`re` — Operaciones con expresiones regulares¶

Código fuente: Lib/re.py

Este módulo proporciona operaciones de coincidencia de expresiones regulares similares a las encontradas en Perl.

Tanto los patrones como las cadenas de texto a buscar pueden ser cadenas de Unicode (`str`) así como cadenas de 8 bits (`bytes`). Sin embargo, las cadenas Unicode y las cadenas de 8 bits no se pueden mezclar: es decir, no se puede hacer coincidir una cadena Unicode con un patrón de bytes o viceversa; del mismo modo, al pedir una sustitución, la cadena de sustitución debe ser del mismo tipo que el patrón y la cadena de búsqueda.

Las expresiones regulares usan el carácter de barra inversa (`'\'`) para indicar formas especiales o para permitir el uso de caracteres especiales sin invocar su significado especial. Esto choca con el uso de Python de este carácter para el mismo propósito con los literales de cadena; por ejemplo, para hacer coincidir una barra inversa literal, se podría escribir `'\\\\'` como patrón, porque la expresión regular debe ser `\\`, y cada barra inversa debe ser expresada como `\\` dentro de un literal de cadena regular de Python. También, notar que cualquier secuencia de escape inválida mientras se use la barra inversa de Python en los literales de cadena ahora genera un `DeprecationWarning` y en el futuro esto se convertirá en un `SyntaxError`. Este comportamiento ocurrirá incluso si es una secuencia de escape válida para una expresión regular.

La solución es usar la notación de cadena *raw* de Python para los patrones de expresiones regulares; las barras inversas no se manejan de ninguna manera especial en un literal de cadena prefijado con `'r'`. Así 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. Normalmente los patrones se expresan en código Python usando esta notación de cadena *raw*.

Es importante señalar que la mayoría de las operaciones de expresiones regulares están disponibles como funciones y métodos a nivel de módulo en expresiones regulares compiladas (expresiones regulares compiladas). Las funciones son atajos que no requieren de compilar un objeto regex primero, aunque pasan por alto algunos parámetros de ajuste.

> **Ver también:** El módulo de terceros regex , cuenta con una API compatible con el módulo de la biblioteca estándar `re`, el cual ofrece una funcionalidad adicional y un soporte Unicode más completo.

Sintaxis de expresiones regulares

Una expresión regular (o RE, por sus siglas en inglés) especifica un conjunto de cadenas que coinciden con ella; las funciones de este módulo permiten comprobar si una determinada

cadena coincide con una expresión regular dada (o si una expresión regular dada coincide con una determinada cadena, que se reduce a lo mismo).

Las expresiones regulares pueden ser concatenadas para formar nuevas expresiones regulares; si *A* y *B* son ambas expresiones regulares, entonces *AB* es también una expresión regular. En general, si una cadena *p* coincide con *A* y otra cadena *q* coincide con *B*, la cadena *porque* coincidirá con AB. Esto se mantiene a menos que *A* o *B* contengan operaciones de baja precedencia; condiciones límite entre *A* y *B*; o tengan referencias de grupo numeradas. Así, las expresiones complejas pueden construirse fácilmente a partir de expresiones primitivas más simples como las que se describen aquí. Para detalles de la teoría e implementación de las expresiones regulares, consulte el libro de Friedl [Frie09], o casi cualquier libro de texto sobre la construcción de compiladores.

A continuación se explica brevemente el formato de las expresiones regulares. Para más información y una presentación más amena, consultar la Regular Expression HOWTO.

Las expresiones regulares pueden contener tanto caracteres especiales como ordinarios. La mayoría de los caracteres ordinarios, como `'A'`, `'a'`, o `'0'` son las expresiones regulares más sencillas; simplemente se ajustan a sí mismas. Se pueden concatenar caracteres ordinarios, así que `last` coincide con la cadena `'last'`. (En el resto de esta sección, se escribirán los RE en `este estilo especial`, normalmente sin comillas, y las cadenas que deban coincidir `'entre comillas simples'`.)

Algunos caracteres, como `'|'` o `'('`, son especiales. Los caracteres especiales representan clases de caracteres ordinarios, o afectan a la forma en que se interpretan las expresiones regulares que los rodean.

Los delimitadores de repetición (`*`, `+`, `?`, `{m,n}`, etc.) no pueden ser anidados directamente. Esto evita la ambigüedad con el sufijo modificador no *greedy* (codiciosos) `?`, y con otros modificadores en otras implementaciones. Para aplicar una segunda repetición a una repetición interna, se pueden usar paréntesis. Por ejemplo, la expresión `(?:a{6})*` coincide con cualquier múltiplo de seis caracteres `'a'`.

Los caracteres especiales son:

`.`
: (Punto.) En el modo predeterminado, esto coincide con cualquier carácter excepto con una nueva línea. Si se ha especificado el indicador `DOTALL`, esto coincide con cualquier carácter que incluya una nueva línea.

`^`
: (Circunflejo.) Coincide con el comienzo de la cadena, y en modo `MULTILINE` también coincide inmediatamente después de cada nueva línea.

`$`
: Coincide con el final de la cadena o justo antes de la nueva línea al final de la cadena, y en modo `MULTILINE` también coincide antes de una nueva línea. `foo` coincide con “foo” y “foobar”, mientras que la expresión regular `foo$` sólo coincide con “foo”. Más interesante aún, al buscar `foo.$` en `'foo1\nfoo2\n'` coincide con “foo2” normalmente, pero solo

"foo1" en `MULTILINE`; si busca un solo `$` en `'foo\n'` encontrará dos coincidencias (vacías): una justo antes de una nueva línea, y otra al final de la cadena.

`*`

Hace que el RE resultante coincida con 0 o más repeticiones del RE precedente, tantas repeticiones como sean posibles. `ab*` coincidirá con "a", "ab" o "a" seguido de cualquier número de "b".

`+`

Hace que la RE resultante coincida con 1 o más repeticiones de la RE precedente. `ab+` coincidirá con "a" seguido de cualquier número distinto de cero de "b"; no coincidirá solo con "a".

`?`

Hace que la RE resultante coincida con 0 o 1 repeticiones de la RE precedente. `ab?` coincidirá con "a" o "ab".

`*?`, `+?`, `??`

Los delimitadores «*», «+» y «*» son todos *greedy* (codiciosos); coinciden con la mayor cantidad de texto posible. A veces este comportamiento no es deseado; si el RE `<.*>` se utiliza para coincidir con `'<a> b <c>'`, coincidirá con toda la cadena, y no sólo con `'<a>'`. Añadiendo `?` después del delimitador hace que se realice la coincidencia de manera *non-greedy* o *minimal*; coincidirá la *mínima* cantidad de caracteres como sea posible. Usando el RE `<.*?>` sólo coincidirá con `'<a>'`.

`{m}`

Especifica que exactamente *m* copias de la RE anterior deben coincidir; menos coincidencias hacen que la RE entera no coincida. Por ejemplo, `a{6}` coincidirá exactamente con seis caracteres `'a'`, pero no con cinco.

`{m,n}`

Hace que el RE resultante coincida de *m* a *n* repeticiones del RE precedente, tratando de coincidir con el mayor número de repeticiones posible. Por ejemplo, `a{3,5}` coincidirá de 3 a 5 caracteres `'a'`. Omitiendo *m* se especifica un límite inferior de cero, y omitiendo *n* se especifica un límite superior infinito. Por ejemplo, `a{4,}b` coincidirá con `'aaaab'` o mil caracteres `'a'` seguidos de una `'b'`, pero no `'aaab'`. La coma no puede ser omitida o el modificador se confundiría con la forma descrita anteriormente.

`{m,n}?`

Hace que el RE resultante coincida de *m* a *n* repeticiones del RE precedente, tratando de coincidir con el *mínimo de* repeticiones posible. Esta es la versión *non-greedy* (no codiciosa) del delimitador anterior. Por ejemplo, en la cadena de 6 caracteres `'aaaaaaa'`, `a{3,5}` coincidirá con 5 caracteres `'a'`, mientras que `a{3,5}?` solo coincidirá con 3 caracteres.

`\`

O bien se escapan a los caracteres especiales (lo que le permite hacer coincidir caracteres como `'*'`, `'?'`, y así sucesivamente), o se señala una secuencia especial; las secuencias especiales se explican más adelante.

Si no se utiliza una cadena *raw* para expresar el patrón, recuerde que Python también utiliza la barra inversa como secuencia de escape en los literales de la cadena; si el analizador sintáctico de Python no reconoce la secuencia de escape, la barra inversa y el carácter subsiguiente se incluyen en la cadena resultante. Sin embargo, si Python quisiera reconocer la secuencia resultante, la barra inversa debería repetirse dos veces. Esto es complicado y difícil de entender, por lo que se recomienda encarecidamente utilizar cadenas *raw* para todas las expresiones salvo las más simples.

`[]`

Se utiliza para indicar un conjunto de caracteres. En un conjunto:

- Los caracteres pueden ser listados individualmente, ej. `[amk]` coincidirá con `'a'`, `'m'`, o `'k'`.

- Los rangos de caracteres se pueden indicar mediante dos caracteres y separándolos con un `'-'`. Por ejemplo, `[a-z]` coincidirá con cualquier letra ASCII en minúscula, `[0-5][0-9]` coincidirá con todos los números de dos dígitos desde el `00` hasta el `59`, y `[0-9A-Fa-f]` coincidirá con cualquier dígito hexadecimal. Si se escapa `-` (por ejemplo, `[a\-z]`) o si se coloca como el primer o el último carácter (por ejemplo, `[-a]` o `[a-]`), coincidirá con un literal `'-'`.
- Los caracteres especiales pierden su significado especial dentro de los sets. Por ejemplo, `[(+*)]` coincidirá con cualquiera de los caracteres literales `'('`, `'+'`, `'*'`, o `')'`.

- Las clases de caracteres como `\w` o `\S` (definidas más adelante) también se aceptan dentro de un conjunto, aunque los caracteres que coinciden dependen de si el modo `ASCII` o `LOCALE` está activo.

- Los caracteres que no están dentro de un rango pueden ser coincidentes con *complementing* el conjunto. Si el primer carácter del conjunto es `'^'`, todos los caracteres que *no* están en el conjunto coincidirán. Por ejemplo, `[^5]` coincidirá con cualquier carácter excepto con `'5'`, y `[^^]` coincidirá con cualquier carácter excepto con `'^'`. `^` no tiene un significado especial si no es el primer carácter del conjunto.
- Para coincidir con un `']'` literal dentro de un set, se debe preceder con una barra inversa, o colocarlo al principio del set. Por ejemplo, tanto `[()[\]{}]` como `[]()[{}]` coincidirá con los paréntesis, corchetes y llaves.

- El soporte de conjuntos anidados y operaciones de conjuntos como en Unicode Technical Standard #18 podría ser añadido en el futuro. Esto cambiaría la sintaxis, así que por el momento se planteará un `FutureWarning` en casos ambiguos para facilitar este cambio. Ello incluye conjuntos que empiecen con un literal `'['` o que contengan secuencias de caracteres literales `'--'`, `'&&'`, `'~~'` y `'||'`. Para evitar una advertencia, utilizar el código de escape con una barra inversa.

Distinto en la versión 3.7: `FutureWarning` se genera si un conjunto de caracteres contiene construcciones que cambiarán semánticamente en el futuro.

`|`

`A|B`, donde *A* y *B* pueden ser RE arbitrarias, crea una expresión regular que coincidirá con *A* or *B*. Un número arbitrario de RE puede ser separado por `'|'` de esta manera. Esto puede también ser usado dentro de grupos (ver más adelante). Cuando la cadena de destino es procesada, los RE separados por `'|'` son probados de izquierda a derecha. Cuando un patrón coincide completamente, esa rama es aceptada. Esto significa que una vez que *A* coincida, *B* no se comprobará más, incluso si se produce una coincidencia general más larga. En otras palabras, el operador de `'|'` nunca es codicioso. Para emparejar un literal `'|'`, se usa `\|`, o se envuelve dentro de una clase de caracteres, como en `[|]`.

`(...)`
Coincide con cualquier expresión regular que esté dentro de los paréntesis, e indica el comienzo y el final de un grupo; el contenido de un grupo puede ser recuperado después de que se haya realizado una coincidencia, y puede coincidir más adelante en la cadena con la secuencia especial `\number`, que se describe más adelante. Para hacer coincidir los literales `` `'('`` o `')'`, se usa `\(` o `\)`, o se envuelve dentro de una clase de caracteres: `[(]`, `[)]`.

`(?...)`
Esta es una notación de extensión (un `'?'` después de un `'('` no tiene ningún otro significado). El primer carácter después de `'?'` determina el significado y la sintaxis de la construcción. Las extensiones normalmente no crean un nuevo grupo; `(?P<name>…)` es la única excepción a esta regla. A continuación se muestran las extensiones actualmente soportadas.

`(?aiLmsux)`
(Una o más letras del conjunto `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`.) El grupo coincide con la cadena vacía; las letras ponen los indicadores correspondientes: `re.A` (coincidencia sólo en ASCII), `re.I` (ignorar mayúsculas o minúsculas), `re. L` (dependiente de la configuración regional), `re.M` (multilínea), `re.S` (el punto coincide con todo), `re.U` (coincidencia con Unicode), y `re.X` (modo *verbose*), para toda la expresión regular. (Los indicadores se describen en Contenidos del módulo.) Esto es útil si se desea incluir los indicadores como parte de la expresión regular, en lugar de pasar un argumento *flag* (indicador) a la función `re.compile()`. Los indicadores deben ser usados primero en la cadena de expresión.

`(?:...)`
Una versión no capturable de los paréntesis regulares. Hace coincidir cualquier expresión regular que esté dentro de los paréntesis, pero la subcadena coincidente con el grupo *no puede* ser recuperada después de realizar una coincidencia o referenciada más adelante en el patrón.

`(?aiLmsux-imsx:...)`
(Cero o más letras del conjunto `'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'`, opcionalmente seguido de `'-'` seguido de una o más letras de `'i'`, `'m'`, `'s'`, `'x'`.) Las letras ponen o quitan los indicadores correspondientes: `re.A` (coincidencia sólo en ASCII), `re.I` (ignorar mayúsculas o minúsculas), `re. L` (dependiente de la configuración regional), `re.M` (multilínea), `re.S` (el punto coincide con todo), `re.U` (coincidencia con Unicode), y

`re.X` (modo *verbose*) para la parte de la expresión. (Los indicadores se describen en Contenidos del módulo.)

Las letras `'a'`, `'L'` y `'u'` se excluyen mutuamente cuando se usan como indicadores en línea, así que no pueden combinarse o ser seguidos por `'-'`. En cambio, cuando uno de ellos aparece en un grupo dentro de la línea, anula el modo de coincidencia en el grupo que lo rodea. En los patrones Unicode, `(?a:…)` cambia al modo de concordancia sólo en ASCII, y `(?u:…)` cambia al modo de concordancia Unicode (por defecto). En el patrón de bytes `(?L:…)` se cambia a una correspondencia en función de la configuración regional, y `(?a:…)` se cambia a una correspondencia sólo en ASCII (predeterminado). Esta anulación sólo tiene efecto para el grupo de línea restringida, y el modo de coincidencia original se restaura fuera del grupo.

Nuevo en la versión 3.6.

Distinto en la versión 3.7: Las letras `'a'`, `'L'` y `'u'` también pueden ser usadas en un grupo.

`(?P<name>...)`

Similar a los paréntesis regulares, pero la subcadena coincidente con el grupo es accesible a través del nombre simbólico del grupo, *name* . Los nombres de grupo deben ser identificadores válidos de Python, y cada nombre de grupo debe ser definido sólo una vez dentro de una expresión regular. Un grupo simbólico es también un grupo numerado, del mismo modo que si el grupo no tuviera nombre.

Los grupos con nombre pueden ser referenciados en tres contextos. Si el patrón es `(?P<quote>['"]).*?(?P=quote)` (es decir, hacer coincidir una cadena citada con comillas simples o dobles):

Contexto de la referencia al grupo *quote* (cita)	**Formas de hacer referencia**
en el mismo patrón en sí mismo	• `(?P=quote)` (como se muestra) • `\1`
cuando se procesa el objeto de la coincidencia *m*	• `m.group('quote')` • `m.end('quote')` (etc.)
en una cadena pasada al argumento *repl* de `re.sub()`	• `\g<quote>` • `\g<1>` • `\1`

`(?P=name)`

Una referencia inversa a un grupo nombrado; coincide con cualquier texto correspondido por el grupo anterior llamado *name*.

`(?#...)`

Un comentario; el contenido de los paréntesis es simplemente ignorado.

`(?=...)`

Coincide si … coincide con el siguiente patrón, pero no procesa nada de la cadena. Esto se llama una *lookahead assertion* (aserción de búsqueda anticipada). Por ejemplo, `Isaac (?=Asimov)` coincidirá con `'Isaac '` sólo si va seguido de `'Asimov'`.

`(?!...)`

Coincide si … no coincide con el siguiente. Esta es una *negative lookahead assertion* (aserción negativa de búsqueda anticipada). Por ejemplo, `Isaac (?!Asimov)` coincidirá con `'Isaac '` sólo si *no* es seguido por `'Asimov'`.

`(?<=...)`

Coincide si la posición actual en la cadena es precedida por una coincidencia para … que termina en la posición actual. Esto se llama una *positive lookbehind assertion* (aserciones positivas de búsqueda tardía). `(?<=abc)def` encontrará una coincidencia en `'abcdef'`, ya que la búsqueda tardía hará una copia de seguridad de 3 caracteres y comprobará si el patrón contenido coincide. El patrón contenido sólo debe coincidir con cadenas de alguna longitud fija, lo que significa que `abc` o `a|b` están permitidas, pero `a*` y `a{3,4}` no lo están. Hay que tener en cuenta que los patrones que empiezan con aserciones positivas de búsqueda tardía no coincidirán con el principio de la cadena que se está buscando; lo más probable es que se quiera usar la función `search()` en lugar de la función `match()`:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

Este ejemplo busca una palabra seguida de un guión:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

Distinto en la versión 3.5: Se añadió soporte a las referencias de grupo de longitud fija.

`(?<!...)`

Coincide si la posición actual en la cadena no está precedida por una coincidencia de «...». Esto se llama una *negative lookbehind assertion* (Aserciones negativas de búsqueda tardía). Similar a las aserciones positivas de búsqueda tardía, el patrón contenido sólo debe coincidir con cadenas de alguna longitud fija. Los patrones que empiezan con aserciones negativas pueden coincidir al principio de la cadena que se busca.

`(?(id/name)yes-pattern|no-pattern)`

Tratará de coincidir con el `yes-pattern` (con patrón) si el grupo con un *id* o *nombre* existe, y con el `no-pattern` (sin patrón) si no existe. El `no-pattern` es opcional y puede ser omitido. Por ejemplo, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` es un patrón de coincidencia de correo electrónico deficiente, ya que coincidirá con `'<user@host.com>'`

así como con `'user@host.com'`, pero no con `'<user@host.com'` ni con `'user@host.com>'`.

Las secuencias especiales consisten en `'\'` y un carácter de la lista que aparece más adelante. Si el carácter ordinario no es un dígito ASCII o una letra ASCII, entonces el RE resultante coincidirá con el segundo carácter. Por ejemplo, `\$` coincide con el carácter `'$'`.

`\number`

Coincide con el contenido del grupo del mismo número. Los grupos se numeran empezando por el 1. Por ejemplo, `(.+) \1` coincide con `'el el'` o `'55 55'`, pero no con `'elel'` (notar el espacio después del grupo). Esta secuencia especial sólo puede ser usada para hacer coincidir uno de los primeros 99 grupos. Si el primer dígito del *número* es 0, o el *número* tiene 3 dígitos octales, no se interpretará como una coincidencia de grupo, sino como el carácter con valor octal *número*. Dentro de los `'['` y `']'` de una clase de caracteres, todos los escapes numéricos son tratados como caracteres.

`\A`

Coincide sólo al principio de la cadena.

`\b`

Coincide con la cadena vacía, pero sólo al principio o al final de una palabra. Una palabra se define como una secuencia de caracteres de palabras. Notar que formalmente, `\b` se define como el límite entre un carácter `\w` y un carácter `\W` (o viceversa), o entre `\w` y el principio/fin de la cadena. Esto significa que `r'\bfoo\b'` coincide con `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` pero no `'foobar'` o `'foo3'`.

Por defecto, los alfanuméricos Unicode son los que se usan en los patrones Unicode, pero esto se puede cambiar usando el indicador `ASCII`. Los límites de las palabras están determinados por la configuración regional actual si se usa el indicador `LOCALE`. Dentro de un rango de caracteres, `\b` representa el carácter de retroceso (*backspace*), para compatibilidad con los literales de las cadenas de Python.

`\B`

Coincide con la cadena vacía, pero sólo cuando *no* está al principio o al final de una palabra. Esto significa que `r'py\B'` coincide con `'python'`, `'py3'`, `'py2'`, pero no con `'py'`, `'py.'` o `'py!'`. `\B` es justo lo opuesto a `\b`, por lo que los caracteres de las palabras en los patrones de Unicode son alfanuméricos o el subrayado, aunque esto puede ser cambiado usando el indicador `ASCII`. Los límites de las palabras están determinados por la configuración regional actual si se usa el indicador `LOCALE`.

`\d`

Para los patrones de Unicode (str):

Coincide con cualquier dígito decimal de Unicode (es decir, cualquier carácter de la categoría de caracteres de Unicode [Nd]). Esto incluye a `[0-9]`, y también muchos otros caracteres de dígitos. Si se usa el indicador `ASCII`, sólo coincide con `[0-9]`.

Para patrones de 8 bits (bytes):

Coincide con cualquier dígito decimal; esto equivale a `[0-9]`.

`\D`

Coincide con cualquier carácter que no sea un dígito decimal. Esto es lo opuesto a `\d`. Si se usa el indicador `ASCII` esto se convierte en el equivalente a `[^0-9]`.

`\s`

Para los patrones de Unicode (str):

Coincide con los caracteres de los espacios en blanco de Unicode (que incluye `[ \t\n\r\f\v]`, y también muchos otros caracteres, por ejemplo los espacios duros exigidos por las reglas tipográficas en muchos idiomas). Si se usa el indicador `ASCII`, sólo `[ \t\n\r\f\v]` coincide.

Para patrones de 8 bits (bytes):

Coincide con los caracteres considerados como espacios en blanco en el conjunto de caracteres ASCII, lo que equivale a `[ \t\n\r\f\v]`.

`\S`

Coincide con cualquier carácter que no sea un carácter de espacio en blanco. Esto es lo opuesto a `\s`. Si se usa el indicador `ASCII` se convierte en el equivalente a *[^ tnrfv]*.

`\w`

Para los patrones de Unicode (str):

Coincide con los caracteres de palabras de Unicode; esto incluye la mayoría de los caracteres que pueden formar parte de una palabra en cualquier idioma, así como los números y el guión bajo. Si se usa el indicador `ASCII`, sólo coincide con `[a-zA-Z0-9_]`.

Para patrones de 8 bits (bytes):

Coincide con los caracteres considerados alfanuméricos en el conjunto de caracteres ASCII; esto equivale a `[a-zA-Z0-9_]`. Si se usa el indicador `LOCALE`, coincide con los caracteres considerados alfanuméricos en la configuración regional actual y el guión bajo.

`\W`

Coincide con cualquier carácter que no sea un carácter de una palabra. Esto es lo opuesto a `\w`. Si se usa el indicador `ASCII` esto se convierte en el equivalente a `[^a-zA-Z0-9_]`. Si se usa el indicador `LOCALE`, coincide con los caracteres que no son ni alfanuméricos en la configuración regional actual ni con el guión bajo.

`\Z`

Coincide sólo el final de la cadena.

La mayoría de los escapes estándar soportados por los literales de la cadena de Python también son aceptados por el analizador de expresiones regulares:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(Notar que `\b` se usa para representar los límites de las palabras, y significa «retroceso» (*backspace*) sólo dentro de las clases de caracteres.)

Las secuencias de escape `'\u'`, `'\U'` y `'\N'` sólo se reconocen en los patrones Unicode. En los patrones de bytes son errores. Los escapes desconocidos de las letras ASCII se reservan para su uso posterior y se consideran errores.

Los escapes octales se incluyen en una forma limitada. Si el primer dígito es un 0, o si hay tres dígitos octales, se considera un escape octal. De lo contrario, es una referencia de grupo. En cuanto a los literales de cadena, los escapes octales siempre tienen como máximo tres dígitos de longitud.

Distinto en la versión 3.3: Se han añadido las secuencias de escape `'\u'` y `'\U'`.

Distinto en la versión 3.6: Los escapes desconocidos que consisten en `'\'` y una letra ASCII ahora son errores.

Distinto en la versión 3.8: Se añadió la secuencia de escape `'\N{name}'`. Como en los literales de cadena, se expande al carácter Unicode nombrado (por ej. `'\N{EM DASH}'`).

Contenidos del módulo

El módulo define varias funciones, constantes y una excepción. Algunas de las funciones son versiones simplificadas de los métodos completos de las expresiones regulares compiladas. La mayoría de las aplicaciones no triviales utilizan siempre la forma compilada.

Distinto en la versión 3.6: Ahora las constantes de indicadores son instancias de `RegexFlag`, que es una subclase de `enum.IntFlag`.

`re.`**`compile`**(*`pattern`*, *`flags=0`*)

Compila un patrón de expresión regular en un objeto de expresión regular, que puede ser usado para las coincidencias usando `match()`, `search()` y otros métodos, descritos más adelante.

El comportamiento de la expresión puede modificarse especificando un valor de *indicadores*. Los valores pueden ser cualquiera de las siguientes variables, combinadas usando el operador OR (el operador `|`).

La secuencia

```
prog = re.compile(pattern)
result = prog.match(string)
```

es equivalente a

```
result = re.match(pattern, string)
```

pero usando `re.compile()` y guardando el objeto resultante de la expresión regular para su reutilización es más eficiente cuando la expresión será usada varias veces en un solo

programa.

> **Nota:** Las versiones compiladas de los patrones más recientes pasaron a `re.compile()` y las funciones de coincidencia a nivel de módulo están en caché, así que los programas que usan sólo unas pocas expresiones regulares a la vez no tienen que preocuparse de compilar expresiones regulares.

`re.`**A**
`re.`**ASCII**

Hace que `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` y `\S` realicen una coincidencia ASCII en lugar de una concordancia Unicode. Esto sólo tiene sentido para los patrones de Unicode, y se ignora para los patrones de bytes. Corresponde al indicador en línea `(?a)`.

Notar que para la compatibilidad con versiones anteriores, el indicador `re.U` todavía existe (así como su sinónimo `re.UNICODE` y su contraparte incrustada `(?u)`), pero estos son redundantes en Python 3 ya que las coincidencias son Unicode por defecto para las cadenas (y no se permite la coincidencia Unicode para los bytes).

`re.`**DEBUG**

Muestra información de depuración (*debug*) sobre la expresión compilada. No hay un indicador en línea que corresponda.

`re.`**I**
`re.`**IGNORECASE**

Realiza una coincidencia insensible a las mayúsculas y minúsculas; expresiones como `[A-Z]` también coincidirán con las minúsculas. La coincidencia completa de Unicode (como Ü coincidencia ü) también funciona a menos que el indicador `re.ASCII` se utilice para desactivar las coincidencias que no sean ASCII. La configuración regional vigente no cambia el efecto de este indicador a menos que también se use el indicador `re.LOCALE`. Corresponde al indicador en línea `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the `IGNORECASE` flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: “İ” (U+0130, Latin capital letter I with dot above), “ı” (U+0131, Latin small letter dotless i), “ſ” (U+017F, Latin small letter long s) and “K” (U+212A, Kelvin sign). If the `ASCII` flag is used, only letters “a” to “z” and “A” to “Z” are matched.

`re.`**L**
`re.`**LOCALE**

Hace que las coincidencias `\w`, `\W`, `\b`, `\B` y las coincidencias insensibles a mayúsculas y minúsculas dependan de la configuración regional vigente. Este indicador sólo puede ser usado con patrones de bytes. Se desaconseja su uso ya que el mecanismo de configuración regional no es fiable, sólo maneja una «cultura» a la vez, y sólo funciona con localizaciones de 8 bits. La coincidencia Unicode ya está activada por defecto en Python 3 para los patrones Unicode (str), y es capaz de manejar diferentes localizaciones/idiomas. Corresponde al indicador en línea `(?L)`.

Distinto en la versión 3.6: `re.LOCALE` sólo se puede usar con patrones de bytes y no es compatible con `re.ASCII`.

Distinto en la versión 3.7: Los objetos expresión regular compilados con el indicador `re.LOCALE` ya no dependen del lugar en el momento de la compilación. Sólo la configuración regional durante la coincidencia afecta al resultado obtenido.

`re.`**M**
`re.`**MULTILINE**

Cuando se especifica, el patrón de caracteres `'^'` coincide al principio de la cadena y al principio de cada línea (inmediatamente después de cada nueva línea); y el patrón de caracteres `'$'` coincide al final de la cadena y al final de cada línea (inmediatamente antes de cada nueva línea). Por defecto, `'^'` coincide sólo al principio de la cadena, y `'$'` sólo al final de la cadena e inmediatamente antes de la nueva línea (si la hay) al final de la cadena. Corresponde al indicador en línea `(?m)`.

`re.`**S**
`re.`**DOTALL**

Hace que el carácter especial `'.'` coincida con cualquier carácter, incluyendo una nueva línea. Sin este indicador, `'.'` coincidirá con cualquier cosa, *excepto* con una nueva línea. Corresponde al indicador en línea `(?s)`.

`re.`**X**
`re.`**VERBOSE**

Este indicador permite escribir expresiones regulares que se ven mejor y son más legibles al facilitar la separación visual de las secciones lógicas del patrón y añadir comentarios. Los espacios en blanco dentro del patrón se ignoran, excepto cuando están en una clase de caracteres, o cuando están precedidos por una barra inversa no escapada, o dentro de fichas como `*?`, `(?:` o `(?P<...>`. Cuando una línea contiene un `#` que no está en una clase de caracteres y no está precedida por una barra inversa no escapada, se ignoran todos los caracteres desde el más a la izquierda (como `#`) hasta el final de la línea.

Esto significa que los dos siguientes objetos expresión regular que coinciden con un número decimal son funcionalmente iguales:

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

Corresponde al indicador en línea `(?x)`.

`re.`**search**(*pattern*, *string*, *flags=0*)

Examina a través de la *string* («cadena») buscando el primer lugar donde el *pattern* («patrón») de la expresión regular produce una coincidencia, y retorna un objeto match correspondiente. Retorna `None` si ninguna posición en la cadena coincide con el patrón; notar que esto es diferente a encontrar una coincidencia de longitud cero en algún punto de la cadena.

re.**match**(*pattern*, *string*, *flags=0*)

Si cero o más caracteres al principio de la *string* («cadena») coinciden con el *pattern* («patrón») de la expresión regular, retorna un objeto match correspondiente. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Notar que incluso en el modo `MULTILINE`, `re.match()` sólo coincidirá al principio de la cadena y no al principio de cada línea.

Si se quiere localizar una coincidencia en cualquier lugar de la *string* («cadena»), se utiliza `search()` en su lugar (ver también search() vs. match()).

re.**fullmatch**(*pattern*, *string*, *flags=0*)

Si toda la *string* («cadena») coincide con el *pattern* («patrón») de la expresión regular, retorna un correspondiente objeto match. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Nuevo en la versión 3.4.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

Divide la *string* («cadena») por el número de ocurrencias del *pattern* («patrón»). Si se utilizan paréntesis de captura en *pattern*, entonces el texto de todos los grupos en el patrón también se retornan como parte de la lista resultante. Si *maxsplit* (máxima divisibilidad) es distinta de cero, como mucho se producen *maxsplit* divisiones, y el resto de la cadena se retorna como elemento final de la lista.

```
>>> 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.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

Si hay grupos de captura en el separador y coincide al principio de la cadena, el resultado comenzará con una cadena vacía. Lo mismo ocurre con el final de la cadena:

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

De esa manera, los componentes de los separadores se encuentran siempre en los mismos índices relativos dentro de la lista de resultados.

Las coincidencias vacías para el patrón dividen la cadena sólo cuando no están adyacentes a una coincidencia vacía anterior.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
```

```
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', ''
```

Distinto en la versión 3.1: Se añadió el argumento de los indicadores opcionales.

Distinto en la versión 3.7: Se añadió el soporte de la división en un patrón que podría coincidir con una cadena vacía.

re.**findall**(*pattern*, *string*, *flags=0*)

Retorna todas las coincidencias no superpuestas del *pattern* («patrón») en la *string* («cadena»), como una lista de cadenas. La cadena es examinada de izquierda a derecha, y las coincidencias son retornadas en el orden en que fueron encontradas. Si uno o más grupos están presentes en el patrón, retorna una lista de grupos; esta será una lista de tuplas si el patrón tiene más de un grupo. Las coincidencias vacías se incluyen en el resultado.

Distinto en la versión 3.7: Las coincidencias no vacías ahora pueden empezar justo después de una coincidencia vacía anterior.

re.**finditer**(*pattern*, *string*, *flags=0*)

Retorna un iterator que produce objetos de coincidencia sobre todas las coincidencias no superpuestas para *pattern* («patrón») de RE en la *string* («cadena»). La *string* es examinada de izquierda a derecha, y las coincidencias son retornadas en el orden en que se encuentran. Las coincidencias vacías se incluyen en el resultado.

Distinto en la versión 3.7: Las coincidencias no vacías ahora pueden empezar justo después de una coincidencia vacía anterior.

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Retorna la cadena obtenida reemplazando las ocurrencias no superpuestas del *pattern* («patrón») en la *string* («cadena») por el reemplazo de *repl*. Si el patrón no se encuentra, se retorna *string* sin cambios. *repl* puede ser una cadena o una función; si es una cadena, cualquier barra inversa escapada en ella es procesada. Es decir, `\n` se convierte en un carácter de una sola línea nueva, `\r` se convierte en un retorno de carro, y así sucesivamente. Los escapes desconocidos de las letras ASCII se reservan para un uso futuro y se tratan como errores. Otros escapes desconocidos como `\&` no se utilizan. Las referencias inversas, como `\6`, se reemplazan por la subcadena que corresponde al grupo 6 del patrón. Por ejemplo:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

Si *repl* es una función, se llama para cada ocurrencia no superpuesta de *pattern*. La función toma un solo argumento objeto match, y retorna la cadena de sustitución. Por ejemplo:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

El patrón puede ser una cadena o un objeto patrón.

El argumento opcional *count* («recuento») es el número máximo de ocurrencias de patrones a ser reemplazados; *count* debe ser un número entero no negativo. Si se omite o es cero, todas las ocurrencias serán reemplazadas. Las coincidencias vacías del patrón se reemplazan sólo cuando no están adyacentes a una coincidencia vacía anterior, así que `sub('x*', '-', 'abxd')` retorna `'-a-b--d-'`.

En los argumentos *repl* de tipo cadena, además de los escapes de caracteres y las referencias inversas descritas anteriormente, `\g<name>` usará la subcadena coincidente con el grupo llamado `name`, como se define en la sintaxis `(?P<name>...)`. `\g<number>` utiliza el número de grupo correspondiente; `\g<2>` es por lo tanto equivalente a `\2`, pero no es ambiguo en un reemplazo como sucede con `\g<2>0`. `\20` se interpretaría como una referencia al grupo 20, no como una referencia al grupo 2 seguido del carácter literal `'0'`. La referencia inversa `\g<0>` sustituye en toda la subcadena coincidente con la RE.

Distinto en la versión 3.1: Se añadió el argumento de los indicadores opcionales.

Distinto en la versión 3.5: Los grupos no coincidentes son reemplazados por una cadena vacía.

Distinto en la versión 3.6: Los escapes desconocidos en el *pattern* que consisten en `'\'` y una letra ASCII ahora son errores.

Distinto en la versión 3.7: Los escapes desconocidos en *repl* que consisten en `'\'` y una letra ASCII ahora son errores.

Distinto en la versión 3.7: Las coincidencias vacías para el patrón se reemplazan cuando están adyacentes a una coincidencia anterior no vacía.

`re.`**subn**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Realiza la misma operación que `sub()`, pero retorna una tupla `(new_string, number_of_subs_made)`.

Distinto en la versión 3.1: Se añadió el argumento de los indicadores opcionales.

Distinto en la versión 3.5: Los grupos no coincidentes son reemplazados por una cadena vacía.

`re.`**escape**(*pattern*)

Caracteres de escape especiales en *pattern* (» patrón»). Esto es útil si quieres hacer coincidir una cadena literal arbitraria que puede tener metacaracteres de expresión

regular en ella. Por ejemplo:

```
>>> print(re.escape('http://www.python.org'))
http://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

Esta función no debe usarse para la cadena de reemplazo en `sub()` y `subn()`, sólo deben escaparse las barras inversas. Por ejemplo:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

Distinto en la versión 3.3: El carácter de `'_'` ya no se escapa.

Distinto en la versión 3.7: Sólo se escapan los caracteres que pueden tener un significado especial en una expresión regular. Como resultado, `'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'` y `"`" ya no se escapan.

`re.`**purge**()

Despeja la caché de expresión regular.

exception `re.`**error**(*msg*, *pattern=None*, *pos=None*)

Excepción señalada cuando una cadena enviada a una de las funciones descritas aquí no es una expresión regular válida (por ejemplo, podría contener paréntesis no coincidentes) o cuando se produce algún otro error durante la compilación o la coincidencia. Nunca es un error si una cadena no contiene ninguna coincidencia para un patrón. La instancia de error tiene los siguientes atributos adicionales:

msg

El mensaje de error sin formato.

pattern

El patrón de expresión regular.

pos

El índice en *pattern* («patrón») donde la compilación falló (puede ser `None`).

lineno

La línea correspondiente a *pos* (puede ser `None`).

colno

La columna correspondiente a *pos* (puede ser `None`).

Distinto en la versión 3.5: Se añadieron atributos adicionales.

Objetos expresión regular

Los objetos expresión regular compilados soportan los siguientes métodos y atributos:

`Pattern.`**`search`**`(`*`string`*`[, `*`pos`*`[, `*`endpos`*`]])`

Escanea a través de la *string* («cadena») buscando la primera ubicación donde esta expresión regular produce una coincidencia, y retorna un objeto match correspondiente. Retorna `None` si ninguna posición en la cadena coincide con el patrón; notar que esto es diferente a encontrar una coincidencia de longitud cero en algún punto de la cadena.

El segundo parámetro opcional *pos* proporciona un índice en la cadena donde la búsqueda debe comenzar; por defecto es `0`. Esto no es completamente equivalente a dividir la cadena; el patrón de carácter `'^'` coincide en el inicio real de la cadena y en las posiciones justo después de una nueva línea, pero no necesariamente en el índice donde la búsqueda va a comenzar.

El parámetro opcional *endpos* limita hasta dónde se buscará la cadena; será como si la cadena fuera de *endpos* caracteres de largo, por lo que sólo se buscará una coincidencia entre los caracteres de *pos* a `endpos - 1`. Si *endpos* es menor que *pos*, no se encontrará ninguna coincidencia; de lo contrario, si *rx* es un objeto de expresión regular compilado, `rx.search(string, 0, 50)` es equivalente a `rx.search(string[:50], 0)`.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

`Pattern.`**`match`**`(`*`string`*`[, `*`pos`*`[, `*`endpos`*`]])`

Si cero o más caracteres en el *beginning* («comienzo») de la *string* («cadena») coinciden con esta expresión regular, retorna un objeto match correspondiente. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Los parámetros opcionales *pos* y *endpos* tienen el mismo significado que para el método `search()`.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "do
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog"
<re.Match object; span=(1, 2), match='o'>
```

Si se quiere encontrar una coincidencia en cualquier lugar de *string*, utilizar `search()` en su lugar (ver también search() vs. match()).

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

Si toda la *string* («cadena») coincide con esta expresión regular, retorna un objeto match correspondiente. Retorna `None` si la cadena no coincide con el patrón; notar que esto es diferente de una coincidencia de longitud cero.

Los parámetros opcionales *pos* y *endpos* tienen el mismo significado que para el método `search()`.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of
>>> pattern.fullmatch("ogre")     # No match as not the full string matche
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

Nuevo en la versión 3.4.

Pattern.**split**(*string*, *maxsplit=0*)

Idéntico a la función `split()`, usando el patrón compilado.

Pattern.**findall**(*string*[, *pos*[, *endpos*]])

Similar a la función `findall()`, usando el patrón compilado, pero también acepta parámetros opcionales *pos* y *endpos* que limitan la región de búsqueda como para `search()`.

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])

Similar a la función `finditer()`, usando el patrón compilado, pero también acepta parámetros opcionales *pos* y *endpos* que limitan la región de búsqueda como para `search()`.

Pattern.**sub**(*repl*, *string*, *count=0*)

Idéntico a la función `sub()`, usando el patrón compilado.

Pattern.**subn**(*repl*, *string*, *count=0*)

Idéntico a la función `subn()`, usando el patrón compilado.

Pattern.**flags**

Los indicadores regex de coincidencia. Esta es una combinación de los indicadores dados a `compile()`, cualquier indicador `(?...)` en línea en el patrón, y los indicadores implícitos como `UNICODE` si el patrón es una cadena de Unicode.

Pattern.**groups**

El número de grupos de captura en el patrón.

Pattern.**groupindex**

Un diccionario que mapea cualquier nombre de grupo simbólico definido por `(?P<id>)` para agrupar números. El diccionario está vacío si no se utilizaron grupos simbólicos en el patrón.

`Pattern.`**`pattern`**

La cadena de patrones a partir de la cual el objeto de patrón fue compilado.

Distinto en la versión 3.7: Se añadió el soporte de `copy.copy()` y `copy.deepcopy()`. Los objetos expresión regular compilados se consideran atómicos.

Objetos de coincidencia

Los objetos de coincidencia siempre tienen un valor booleano de `True` («Verdadero»). Ya que `match()` y `search()` retornan `None` cuando no hay coincidencia. Se puede probar si hubo una coincidencia con una simple declaración `if`:

```
match = re.search(pattern, string)
if match:
    process(match)
```

Los objetos de coincidencia admiten los siguientes métodos y atributos:

`Match.`**`expand`**(*`template`*)

Retorna la cadena obtenida al hacer la sustitución de la barra inversa en la cadena de la plantilla *template*, como se hace con el método `sub()`. Escapes como `\n` son convertidos a los caracteres apropiados, y las referencias inversas numéricas (`\1`, `\2`) y las referencias inversas con nombre (`\g<1>`, `\g<name>`) son reemplazadas por el contenido del grupo correspondiente.

Distinto en la versión 3.5: Los grupos no coincidentes son reemplazados por una cadena vacía.

`Match.`**`group`**([*`group1`*, `...`])

Retorna uno o más subgrupos de la coincidencia. Si hay un solo argumento, el resultado es una sola cadena; si hay múltiples argumentos, el resultado es una tupla con un elemento por argumento. Sin argumentos, *group1* tiene un valor por defecto de cero (se retorna la coincidencia completa). Si un argumento *groupN* es cero, el valor de retorno correspondiente es toda la cadena coincidente; si está en el rango inclusivo [1..99], es la cadena coincidente con el grupo correspondiente entre paréntesis. Si un número de grupo es negativo o mayor que el número de grupos definidos en el patrón, se produce una excepción `IndexError`. Si un grupo está contenido en una parte del patrón que no coincidió, el resultado correspondiente es `None`. Si un grupo está contenido en una parte del patrón que coincidió varias veces, se retorna la última coincidencia.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

Si la expresión regular usa la sintaxis `(?P<name>...)`, los argumentos *groupN* también pueden ser cadenas que identifican a los grupos por su nombre de grupo. Si un argumento de cadena no se usa como nombre de grupo en el patrón, se produce una excepción `IndexError`.

Un ejemplo moderadamente complicado:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reyno
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

Los grupos nombrados también pueden ser referidos por su índice:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

Si un grupo coincide varias veces, sólo se puede acceder a la última coincidencia:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

`Match.`**`__getitem__`**(*g*)

Esto es idéntico a `m.group(g)`. Esto permite un acceso más fácil a un grupo individual de una coincidencia:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Nuevo en la versión 3.6.

`Match.`**`groups`**(*`default=None`*)

Retorna una tupla que contenga todos los subgrupos de la coincidencia, desde 1 hasta tantos grupos como haya en el patrón. El argumento *default* («por defecto») se utiliza para los grupos que no participaron en la coincidencia; por defecto es `None`.

Por ejemplo:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

Si hacemos que el decimal y todo lo que sigue sea opcional, no todos los grupos podrían participar en la coincidencia. Estos grupos serán por defecto `None` a menos que se utilice el argumento *default*:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

`Match.`**`groupdict`**(*`default=None`*)

Retorna un diccionario que contiene todos los subgrupos *nombrados* de la coincidencia, tecleado por el nombre del subgrupo. El argumento *por defecto* se usa para los grupos que no participaron en la coincidencia; por defecto es `None`. Por ejemplo:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

`Match.`**`start`**([*`group`*])

`Match.`**`end`**([*`group`*])

Retorna los índices del comienzo y el final de la subcadena coincidiendo con el *group*; el *group* por defecto es cero (es decir, toda la subcadena coincidente). Retorna `-1` si *grupo* existe pero no ha contribuido a la coincidencia. Para un objeto coincidente *m*, y un grupo *g* que sí contribuyó a la coincidencia, la subcadena coincidente con el grupo *g* (equivalente a `m.group(g)`) es

```
m.string[m.start(g):m.end(g)]
```

Notar que `m.start(group)` será igual a `m.end(group)` si *group* coincidió con una cadena nula. Por ejemplo, después de `m = re.search('b(c?)', 'cba')`, `m.start(0)` es 1, `m.end(0)` es 2, `m.start(1)` y `m.end(1)` son ambos 2, y `m.start(2)` produce una excepción `IndexError`.

Un ejemplo que eliminará *remove_this* («quita esto») de las direcciones de correo electrónico:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

`Match.`**`span`**([*`group`*])

Para una coincidencia *m*, retorna la tupla-2 `(m.inicio(grupo), m.fin(grupo))`. Notar que si *group* no contribuyó a la coincidencia, esto es `(-1, -1)`. *group* por se convierte a cero para toda la coincidencia.

`Match.`**`pos`**

El valor de *pos* que fue pasado al método `search()` o `match()` de un objeto regex. Este es el índice de la cadena en la que el motor RE comenzó a buscar una coincidencia.

`Match.`**`endpos`**

El valor de *endpos* que se pasó al método `search()` o `match()` de un objeto regex. Este es el índice de la cadena más allá de la cual el motor RE no irá.

`Match.`**`lastindex`**

El índice entero del último grupo de captura coincidente, o``None`` si no hay ningún grupo coincidente. Por ejemplo, las expresiones `(a)b`, `((a)(b))` y `((ab))` tendrán `lastindex == 1` si se aplican a la cadena `'ab'`, mientras que la expresión `(a)(b)` tendrá `lastindex == 2`, si se aplica a la misma cadena.

`Match.`**`lastgroup`**

El nombre del último grupo capturador coincidente, o``None`` si el grupo no tenía nombre, o si no había ningún grupo coincidente.

`Match.`**`re`**

El objeto de expresión regular cuyo método `match()` o `search()` produce esta instancia de coincidencia.

`Match.`**`string`**

La cadena pasada a `match()` o `search()`.

Distinto en la versión 3.7: Se añadió el soporte de `copy.copy()` y `copy.deepcopy()`. Los objetos de coincidencia se consideran atómicos.

Ejemplos de expresiones regulares

Buscando un par

En este ejemplo, se utilizará la siguiente función de ayuda para mostrar los objetos de coincidencia con un poco más de elegancia:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

Supongamos que se está escribiendo un programa de póquer en el que la mano de un jugador se representa como una cadena de 5 caracteres en la que cada carácter representa una carta, «a» para el as, «k» para el rey, «q» para la reina, «j» para la jota, «t» para el 10, y del » 2» al «9» representando la carta con ese valor.

Para ver si una cadena dada es una mano válida, se podría hacer lo siguiente:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
```

```
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

Esa última mano, `"727ak"`, contenía un par, o dos de las mismas cartas de valor. Para igualar esto con una expresión regular, se podrían usar referencias inversas como tales:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

Para averiguar en qué carta consiste el par, se podría utilizar el método `group()` del objeto de coincidencia de la siguiente manera:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

Simular scanf()

Python no tiene actualmente un equivalente a `scanf()`. Las expresiones regulares son generalmente más poderosas, aunque también más verbosas, que las cadenas de formato `scanf()`. La tabla siguiente ofrece algunos mapeos más o menos equivalentes entre tokens de formato `scanf()` y expresiones regulares.

Token `scanf()`	**Expresión regular**
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e`, `%E`, `%f`, `%g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`

Token `scanf()`	**Expresión regular**
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

Para extraer el nombre de archivo y los números de una cadena como

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

se usaría un formato `scanf()` como

```
%s - %d errors, %d warnings
```

La expresión regular equivalente sería

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python ofrece dos operaciones primitivas diferentes basadas en expresiones regulares: `re.match()` comprueba si hay una coincidencia sólo al principio de la cadena, mientras que `re.search()` comprueba si hay una coincidencia en cualquier parte de la cadena (esto es lo que hace Perl por defecto).

Por ejemplo:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
```

Las expresiones regulares que comienzan con `'^'` pueden ser usadas con `search()` para restringir la coincidencia al principio de la cadena:

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

Notar, sin embargo, que en el modo `MULTILINE` `match()` sólo coincide al principio de la cadena, mientras que usando `search()` con una expresión regular que comienza con `'^'` coincidirá al principio de cada línea.

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

Haciendo una guía telefónica

`split()` divide una cadena en una lista delimitada por el patrón recibido. El método es muy útil para convertir datos textuales en estructuras de datos que pueden ser fácilmente leídas y modificadas por Python, como se demuestra en el siguiente ejemplo en el que se crea una guía telefónica.

Primero, aquí está la información. Normalmente puede venir de un archivo, aquí se usa la sintaxis de cadena de triple comilla

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

Las entradas (*entries*) están separadas por una o más líneas nuevas. Ahora se convierte la cadena en una lista en la que cada línea no vacía tiene su propia entrada:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finalmente, se divide cada entrada en una lista con nombre, apellido, número de teléfono y dirección. Se utiliza el parámetro `maxsplit` (división máxima) de `split()` porque la dirección tiene espacios dentro del patrón de división:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

El patrón `:?` coincide con los dos puntos después del apellido, de manera que no aparezca en la lista de resultados. Con `maxsplit` de `4`, se podría separar el número de casa del nombre de la calle:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

Mungear texto

`sub()` reemplaza cada ocurrencia de un patrón con una cadena o el resultado de una función. Este ejemplo demuestra el uso de `sub()` con una función para «mungear» (*munge*)

el texto, o aleatorizar el orden de todos los caracteres en cada palabra de una frase excepto el primer y último carácter:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

Encontrar todos los adverbios

`findall()` coincide con *todas* las ocurrencias de un patrón, no sólo con la primera, como lo hace `search()`. Por ejemplo, si un escritor quisiera encontrar todos los adverbios en algún texto, podría usar `findall()` de la siguiente manera:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

Encontrar todos los adverbios y sus posiciones

Si uno quiere más información sobre todas las coincidencias de un patrón en lugar del texto coincidente, `finditer()` es útil ya que proporciona objetos de coincidencia en lugar de cadenas. Continuando con el ejemplo anterior, si un escritor quisiera encontrar todos los adverbios *y sus posiciones* en algún texto, usaría `finditer()` de la siguiente manera:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

Notación de cadena *raw*

La notación de cadena *raw* (`r "text"`) permite escribir expresiones regulares razonables. Sin ella, para «escapar» cada barra inversa (`'\'`) en una expresión regular tendría que ser precedida por otra. Por ejemplo, las dos siguientes líneas de código son funcionalmente idénticas:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

Cuando uno quiere igualar una barra inversa literal, debe escaparse en la expresión regular. Con la notación de cadena *raw*, esto significa `r"\\"`. Sin la notación de cadena, uno debe usar `"\\\\"`, haciendo que las siguientes líneas de código sean funcionalmente idénticas:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

Escribir un Tokenizador

Un tokenizador o analizador léxico analiza una cadena para categorizar grupos de caracteres. Este es un primer paso útil para escribir un compilador o intérprete.

Las categorías de texto se especifican con expresiones regulares. La técnica consiste en combinarlas en una única expresión regular maestra y en hacer un bucle sobre las sucesivas coincidencias:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
```

```
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

El tokenizador produce el siguiente resultado:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

[Frie09] Friedl, Jeffrey. *Mastering Regular Expressions*. 3a ed., O'Reilly Media, 2009. La tercera edición del libro ya no abarca a Python en absoluto, pero la primera edición cubría la escritura de buenos patrones de expresiones regulares con gran detalle.