
Expressões Regulares

Release 3.11.13

Guido van Rossum and the Python development team

julho 07, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Introdução	2
2	Padrões simples	2
2.1	Correspondendo caracteres	2
2.2	Repetindo coisas	4
3	Usando expressões regulares	5
3.1	Compilando expressões regulares	5
3.2	A praga da contrabarra	5
3.3	Executando correspondências	6
3.4	Funções de nível de módulo	8
3.5	Sinalizadores de compilação	8
4	Mais poder dos padrões	10
4.1	Mais metacaracteres	10
4.2	Agrupamento	12
4.3	Não captura e grupos nomeados	13
4.4	Asserções lookahead	14
5	Modificando strings	15
5.1	Dividindo as strings	16
5.2	Busca e Substituição	16
6	Problemas comuns	18
6.1	Usando métodos de string	18
6.2	match() versus search()	18
6.3	Gulosos versus não-gulosos	19
6.4	Usando re.VERBOSE	19
7	Comentários	20

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 corresponder; esse conjunto pode conter sentenças em português, endereços de e-mail, 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 correspondência 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 expressão regular é relativamente pequena e restrita, por isso nem todas as tarefas de processamento de strings possíveis podem ser feitas usando expressões regulares. Existem também tarefas que podem ser feitas com expressões regulares, mas as expressões acabam por se tornarem muito complicadas. Nestes casos, pode ser melhor para você escrever um código Python para fazer o processamento; embora um código Python seja mais lento do que uma expressão regular elaborada, ele provavelmente será mais compreensível.

2 Padrões simples

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

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

2.1 Correspondendo caracteres

A maioria das letras e caracteres simplesmente irão corresponder entre si. Por exemplo, a expressão regular `teste` irá combinar totalmente com a string `teste`. (Você pode habilitar o modo de maiúsculas e minúsculas que faria a RE corresponder com `Test` ou `TEST` também; veremos mais 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, repetindo-as ou alterando seus significados. Grande parte deste documento é dedicada à discussão de 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 estudar 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 intervalo de caracteres pode ser indicado fornecendo 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 intervalo para expressar o mesmo conjunto de caracteres. Se você deseja encontrar apenas letras minúsculas, sua RE seria `[a-z]`.

Metacaracteres (exceto `\`) não são ativos dentro de classes. Por exemplo, `[akm$]` irá corresponder com qualquer dos caracteres `'a'`, `'k'`, `'m'` ou `'$'`; `'$'` 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 seja 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 `[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 incluídas dentro de uma classe de caractere. Por exemplo, `[\s,.]` É uma classe de 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 caracteres `'a'`), `'cat'` (1 `'a'`), `'caaat'` (3 caracteres `'a'`), e assim por diante.

Repetições tais como `*` são gulosas; 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 menos 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	Correspon- dência	Explicação
1	a	O caractere a na RE tem correspondência.
2	abcbd	O motor corresponde com <code>[bcd]*</code> , indo tão longe quanto possível, que é o fim do string.
3	<i>Falha</i>	O motor tenta corresponder com b, mas a posição corrente está no final da string, então ele falha.
4	abcb	Voltando, de modo que <code>[bcd]*</code> corresponde a um caractere a menos.
5	<i>Falha</i>	Tenta b novamente, mas a posição corrente é a do último caractere, que é um <code>'d'</code> .
6	abc	Voltando novamente, de modo que <code>[bcd]*</code> está correspondendo com <code>bc</code> somente.
6	abcb	Tenta b novamente. Desta vez, o caractere na posição corrente é <code>'b'</code> , 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 a volta progressivamente e tentará novamente o restante da RE. Ele fará voltas 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.

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

Existem mais dois quantificadores ou operadores de repetição. O caractere de ponto de interrogação, `?`, corresponde a uma ou zero vez; você pode pensar nisso como a marcação de algo sendo opcional. Por exemplo, `home-?brew` corresponde tanto a `'homebrew'` quanto a `'home-brew'`.

O qualificador mais complicado é o `{m, n}`, no qual `m` e `n` são números inteiros decimais. Este qualificador 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 corresponder a `'ab'`, que não tem barras ou a `'a////b'`, que tem quatro.

Você pode omitir tanto `m` quanto `n`; nesse caso, um valor razoável é presumido para o valor em falta. A omissão de `m` é interpretada como o limite inferior de 0, enquanto a omissão de `n` resulta como limite superior o infinito.

O caso mais simples `{m}` corresponde ao item precedente exatamente `m` vezes. Por exemplo, `a/{2}b` corresponderá somente a `'a//b'`.

Os leitores de uma inclinação reducionista podem notar que os três outros quantificadores 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*, utilizado 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 contrabarra

Como afirmado anteriormente, expressões regulares usam o caractere de contrabarra (`\`) 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 contrabarra e outros metacaracteres com uma contrabarra, 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 contrabarras devem ser precedidas com uma contrabarra novamente.

Caracteres	Etapa
<code>\section</code>	String de texto a ser correspondida
<code>\\section</code>	Preceder com contrabarra para <code>re.compile()</code>
<code>\\\\section</code>	Contrabarras precedidas novamente para uma string literal

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

A solução é usar a notação de string bruta (raw) do Python para expressões regulares; contrabarras 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 bruta (raw).

Além disso, sequências de escape especiais que são válidas em expressões regulares, mas não válidas como literais de string do Python, agora resultam em uma `DeprecationWarning` e eventualmente se tornarão uma `SyntaxError`, o que significa que as sequências serão inválidas se a notação de string bruta ou o escape das contrabarras não forem usados.

String regular	String bruta
<code>"ab*"</code>	<code>r"ab*"</code>
<code>"\\\\section"</code>	<code>r"\\section"</code>
<code>"\\w+\\s+\\1"</code>	<code>r"\\w+\\s+\\1"</code>

3.3 Executando correspondências

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

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

`match()` e `search()` retornam `None` se não existir nenhuma correspondência encontrada. Se tiveram sucesso, uma instância de objeto correspondência é retornada, contendo informações sobre a correspondência: onde ela começa e termina, a substring com a qual ela teve correspondência, e mais.

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 documento usa o interpretador Python padrão para seus exemplos. Primeiro, execute o interpretador Python, importe o módulo `re`, e compile uma RE:

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

Agora, você pode tentar corresponder várias strings com a RE `[a-z]+`. Mas uma string vazia não deveria corresponder com nada, uma vez que `+` significa '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 de `match()` para deixar isso claro.

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

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

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

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

Método/Atributo	Propósito
<code>group()</code>	Retorna a string que corresponde com a RE
<code>start()</code>	Retorna a posição inicial da string correspondente
<code>end()</code>	Retorna a posição final da string correspondente
<code>span()</code>	Retorna uma tupla contendo as posições (inicial, final) da string correspondente

Experimentando estes métodos teremos seus significado esclarecidos:

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

`group()` retorna a substring que correspondeu com a RE. `start()` e `end()` retornam os índices inicial e o final da substring correspondente. `span()` retorna tanto os índices inicial e final em uma única tupla. Como o método `match()` somente verifica se a RE corresponde ao início de uma string, `start()` será sempre zero. No entanto, o método `search()` dos objetos padrão, varre toda a string, de modo que a substring correspondente pode não iniciar em zero nesse caso.

```
>>> 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 correspondência 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')
```

Dois métodos padrão retornam todas as correspondências de um padrão. `findall()` retorna uma lista de strings correspondentes:

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

O prefixo `r`, tornando literal uma literal de string bruta, é necessário neste exemplo porque sequências de escape em uma literal de string “cozida” normal que não são reconhecidas pelo Python, ao contrário de expressões regulares, agora resultam em uma `DeprecationWarning` e eventualmente se tornarão uma `SyntaxError`. Veja [A praga da contrabarra](#).

`findall()` tem que criar a lista inteira antes de poder devolvê-la como resultado. O método `finditer()` retorna uma sequência de instâncias objeto correspondência como um iterator:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
```

(continua na próxima página)

```
<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

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

```
>>> 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 ' >
```

Sob o capô, estas funções simplesmente criam um objeto padrão para você e chamam o método apropriado para ele. Elas também armazenam o objeto compilado em um cache, para que futuras chamadas usando a mesma RE não precisem analisar o padrão de novo e de novo.

Você deve usar essas funções de nível de módulo ou deve obter o padrão e chamar seus métodos você mesmo? Se estiver acessando uma expressão regular dentro de um laço de repetição, pré-compilá-la economizará algumas chamadas de função. Fora dos laços, não há muita diferença graças ao cache interno.

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:

Sinalizador	Significado
ASCII, A	Faz com que vários escapes como <code>\w</code> , <code>\b</code> , <code>\s</code> e <code>\d</code> correspondam apenas a caracteres ASCII com a respectiva propriedade.
DOTALL, S	Faz o <code>.</code> corresponder a qualquer caractere, incluindo novas linhas.
IGNORECASE, I	Faz combinações sem diferenciar maiúsculo de minúsculo.
LOCALE, L	Faz uma correspondência considerando a localidade.
MULTILINE, M	Correspondência multilinha, afetando <code>^</code> e <code>\$</code> .
VERBOSE, X (de 'extended'; estendido em inglês)	Habilita REs detalhadas, que podem ser organizadas de forma mais clara e compreensível.

`re.I`

`re.IGNORECASE`

Faz correspondência sem distinção entre maiúsculas e minúsculas; a classe de caracteres e as strings literais corresponderão às letras ignorando maiúsculas e minúsculas. Por exemplo, `[A-Z]` corresponderá às letras minúsculas também. A correspondência Unicode completa também funciona, a menos que o sinalizador `ASCII` seja usado para desabilitar correspondências não ASCII. Quando os padrões Unicode `[a-z]` ou `[A-Z]` são usados em combinação com o sinalizador `IGNORECASE`, eles corresponderão às 52 letras ASCII e 4 letras não

ASCII adicionais: 'İ' (U+0130, letra maiúscula latina I com ponto acima), 'ı' (U+0131, letra minúscula latina i sem ponto), 'ſ' (U+017F, letra minúscula latina s longo) e 'K' (U+212A, sinal Kelvin). `Spam` corresponderá a `'Spam'`, `'spam'`, `'spAM'` ou `'ſpam'` (o último é correspondido apenas no modo Unicode). Essa capitalização em minúsculas não leva em conta a localidade atual; levará se você também definir o sinalizador `LOCALE`.

`re.L`

`re.LOCALE`

Faz com que `\w`, `\W`, `\b`, `\B` e a correspondência sem diferenciação de maiúsculas e minúsculas dependam da localidade atual em vez do banco de dados Unicode.

Localidades são um recurso da biblioteca C destinado a ajudar na escrita de programas que levam em conta as diferenças de idioma. Por exemplo, se você estiver processando texto codificado em francês, você gostaria de ser capaz de escrever `\w+` para corresponder a palavras, mas `\w` corresponde apenas à classe de caracteres `[A-Za-z]` em padrões de bytes; ele não corresponderá a bytes correspondentes a `é` ou `ç`. Se seu sistema estiver configurado corretamente e uma localidade francesa for selecionada, certas funções C dirão ao programa que o byte correspondente a `é` também deve ser considerado uma letra. Definir o sinalizador `LOCALE` ao compilar uma expressão regular fará com que o objeto compilado resultante use essas funções C para `\w`; isso é mais lento, mas também permite que `\w+` corresponda a palavras francesas como você esperaria. O uso deste sinalizador é desencorajado no Python 3, pois o mecanismo de localidade é muito pouco confiável, ele só manipula uma “cultura” por vez e só funciona com localidades de 8 bits. A correspondência Unicode já está habilitada por padrão no Python 3 para padrões Unicode (str), e é capaz de manipular diferentes localidades/idiomas.

`re.M`

`re.MULTILINE`

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

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

`re.S`

`re.DOTALL`

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

`re.A`

`re.ASCII`

Faz com que `\w`, `\W`, `\b`, `\B`, `\s` e `\S` executem a correspondência somente ASCII em vez da correspondência Unicode completa. Isso é significativo apenas para padrões Unicode e é ignorado para padrões de bytes.

`re.X`

`re.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`; consegue ver o quanto mais fácil de ler é?

```
charref = re.compile(r"""
&[#]           # Start of a numeric entity reference
(
    0[0-7]+     # Octal form
```

(continua na próxima página)

```

| [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 asserção 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.

|

Alternância, ou operador “or”. Se *A* e *B* são expressões regulares, *A|B* irá corresponder com qualquer string que corresponder com *A* ou *B*. `|` tem uma prioridade muito baixa, a fim de fazê-lo funcionar razoavelmente quando você está alternando entre strings de vários caracteres. `Crow|Servo` irá corresponder tanto com `'Crow'` quanto com `'Servo'`, e não com `'Cro'`, `'w'` ou `'S'`, e `'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, a RE 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

```

Para corresponder a um `'^'` 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 asserção de largura zero que corresponde apenas ao início ou ao fim de uma palavra. Uma palavra é definida como uma sequência de caracteres alfanuméricos, então o fim de uma palavra é indicado por espaço 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 que 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 brutas, 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 asserção, \b representa o caractere backspace, para compatibilidade com strings literais do Python

\B

Outra asserção 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

Frequentemente é necessário obter mais informações do que apenas se a RE teve correspondência ou não. As expressões regulares são muitas vezes utilizadas para dissecar strings escrevendo uma RE dividida em vários subgrupos que correspondem a diferentes componentes de interesse. Por exemplo, uma linha de cabeçalho RFC-822 é dividida em um nome de cabeçalho e um valor, separados por um ' : ', como essa:

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

Isto pode ser tratado escrevendo 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.

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

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, de forma que os métodos de objeto de correspondência têm todos 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('(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() pode receber vários números de grupos de uma vez, e nesse caso ele irá retornar uma tupla contendo os valores correspondentes desses 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 o último.

```
>>> 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 falha caso contrário. Lembre-se que as strings literais do Python também usam a contrabarra seguida por números para permitir a inclusão de caracteres arbitrários em uma string, por isso certifique-se de usar strings brutas 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 em 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 primeiro para isso.

O Perl 5 é bem conhecido por suas poderosas adições às expressões regulares padrão. Para esses novos recursos, os desenvolvedores do Perl não podiam escolher novos metacaracteres de um único caractere ou novas sequências especiais começando com `\` sem tornar as expressões regulares do Perl confusamente diferentes das REs padrão. Se eles escolhessem `&` como um novo metacaractere, por exemplo, as expressões antigas estariam presumindo que `&` era um caractere regular e não teriam escapado dele escrevendo `\&` ou `[&]`.

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 asserção *lookahead* positiva) e `(?:foo)` é outra coisa (um grupo de não-captura contendo a subexpressão `foo`).

Python oferece suporte a diversas extensões do Perl e adiciona uma sintaxe de extensão à sintaxe de extensão do Perl. Se o primeiro caractere após o ponto de interrogação for um `P`, você sabe que se trata de uma extensão específica do Python.

Agora que vimos a sintaxe geral da extensão, podemos retornar aos recursos que simplificam o trabalho com grupos em REs complexas.

Às vezes você vai querer usar um grupo para representar uma parte de uma expressão regular, mas não está interessado em recuperar o conteúdo do grupo. Você pode tornar isso explícito usando um grupo de não-captura: `(?:...)`, onde você pode substituir o `...` por 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 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.

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 se comportam exatamente como os grupos de captura, e, adicionalmente, associam um nome a um grupo. Todos os métodos de objeto correspondência que lidam com grupos de captura aceitam tanto inteiros que se referem ao grupo por número ou strings que contêm o nome do grupo desejado. Os grupos nomeados ainda recebem números, então você pode 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'
```

Além disso, você pode recuperar grupos nomeados como um dicionário com `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.

A sintaxe para referências anteriores em uma expressão, tal como `(...)\1`, faz referência ao número do grupo. Existe, 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'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

4.4 Asserções lookahead

Outra asserção de “largura zero” é a asserção lookahead. Asserções lookahead estão disponíveis tanto na forma positiva quanto na negativa, e se parece com isto:

`(?=...)`

Asserção lookahead positiva. É bem-sucedida se a expressão regular informada, aqui representada por `..`, corresponde com o conteúdo da localização atual, e 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 asserção foi iniciada.

`(?!...)`

Asserção lookahead negativa. É o oposto da asserção 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 uma lookahead é útil. Considere um padrão simples para corresponder com um nome de arquivo e dividi-lo 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:

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

Observe que o `.` precisa ser tratado de forma especial, pois é um metacaractere e, portanto, está dentro de uma classe de caracteres para corresponder apenas a esse caractere específico. Observe também o `$` ao final; ele é adicionado para garantir que todo o restante da string seja incluído na extensão. Esta expressão regular corresponde a `foo.bar`, `autoexec.bat`, `sendmail.cf` e `printers.conf`.

Agora, considere complicar um pouco o problema; e se você deseja 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 `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`.

Agora, o padrão está ficando realmente muito complicado, 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.

Uma 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 asserção. 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	Propósito
<code>split()</code>	Divide a string em uma lista, dividindo-a onde quer que haja correspondência com a RE
<code>sub()</code>	Encontra todas as substrings que correspondem com a RE e faz a substituição por uma string diferente
<code>subn()</code>	Faz a mesma coisa que <code>sub()</code> , mas retorna a nova string e o número de substituições

5.1 Dividindo as strings

O método `split()` de um padrão divide uma string em pedaços onde quer que a RE corresponda, retornando uma lista formada por esses pedaços. É semelhante ao método `split()` de strings, mas oferece muito mais generalidade nos delimitadores que pode usar para fazer a divisão; `split()` só implementa a divisão por espaço em branco ou por uma string fixa. Como você deve deduzir, existe também uma função a nível de módulo `re.split()`.

.split (*string* [, *maxsplit*=0])

Divide *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, serão feitas no máximo *maxsplit* divisões.

Você pode limitar o número de divisões feitas, passando um valor para *maxsplit*. Quando *maxsplit* é diferente de zero, serão feitas no máximo *maxsplit* divisões, 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 interessado apenas no texto que está entre os delimitadores, 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, fora isso, é 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

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 da RE em *string* pelo valor de 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 substituir todas as ocorrências.

Aqui está um exemplo simples do uso do método `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'
```

(continua na próxima página)


```
>>> 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 com duas informações: uma string com novo valor 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 (próxima) a uma correspondência vazia anterior.

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

Se o valor de substituição (*replacement*) é uma string, qualquer barra invertida é interpretada e processada. Isto é, `\n` é convertido a um único caractere de nova linha, `\r` é convertido em um retorno do carro, e assim por diante. Casos desconhecidos, como `\&` são ignorados. Referências anteriores, como `\6`, são substituídas com a substring correspondida pelo grupo correspondente na RE. Isso permite que você incorpore partes do texto original na string de substituição resultante.

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, 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 métodos de string

Às vezes, usar o módulo `re` é um equívoco. Se você está fazendo correspondência com uma string fixa, ou uma classe de caractere única, e você não está usando nenhum recurso de `re` como o sinalizador `IGNORECASE`, então pode não ser necessário todo o poder das expressões regulares. Strings possuem vários métodos para executar operações com strings fixas e eles são, geralmente, muito mais rápidos, porque a implementação é um único e pequeno laço de repetição em C que foi otimizado para esse propósito, em vez do grande e mais generalizado mecanismo das expressões regulares.

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 uma RE ingênua 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 delimitador de palavra em ambos os lados. Isso leva a tarefa para além da capacidade de `replace()`.)

Outra tarefa comum é apagar todas as ocorrências de um único caractere de uma string ou substituí-lo por um outro caractere único. Você pode fazer isso com algo como `re.sub('\n', ' ', S)`, mas `translate()` é capaz de fazer ambas as tarefas e será mais rápida 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 de string mais rápido e mais simples.

6.2 match() versus search()

A função `match()` somente verifica se a RE corresponde ao início da string, enquanto `search()` fará a varredura percorrendo a string procurando por uma correspondência. É importante manter esta distinção em mente. Lembre-se, `match()` só irá relatar uma correspondência bem-sucedida que começa em 0; se a correspondência não começar em zero, `match()` não vai reportá-la.

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

Por outro lado, `search()` fará a varredura percorrendo a string e relatando a primeira correspondência que encontrar.

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

(continua na próxima página)

```
>>> 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>
```

A RE corresponde a `'<'` em `'<html>'`, e o `.` consome o resto da string. Existe ainda mais coisa existente na RE, no entanto, e o `>` pode não corresponder com o final da string, de modo que o mecanismo de expressão regular tem que recuar caractere por caractere até encontrar uma correspondência para `>`. 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 quantificadores não-gulosos `*?`, `+?`, `??` ou `{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.

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

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, que 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.

O livro mais completo sobre expressões regulares é quase certamente o *Mastering Regular Expressions* de Jeffrey Friedl's, publicado pela O'Reilly. Infelizmente, ele se concentra exclusivamente em sabores de expressões regulares do Perl e do Java, e não contém qualquer material relativo a Python, por isso não vai ser útil como uma referência para a programação em Python. (A primeira edição cobre o módulo `regex` agora removido do Python, o que não vai te ajudar muito.) Considere removê-lo de sua biblioteca.