta confusão:

`.*[.](?!bat$).*$` O lookahead negativo significa: se a expressão `bat` não corresponder até este momento, tente o resto do padrão; se `bat$` tem correspondência, todo o padrão irá falhar. O final `$` é necessário para garantir que algo como `sample.batch`, onde a extensão só começa com o `bat`, será permitido.

Excluir uma outra extensão de nome de arquivo agora é fácil; basta fazer a adição de uma alternativa dentro da afirmação. O padrão a seguir exclui os nomes de arquivos que terminam com `bat` ou `exe`:

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

5 Modificando Strings

Até este ponto, nós simplesmente realizamos pesquisas em uma string estática. As expressões regulares também são comumente usadas para modificar strings através de várias maneiras, usando os seguintes métodos padrão:

Método/Atributo	Purpose
`split()`	Divide a string em uma lista, dividindo-a onde quer que haja correspondência com a RE
`sub()`	Encontra todas as substrings que correspondem com a RE e faz a substituição por uma string diferente
`subn()`	Does the same thing as `sub()`, but returns the new string and the number of replacements

5.1 Dividindo as Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; string `split()` only supports splitting by whitespace or by a fixed string. As you'd expect, there's a module-level `re.split()` function, too.

`.split`(*string*[, *maxsplit=0*])
: Divide a string usando a correspondência com uma expressão regular. Se os parênteses de captura forem utilizados na RE, então seu conteúdo também será retornado como parte da lista resultante. Se maxsplit é diferente de zero, um número de divisões `maxsplit` será executado.

Você pode limitar o número de divisões feitas, passando um valor para maxsplit. Quando `maxsplit` é diferente de zero, um determinado número de divisões `maxsplit` será executado, 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 contém, mas também precisa saber qual o delimitador foi usado. Se os parênteses de captura são utilizados na RE, então 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 de nível de módulo `re.split()` adiciona a RE a ser utilizada como o primeiro argumento, mas é, em determinadas circunstâncias, a mesma.

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

5.2 Busca e Substituição

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.

.sub (*replacement*, *string*[, *count=0*])

Retorna a string obtida substituindo as ocorrências mais à esquerda não sobrepostas da RE em `string` pela substituição `replacement`. Se o padrão não for encontrado, a `string` é retornada inalterada.

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

Here's a simple example of using the `sub()` method. It replaces colour names with the word `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'
```

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> 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)
```

Empty matches are replaced only when they're not adjacent to a previous empty match.

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

If *replacement* is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

Este exemplo corresponde com a 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 sintaxe `(?P<name>...)`. `\g<name>` usará a substring correspondida pelo grupo com nome `name` e `\g<number>` utiliza o número do grupo correspondente. `.\g<2>` é, portanto, equivalente a `\2`, mas não é ambígua em uma string de substituição (replacement), tal como `\g<2>0`. (`\20` seria interpretado como uma referência ao grupo de `20`, e não uma referência ao grupo `2` seguido pelo caractere literal `0`). As substituições a seguir são todas equivalentes, mas usam todas as três variações da string 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}'
```

replacement também pode ser uma função, que lhe dá ainda mais controle. Se *replacement* for uma função, a função será chamada para todas as ocorrências não sobrepostas de *pattern*. Em cada chamada, a função recebe um argumento de objeto Match para a correspondência e pode usar essas informações para calcular a string de substituição desejada e retorná-la.

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 fornecido como um objeto ou como uma string; se você precisa especificar sinalizadores de expressões regulares, você deve usar um objeto padrão como o primeiro parâmetro, ou usar modificadores embutidos na string padrão, por exemplo, `sub("(?i)b+", "x", "bbbb BBBB")` retorna `'x x'`.

6 Problemas Comuns

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

6.1 Usando String Methods

Sometimes using the `re` module is a mistake. If you're matching a fixed string, or a single character class, and you're not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they're usually much faster, because the implementation is a single small C loop that's been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`'s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

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.

6.2 match() versus search()

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn't start at zero, `match()` will *not* report it.

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

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

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

Às vezes, você vai ficar tentado a continuar usando `re.match()`, e apenas adicionar `.*` ao início 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 correspondência. Tal análise descobre o que o primeiro caractere de uma string deve ser; por exemplo, um padrão começando com `Crow` deve corresponder com algo iniciando com `'C'`. A análise permite que o mecanismo faça a varredura rapidamente através da string a procura do caractere inicial, apenas tentando a combinação completa se um `'C'` for encontrado.

Adicionar um `.*` evita essa otimização, sendo necessário a varredura até o final da string e, em seguida, retroceder para encontrar uma correspondência para o resto da RE. Use `re.search()` em vez disso.

6.3 Gulosos versus não Gulosos

Ao repetir uma expressão regular, como em `a*`, a ação resultante é consumir o tanto do padrão quanto possível. Este fato, muitas vezes derruba você quando você está tentando corresponder com um par de delimitadores balanceados, tal 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>
```

The RE matches the `'<'` in `'<html>'`, and the `.*` consumes the rest of the string. There's still more left in the RE, though, and the > can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the `'<'` in `'<html>'` to the `'>'` in `'</title>'`, which isn't what you want.

Neste caso, a solução é usar os qualificadores não-gulosos `*?`, `+?`, `??`, or `{m,n}?`, que corresponde com o mínimo de texto possível. No exemplo acima, o > é tentado imediatamente após a primeira correspondência de <, e quando ele falhar, o mecanismo avança um caractere de cada vez, experimentado > a cada passo. Isso produz justamente o resultado correto:

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

(Note que a análise de HTML ou XML com expressões regulares é dolorosa. Padrões "sujos e rápidos" irão lidar com casos comuns, mas HTML e XML tem casos especiais que irão quebrar expressões regulares óbvias; com o tempo, expressões regulares que você venha a escrever para lidar com todos os casos possíveis, se tornarão um padrão muito complicado. Use um módulo de análise de HTML ou XML para tais tarefas.)

6.4 Usando re.VERBOSE

Nesse momento, você provavelmente deve ter notado que as expressões regulares são de uma notação muito compacta, mas não é possível dizer que são legíveis. REs de complexidade moderada podem se tornar longas coleções de barras invertidas, parênteses e metacaracteres, fazendo com que se tornem difíceis de ler e compreender.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

O sinalizador `re.VERBOSE` produz 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 uma RE; comentários se estendem de um caractere `#` até a próxima nova linha. Quando usados junto com strings de aspas triplas, isso permite as REs serem formatadas 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)
```

Isso é muito mais legível do que:

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

7 Comentários

Expressões regulares são um tópico complicado. Esse documento ajudou você a compreendê-las? Existem partes que foram pouco claras, ou situações que você vivenciou que não foram abordadas aqui? Se assim for, por favor, envie sugestões de melhorias para o autor.

The most complete book on regular expressions is almost certainly Jeffrey Friedl's Mastering Regular Expressions, published by O'Reilly. Unfortunately, it exclusively concentrates on Perl and Java's flavours of regular expressions, and doesn't contain any Python material at all, so it won't be useful as a reference for programming in Python. (The first edition covered Python's now-removed `regex` module, which won't help you much.) Consider checking it out from your library.