
Expresiones regulares COMOS (HOWTO)

Versión 3.13.3

Guido van Rossum and the Python development team

abril 25, 2025

Python Software Foundation
Email: docs@python.org

Índice general

1	Introducción	2
2	Patrones simples	2
2.1	Coincidencia de caracteres (<i>Matching Characters</i>)	2
2.2	Repitiendo cosas	3
3	Usando expresiones regulares	4
3.1	Compilando expresiones regulares	4
3.2	La plaga de la barra invertida (<i>The Backslash Plague</i>)	5
3.3	Realizando coincidencias	6
3.4	Funciones a nivel de módulo	8
3.5	Los flags de compilación	8
4	Más poder de patrones	10
4.1	Más metacaracteres	10
4.2	Agrupando	11
4.3	Grupos con nombre y sin captura	13
4.4	Aserciones anticipadas	14
5	Modificando cadenas de caracteres	15
5.1	Separando cadenas de caracteres	15
5.2	Búsqueda y reemplazo	16
6	Problemas comunes	17
6.1	Uso de métodos de cadenas de caracteres	17
6.2	<i>match()</i> versus <i>search()</i>	18
6.3	Codiciosa versus no codiciosa (<i>Greedy versus Non-Greedy</i>)	18
6.4	Usando <code>re.VERBOSE</code>	19
7	Feedback	19

Autor

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

Resumen

Este documento es un tutorial de introducción al uso de expresiones regulares en Python con el módulo `re`. Proporciona una introducción más apacible que la sección correspondiente en la Referencia de la Biblioteca.

1 Introducción

Las expresiones regulares (llamadas RE, o regex, o patrones de regex) son esencialmente en un lenguaje de programación diminuto y altamente especializado incrustado dentro de Python y disponible a través del módulo `re`. Usando este pequeño lenguaje, especificas las reglas para el conjunto de cadenas de caracteres posibles que deseas hacer coincidir; este conjunto puede contener frases en inglés, o direcciones de correo electrónico, o comandos TeX, o cualquier cosa que desee. A continuación, puede hacer preguntas como «¿Coincide esta cadena con el patrón?» o «¿Hay alguna coincidencia con el patrón en alguna parte de esta cadena?». También puede utilizar RE para modificar una cadena de caracteres o dividirla de varias formas.

Los patrones de expresiones regulares se compilan en una serie de códigos de bytes que luego son ejecutados por un motor de coincidencia escrito en C. Para un uso avanzado, puede ser necesario prestar mucha atención a cómo el motor ejecutará una RE dado y escribir la RE en un de cierta manera para producir un código de bytes que se ejecute más rápido. La optimización no se trata en este documento, porque requiere que tenga un buen conocimiento de los componentes internos del motor de coincidencia.

El lenguaje de expresiones regulares es relativamente pequeño y restringido, por lo que no todas las posibles tareas de procesamiento de cadenas de caracteres se pueden realizar utilizando expresiones regulares. También hay tareas que *se pueden* hacer con expresiones regulares, pero las expresiones resultan ser muy complicadas. En estos casos, es mejor que escriba código Python para realizar el procesamiento; Si bien el código Python será más lento que una expresión regular elaborada, probablemente también será más comprensible.

2 Patrones simples

Comenzaremos aprendiendo sobre las expresiones regulares más simples posibles. Dado que las expresiones regulares se utilizan para operar en cadenas de caracteres, comenzaremos con la tarea más común: hacer coincidir caracteres.

Para obtener una explicación detallada de la informática que subyace a las expresiones regulares (autómatas finitos deterministas y no deterministas), puede consultar casi cualquier libro de texto sobre la escritura de compiladores.

2.1 Coincidencia de caracteres (*Matching Characters*)

La mayoría de letras y caracteres simplemente coincidirán. Por ejemplo, la expresión regular `test` coincidirá exactamente con la cadena `test`. (Puede habilitar un modo que no distinga entre mayúsculas y minúsculas que permitirá que este RE coincida con `test` o `TEST` también; más sobre esto más adelante.)

Hay excepciones a esta regla; algunos caracteres son especiales *metacharacters*, y no coinciden. En cambio, señalan que debe coincidir con algo fuera de lo común, o afectan otras partes de la RE repitiéndolos o cambiando su significado. Gran parte de este documento está dedicado a discutir varios metacaracteres y lo que hacen.

Aquí hay una lista completa de los metacaracteres; sus significados se discutirán en el resto de este COMO (*HOWTO*).

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

Los primeros metacaracteres que veremos son `[` and `]`. Se utilizan para especificar una clase de carácter, que es un conjunto de caracteres que desea hacer coincidir. Los caracteres se pueden enumerar individualmente, o se puede indicar un rango de caracteres dando dos caracteres y separándolos con un `-`. Por ejemplo, `[abc]` coincidirá con cualquiera de los caracteres `a`, `b` o `c`; esto es lo mismo que `[a-c]`, que usa un rango para expresar el mismo conjunto de caracteres. Si quisiera hacer coincidir solo letras minúsculas, su RE sería `[a-z]`.

Los metacaracteres (excepto `\`) no están activos dentro de las clases. Por ejemplo, `[akm$]` coincidirá con cualquiera de los caracteres `'a'`, `'k'`, `'m'`, o `'$'`; `'$'` suele ser un metacarácter, pero dentro de una clase de carácter se le quita su naturaleza especial.

Puede hacer coincidir los caracteres que no figuran en la clase mediante el conjunto *complementing*. Esto se indica mediante la inclusión de un '^' como primer carácter de la clase. Por ejemplo, `[^5]` coincidirá con cualquier carácter excepto con '5'. Si el símbolo de intercalación aparece en otra parte de una clase de carácter, no tiene un significado especial. Por ejemplo: `[5^]` coincidirá con un '5' o un '^'.

Quizás el metacarácter más importante es la barra invertida, `\`. Al igual que en los literales de cadena de Python, la barra invertida puede ir seguida de varios caracteres para señalar varias secuencias especiales. También se usa para escapar de todos los metacaracteres, de modo que aún pueda emparejarlos en patrones; por ejemplo, si necesita hacer coincidir un `[o \`, puede precederlos con una barra invertida para eliminar su significado especial: `\[o \\`.

Algunas de las secuencias especiales que comienzan con `'\'` representan conjuntos predefinidos de caracteres que a menudo son útiles, como el conjunto de dígitos, el conjunto de letras o el conjunto de cualquier cosa que no sea un espacio en blanco.

Tomemos un ejemplo: `\w` coincide con cualquier carácter alfanumérico. Si el patrón de expresiones regulares se expresa en bytes, esto es equivalente a la clase `[a-zA-Z0-9_]`. Si el patrón de expresiones regulares es una cadena de caracteres, `\w` coincidirá con todos los caracteres marcados como letras en la base de datos Unicode proporcionada por el módulo `unicodedata`. Puede usar la definición más restringida de `\w` en un patrón de cadena proporcionando el indicador `re.ASCII` al compilar la expresión regular.

La siguiente lista de secuencias especiales no está completa. Para obtener una lista completa de secuencias y definiciones de clases expandidas para patrones de cadenas Unicode, consulte la última parte de Regular Expression Syntax en la referencia de la biblioteca estándar. En general, las versiones Unicode coinciden con cualquier carácter que esté en la categoría apropiada en la base de datos Unicode.

- `\d`
Coincide con cualquier dígito decimal; esto es equivalente a la clase `[0-9]`.
- `\D`
Coincide con cualquier carácter que no sea un dígito; esto es equivalente a la clase `[^0-9]`.
- `\s`
Coincide con cualquier carácter de espacio en blanco; esto es equivalente a la clase `[\t\n\r\f\v]`.
- `\S`
Coincide con cualquier carácter que no sea un espacio en blanco; esto es equivalente a la clase `[^\t\n\r\f\v]`.
- `\w`
Coincide con cualquier carácter alfanumérico; esto es equivalente a la clase `[a-zA-Z0-9_]`.
- `\W`
Coincide con cualquier carácter no alfanumérico; esto es equivalente a la clase `[^a-zA-Z0-9_]`.

Estas secuencias se pueden incluir dentro de una clase de carácter. Por ejemplo, `[\s, .]` es una clase de carácter que coincidirá con cualquier carácter de espacio en blanco, o ',', o '.'.

El metacarácter final en esta sección es `..` Coincide con cualquier cosa excepto un carácter de nueva línea, y hay un modo alternativo (`re.DOTALL`) donde coincidirá incluso con una nueva línea. `.` se usa a menudo cuando se desea hacer coincidir «cualquier carácter».

2.2 Repitiendo cosas

Ser capaz de hacer coincidir diferentes conjuntos de caracteres es lo primero que pueden hacer las expresiones regulares que ya no es posible con los métodos disponibles en cadenas de caracteres. Sin embargo, si esa fuera la única capacidad adicional de las expresiones regulares, no serían un gran avance. Otra capacidad es que puede especificar que partes de la RE deben repetirse un cierto número de veces.

El primer metacarácter para repetir cosas que veremos es `*`. `*` no coincide con el carácter literal '*'; en cambio, especifica que el carácter anterior puede coincidir cero o más veces, en lugar de exactamente una vez.

Por ejemplo, `ca*t` coincidirá con `'ct'` (0 'a' caracteres), `'cat'` (1 'a'), `'caaat'` (3 'a' caracteres), etc.

Las repeticiones como `*` son *greedy*; al repetir una RE, el motor de emparejamiento intentará repetirlo tantas veces como sea posible. Si las partes posteriores del patrón no coinciden, el motor de coincidencia hará una copia de seguridad y volverá a intentarlo con menos repeticiones.

Un ejemplo paso a paso hará que esto sea más obvio. Consideremos la expresión `a[bcd]*b`. Esto coincide con la letra 'a', cero o más letras de la clase `[bcd]`, y finalmente termina con una 'b'. Ahora imagina hacer coincidir este RE con la cadena de caracteres 'abcbd'.

Pa-sos	Coinciden-cias	Explicación
1	a	La a en las RE coincide.
2	abcbcd	El motor coincide con <code>[bcd]*</code> , yendo tan lejos como puede, que es hasta el final de la cadena de caracteres.
3	<i>Failure</i>	El motor intenta hacer coincidir b, pero la posición actual está al final de la cadena de caracteres, por lo que falla.
4	abcb	Hace una copia de seguridad para que <code>[bcd]*</code> coincida con un carácter menos.
5	<i>Failure</i>	Intente b de nuevo, pero la posición actual está en el último carácter, que es un 'd'.
6	abc	Haga una copia de seguridad de nuevo, de modo que <code>[bcd]*</code> solo coincida con bc.
6	abcb	Intente b de nuevo. Esta vez, el carácter en la posición actual es 'b', por lo que tiene éxito.

Se ha alcanzado el final de la RE y ha coincidido con 'abcb'. Esto demuestra cómo el motor de coincidencias llega tan lejos como puede al principio, y si no se encuentra ninguna coincidencia, retrocederá progresivamente y volverá a intentar de la RE una y otra vez. Hará una copia de seguridad hasta que haya probado cero coincidencias para `[bcd]*`, y si eso falla posteriormente, el motor concluirá que la cadena no coincide con la RE en absoluto.

Otro metacarácter que se repite es +, que coincide una o más veces. Preste especial atención a la diferencia entre * and +; coincide con *cero* o más veces, por lo que cualquier cosa que se repita puede no estar presente en absoluto, mientras que + requiere al menos *one* aparición. Para usar un ejemplo similar, 'cat' (1 'a'), 'caaat' (3 'a's), pero no coincidirá con 'ct'.

Hay otros dos operadores de repetición o cuantificadores. El carácter de interrogación, ?, coincide una vez o cero veces; puede pensar en ello como marcar algo como opcional. Por ejemplo, `home-?brew` coincide con 'homebrew' o 'home-brew'.

El cuantificador más complicado es `{m,n}`, donde *m* y *n* son enteros decimales. Este cuantificador significa que debe haber al menos *m* repeticiones y como máximo *n*. Por ejemplo, `a/{1,3}b` coincidirá con 'a/b', 'a//b', y 'a///b'. No coincidirá con 'ab', que no tiene barras diagonales, ni con 'a////b', que tiene cuatro.

Puede omitir *m* o *n*; en ese caso, se asume un valor razonable para el valor faltante. Omitir *m* se interpreta como un límite inferior de 0, mientras que omitir *n* da como resultado un límite superior de infinito.

The simplest case `{m}` matches the preceding item exactly *m* times. For example, `a/{2}b` will only match 'a//b'.

Los lectores con una inclinación reduccionista pueden notar que los otros tres cuantificadores se pueden expresar todos utilizando esta notación. `{0,}` es lo mismo que *, `{1,}` es equivalente a +, y `{0,1}` es lo mismo que ?. Es mejor usar *, +, o ? cuando sea posible, simplemente porque son más cortos y más fáciles de leer.

3 Usando expresiones regulares

Ahora que hemos visto algunas expresiones regulares simples, ¿cómo las usamos realmente en Python? El módulo `re` proporciona una interfaz para el motor de expresiones regulares, lo que le permite compilar RE en objetos y luego realizar coincidencias con ellos.

3.1 Compilando expresiones regulares

Las expresiones regulares se compilan en objetos de patrón, que tienen métodos para diversas operaciones, como buscar coincidencias de patrones o realizar sustituciones de cadenas de caracteres.

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

`re.compile()` también acepta un argumento opcional *flags*, usado para habilitar varias características especiales y variaciones de sintaxis. Repasaremos las configuraciones disponibles más adelante, pero por ahora un solo ejemplo servirá:

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

La RE se pasa a `re.compile()` como una cadena de caracteres. Las RE se manejan como cadenas de caracteres porque las expresiones regulares no son parte del lenguaje central de Python y no se creó una sintaxis especial para expresarlas. (Hay aplicaciones que no necesitan RE en absoluto, por lo que no hay necesidad de aumentar la especificación del lenguaje incluyéndolas). En cambio, el módulo `re` es simplemente un módulo de extensión C incluido en Python, al igual que los módulos `socket` o `zlib`.

Poner RE en cadenas de caracteres mantiene el lenguaje Python más simple, pero tiene una desventaja que es el tema de la siguiente sección.

3.2 La plaga de la barra invertida (*The Backslash Plague*)

Como se indicó anteriormente, las expresiones regulares usan el carácter de barra invertida (`'\'`) para indicar formas especiales o para permitir que se usen caracteres especiales sin invocar su significado especial. Esto entra en conflicto con el uso de Python del mismo carácter para el mismo propósito en cadenas literales.

Supongamos que desea escribir una RE que coincida con la cadena de caracteres `\section`, que podría encontrarse en un archivo LaTeX. Para averiguar qué escribir en el código del programa, comience con la cadena deseada para que coincida. A continuación, debe escapar de las barras invertidas y otros metacaracteres precediéndolos con una barra invertida, lo que da como resultado la cadena `\\section`. La cadena resultante que debe pasarse a `re.compile()` debe ser `\\\\section`. Sin embargo, para expresar esto como una cadena literal de Python, ambas barras invertidas deben escaparse *nuevamente*.

Caracteres	Explicación
<code>\section</code>	Cadena de texto que debe coincidir
<code>\\section</code>	Barra invertida de escape para <code>re.compile()</code>
<code>\\\\section</code>	Barra invertida de escape para un literal de cadena de caracteres

En resumen, para hacer coincidir una barra invertida literal, uno tiene que escribir `'\\\\'` como la cadena RE, porque la expresión regular debe ser `\\`, y cada barra invertida debe expresarse como `\\` dentro de un literal de cadena Python normal. En las RE que presentan barras invertidas repetidamente, esto genera muchas barras invertidas repetidas y dificulta la comprensión de las cadenas resultantes.

La solución es utilizar la notación de cadena de caracteres sin formato de Python para expresiones regulares; las barras invertidas no se manejan de ninguna manera especial en una cadena literal con el prefijo `'r'`, por lo que `r"\n"` es una cadena de dos caracteres que contiene `'\'` y `'n'`, mientras que `"\n"` es una cadena de un carácter que contiene una nueva línea. Las expresiones regulares a menudo se escribirán en código Python utilizando esta notación de cadena sin formato.

Además, las secuencias de escape especiales que son válidas en expresiones regulares, pero no válidas como literales de cadena de Python, ahora dan como resultado `DeprecationWarning` y eventualmente se convertirán en `SyntaxError`, lo que significa que las secuencias no serán válidas, si no se utiliza la notación de cadena sin formato o el escape de las barras invertidas.

Cadena de caracteres regulares	Cadena de caracteres crudas (<i>Raw string</i>)
<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 Realizando coincidencias

Una vez que tiene un objeto que representa una expresión regular compilada, ¿qué hace con él? Los objetos de patrón tienen varios métodos y atributos. Aquí solo se cubrirán los más importantes; consulte los documentos `re` para obtener una lista completa.

Método/atributo	Objetivo
<code>match()</code>	Determina si la RE coincide con el comienzo de la cadena de caracteres.
<code>search()</code>	Escanea una cadena, buscando cualquier ubicación donde coincida este RE.
<code>findall()</code>	Encuentra todas las subcadenas de caracteres donde coincide la RE y las retorna como una lista.
<code>finditer()</code>	Encuentra todas las subcadenas donde la RE coincide y las retorna como un término iterado iterator.

`match()` y `search()` retornan `None` si la coincidencia no puede ser encontrada. Si tienen éxito, se retorna una instancia `match object`, que contiene información sobre la coincidencia: dónde comienza y termina, la subcadena de caracteres con la que coincidió, y más.

Puedes aprender sobre esto experimentando de forma interactiva con el módulo `re`.

Este CÓMO (HOWTO) utiliza el intérprete estándar de Python para sus ejemplos. Primero, ejecute el intérprete de Python, importe el módulo `re` y compile en RE:

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

Ahora, puede intentar hacer coincidir varias cadenas con la RE `[a-z]+`. Una cadena de caracteres vacía no debería coincidir en absoluto, ya que `+` significa que “una o más repeticiones”. `match()` debería retornar `None` en este caso, lo que hará que el intérprete no imprima ningún resultado. Puede imprimir explícitamente el resultado de `match()` para aclarar esto.

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

Ahora, intentémoslo en una cadena de caracteres que debería coincidir, como `tempo`. En este caso, `match()` retornará un `match object`, por lo que debe almacenar el resultado en una variable para su posterior uso.

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

Ahora puede consultar `match object` para obtener información sobre la cadena coincidente. Las instancias de objetos coincidentes también tienen varios métodos y atributos; los más importantes son:

Método/atributo	Objetivo
<code>group()</code>	Retorna la cadena de caracteres que coincide con la RE
<code>start()</code>	Retorna la posición de inicio de la coincidencia
<code>end()</code>	Retorna la posición final de la coincidencia
<code>span()</code>	Retorna una tupla que contiene (inicio, final) las posiciones de coincidencia

Probando estos métodos pronto aclarará sus significados:

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

`group()` retorna la subcadena de caracteres que coincide con la RE. `start()` y `end()` retornan el índice inicial y final de la coincidencia. `span()` retorna el índice inicial y final en una única tupla. Dado que el método `match()` solo verifica si la RE coincide al comienzo de una cadena de caracteres, `start()` siempre será cero. Sin embargo, el método de patrones `search()` escanea a través de la cadena de caracteres, por lo que es posible que la coincidencia no comience en cero en ese 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)
```

En programas reales, el estilo más común es almacenar match object en una variable, y luego verificar si era `None`. Esto generalmente se ve así:

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

Dos métodos de patrón retornan todas las coincidencias de un patrón. `findall()` retorna una lista de cadenas de caracteres coincidentes:

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

El prefijo `r`, que convierte al literal en una cadena de caracteres literal sin formato, es necesario en este ejemplo porque las secuencias de escape en un cadena de caracteres literal «cocinado» normal que no son reconocidas por Python, a diferencia de las expresiones regulares, ahora dan como resultado `DeprecationWarning` y eventualmente se convertirá en `SyntaxError`. Ver *La plaga de la barra invertida (The Backslash Plague)*.

`findall()` tiene que crear la lista completa antes de que pueda retornarse como resultado. El método `finditer()` retorna una secuencia de match object instancias como iterados `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 Funciones a nivel de módulo

No es necesario crear un objeto patrón y llamar a sus métodos; el módulo `re` también proporciona funciones de nivel superior llamadas `match()`, `search()`, `findall()`, `sub()`, y así sucesivamente. Estas funciones toman los mismos argumentos que el método de patrón correspondiente con la cadena de RE agregada como primer argumento, y aún así retornan una instancia de `None` o `match object`.

```
>>> 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 ' >
```

Bajo el capó (*hood*), estas funciones simplemente crean un objeto patrón para usted y llaman al método apropiado en él. También almacenan el objeto compilado en un caché, por lo que las futuras llamadas que usen el mismo RE no necesitarán analizar el patrón una y otra vez.

¿Debería utilizar estas funciones a nivel de módulo o debería obtener el patrón y llamar a sus métodos usted mismo? Si está accediendo a una expresión regular dentro de un bucle, la compilación previa guardará algunas llamadas a funciones. Fuera de los bucles, no hay mucha diferencia gracias al caché interno.

3.5 Los flags de compilación

Las flags de compilación le permiten modificar algunos aspectos de cómo funcionan las expresiones regulares. Las flags están disponibles en el módulo `re` con dos nombres, un nombre largo como `IGNORECASE` y una forma corta de una letra como `I`. (Si está familiarizado con los modificadores de patrones de Perl, las formas de una letra usan las mismas letras; la forma corta de `re.VERBOSE` es `re.X`, por ejemplo). Se pueden especificar varios indicadores uniéndolos con *OR* bit a bit; `re.I | re.M` establece los flags `I` and `M`, por ejemplo.

Aquí hay una tabla de las flags disponibles, seguida de una explicación más detallada de cada una.

Flag	Significado
ASCII, A	Hace que varios escapes como <code>\w</code> , <code>\b</code> , <code>\s</code> y <code>\d</code> coincidan solo en caracteres ASCII con la propiedad respectiva.
DOTALL, S	Hace que <code>.</code> coincida con cualquier caracter, incluidas las nuevas líneas.
IGNORECASE, I	Hace coincidencias que no distingan entre mayúsculas y minúsculas.
LOCALE, L	Hace una coincidencia con reconocimiento de configuración regional.
MULTILINE, M	Coincidencia de varias líneas, que afecta a <code>^</code> y <code>\$</code> .
VERBOSE, X (for “extended”)	Habilite RE detallados, que se pueden organizar de manera más limpia y comprensible.

`re.I`

`re.IGNORECASE`

Realiza una coincidencia que no distinga entre mayúsculas y minúsculas; la clase de caracteres y las cadenas de caracteres literales coincidirán con las letras ignorando las mayúsculas. Por ejemplo, `[A-Z]` también coincidirá con letras minúsculas. La coincidencia completa de Unicode también funciona a menos que se utilice la flag `ASCII` para deshabilitar las coincidencias que no sean ASCII. Cuando los patrones Unicode `[a-z]` o `[A-Z]` se utilizan en combinación con el indicador `IGNORECASE`, coincidirán con las 52 letras ASCII y 4 letras adicionales no ASCII “İ” (U+0130, letra mayúscula latina I con un punto arriba), “ı” (U+0131, letra minúscula latina sin punto i), “ſ” (U+017F, letra minúscula latina larga s) y “K” (U+212A, signo de Kelvin). `Spam` coincidirá `'Spam'`, `'spam'`, `'spAM'`, o `'ƒpam'` (este último solo coincide en modo Unicode). Estas minúsculas no tiene en cuenta la configuración regional actual; lo hará si también establece la flag `LOCALE`.

`re.L`

`re.LOCALE`

Hace que `\w`, `\W`, `\b`, `\B` y la coincidencia que no distinga entre mayúsculas y minúsculas dependan de la configuración regional actual en lugar de la base de datos Unicode.

Las configuraciones regionales son una característica de la biblioteca C destinada a ayudar a escribir programas que tengan en cuenta las diferencias de idioma. Por ejemplo, si está procesando texto en francés codificado,

querrá poder escribir `\w+` para que coincida con las palabras, pero `\w` solo coincide con la clase de caracteres `[A-Za-z]` en patrones de bytes; no coincidirá con los bytes correspondientes a `é` or `ç`. Si su sistema está configurado correctamente y se selecciona una configuración regional francesa, ciertas funciones de C le dirán al programa que el byte correspondiente a `é` también debe considerarse una letra. Establecer el indicador `LOCALE` al compilar una expresión regular hará que el objeto compilado resultante use estas funciones C para `\w`; esto es más lento, pero también permite que `\w+` coincida con las palabras en francés como era de esperar. Se desaconseja el uso de esta flag en Python 3 ya que el mecanismo de configuración regional es muy poco confiable, solo maneja una «cultura» a la vez y solo funciona con configuraciones regionales de 8 bits. La coincidencia Unicode ya está habilitada de forma predeterminada en Python 3 para patrones Unicode (str), y puede manejar diferentes configuraciones regionales/idiomas.

`re.M`

`re.MULTILINE`

(`^` y `$` aún no se han explicado; se presentarán en la sección [Más metacaracteres.](#))

Por lo general, `^` coincide solo al principio de la cadena de caracteres, y `$` solo coincide con el final de la cadena de caracteres e inmediatamente antes del salto de línea (si existe) al final de la cadena de caracteres. Cuando se especifica esta bandera, `^` coincide al principio de la cadena y al comienzo de cada línea dentro de la cadena, inmediatamente después de cada nueva línea. De manera similar, el metacarácter `$` coincide al final de la cadena de caracteres y al final de cada línea (inmediatamente antes de cada nueva línea).

`re.S`

`re.DOTALL`

Hace que el carácter especial `.` coincida con cualquier carácter, incluida una nueva línea; sin esta bandera, `.` coincidirá con cualquier cosa *except* una nueva línea.

`re.A`

`re.ASCII`

Haga que `\w`, `\W`, `\b`, `\B`, `\s` y `\S` realicen una coincidencia solo en ASCII en lugar de una coincidencia Unicode completa. Esto solo es significativo para los patrones Unicode y se ignora para los patrones de bytes.

`re.X`

`re.VERBOSE`

Esta flag le permite escribir expresiones regulares que son más legibles al otorgarle más flexibilidad en cómo puede formatearlas. Cuando se ha especificado esta flag, los espacios en blanco dentro de la cadena de caracteres de la RE se ignoran, excepto cuando el espacio en blanco está en una clase de caracteres o está precedido por una barra invertida sin escape; esto le permite organizar e indentar la RE más claramente. Esta flag también le permite poner comentarios dentro de una RE que serán ignorados por el motor (*engine*); los comentarios están marcados con un `#` que no está en una clase de carácter ni está precedido por una barra invertida sin escape.

Por ejemplo, aquí hay una RE que usa `re.VERBOSE`; ¿Ves lo fácil que es leer?

```
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)
```

Sin la configuración detallada, la RE se vería así:

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

En el ejemplo anterior, la concatenación automática de cadenas de caracteres literales de Python se ha utilizado para dividir la RE en partes más pequeñas, pero aún es más difícil de entender que la versión que usa `re.VERBOSE`.

4 Más poder de patrones

Hasta ahora solo hemos cubierto una parte de las características de las expresiones regulares. En esta sección, cubriremos algunos metacaracteres nuevos y cómo usar grupos para recuperar partes del texto que coincidió.

4.1 Más metacaracteres

Hay algunos metacaracteres que aún no hemos cubierto. La mayoría de ellos se tratarán en esta sección.

Algunos de los metacaracteres restantes que se discutirán son *zero-width assertions*. No hacen que el motor avance a través de la cadena de caracteres; en cambio, no consumen caracteres en absoluto y simplemente tienen éxito o fracasan. Por ejemplo, `\b` es una flag de que la posición actual se encuentra en el límite de una palabra; la posición no cambia por la `\b` en absoluto. Esto significa que las aserciones de ancho cero nunca deben repetirse, porque si coinciden una vez en una ubicación determinada, obviamente pueden coincidir un número infinito de veces.

|

Alternancia, o el operador «or». Si *A* y *B* son expresiones regulares, *A|B* coincidirá con cualquier cadena de caracteres que coincida con *A* o *B*. | tiene una precedencia muy baja para que funcione razonablemente cuando está alternando cadenas de varios caracteres. `Crow|Servo` coincidirá con `'Crow'` o `'Servo'`, no `'Cro'`, un `'w'` o un `'S'`, y `'ervo'`.

Para hacer coincidir un literal `'|'`, use `\|`, o enciérrelo dentro de una clase de carácter, como en `[|]`.

^

Coincide con el comienzo de las líneas. A menos que se haya establecido la flag `MULTILINE`, esto solo coincidirá al principio de la cadena de caracteres. En modo `MULTILINE`, esto también coincide inmediatamente después de cada nueva línea dentro de la cadena.

Por ejemplo, si desea hacer coincidir la palabra `From` solo al principio de una línea, la RE que debe usar es `^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 una coincidencia literal `'^'`, usar `\^`.

\$

Coincide con el final de una línea, que se define como el final de la cadena o cualquier ubicación seguida de un carácter de nueva línea.

```
>>> 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 hacer coincidir un literal `'$'`, usar `\$` o enciérrelo dentro de una clase de carácter, como en `[$]`.

\A

Coincide solo al comienzo de la cadena de caracteres. Cuando no está en el modo `MULTILINE`, `\A` y `^` son efectivamente lo mismo. En el modo `MULTILINE`, son diferentes: `\A` todavía coincide solo al principio de la cadena, pero `^` puede coincidir en cualquier ubicación dentro de la cadena de caracteres que sigue a un carácter de nueva línea.

\Z

Coincidencias solo al final de la cadena de caracteres.

\b

Límite de palabra. Esta es una aserción de ancho cero que coincide solo al principio o al final de una palabra. Una palabra se define como una secuencia de caracteres alfanuméricos, por lo que el final de una palabra se indica mediante un espacio en blanco o un carácter no alfanumérico.

El siguiente ejemplo coincide con `class` solo cuando es una palabra completa; no coincidirá cuando esté contenido dentro de otra palabra.

```
>>> 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
```

Hay dos sutilezas que debe recordar al usar esta secuencia especial. Primero, esta es la peor colisión entre las cadenas literales de caracteres de Python y las secuencias de expresiones regulares. En las cadenas de caracteres literales de Python, `\b` es el carácter de retroceso (*backspace*), valor ASCII 8. Si no está utilizando cadenas de caracteres sin procesar, Python convertirá la `\b` en una línea de retroceso, y su RE no lo hará coincidir como lo espera. El siguiente ejemplo tiene el mismo aspecto que nuestra RE anterior, pero omite la `'r'` delante de la cadena de caracteres de 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'>
```

En segundo lugar, dentro de una clase de caracteres, donde no hay uso para esta aserción, `\b` representa el carácter de retroceso, por compatibilidad con las cadenas de caracteres literales de Python.

\B

Otra flag de ancho cero, esto es lo opuesto a `\b`, solo coincide cuando la posición actual no está en el límite de una palabra.

4.2 Agrupando

Con frecuencia, necesita obtener más información que solo si la RE coincide o no. Las expresiones regulares se utilizan a menudo para diseccionar cadenas de caracteres escribiendo una RE dividido en varios subgrupos que coinciden con diferentes componentes de interés. Por ejemplo, una línea de encabezado RFC-822 se divide en un nombre de encabezado y un valor, separados por un `:`, así:

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

Esto se puede manejar escribiendo una expresión regular que coincida con una línea de encabezado completa y que tenga un grupo que coincida con el nombre del encabezado y otro grupo que coincida con el valor del encabezado.

Los grupos se marcan con los metacaracteres `'(' y ')'`. `'(' y ')'` tienen prácticamente el mismo significado que en las expresiones matemáticas; agrupan las expresiones contenidas dentro de ellos, y puedes repetir el contenido de un grupo con un cuantificador, como `*`, `+`, `?`, o `{m,n}`. Por ejemplo, `(ab)*` coincidirá con cero o más repeticiones de `ab`.

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

Los grupos indicados con ' (, ') ' también capturan el índice inicial y final del texto que coinciden; esto se puede recuperar pasando un argumento a `group()`, `start()`, `end()`, y `span()`. Los grupos se numeran empezando por 0. El grupo 0 siempre está presente; es todo las RE, entonces todos los métodos match object tienen el grupo 0 como argumento predeterminado. Más adelante veremos cómo expresar grupos que no capturan el espacio de texto que coinciden.

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

Los subgrupos están numerados de izquierda a derecha, de 1 en adelante. Los grupos se pueden anidar; para determinar el número, simplemente cuente los caracteres del paréntesis de apertura, de izquierda a derecha.

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

`group()` se pueden pasar varios números de grupo a la vez, en cuyo caso retornará una tupla que contiene los valores correspondientes para esos grupos.

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

El método `groups()` retorna una tupla que contiene las cadenas de caracteres de todos los subgrupos, desde 1 hasta la cantidad que haya.

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

Las referencias inversas en un patrón le permiten especificar que el contenido de un grupo de captura anterior también debe encontrarse en la ubicación actual en la cadena. Por ejemplo, “1” tendrá éxito si el contenido exacto del grupo 1 se puede encontrar en la posición actual y falla en caso contrario. Recuerde que las cadenas de caracteres literales de Python también usan una barra invertida seguida de números para permitir la inclusión de caracteres arbitrarios en una cadena de caracteres, así que asegúrese de usar una cadena de caracteres sin procesar al incorporar referencias inversas en una RE.

Por ejemplo, la siguiente RE detecta palabras duplicadas en una cadena.

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

Las referencias inversas como esta no suelen ser útiles para buscar a través de una cadena — hay pocos formatos de texto que repiten datos de esta manera — pero pronto descubrirá que son *muy* útiles al realizar sustituciones de cadenas de caracteres.

4.3 Grupos con nombre y sin captura

Las RE elaboradas pueden utilizar muchos grupos, tanto para capturar subcadenas de caracteres de interés como para agrupar y estructurar la propia RE. En las RE complejas, resulta difícil realizar un seguimiento de los números de los grupos. Hay dos funciones que ayudan con este problema. Ambos usan una sintaxis común para las extensiones de expresiones regulares, así que veremos eso primero.

Perl 5 es bien conocido por sus poderosas adiciones a las expresiones regulares estándar. Para estas nuevas características, los desarrolladores de Perl no podían elegir nuevos metacaracteres de una sola pulsación de tecla o nuevas secuencias especiales que comienzan con `\` sin hacer que las expresiones regulares de Perl sean confusamente diferentes de las RE estándar. Si eligieran `&` como un nuevo metacarácter, por ejemplo, las expresiones antiguas supondrían que `&` era un carácter regular y no se habría escapado escribiendo `\&` o `[&]`.

La solución elegida por los desarrolladores de Perl fue utilizar `(? . . .)` como sintaxis de extensión. `?` inmediatamente después de un paréntesis había un error de sintaxis porque el `?` no tendría nada que repetir, por lo que esto no introdujo ningún problema de compatibilidad. Los caracteres inmediatamente después de `?` indican qué extensión se está utilizando, por lo que `(?=foo)` es una cosa (una flag de anticipación positiva) y `(?:foo)` es otra cosa (un grupo de no captura que contiene la subexpresión `foo`).

Python admite varias de las extensiones de Perl y agrega una sintaxis de extensión a la sintaxis de extensión de Perl. Si el primer carácter después del signo de interrogación es una `P`, sabrá que es una extensión específica de Python.

Ahora que hemos visto la sintaxis de la extensión general, podemos volver a las características que simplifican el trabajo con grupos en RE complejos.

A veces querrá usar un grupo para denotar una parte de una expresión regular, pero no está interesado en recuperar el contenido del grupo. Puede hacer que este hecho sea explícito utilizando un grupo de no captura: `(?: . . .)`, donde puede reemplazar el `. . .` con cualquier otra expresión regular.

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

Excepto por el hecho de que no puede recuperar el contenido de lo que coincide con el grupo, un grupo que no captura se comporta exactamente igual que un grupo que captura; puede poner cualquier cosa dentro de él, repetirlo con un metacarácter de repetición como `*` y anidarlos dentro de otros grupos (capturando o no capturando). `(?: . . .)` es particularmente útil cuando se modifica un patrón existente, ya que puede agregar nuevos grupos sin cambiar cómo se numeran todos los demás grupos. Cabe mencionar que no hay diferencia de rendimiento en la búsqueda entre grupos de captura y no captura; ninguna forma es más rápida que la otra.

Una característica más significativa son los grupos nombrados: en lugar de referirse a ellos por números, los grupos pueden ser referenciados por un nombre.

La sintaxis de un grupo con nombre es una de las extensiones específicas de Python: `(?P<name> . . .)`. `name` es, obviamente, el nombre del grupo. Los grupos con nombre se comportan exactamente como los grupos de captura y, además, asocian un nombre con un grupo. Los métodos `match object` que tratan con la captura de grupos aceptan enteros que se refieren al grupo por número o cadenas de caracteres que contienen el nombre del grupo deseado. Los grupos con nombre todavía reciben números, por lo que puede recuperar información sobre un grupo de dos maneras:

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

Además, puede recuperar grupos nombrados como un diccionario con `groupdict()`:

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

Los grupos nombrados son útiles porque le permiten usar nombres fáciles de recordar en lugar de tener que recordar números. Aquí hay un ejemplo de una RE del 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'")')
```

Obviamente, es mucho más fácil recuperar `m.group('zonem')`, en lugar de tener que recordar recuperar el grupo 9.

La sintaxis de las referencias inversas en una expresión como `(...)\1` se refiere al número del grupo. Naturalmente, existe una variante que usa el nombre del grupo en lugar del número. Esta es otra extensión de Python: `(?P=name)` indica que el contenido del grupo llamado *name* debe coincidir nuevamente en el punto actual. La expresión regular para encontrar palabras duplicadas, `\b(\w+)\s+\1\b` también se puede escribir como `\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 Aserciones anticipadas

Otra aserción de ancho cero es la aserción de anticipación. Las afirmaciones anticipadas están disponibles tanto en forma positiva como negativa, y tienen este aspecto:

(?=...)

Aserción de anticipación positiva. Esto tiene éxito si la expresión regular contenida, representada aquí por `...`, coincide con éxito en la ubicación actual y falla en caso contrario. Pero, una vez que se ha probado la expresión contenida, el motor de comparación no avanza en absoluto; el resto del patrón se intenta justo donde comenzó la aserción.

(?!...)

Aserción de anticipación negativa. Esto es lo opuesto a la flag positiva; tiene éxito si la expresión contenida *no* coincide con la posición actual en la cadena.

Para que esto sea concreto, veamos un caso en el que una búsqueda anticipada es útil. Considere un patrón simple para hacer coincidir un nombre de archivo y dividirlo en un nombre base y una extensión, separados por un `..`. Por ejemplo, en `news.rc`, `news` es el nombre base y `rc` es la extensión del nombre del archivo.

El patrón para que coincida con esto es bastante simple:

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

Tenga en cuenta que el `.` Debe tratarse especialmente porque es un metacarácter, por lo que está dentro de una clase de carácter para coincidir solo con ese carácter específico. También observe el final `$`; esto se agrega para garantizar que todo el resto de la cadena deba incluirse en la extensión. Esta expresión regular coincide con `foo.bar` y `autoexec.bat` y `sendmail.cf` y `printers.conf`.

Ahora, considere complicar un poco el problema; ¿Qué sucede si desea hacer coincidir los nombres de archivo donde la extensión no es `bat`? Algunos intentos incorrectos:

`.*[.][^b].*$` El primer intento anterior intenta excluir `bat` requiriendo que el primer carácter de la extensión no sea una `b`. Esto está mal, porque el patrón tampoco coincide `foo.bar`.

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

La expresión se vuelve más desordenada cuando intenta parchear la primera solución requiriendo que coincida uno de los siguientes casos: el primer carácter de la extensión no es `b`; el segundo carácter no es `a`; o el tercer carácter no es `t`. Esto acepta `foo.bar` y rechaza `autoexec.bat`, pero requiere una extensión de tres letras y no acepta un nombre de archivo con una extensión de dos letras como `sendmail.cf`. Complicaremos el patrón nuevamente en un esfuerzo por arreglarlo.

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

En el tercer intento, la segunda y tercera letras se hacen opcionales para permitir extensiones coincidentes de menos de tres caracteres, como `sendmail.cf`.

El patrón se está volviendo realmente complicado ahora, lo que dificulta su lectura y comprensión. Peor aún, si el problema cambia y desea excluir tanto `bat` y `exe` como extensiones, el patrón se volvería aún más complicado y confuso.

Una mirada anticipada negativa atraviesa toda esta confusión:

`.*[.](?!bat$)[^.]*$` La búsqueda anticipada negativa significa: si la expresión `bat` no coincide en este punto, pruebe el resto del patrón; si `bat$` coincide, todo el patrón fallará. El `$` final es necesario para garantizar que se permita algo como `sample.batch`, donde la extensión solo comienza con `bat`. El `[^.]` asegura que el patrón funcione cuando hay varios puntos en el nombre del archivo.

Ahora es fácil excluir otra extensión de nombre de archivo; simplemente agréguelo como una alternativa dentro de la aserción. El siguiente patrón excluye los nombres de archivo que terminan en `bat` o `exe`:

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

5 Modificando cadenas de caracteres

Hasta este punto, simplemente hemos realizado búsquedas en una cadena de caracteres estática. Las expresiones regulares también se utilizan comúnmente para modificar cadenas de varias formas, utilizando los siguientes métodos de patrón:

Método/atributo	Objetivo
<code>split()</code>	Divida la cadena de caracteres en una lista, dividiéndola donde coincida la RE
<code>sub()</code>	Encuentra todas las subcadenas de caracteres donde coincida la RE y las reemplaza con una cadena de caracteres diferente
<code>subn()</code>	Hace lo mismo que <code>sub()</code> , pero retorna la nueva cadena de caracteres y el número de reemplazos

5.1 Separando cadenas de caracteres

El método `split()` de un patrón que divide una cadena de caracteres donde la RE coincide, retornando una lista de las piezas. Es similar al método de cadenas de caracteres `split()` pero proporciona mucha más generalidad en los delimitadores por los que puede dividir; cadena de caracteres `split()` solo admite la división por espacios en blanco o por una cadena fija. Como era de esperar, también hay una función a nivel de módulo `re.split()`.

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

Dividir *string* por las coincidencias de la expresión regular. Si se utilizan paréntesis de captura en la RE, su contenido también se retornará como parte de la lista resultante. Si *maxsplit* es distinto de cero, se realizan como máximo divisiones *maxsplit*.

Puede limitar el número de divisiones realizadas, pasando un valor para *maxsplit*. Cuando *maxsplit* es distinto de cero, se realizarán como máximo *maxsplit* divisiones, y el resto de la cadena de caracteres se retorna como el elemento final de la lista. En el siguiente ejemplo, el delimitador es cualquier secuencia de caracteres no 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', '']
```

(continúe en la próxima página)

(proviene de la página anterior)

```
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

A veces, no solo le interesa cuál es el texto entre delimitadores, sino que también necesita saber cuál era el delimitador. Si se utilizan paréntesis de captura en la RE, sus valores también se retornan como parte de la lista. Compare las siguientes llamadas:

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

La función de nivel de módulo `re.split()` agrega la RE que se usará como primer argumento, pero por lo demás es el mismo.

```
>>> 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 Búsqueda y reemplazo

Otra tarea común es encontrar todas las coincidencias para un patrón y reemplazarlas con una cadena de caracteres diferente. El método `sub()` toma un valor de reemplazo, que puede ser una cadena de caracteres o una función, y la cadena de caracteres a procesar.

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

Retorna la cadena de caracteres obtenida al reemplazar las apariciones no superpuestas del extremo izquierdo de la RE en *string* por el reemplazo *replacement*. Si no se encuentra el patrón, el *string* se retorna sin cambios.

El argumento opcional *count* es el número máximo de ocurrencias de patrones que se reemplazarán; *count* debe ser un número entero no negativo. El valor predeterminado de 0 significa reemplazar todas las ocurrencias.

Aquí hay un ejemplo simple del uso del método `sub()`. Reemplaza los nombres de los colores con la palabra `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'
```

El método `subn()` hace el mismo trabajo, pero retorna una tupla de 2 que contiene el nuevo valor de cadena de caracteres y el número de reemplazos que se realizaron:

```
>>> 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)
```

Las coincidencias vacías se reemplazan solo cuando no son adyacentes a una coincidencia vacía anterior.

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

Si *replacement* es una cadena, se procesan los escapes de barra invertida que contenga. Es decir, `\n` se convierte en un solo carácter de nueva línea, `\r` se convierte en un RETorno de carro, y así sucesivamente. Los escapes desconocidos como `&` se dejan en paz. Las referencias inversas, como `\6`, se reemplazan con la subcadena de caracteres que coincide con el grupo correspondiente a la RE. Esto le permite incorporar partes del texto original en la cadena de reemplazo resultante.

Este ejemplo hace coincidir la palabra `section` seguida de una cadena encerrada entre `{, }`, y cambia `section` a `subsection`:

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

También hay una sintaxis para hacer referencia a grupos con nombre según lo definido por la sintaxis `(?P<name>...)`. `\g<name>` usará la subcadena de caracteres que coincide con el grupo llamado `name`, y `\g<number>` usa el número de grupo correspondiente. `\g<2>` es equivalente a `\2`, pero no es ambiguo en una cadena de reemplazo como `\g<2>0`. (`\20` se interpretaría como una referencia al grupo 20, no como una referencia al grupo 2 seguido del carácter literal '0'.) Las siguientes sustituciones son todas equivalentes, pero use las tres variaciones de la cadena de reemplazo.

```
>>> 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 también puede ser una función, lo que le brinda aún más control. Si *replacement* es una función, la función se llama para cada ocurrencia no superpuesta de *pattern*. En cada llamada, a la función se le pasa un argumento `match object` para la coincidencia y puede usar esta información para calcular la cadena de reemplazo deseada y retornarla.

En el siguiente ejemplo, la función de reemplazo traduce decimales a hexadecimales:

```
>>> 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.'
```

Cuando se usa la función *module-level* `re.sub()`, el patrón se pasa como primer argumento. El patrón puede proporcionarse como un objeto o como una cuerda; Si necesita especificar marcas de expresión regular, debe usar un objeto de patrón como primer parámetro o usar modificadores incrustados en la cadena de patrón, por ejemplo `sub("(?i)b+", "x", "bbbb BBBB")` retorna `'x x'`.

6 Problemas comunes

Las expresiones regulares son una herramienta poderosa para algunas aplicaciones, pero de alguna manera su comportamiento no es intuitivo y, a veces, no se comportan de la forma esperada. Esta sección señalará algunos de los errores más comunes.

6.1 Uso de métodos de cadenas de caracteres

A veces, usar el módulo `re` es un error. Si está haciendo coincidir una cadena de caracteres fija, o una clase de un solo carácter, y no está usando ninguna característica `re` como la marca `IGNORECASE`, entonces todo el poder de las expresiones regulares puede que no sea necesario. Las cadenas tienen varios métodos para realizar operaciones con

cadenas fijas y, por lo general, son mucho más rápidas, porque la implementación es un único bucle C pequeño que se ha optimizado para este propósito, en lugar del motor de expresión regular más grande y generalizado.

Un ejemplo podría ser reemplazar una sola cadena fija por otra; por ejemplo, puede reemplazar `word` por `deed`. `re.sub()` parece la función a utilizar para esto, pero considere el método `replace()`. Tenga en cuenta que `replace()` también reemplazará `word` dentro de las palabras, convirtiendo `swordfish` en `sdeedfish`, pero la RE naïf `word` también lo habría hecho. (Para evitar realizar la sustitución en partes de palabras, el patrón tendría que ser `\bword\b`, para requerir que `word` tenga un límite de palabra en cada lado. Esto lleva el trabajo más allá de las habilidades de `replace()`.)

Otra tarea común es eliminar cada aparición de un solo carácter de una cadena o reemplazarlo con otro solo carácter. Puede hacer esto con algo como `re.sub('\n', ' ', S)`, pero `translate()` es capaz de realizar ambas tareas y será más rápido que cualquier expresión regular la operación puede ser.

En resumen, antes de pasar al módulo `re`, considere si su problema puede resolverse con un método de cadena de caracteres más rápido y simple.

6.2 `match()` versus `search()`

La función `match()` solo verifica si la RE coincide con el comienzo de la cadena de caracteres, mientras que `search()` buscará una coincidencia en la cadena de caracteres. Es importante tener en cuenta esta distinción. Recuerde `match()` solo informará una coincidencia exitosa que comenzará en 0; si la coincidencia no comienza en cero, `match()` *no* lo informará.

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

Por otro lado, `search()` escaneará hacia adelante a través de la cadena de caracteres, informando la primera coincidencia que encuentre.

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

A veces, tendrá la tentación de seguir usando `re.match()`, y simplemente agregar `.*` al frente de su RE. Resista esta tentación y use `re.search()` en su lugar. El compilador de expresiones regulares realiza un análisis de las RE para acelerar el proceso de búsqueda de coincidencias. Uno de esos análisis determina cuál debe ser el primer carácter de una coincidencia; por ejemplo, un patrón que comienza con `Crow` debe coincidir con una `'C'`. El análisis permite que el motor escanee rápidamente a través de la cadena en busca del carácter inicial, solo probando la coincidencia completa si se encuentra una `'C'`.

Agregar `.*` anula esta optimización, lo que requiere escanear hasta el final de la cadena y luego retroceder para encontrar una coincidencia para el resto de la RE. Utilice `re.search()` en su lugar.

6.3 Codiciosa versus no codiciosa (*Greedy versus Non-Greedy*)

Al repetir una expresión regular, como en `a*`, la acción resultante es consumir la mayor cantidad posible del patrón. Este hecho suele molestarle cuando intenta hacer coincidir un par de delimitadores equilibrados, como los corchetes angulares que rodean una etiqueta HTML. El patrón ingenuo para hacer coincidir una sola etiqueta HTML no funciona debido a la naturaleza codiciosa de `.*`.

```
>>> s = '<html><head><title>Title</title>'\n>>> len(s)\n32\n>>> print(re.match('<.*>', s).span())\n(0, 32)\n>>> print(re.match('<.*>', s).group())\n<html><head><title>Title</title>
```

La RE coincide con el '`<`' en '`<html>`', y el `.*` consume el resto de la cadena de caracteres. Sin embargo, aún queda más en la RE y el '`>`' no puede coincidir al final de la cadena de caracteres, por lo que el motor de la expresión regular tiene que retroceder carácter por carácter hasta que encuentre una coincidencia para el '`>`'. La coincidencia final se extiende desde el '`<`' en '`<html>`' al '`>`' en '`</title>`', que no es lo que queremos.

En este caso, la solución es utilizar los cuantificadores no codiciosos `*?`, `+?`, `??`, o `{m,n}?`, que coinciden con la *menor* cantidad de texto posible. En el ejemplo anterior, el '`>`' se prueba inmediatamente después de que el primer '`<`' coincida, y cuando falla, el motor avanza un carácter a la vez, volviendo a intentar el '`>`' en cada paso. Esto produce el resultado correcto:

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

(Tenga en cuenta que analizar HTML o XML con expresiones regulares es doloroso. Los patrones rápidos y sucios manejarán casos comunes, pero HTML y XML tienen casos especiales que romperán la expresión regular obvia; para cuando haya escrito una expresión regular que maneja todos los casos posibles, los patrones serán *muy* complicados. Utilice un módulo analizador HTML o XML para tales tareas.)

6.4 Usando re.VERBOSE

Probablemente ya haya notado que las expresiones regulares son una notación muy compacta, pero no son muy legibles. Las RE de complejidad moderada pueden convertirse en largas colecciones de barras invertidas, paréntesis y metacaracteres, lo que dificulta su lectura y comprensión.

Para tales RE, especificar el flag `re.VERBOSE` al compilar la expresión regular puede ser útil, porque le permite formatear la expresión regular con mayor claridad.

La flag `re.VERBOSE` tiene varios efectos. Los espacios en blanco en la expresión regular que *no están* dentro de una clase de caracteres se ignoran. Esto significa que una expresión como `dog | cat` es equivalente al menos legible `dog | cat`, pero `[a b]` seguirá coincidiendo con los caracteres '`a`', '`b`' o un espacio. Además, también puede poner comentarios dentro de una RE; los comentarios se extienden desde un carácter `#` hasta la siguiente línea nueva. Cuando se usa con cadenas entre comillas triples, esto permite que las REs sean formateados de manera más ordenada:

```
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)
```

Esto es mas legible que:

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

7 Feedback

Las expresiones regulares son un tema complicado. ¿Le ayudó este documento a comprenderlas? ¿Hubo partes que no estaban claras o problemas que encontró que no se trataron aquí? Si es así, envíe sugerencias de mejora al autor.

El libro más completo sobre expresiones regulares es casi con certeza *Mastering Regular Expressions* de Jeffrey Friedl, publicado por *O'Reilly*. Desafortunadamente, se concentra exclusivamente en los tipos de expresiones regulares de Perl y Java, y no contiene ningún material de Python, por lo que no será útil como referencia para la programación en Python. (La primera edición cubría el módulo `regex` de Python, ahora eliminado, que no le ayudará mucho.) Considere sacarlo de su biblioteca.