EBook Gratis

APRENDIZAJE Regular Expressions

Free unaffiliated eBook created from
Stack Overflow contributors.

#regex

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: regular-expressions

It is an unofficial and free Regular Expressions ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Regular Expressions.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Expresiones Regulares

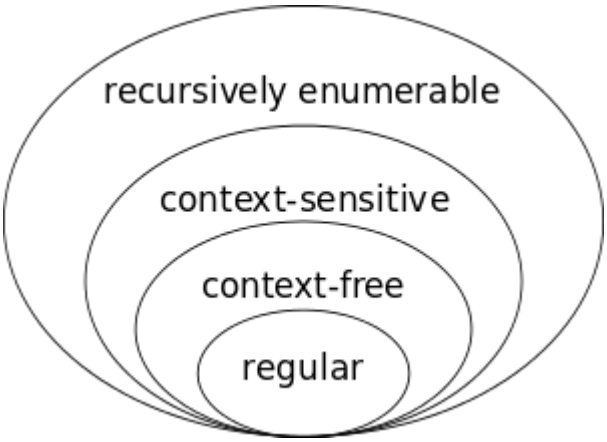
Observaciones

Para muchos programadores, la *expresión regular* es una especie de espada mágica que lanzan para resolver cualquier tipo de situación de análisis de texto. Pero esta herramienta no es nada mágico, y aunque es muy bueno en lo que hace, no es un lenguaje de programación con todas las funciones *(es decir,* **no** es Turing completo).

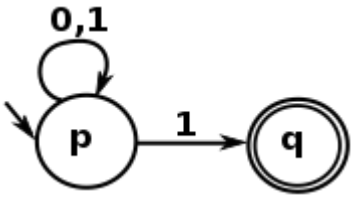
¿Qué significa 'expresión regular'?

Las expresiones regulares expresan un lenguaje definido por una *gramática regular* que se puede resolver con un *autómata finito no determinista* (NFA), donde la coincidencia está representada por los estados.

Una *gramática regular* es la *gramática* más simple expresada por la *Jerarquía de Chomsky* .



En pocas palabras, un lenguaje regular se expresa visualmente por lo que puede expresar una NFA, y aquí hay un ejemplo muy simple de NFA:



Y el lenguaje de *Expresión Regular* es una representación textual de tal autómata. Ese último ejemplo es expresado por el siguiente regex:

```
^[01]*1$
```

Que coincide con cualquier cadena que comience con `0` o `1` , repitiendo 0 o más veces, que

termine con `1` . En otras palabras, es una expresión regular para hacer coincidir los números impares de su representación binaria.

¿Son todas las expresiones *regulares* una gramática *regular* ?

En realidad no lo son. Muchos motores de expresiones regulares han mejorado y están usando *autómatas de empuje hacia abajo* , que pueden acumularse y desplegar información a medida que se ejecuta. Esos autómatas definen lo que se llama gramáticas *libres de* contexto en la Jerarquía de Chomsky. El uso más típico de aquellos en *expresiones* regulares no regulares, es el uso de un patrón recursivo para la comparación de paréntesis.

Una expresión regular recursiva como la siguiente (que coincide con paréntesis) es un ejemplo de tal implementación:

```
{((?>[^\(\)]+|(?R))*)}
```

(este ejemplo no funciona con el motor de `re` de Python, sino con el motor de `regex` o con el motor de PCRE).

Recursos

Para obtener más información sobre la teoría detrás de las expresiones regulares, puede consultar los siguientes cursos disponibles en el MIT:

- Autómatas, computabilidad y complejidad.
- Expresiones regulares y gramáticas
- Especificar idiomas con expresiones regulares y gramáticas libres de contexto

Cuando está escribiendo o depurando una expresión regular compleja, existen herramientas en línea que pueden ayudar a visualizar las expresiones regulares como autómatas, como el sitio de debuggex .

Versiones

PCRE

Versión	Publicado
2	2015-01-05
1	1997-06-01

Utilizado por: PHP 4.2.0 (y superior), Delphi XE (y superior), Julia , Notepad ++

Perl

Versión	Publicado
1	1987-12-18
2	1988-06-05
3	1989-10-18
4	1991-03-21
5	1994-10-17
6	2009-07-28

.RED

Versión	Publicado
1	2002-02-13
4	2010-04-12

Idiomas: C #

Java

Versión	Publicado
4	2002-02-06
5	2004-10-04
7	2011-07-07
SE8	2014-03-18

JavaScript

Versión	Publicado
1.2	1997-06-11

Versión	Publicado
1.8.5	2010-07-27

Pitón

Versión	Publicado
1.4	1996-10-25
2.0	2000-10-16
3.0	2008-12-03
3.5.2	2016-06-07

Oniguruma

Versión	Publicado
Inicial	2002-02-25
5.9.6	2014-12-12
Onigmo	2015-01-20

Aumentar

Versión	Publicado
0	1999-12-14
1.61.0	2016-05-13

POSIX

Versión	Publicado
BRE	1997-01-01
ANTES DE	2008-01-01

Idiomas: Bash

Examples

Guia de personajes

Tenga en cuenta que algunos elementos de sintaxis tienen un comportamiento diferente según la expresión.

Sintaxis	Descripción
`?`	Haga coincidir el carácter o subexpresión anterior 0 o 1 veces. También se usa para grupos que no capturan, y grupos de captura nombrados.
`*`	Coincidir con el carácter o subexpresión anterior 0 o más veces.
`+`	Coincidir con el carácter o subexpresión anterior 1 o más veces.
`{n}`	Haga coincidir el carácter o subexpresión precedente exactamente *n* veces.
`{min,}`	Coincidir con el carácter anterior o subexpresión *mínimo* o más veces.
`{,max}`	Coincidir con el carácter anterior o subexpresión *máximo* o menos veces.
`{min,max}`	Haga coincidir el carácter o subexpresión precedente al menos un *mínimo de* veces, pero no más de un `max` veces.
`-`	Cuando se incluye entre corchetes indica `to` ; por ejemplo, [3-6] coincide con los caracteres 3, 4, 5 o 6.
`^`	Inicio de cadena (o inicio de línea si se especifica la opción multilínea `/m`), o niega una lista de opciones (es decir, si se encuentra entre corchetes `[]`)
`$`	Fin de cadena (o final de una línea si se especifica la opción multilínea `/m`).
`( ... )`	Subexpresiones de grupos, captura contenido coincidente en variables especiales (`\1` , `\2` , etc.) que se pueden usar más adelante dentro de la misma expresión regular, por ejemplo `(\w+)\s\1\s` coincide con la repetición de palabras
`(?<name> ... )`	Subexpresiones de grupos, y las captura en un grupo nombrado
`(?: ... )`	Subexpresiones de grupos sin captura.
`.`	Coincide con cualquier carácter, excepto los saltos de línea (`\n` , y normalmente `\r`).
`[ ... ]`	Cualquier carácter entre estos paréntesis debe coincidir una vez. NB: `^` siguiendo el corchete abierto niega este efecto. `-` aparecer dentro de los corchetes, se puede especificar un rango de valores (a menos que sea el primer o último carácter, en cuyo caso solo representa un guión normal).

Sintaxis	Descripción
`\`	Escapa del siguiente personaje. También se usa en secuencias meta - toques de expresión regular con significado especial.
`\$`	dólar (es decir, un personaje especial escapado)
`\(`	paréntesis abiertos (es decir, un carácter especial escapado)
`\)`	paréntesis de cierre (es decir, un carácter especial escapado)
`\*`	asterisco (es decir, un carácter especial escapado)
`\.`	punto (es decir, un carácter especial escapado)
`\?`	signo de interrogación (es decir, un carácter especial escapado)
`\[`	corchete izquierdo (abierto) (es decir, un carácter especial escapado)
`\\`	barra invertida (es decir, un carácter especial escapado)
`\]`	corchete derecho (cerrado) (es decir, un carácter especial escapado)
`\^`	caret (es decir, un personaje especial escapado)
`\{`	corchete / soporte izquierdo (abierto) (es decir, un carácter especial escapado)
`|`	pipa (es decir, un personaje especial escapado)
`\}`	corchete / soporte derecho (cerrado) (es decir, un carácter especial escapado)
`\+`	más (es decir, un personaje especial escapado)
`\A`	comienzo de una cuerda
`\Z`	final de una cadena
`\z`	absoluto de una cuerda
`\b`	palabra (secuencia alfanumérica) límite
`\1` , `\2` , etc.	las referencias anteriores a subexpresiones coincidentes anteriormente, agrupadas por `()` , `\1` significa la primera coincidencia, `\2` significa la segunda coincidencia, etc.
`[\b]`	retroceso: cuando `\b` está dentro de una clase de caracteres (`[]`) coincide con retroceso
`\B`	negated `\b` - coincide en cualquier posición entre caracteres de dos palabras así como en cualquier posición entre dos caracteres que no sean palabras
`\D`	sin dígitos

Sintaxis	Descripción
`\d`	dígito
`\e`	escapar
`\f`	form feed
`\n`	linea de alimentación
`\r`	retorno de carro
`\S`	espacio no blanco
`\s`	espacio en blanco
`\t`	lengüeta
`\v`	pestaña vertical
`\W`	no palabra
`\w`	palabra (es decir, carácter alfanumérico)
`{ ... }`	conjunto de caracteres con nombre
`|`	o; Es decir, delinea las opciones anteriores y anteriores.

Lea Empezando con Expresiones Regulares en línea:
https://riptutorial.com/es/regex/topic/259/empezando-con-expresiones-regulares

Capítulo 2: Agrupación atómica

Introducción

Los grupos regulares que no capturan permiten que el motor vuelva a ingresar al grupo e intente hacer coincidir algo diferente (como una alternancia diferente, o hacer coincidir menos caracteres cuando se usa un cuantificador).

Los grupos atómicos difieren de los grupos regulares que no capturan, ya que está prohibido el retroceso. Una vez que el grupo sale, toda la información de seguimiento se descarta, por lo que no se pueden intentar coincidencias alternativas.

Observaciones

Un cuantificador posesivo se comporta como un grupo atómico en el sentido de que el motor no podrá retroceder sobre un token o grupo.

Los siguientes son equivalentes en términos de funcionalidad, aunque algunos serán más rápidos que otros:

```
a*+abc
(?>a*)abc
(?:a+)*+abc
(?:a)*+abc
(?:a*)*+abc
(?:a*)++abc
```

Examples

Agrupando con (?>)

Usando un grupo atómico

Los grupos atómicos tienen el formato `(?>...)` con un `?>` Después del parén abierto.

Considere el siguiente texto de muestra:

```
ABC
```

La expresión regular intentará coincidir comenzando en la posición 0 del texto, que está antes de la `A` en `ABC` .

Si se utilizara `(?>a*)abc` expresión insensible a mayúsculas `(?>a*)abc` , `(?>a*)` coincidiría con 1 carácter `A` , dejando

```
BC
```

como el texto restante para que coincida. Se sale del grupo `(?>a*)` y se intenta `abc` en el texto restante, que no coincide.

El motor no puede retroceder en el grupo atómico, por lo que el paso actual falla. El motor se mueve a la siguiente posición en el texto, que estaría en la posición 1, que está después de la `A` y antes de la `B` de `ABC` .

La expresión regular `(?>a*)abc` se intenta de nuevo, y `(?>a*)` coincide con `A` 0 veces, dejando

```
BC
```

como el texto restante para que coincida. Se sale del grupo `(?>a*)` y se intenta `abc` , que falla.

Nuevamente, el motor no puede retroceder en el grupo atómico, por lo que el paso actual falla. La expresión regular continuará fallando hasta que todas las posiciones en el texto se hayan agotado.

Usando un grupo no atómico

Los grupos regulares que no capturan tienen el formato `(?:...)` con un `?:` Después del parén abierto.

Dado el mismo texto de ejemplo, pero con la expresión que no distingue entre mayúsculas y minúsculas `(?:a*)abc` , se producirá una coincidencia, ya que se permite que se produzca un retroceso.

Al principio, `(?:a*)` consumirá la letra `A` en el texto

```
ABC
```

dejando

```
BC
```

como el texto restante para que coincida. Se sale del grupo `(?:a*)` y se intenta `abc` en el texto restante, que no coincide.

El motor retrocede al grupo `(?:a*)` e intenta hacer coincidir 1 carácter menos: en lugar de hacer coincidir el carácter de 1 `A` , intenta hacer coincidir los caracteres de 0 `A` , y se abandona el grupo `(?:a*)` . Esto deja

```
ABC
```

como el texto restante para que coincida. El regex `abc` ahora puede coincidir exitosamente con el texto restante.

Otro ejemplo de texto

Considere este texto de ejemplo, con grupos atómicos y no atómicos (de nuevo, sin distinción de mayúsculas y minúsculas):

```
AAAABC
```

La expresión regular intentará coincidir comenzando en la posición 0 del texto, que está antes de la primera `A` en `AAAABC` .

El patrón que usa el grupo atómico `(?>a*)abc` **no** podrá coincidir, se comportará de manera casi idéntica al ejemplo `ABC` atómico anterior: los 4 caracteres `A` primero se comparan con `(?>a*)` (dejando a `BC` como el texto restante para coincidir), y `abc` no puede coincidir en ese texto. El grupo **no** puede volver a ingresarse, por lo que la coincidencia falla.

El patrón que usa el grupo no atómico `(?:a*)abc` **podrá** coincidir, comportándose de manera similar al ejemplo `ABC` no atómico anterior: todos los 4 caracteres `A` se comparan primero con `(?:a*)` (dejando `BC` como el texto restante para coincidir), y `abc` no puede coincidir en ese texto. El grupo **es** capaz de volver a introducir, por lo que uno menos `A` se intenta: 3 `A` los personajes se emparejan en lugar de 4 (dejando `ABC` como el resto del texto para que coincida), y `abc` es capaz de igualar con éxito en ese texto.

Lea Agrupación atómica en línea: https://riptutorial.com/es/regex/topic/8770/agrupacion-atomica

Capítulo 3: Caracteres del ancla: Dólar ($)

Observaciones

Una gran cantidad de motores de expresiones regulares utilizan un modo "multilínea" para buscar varias líneas en un archivo de forma independiente.

Por lo tanto, al usar `$` , estos motores coincidirán con los finales de todas las líneas. Sin embargo, los motores que no utilizan este tipo de modo multilínea solo coincidirán con la última posición de la cadena proporcionada para la búsqueda.

Examples

Une una letra al final de una línea o cadena

```
g$
```

Lo anterior coincide con una letra (la letra `g`) al final de una *cadena* en la mayoría de los motores de expresiones regulares (no en Oniguruma , donde `$` delimitador coincide con el final de una línea de forma predeterminada, y el modificador `m` (*MULTILINE*) se usa para hacer un `.` encontrado caracteres, incluyendo caracteres de salto de línea, como un modificador dotall en la mayoría de otros sabores de expresiones regulares NFA). El ancla `$` coincidirá con la primera aparición de una letra `g` antes del final de las siguientes cadenas:

En las siguientes oraciones, solo las letras en **negrita** coinciden:

> Las anclas son personajes que, de hecho, no coinciden con ningún carácter en un strin **g**
>
> Su objetivo es hacer coincidir una posición específica en esa cadena.
>
> Bob era helpin **g**
>
> ¡Pero su edición introdujo ejemplos que no coincidían!

En la mayoría de los tipos de expresiones regulares, `$` anchor también puede coincidir antes de un carácter de nueva línea o de salto de línea (secuencia), en un modo *MULTILINE* , donde `$` coincide al final de cada línea en lugar de solo al final de una cadena. Por ejemplo, usando `g$` como nuestra expresión regular de nuevo, en modo multilínea, los caracteres en cursiva en la siguiente cadena coincidirán:

```
tvxlt obofh necpu riist g\n aelxk zlhdx lyogu vcbke pzyay wtsea wbrju jztg\n drosf ywhed bykie lqmzg wgyhc lg\n qewrx ozrvm jwenx
```

Lea Caracteres del ancla: Dólar ($) en línea: https://riptutorial.com/es/regex/topic/1603/caracteres-del-ancla--dolar----

Capítulo 4: Clases de personajes

Observaciones

Clases simples

Regex	Partidos
`[abc]`	Cualquiera de los siguientes caracteres: `a` , `b` , o `c`
`[az]`	Cualquier personaje de `a` de `z` , **incluido** (esto se llama un *rango)*
`[0-9]`	Cualquier dígito de `0` a `9` , **inclusive**

Clases comunes

Algunos grupos / rangos de caracteres se usan tan a menudo, tienen abreviaturas especiales:

Regex	Partidos
`\w`	Caracteres alfanuméricos más el subrayado (también conocidos como "caracteres de palabras")
`\W`	Caracteres sin palabras (igual que `[^\w]`)
`\d`	Dígitos (*más anchos* que `[0-9]` ya que incluyen dígitos persas, indios, etc.)
`\D`	Sin dígitos (*más corto* que `[^0-9]` desde que rechazó los dígitos persas, los indios, etc.)
`\s`	Caracteres de espacios en blanco (espacios, tabulaciones, etc.) **Nota** : puede variar dependiendo de su motor / contexto
`\S`	Caracteres no en blanco

Clases de negacion

Una careta (^) después del corchete de apertura funciona como una negación de los personajes que lo siguen. Esto coincidirá con todos los caracteres que no están en la clase de caracteres.

Las clases de caracteres negados también coinciden con los caracteres de salto de línea, por lo tanto, si estos no deben coincidir, los caracteres de salto de línea específicos deben agregarse a

la clase (\ r y / o \ n).

Regex	Partidos
`[^AB]`	Cualquier personaje que **no** sea `A` y `B`
`[^\d]`	Cualquier carácter, **excepto los** dígitos.

Examples

Los basicos

Supongamos que tenemos una lista de equipos, llamada así: `Team A` , `Team B` , ..., `Team Z` Entonces:

- `Team [AB]` : Esto coincidirá con el `Team A` o el `Team B`
- `Team [^AB]` : Esto coincidirá con cualquier equipo, **excepto el** `Team A` o el `Team B`

A menudo necesitamos hacer coincidir los caracteres que "pertenecen" en algún contexto u otro (como las letras de la `A` a la `Z`), y para eso son las clases de caracteres.

Unir diferentes palabras similares

Considere la clase de caracteres `[aeiou]` . Esta clase de caracteres se puede usar en una expresión regular para hacer coincidir un conjunto de palabras escritas de manera similar.

`b[aeiou]t` coincide:

- murciélago
- apuesta
- poco
- larva del moscardón
- pero

No concuerda:

- combate
- btt
- bt

Las clases de personajes en su propia coinciden con un solo personaje a la vez.

Coincidencia no alfanumérica (clase de caracteres negados)

```
[^0-9a-zA-Z]
```

Esto coincidirá con todos los caracteres que no sean ni números ni letras (caracteres alfanuméricos). Si el carácter de subrayado _ también se va a negar, la expresión se puede

acortar a:

```
[^\w]
```

O:

```
\W
```

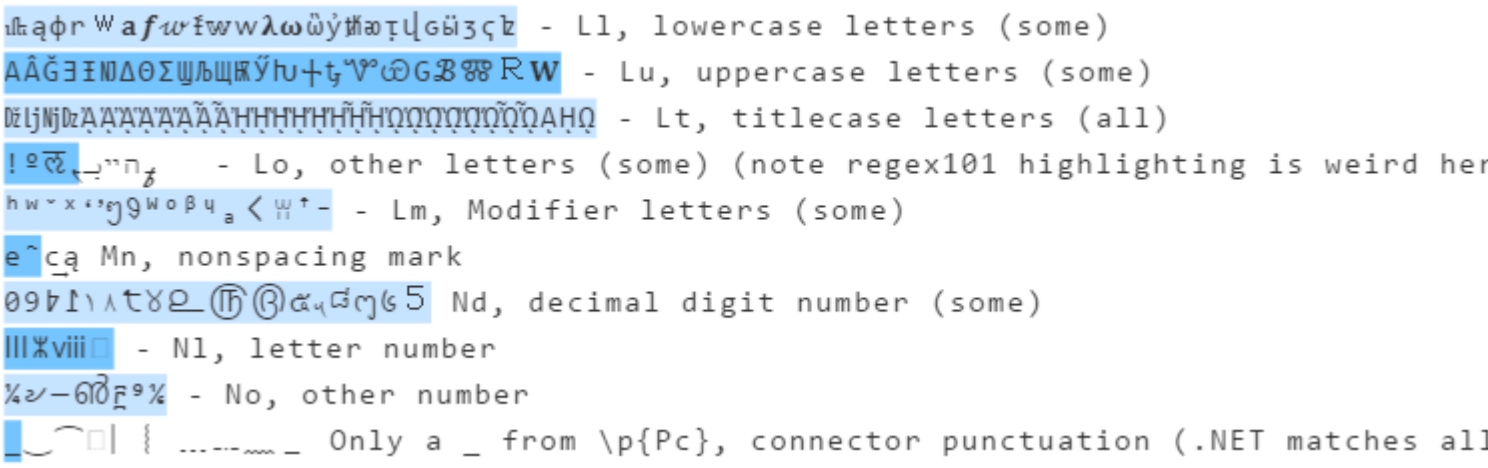
En las siguientes oraciones:

1. ¿Hola que tal?
2. No puedo esperar para el 2017 !!!

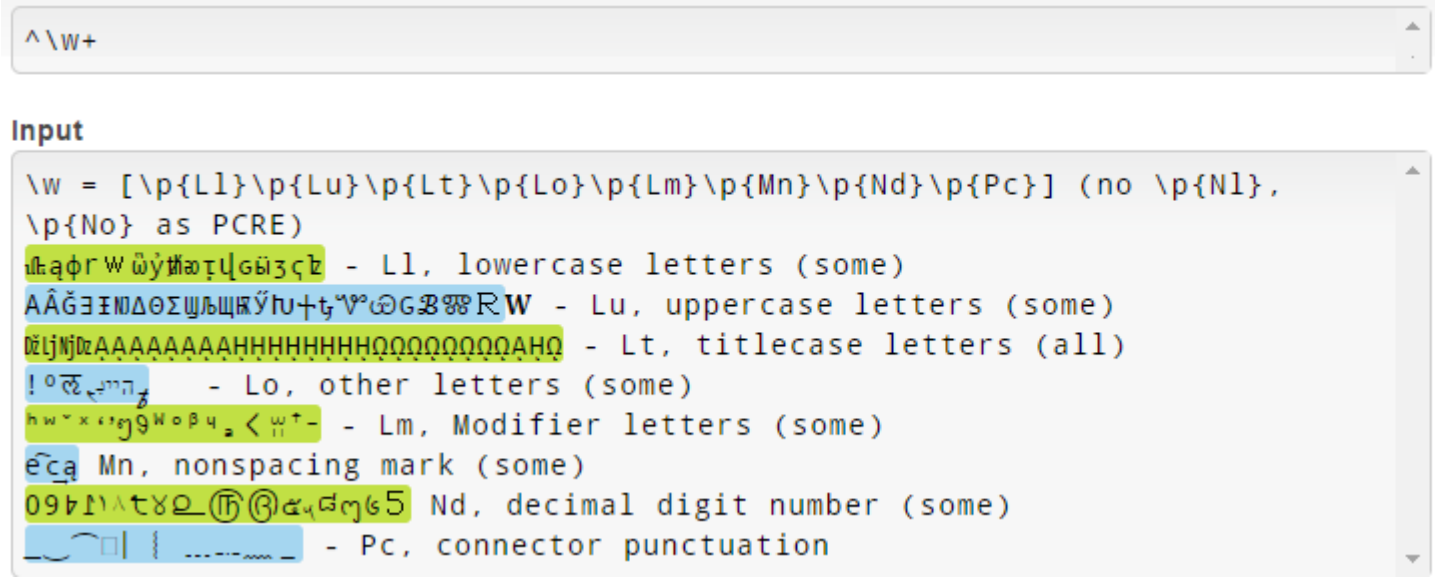
Los siguientes caracteres coinciden:

1. ¿ , , , ' , ? y el final del carácter de línea.
2. ' , , ! y el final del carácter de línea.

NOTA UNICODE
Tenga en cuenta que algunos tipos con compatibilidad de propiedades de caracteres Unicode pueden interpretar `\w` y `\W` como `[\p{L}\p{N}_]` y `[^\p{L}\p{N}_]` que significa otras letras Unicode y también se incluirán caracteres numéricos (ver documentos PCRE). Aquí hay una prueba de PCRE `\w` :

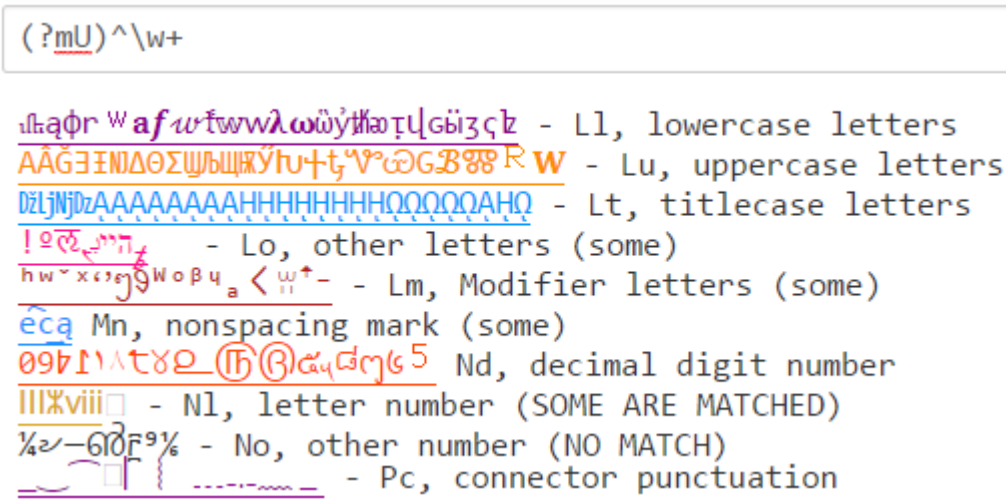


En .NET, `\w` = `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}]` , y tenga en cuenta que no coincide con `\p{Nl}` y `\p{No}` diferencia de PCRE (consulte la documentación de `\w` .NET):



Tenga en cuenta que, por alguna razón, Unicode 3.1 letras minúsculas (como ▯▯▯▯▯▯▯▯▯) no coinciden.

Java `(?U)\w` coincidirá con una combinación de lo que `\w` coincide en PCRE y .NET:



Coincidencia sin dígitos (clase de caracteres negados)

```
[^0-9]
```

Esto coincidirá con todos los caracteres que no sean dígitos ASCII.

Si también se van a negar los dígitos de Unicode, se puede usar la siguiente expresión, dependiendo de su configuración de sabor / idioma:

```
[^\d]
```

Esto se puede reducir a:

```
\D
```

Es posible que deba habilitar el soporte de propiedades de caracteres Unicode explícitamente mediante el uso del modificador `u` o mediante programación en algunos idiomas, pero esto puede no ser obvio. Para transmitir la intención explícitamente, se puede usar la siguiente construcción (cuando hay soporte disponible):

```
\P{N}
```

Lo que *por definición* significa: cualquier carácter que no sea un carácter numérico en ningún script. En un rango de caracteres negados, puedes usar:

```
[^\p{N}]
```

En las siguientes oraciones:

1. ¿Hola que tal?
2. No puedo esperar para el 2017 !!!

Los siguientes personajes serán emparejados:

1. `,` , , `'` , `?` , el carácter de fin de línea y todas las letras (minúsculas y mayúsculas).
2. `'` , , `!` , el carácter de fin de línea y todas las letras (minúsculas y mayúsculas).

Clase de personajes y problemas comunes que enfrentan los principiantes.

1. Clase de personaje

La clase de personaje se denota por `[]` . El contenido dentro de una clase de caracteres se trata como un `single character separately` . por ejemplo, supongamos que usamos

```
[12345]
```

En el ejemplo anterior, significa coincidencia `1 or 2 or 3 or 4 or 5` . En palabras simples, puede entenderse como `or condition for single characters` (**énfasis en un solo carácter**)

1.1 Palabra de precaución

- En la clase de caracteres, no hay concepto de hacer coincidir una cadena. Por lo tanto, si está utilizando regex `[cat]` , no significa que deba coincidir con la palabra `cat` literalmente, sino que debe coincidir con `c` o `a` o `t` . Este es un malentendido muy común que existe entre las personas que son más nuevas que las expresiones regulares.
- A veces la gente usa `|` (alternancia) dentro de la clase de personaje pensando que actuará como `OR condition` que es incorrecta. por ejemplo, utilizando `[a|b]` significa en realidad partido `a` o `|` (literalmente) o `b` .

2. Rango en la clase de personaje

El rango en la clase de caracteres se denota usando `-` signo. Supongamos que queremos

encontrar cualquier carácter dentro de los alfabetos en inglés de la `A` a la `Z` Esto se puede hacer usando la siguiente clase de caracteres

```
[A-Z]
```

Esto se podría hacer para cualquier rango ASCII o Unicode válido. Los rangos más utilizados incluyen `[AZ]` , `[az]` o `[0-9]` . Además, estos rangos se pueden combinar en la clase de caracteres como

```
[A-Za-z0-9]
```

Esto significa que coinciden con cualquier carácter en el rango `A to Z` o `a to z` o `0 to 9` . El pedido puede ser cualquier cosa. Por lo tanto, lo anterior es equivalente a `[a-zA-Z0-9]` siempre que el rango que defina sea correcto.

2.1 Palabra de precaución

- A veces, al escribir rangos para la `A` a la `Z` gente lo escribe como `[Az]` . Esto es incorrecto en la mayoría de los casos porque estamos usando `z` lugar de `Z` Por lo tanto, esto denota una coincidencia con cualquier carácter del rango ASCII `65` (de A) al `122` (de z), que incluye muchos caracteres no deseados después del rango ASCII `90` (de Z). **SIN EMBARGO** , `[Az]` se puede usar para hacer coincidir todas las letras `[a-zA-Z]` en expresiones regulares de estilo POSIX cuando se establece la intercalación para un idioma en particular. `[[ "ABCEDEF[]_abcdef" =~ ([Az]+) ]] && echo "${BASH_REMATCH[1]}"` en Cygwin con `LC_COLLATE="en_US.UTF-8"` produce `ABCEDF` . Si establece `LC_COLLATE` en `C` (en Cygwin, hecho con la `export`), le dará el `ABCEDEF[]_abcdef` esperado `ABCEDEF[]_abcdef` .

- Significado de `-` dentro de la clase de carácter es especial. Denota el rango como se explicó anteriormente. *¿Y si queremos emparejar `-` personaje literalmente?* No podemos ponerlo en ninguna parte, de lo contrario, denotará rangos si se coloca entre dos caracteres. En ese caso, tenemos que poner `-` en el inicio de la clase de caracteres como `[-AZ]` o en el final de la clase de caracteres como `[AZ-]` o `escape it` si quieres usarlo en el medio como `[AZ\-az]` .

3. Clase de personaje negado

La clase de caracteres negados se denota por `[^..]` . El signo de careta `^` denota coincidir con cualquier carácter excepto el presente en la clase de carácter. p.ej

```
[^cat]
```

significa emparejar cualquier carácter excepto `c` o `a` o `t` .

3.1 Palabra de precaución

- El significado de caret sign `^` asigna a la negación solo si está en el inicio de la clase de carácter. Si se encuentra en cualquier otro lugar de la clase de caracteres, se trata como un carácter de intercalación literal sin ningún significado especial.
- Algunas personas escriben expresiones regulares como `[^]` . En la mayoría de los motores

regex, esto da un error. La razón es que cuando está usando `^` en la posición inicial, se espera que al menos un carácter deba ser negado. Sin embargo, en *JavaScript* , esta es una construcción válida que coincide con *cualquier cosa pero nada* , es decir, coincide con cualquier símbolo posible (pero con signos diacríticos, al menos en ES5).

Clases de personajes POSIX

Las clases de caracteres POSIX son secuencias predefinidas para un determinado conjunto de caracteres.

Clase de personaje	Descripción
`[:alpha:]`	Caracteres alfabéticos
`[:alnum:]`	Caracteres alfabéticos y dígitos
`[:digit:]`	Dígitos
`[:xdigit:]`	Dígitos hexadecimales
`[:blank:]`	Espacio y tab
`[:cntrl:]`	Personajes de control
`[:graph:]`	Caracteres visibles (cualquier cosa excepto espacios y caracteres de control)
`[:print:]`	Caracteres y espacios visibles.
`[:lower:]`	Letras minusculas
`[:upper:]`	Letras mayúsculas
`[:punct:]`	Puntuación y símbolos.
`[:space:]`	Todos los caracteres de espacio en blanco, incluidos los saltos de línea

Las clases de caracteres adicionales pueden estar disponibles según la implementación y / o la configuración regional.

Clase de personaje	Descripción
`[:<:]`	Principio de palabra
`[:>:]`	Fin de la palabra
`[:ascii:]`	Personajes ASCII

Clase de personaje	Descripción
`[:word:]`	Letras, dígitos y guiones bajos. Equivalente a `\w`

Para utilizar el interior de una secuencia de corchetes (también conocida como clase de caracteres), también debe incluir los corchetes. Ejemplo:

```
[[:alpha:]]
```

Esto coincidirá con un carácter alfabético.

```
[[:digit:]-]{2}
```

Esto coincidirá con 2 caracteres, que son dígitos o `-` . Lo siguiente coincidirá:

- `--`
- `11`
- `-2`
- `3-`

Más información está disponible en: Regular-expressions.info

Lea Clases de personajes en línea: https://riptutorial.com/es/regex/topic/1757/clases-de-personajes

Capítulo 5: Combinadores de UTF-8: letras, marcas, puntuación, etc.

Examples

Cartas a juego en diferentes alfabetos

Los siguientes ejemplos se dan en Ruby, pero los mismos emparejadores deberían estar disponibles en cualquier idioma moderno.

Digamos que tenemos la cadena `"AஃNaï ve"` , producida por Messy Artificial Intelligence. Se compone de letras, pero genérica `\w` matcher no coincidirá tanto:

```
▶ "AஃNaï ve"[/\w+/]
#⇒ "A"
```

La forma correcta de hacer coincidir la letra Unicode con marcas combinadas es usar `\X` para especificar un grupo de grafemas. Sin embargo, hay una advertencia para Ruby. Onigmo, el motor de expresiones regulares para Ruby, todavía usa la antigua definición de un grupo de grafemas . Aún no se ha actualizado a Extended Grapheme Cluster como se define en el Anexo 29 de Unicode Standard .

Por lo tanto, para Ruby podríamos tener una solución: `\p{L}` hará casi bien, salvo porque falla en el acento diacrítico combinado en `i` :

```
▶ "AஃNaï ve"[/\p{L}+/]
#⇒ "AஃNai"
```

Al agregar los "símbolos de marca" a la expresión, finalmente podemos hacer coincidir todo:

```
▶ "AஃNaï ve"[/[\p{L}\p{M}]+/]
#⇒ "AஃNaï ve"
```

Lea Combinadores de UTF-8: letras, marcas, puntuación, etc. en línea: https://riptutorial.com/es/regex/topic/1527/combinadores-de-utf-8--letras--marcas--puntuacion--etc-

Capítulo 6: Cuando NO debes usar Expresiones Regulares

Observaciones

Debido a que las expresiones regulares se limitan a una gramática regular o una gramática libre de contexto, hay muchos usos incorrectos comunes de las expresiones regulares. Entonces, en este tema hay algunos ejemplos de cuándo *NO* debe usar expresiones regulares, sino usar su idioma favorito en su lugar.

> *Algunas personas, cuando se enfrentan a un problema, piensan:*
> *"Lo sé, usaré expresiones regulares".*
> *Ahora ellos tienen dos problemas.*
> - Jamie Zawinski

Examples

Parejas coincidentes (como paréntesis, paréntesis ...)

Algunos motores de expresiones regulares (como .NET) pueden manejar expresiones sin contexto, y funcionarán. Pero ese no es el caso para la mayoría de los motores estándar. E incluso si lo hacen, terminará teniendo una expresión compleja de difícil lectura, mientras que el uso de una biblioteca de análisis podría facilitar el trabajo.

- ¿Cómo encontrar todas las posibles coincidencias de expresiones regulares en python?

Operaciones de cadena simples

Debido a que las *expresiones regulares* pueden hacer mucho, es tentador usarlas para las operaciones más simples. Pero usar un motor de expresiones regulares tiene un costo en la memoria y en el uso del procesador: necesita compilar la expresión, almacenar el autómata en la memoria, inicializarlo y luego alimentarlo con la cadena para ejecutarlo.

¡Y hay muchos casos en los que simplemente no es necesario usarlo! Sea cual sea el idioma que elija, siempre tiene las herramientas básicas de manipulación de cadenas. Entonces, como regla general, cuando hay una herramienta para realizar una acción en su biblioteca estándar, use esa herramienta, no una expresión regular:

- dividir una cadena?

Por ejemplo, el siguiente fragmento de código funciona en Python, Ruby y Javascript:

```
'foo.bar'.split('.')
```

Es más fácil de leer y entender, y es mucho más eficiente que la expresión regular equivalente (de alguna manera):

```
(\w+)\.(\w+)
```

- ¿Desvío de espacios?

¡Lo mismo se aplica a los espacios finales!

```
'foobar     '.strip() # python or ruby
'foobar     '.trim() // javascript
```

Lo que sería equivalente a la siguiente expresión:

```
([^\n]*)\s*$ # keeping \1 in the substitution
```

Análisis de HTML (o XML, o JSON, o código C, o ...)

Si desea extraer algo de una página web (o cualquier lenguaje de representación / programación), una expresión regular es la herramienta incorrecta para la tarea. En su lugar, debe utilizar las bibliotecas de su idioma para lograr la tarea.

Si desea leer HTML, XML o JSON, simplemente use la biblioteca que lo analiza correctamente y lo sirve como objetos utilizables en su idioma favorito. Terminarás con un código legible y más mantenible, y no terminarás

- RegEx coincide con las etiquetas abiertas excepto las etiquetas autocontenidas XHTML
- Python analizando HTML usando expresiones regulares
- ¿Existe una expresión regular para generar todos los enteros para un determinado lenguaje de programación?

Lea Cuando NO debes usar Expresiones Regulares en línea:
https://riptutorial.com/es/regex/topic/4527/cuando-no-debes-usar-expresiones-regulares

Capítulo 7: Cuantificadores Posesivos

Observaciones

NB emulando cuantificadores posesivos.

Examples

Uso básico de cuantificadores posesivos

Los cuantificadores posesivos son otra clase de cuantificadores en muchos tipos de expresiones regulares que permiten que el retroceso se deshabilite, efectivamente, para un token dado. Esto puede ayudar a mejorar el rendimiento, así como a prevenir coincidencias en ciertos casos.

La clase de cuantificadores posesivos se puede distinguir de los cuantificadores perezosos o codiciosos mediante la adición de a `+` después del cuantificador, como se ve a continuación:

Cuantificador	Codicioso	Perezoso	Posesivo
Cero o mas	`*`	`*?`	`*+`
Uno o mas	`+`	`+?`	`++`
Cero o uno	`?`	`??`	`?+`

Considere, por ejemplo, los dos patrones `".*"` Y `".*+"` , Que operan en la cadena `"abc"d` . En ambos casos, el `"` al principio de la cadena coincide", pero después de eso los dos patrones tendrán diferentes comportamientos y resultados.

El codificador codicioso luego sorbirá el resto de la cadena, `abc"d` . Debido a que esto no coincide con el patrón, entonces retrocederá y caerá la `d` , dejando el **cuantificador que** contiene `abc"` . Debido a que esto aún no coincide con el patrón, el cuantificador eliminará el `"` , dejando que contenga solo `abc` . Esto coincide con el patrón (ya que `"` coincide con un literal, en lugar del cuantificador), y el regex informa de éxito.

El cuantificador posesivo también absorberá el resto de la cadena, pero, a diferencia del cuantificador codicioso, no retrocederá. Dado que su contenido, `abc"d` , no permite el resto del patrón de la coincidencia, la expresión regular se detendrá e informará que la falla no coincide.

Debido a que los cuantificadores posesivos no hacen retroceso, pueden resultar en un aumento significativo del rendimiento en patrones largos o complejos. Sin embargo, pueden ser peligrosos (como se ilustra arriba) si uno no es consciente de cómo, precisamente, los cuantificadores funcionan internamente.

Lea Cuantificadores Posesivos en línea:
https://riptutorial.com/es/regex/topic/5916/cuantificadores-posesivos

Capítulo 8: Cuantitativos codiciosos y perezosos

Parámetros

Cuantificadores	Descripción
`?`	Haga coincidir el carácter o subexpresión anterior 0 o 1 veces (preferiblemente 1).
`*`	Haga coincidir el carácter o subexpresión anterior 0 o más veces (tantas como sea posible).
`+`	Haga coincidir el carácter o subexpresión anterior 1 o más veces (tantas como sea posible).
`{n}`	Haga coincidir el carácter o subexpresión precedente exactamente *n* veces.
`{min,}`	Haga coincidir el carácter o subexpresión anterior un *mínimo* o más veces (el mayor número posible).
`{0,max}`	Haga coincidir el carácter o subexpresión anterior *máximo* o menos veces (lo más cerca posible del *máximo*).
`{min,max}`	Haga coincidir el carácter o subexpresión precedente al menos un *mínimo de* veces, pero no más de un *máximo de* veces (lo más cerca posible del *máximo*).
Cuantificadores perezosos	**Descripción**
`??`	Haga coincidir el carácter o subexpresión anterior 0 o 1 veces (preferiblemente 0).
`*?`	Haga coincidir el carácter o subexpresión anterior 0 o más veces (el menor número posible).
`+?`	Haga coincidir el carácter o subexpresión anterior 1 o más veces (lo menos posible).
`{n}?`	Haga coincidir el carácter o subexpresión precedente exactamente *n* veces. No hay diferencia entre la versión codiciosa y la perezosa.
`{min,}?`	Haga coincidir el carácter o subexpresión anterior *mín.* O más veces (lo más cerca posible al *mínimo*).

Cuantificadores	Descripción
`{0,max}?`	Haga coincidir el carácter o subexpresión precedente como *máximo* o menos (lo menos posible).
`{min,max}?`	Haga coincidir el carácter o la subexpresión precedente al menos un *mínimo de* veces, pero no más de un *máximo de* veces (lo más cerca posible del *mínimo*).

Observaciones

Codicia

Un cuantificador *codicioso* siempre intenta repetir el sub-patrón tantas veces como sea posible antes de explorar coincidencias más cortas por retroceso.

En general, un patrón codicioso coincidirá con la cadena más larga posible.

Por defecto, todos los cuantificadores son codiciosos.

pereza

Un cuantificador *perezoso* (también llamado *no codicioso* o *reacio*) siempre intenta repetir el sub-patrón tan *pocas* veces como sea posible, antes de explorar coincidencias más largas por expansión.

En general, un patrón perezoso coincidirá con la cadena más corta posible.

Para hacer que los cuantificadores sean perezosos, simplemente agregue `?` al cuantificador existente, por ejemplo, `+?` , `{0,5}?` .

El concepto de avaricia y pereza solo existe en los motores de retroceso.

La noción de cuantificador codicioso / perezoso solo existe en los motores de regex de retroceso. En los motores de expresiones regulares sin retroceso o los motores de expresiones regulares que cumplen con POSIX, los cuantificadores solo especifican el límite superior y el límite inferior de la repetición, sin especificar cómo encontrar la coincidencia; esos motores siempre coincidirán con la cadena más larga de la izquierda, independientemente.

Examples

La codicia contra la pereza

Dada la siguiente entrada:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
```

Usaremos dos patrones: uno codicioso: `A.*Z` , y uno perezoso: `A.*?Z` Estos patrones producen las siguientes coincidencias:

- `A.*Z` produce 1 coincidencia: `AlazyZgreeedyAlaaazyZ` (ejemplos: Regex101 , Rubular)
- `A.*?Z` produce 2 coincidencias: `AlazyZ` y `AlaaazyZ` (ejemplos: Regex101 , Rubular)

Primero enfócate en lo que `A.*Z` hace. Cuando coincidió con la primera `A` , el `.*` , Siendo codicioso, luego trata de hacer coincidir tantos `.` como sea posible.

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________________/
      A.* matched, Z can't match
```

Dado que la `Z` no coincide, el motor retrocede, y `.*` Debe coincidir con uno menos `.` :

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________________/
      A.* matched, Z can't match
```

Esto sucede unas cuantas veces más, hasta que finalmente se llega a esto:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \___________________/
      A.* matched, Z can now match
```

Ahora `Z` puede coincidir, por lo que el patrón general coincide:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________/
      A.*Z matched
```

Por el contrario, la repetición reacia (perezosa) en `A.*?Z` primero coincide con pocas `.` como sea posible, y luego tomar más `.` según sea necesario. Esto explica por qué encuentra dos coincidencias en la entrada.

Aquí hay una representación visual de lo que los dos patrones combinaron:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____/l      \______/l      l = lazy
     \_________g_________/       g = greedy
```

Ejemplo basado en respuesta hecha por poligenelubricantes .

El estándar POSIX no incluye el `?` Operador, muchos motores de expresiones regulares POSIX no tienen una comparación perezosa. Mientras que la refactorización, especialmente con el "mejor truco" , puede ayudar a emparejar, en algunos casos, la única manera de tener un verdadero juego perezoso es usar un motor que lo soporte.

Límites con múltiples coincidencias

Cuando tiene una entrada con límites bien definidos y espera más de una coincidencia en su cadena, tiene dos opciones:

- Usando cuantificadores perezosos;
- Usando una clase de personaje negada.

Considera lo siguiente:

Tienes un motor de plantillas simple, quieres reemplazar subcadenas como `$[foo]` donde `foo` puede ser cualquier cadena. Desea reemplazar esta subcadena con lo que esté basado en la parte entre `[]` .

Puedes probar algo como `\$\[(.*)\]` , Y luego usar el primer grupo de captura.

El problema con esto es que si tienes una cadena como `something $[foo] lalala $[bar] something else` tu pareja será

```
something $[foo] lalala $[bar] something else
          | \_______CG1_______/|
          \________Match_______/
```

El grupo de captura es `foo] lalala $[bar` que puede o no ser válida.

Tienes dos soluciones

1. Uso de la pereza: en este caso, hacer `*` perezoso es una forma de encontrar las cosas correctas. Así que cambias tu expresión a `\$\[(.*?)\]`

2. Usando la clase de caracteres negados: `[^\]]` cambias tu expresión a `\$\[([^\]]*)\]` .

En ambas soluciones, el resultado será el mismo:

```
something $[foo] lalala $[bar] something else
          | \_/|         | \_/|
          \____/         \____/
```

Con el grupo de captura siendo respectivamente `foo` y `bar` .

El uso de la clase de caracteres negados reduce el problema del seguimiento y puede ahorrarle mucho tiempo a la CPU cuando se trata de entradas grandes.

Lea Cuantitativos codiciosos y perezosos en línea:
https://riptutorial.com/es/regex/topic/429/cuantitativos-codiciosos-y-perezosos

Capítulo 9: Escapando

Examples

Literales crudos de cuerda

Es mejor para la legibilidad (y su cordura) para evitar escapar de los escapes. Ahí es donde entran en juego los literales de cadenas en bruto. (Tenga en cuenta que algunos idiomas permiten delimitadores, que generalmente se prefieren a las cadenas. Pero esa es otra sección).

Por lo general, funcionan de la misma manera que esta respuesta describe :

> [A] barra invertida, `\` , se toma como que significa "solo una barra invertida" (excepto cuando aparece justo antes de una cita que de otra manera terminaría el literal) - no hay "secuencias de escape" para representar nuevas líneas, tabulaciones, espacios en blanco, feeds de formularios , y así.

No todos los idiomas los tienen, y los que usan una sintaxis variable. C # en realidad los llama *literales de cadena literal* , pero es lo mismo.

Pitón

```
pattern = r"regex"
```

```
pattern = r'regex'
```

C ++ (11+)

La sintaxis aquí es extremadamente versátil. La única regla es usar un delimitador que no aparezca en ninguna parte de la expresión regular. Si lo hace, no es necesario que escape más para nada en la cadena. Tenga en cuenta que los paréntesis `()` no forman parte de la expresión regular:

```
pattern = R"delimiter(regex)delimiter";
```

VB.NET

Solo usa una cuerda normal. Las barras invertidas son siempre literales .

DO#

```
pattern = @"regex";
```

Tenga en cuenta que esta sintaxis también permite `""` (dos comillas dobles) como una forma de escape de `"` .

Instrumentos de cuerda

En la mayoría de los lenguajes de programación, para tener una barra invertida en una cadena generada a partir de una cadena literal, cada barra invertida debe duplicarse en la cadena literal. De lo contrario, se interpretará como un escape para el siguiente personaje.

Desafortunadamente, cualquier barra invertida requerida por la expresión regular debe ser una barra invertida literal. Es por esto que se hace necesario tener "escapes de escape" (`\\`) cuando las expresiones regulares se generan a partir de literales de cadena.

Además, las comillas (`"` o `'`) en el literal de cadena pueden ser eliminadas, dependiendo de lo que rodee al literal de cadena. En algunos idiomas, es posible usar cualquiera de los estilos de comillas para una cadena (elija el más legible para escapando de toda la cadena literal).

En algunos idiomas (por ejemplo, Java <= 7), las expresiones regulares no se pueden expresar directamente como literales como `/\w/` ; deben generarse a partir de cadenas, y normalmente se utilizan literales de cadena, en este caso, `"\\w"` . En estos casos, los caracteres literales, como comillas, barras invertidas, etc., deben escaparse. La forma más fácil de lograr esto puede ser usando una herramienta (como *RegexPlanet*). Esta herramienta específica está diseñada para Java, pero funcionará para cualquier idioma con una sintaxis de cadena similar.

¿Qué personajes necesitan ser escapados?

El escape de caracteres es lo que permite buscar y encontrar literalmente ciertos caracteres (reservados por el motor de expresiones regulares para manipular búsquedas) en la cadena de entrada. El escape depende del contexto, por lo tanto, este ejemplo no cubre el escape de la cadena o delimitador .

Barras invertidas

Decir que la barra invertida es el carácter de "escape" es un poco confuso. La barra invertida se escapa y la barra invertida trae; en realidad, activa o desactiva el metacarácter frente al estado literal del personaje que se encuentra frente a él.

Para utilizar una barra invertida literal en cualquier parte de una expresión regular, debe ser escapada por otra barra invertida.

Escape (fuera de las clases de personajes)

Hay varios personajes que necesitan ser escapados para ser tomados literalmente (al menos fuera de las clases char):

- Soportes: `[]`
- Paréntesis: `()`
- Aparatos ortopédicos: `{}`
- Operadores: `*` , `+` `?` , `|`
- Anclajes: `^` , `$`
- Otros: `.` , `\`
- Para utilizar un literal `^` al inicio o un `$` literal al final de una expresión regular, el carácter debe ser escapado.
- Algunos sabores solo usan `^` y `$` como metacaracteres cuando están al principio o al final de la expresión regular, respectivamente. En esos sabores, ningún escape adicional es necesario. Por lo general, es mejor escapar de todos modos.

Escapar dentro de las clases de personajes

- Es una buena práctica escapar de los corchetes (`[` y `]`) cuando aparecen como literales en una clase char. Bajo ciertas condiciones, no se requiere, dependiendo del sabor , pero daña la legibilidad.
- El caret, `^` , es un meta carácter cuando se coloca como primer carácter en una clase char: `[^aeiou]` . En cualquier otro lugar de la clase char, es solo un carácter literal.
- El guión, `-` , es un meta carácter, a menos que esté al principio o al final de una clase de carácter. Si el primer carácter de la clase char es un carácter de carácter `^` , entonces será un literal si es el segundo carácter de la clase char.

Escapar de la sustitución

También hay reglas para escapar dentro del reemplazo, pero ninguna de las reglas anteriores se aplican. Los únicos metacaracteres son `$` y `\` , al menos cuando `$` se puede usar para hacer referencia a grupos de captura (como `$1` para el grupo 1). Para usar un `$` literal, escápalo: `\$5.00` . Del mismo modo `\` : `C:\\Program Files\\` .

Excepciones BRE

Mientras que ERE (expresiones regulares extendidas) refleja la sintaxis típica de estilo Perl, BRE (expresiones regulares básicas) tiene diferencias significativas cuando se trata de escapar:

- Hay diferentes sintaxis abreviada. Todos los `\d` , `\s` , `\w` y así sucesivamente se han ido. En su lugar, tiene su propia sintaxis (que POSIX confusamente llama "clases de caracteres"), como `[:digit:]` . Estas construcciones deben estar dentro de una clase de caracteres.
- Hay pocos metacaracteres (`.` , `*` , `^` , `$`) Que se pueden usar normalmente. TODOS los otros metacaracteres deben escaparse de manera diferente:

Llaves `{}`

- `a{1,2}` coincide con `a{1,2}` . Para hacer coincidir `a` o `aa` , use `a\{1,2\}`

Paréntesis `()`

- `(ab)\1` no es válido, ya que no hay un grupo de captura 1. Para solucionarlo y hacer coincidir `abab` use `\(ab\)\1`

Barra invertida

- Dentro de las clases de caracteres (que se denominan expresiones de corchete en POSIX), la barra diagonal inversa no es un metacarácter (y no necesita escaparse). `[\d]` coincide con `\` o `d` .
- En cualquier otro lugar, escapar como de costumbre.

Otro

- `+` y `?` son literales. Si el motor BRE los admite como metacaracteres, deben escaparse como `\?` y `\+` .

/ Delimitadores /

Muchos idiomas permiten que las expresiones regulares se incluyan o delimiten entre un par de caracteres específicos, generalmente la barra diagonal hacia adelante `/` .

Los delimitadores tienen un impacto en el escape: si el delimitador es `/` y la expresión regular necesita buscar `/` literales, entonces la barra inclinada debe escaparse antes de que pueda ser un literal (`\/`).

La fuga excesiva perjudica la legibilidad, por lo que es importante considerar las opciones disponibles:

Javascript es único porque permite la barra inclinada como un delimitador, pero nada más (aunque sí permite expresiones regulares de cadena).

Perl 1

Perl, por ejemplo, permite que casi cualquier cosa sea un delimitador. Incluso los caracteres árabes:

```
$str =~ m ش ش
```

Reglas específicas son mencionadas en la documentación de Perl .

PCRE permite dos tipos de delimitadores: delimitadores emparejados y delimitadores de estilo de corchete. Los delimitadores combinados hacen uso de un par de un solo carácter, mientras que los delimitadores de estilo corchete hacen uso de un par de caracteres que representan un par de apertura y cierre.

- Delimitadores coincidentes:! `!"#$%&'*+,./:;=?@^_`|~-`
- Delimitadores de estilo de corchete: `()` , `{}` , `[]` , `<>`

Lea Escapando en línea: https://riptutorial.com/es/regex/topic/4524/escapando

Capítulo 10: Grupos de captura

Examples

Grupos de captura básicos

Un *grupo* es una sección de una expresión regular entre paréntesis `()` . Esto se denomina comúnmente "subexpresión" y tiene dos propósitos:

- Hace que la subexpresión sea atómica, es decir, que coincida, falle o se repita como un todo.
- La parte del texto que coincidió es accesible en el resto de la expresión y el resto del programa.

Los grupos están numerados en los motores de expresiones regulares, comenzando con 1. Tradicionalmente, el número máximo de grupos es 9, pero muchos de los sabores de expresiones regulares modernas admiten un mayor número de grupos. El grupo 0 siempre coincide con todo el patrón, de la misma manera que rodea toda la expresión regular con corchetes.

El número ordinal aumenta con cada paréntesis de apertura, independientemente de si los grupos se colocan uno tras otro o anidados:

```
foo(bar(baz)?) (qux)+|(bla)
   1   2       3      4
```

grupos y sus numeros

Después de que una expresión logre una coincidencia global, todos sus grupos estarán en uso, ya sea que un grupo en particular haya logrado igualar algo o no.

Un grupo puede ser opcional, como `(baz)?` arriba, o en una parte alternativa de la expresión que no se usó de la coincidencia, como `(bla)` arriba. En estos casos, los grupos no coincidentes simplemente no contendrán ninguna información.

Si se coloca un cuantificador detrás de un grupo, como en `(qux)+` arriba, el recuento global de grupos de la expresión permanece igual. Si un grupo coincide más de una vez, su contenido será la última vez que se produzca la coincidencia. Sin embargo, las versiones modernas de expresiones regulares permiten acceder a todas las apariciones de subincidencia.

Si desea recuperar la fecha y el nivel de error de una entrada de registro como esta:

```
2012-06-06 12:12.014 ERROR: Failed to connect to remote end
```

Podrías usar algo como esto:

```
^(\d{4}-\d{2}-\d{2}) \d{2}:\d{2}.\d{3} (\w*): .*$
```

Esto extraería la fecha de la entrada de registro `2012-06-06` como grupo de captura 1 y el nivel de error `ERROR` como grupo de captura 2.

Referencias y grupos no capturadores

Dado que los grupos están "numerados", algunos motores también admiten la coincidencia con lo que un grupo ha vuelto a hacer coincidir previamente.

Suponiendo que quisiera hacer coincidir algo donde dos cuerdas iguales de la longitud tres se dividen por un `$` que usaría:

```
(.{3})\$\1
```

Esto coincidiría con cualquiera de las siguientes cadenas:

```
"abc$abc"
"a b$a b"
"af $af "
"   $   "
```

Si desea que un grupo no esté numerado por el motor, puede declararlo como no capturado. Un grupo que no captura se ve así:

```
(?:)
```

Son particularmente útiles para repetir un cierto patrón cualquier cantidad de veces, ya que un grupo también puede usarse como un "átomo". Considerar:

```
(\d{4}(?:-\d{2}){2} \d{2}:\d{2}.\d{3}) (.*)[\r\n]+\1 \2
```

Esto coincidirá con dos entradas de registro en las líneas adyacentes que tienen la misma marca de tiempo y la misma entrada.

Grupos de captura con nombre

Algunos sabores de expresiones regulares permiten *grupos de captura con nombre* . En lugar de hacerlo mediante un índice numérico, puede referirse a estos grupos por su nombre en el código posterior, es decir, en las referencias inversas, en el patrón de reemplazo y en las siguientes líneas del programa.

Los índices numéricos cambian a medida que cambia el número o la disposición de los grupos en una expresión, por lo que son más frágiles en comparación.

Por ejemplo, para hacer coincidir una palabra (`\w+`) entre comillas simples o dobles (`['"]`), podríamos usar:

```
(?<quote>['"])\w+\k{quote}
```

Lo que equivale a:

```
(['"])\w+\1
```

En una situación simple como esta, un grupo de captura numerado y regular no tiene ningún inconveniente.

En situaciones más complejas, el uso de grupos con nombre hará que la estructura de la expresión sea más evidente para el lector, lo que mejora la capacidad de mantenimiento.

El análisis de archivos de registro es un ejemplo de una situación más compleja que se beneficia de los nombres de grupo. Este es el formato de registro común de Apache (CLF):

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

La siguiente expresión captura las partes en grupos nombrados:

```
(?<ip>\S+) (?<logname>\S+) (?<user>\S+) (?<time>\[[^]]+\]) (?<request>"[^"]+") (?<status>\S+)
(?<bytes>\S+)
```

La sintaxis depende del sabor, los más comunes son:

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

Backreferences:

- `\k<name>`
- `\k{name}`
- `\k'name'`
- `\g{name}`
- `(?P=name)`

En la versión .NET, puede tener varios grupos compartiendo el mismo nombre, usarán pilas de captura .

En PCRE, debe habilitarlo explícitamente utilizando el modificador `(?J)` (`PCRE_DUPNAMES`), o utilizando el grupo de restablecimiento de rama `(?|)` . Sin embargo, solo el último valor capturado será accesible.

```
(?J)(?<a>...)(?<a>...)
(?|(?<a>...)|(?<a>...))
```

Lea Grupos de captura en línea: https://riptutorial.com/es/regex/topic/660/grupos-de-captura

Capítulo 11: Grupos de captura con nombre

Sintaxis

- Cree un grupo de captura con nombre (siendo `x` el patrón que desea capturar):

  (? 'nombre'X) (? X) (? PX)

- Referencia a un grupo de captura con nombre:

  $ {nombre} \ {nombre} g \ {nombre}

Observaciones

Python y Java no permiten que múltiples grupos usen el mismo nombre.

Examples

Cómo se ve un grupo de captura con nombre

Dados los sabores, el grupo de captura nombrado puede tener este aspecto:

```
(?'name'X)
(?<name>X)
(?P<name>X)
```

Con `x` siendo el patrón que quieres capturar. Consideremos la siguiente cadena:

> Érase una vez una *niña bonita* ...
>
> Érase una vez un *unicornio con sombrero* ...
>
> Érase una vez un *barco con una bandera pirata* ...

En el que quiero capturar el tema (en *cursiva)* de cada línea. Usaré la siguiente expresión `.* was a (?<subject>[\w ]+)[.]{3}.`

El resultado coincidente tendrá:

```
MATCH 1
subject    [29-47]    `pretty little girl`
MATCH 2
subject    [80-99]    `unicorn with an hat`
MATCH 3
subject    [132-155]    `boat with a pirate flag`
```

Hacer referencia a un grupo de captura nombrado

Como puede (o no) saber, puede hacer referencia a un grupo de captura con:

```
$1
```

`1` es el número de grupo.

De la misma manera, puede hacer referencia a un grupo de captura nombrado con:

```
${name}
\{name}
g\{name}
```

Tomemos el ejemplo anterior y reemplacemos las coincidencias con

```
The hero of the story is a ${subject}.
```

El resultado que obtendremos es:

```
The hero of the story is a pretty little girl.
The hero of the story is a unicorn with an hat.
The hero of the story is a boat with a pirate flag.
```

Lea Grupos de captura con nombre en línea: https://riptutorial.com/es/regex/topic/744/grupos-de-captura-con-nombre

Capítulo 12: Límite de palabra

Sintaxis

- Estilo POSIX, final de la palabra: `[[:>:]]`
- Estilo POSIX, comienzo de palabra: `[[:<:]]`
- Estilo POSIX, límite de palabra: `[[:<:][:>:]]`
- SVR4 / GNU, fin de la palabra: `\>`
- SVR4 / GNU, comienzo de la palabra: `\<`
- Perl / GNU, límite de palabra: `\b`
- Tcl, fin de la palabra: `\M`
- Tcl, comienzo de la palabra: `\m`
- Tcl, límite de palabra: `\y`
- ERE portátil, comienzo de la palabra: `(^|[^[:alnum:]_])`
- ERE portátil, final de la palabra: `([^[:alnum:]_]|$)`

Observaciones

Recursos adicionales

- Capítulo POSIX sobre expresiones regulares.
- Perl documentación de expresiones regulares
- Tcl re_syntax manual de la página
- Expresiones de barra invertida grep de GNU
- BSD re_format
- Más lectura

Examples

Coincidir palabra completa

```
\bfoo\b
```

coincidirá con la palabra completa sin alfanumérico y _ precedente o siguiente.

Tomando de regularexpression.info

> Hay tres posiciones diferentes que califican como límites de palabras:
>
> 1. Antes del primer carácter de la cadena, si el primer carácter es un carácter de palabra.
> 2. Después del último carácter de la cadena, si el último carácter es un carácter de palabra.
> 3. Entre dos caracteres en la cadena, donde uno es un carácter de palabra y el otro

no es un carácter de palabra.

El término *palabra carácter* aquí significa cualquiera de los siguientes

1. Alfabeto (`[a-zA-Z]`)
2. Número (`[0-9]`)
3. Subrayar _

En resumen, el *carácter de la palabra* = `\w` = `[a-zA-Z0-9_]`

Encuentra patrones al principio o al final de una palabra

Examine las siguientes cadenas:

```
foobarfoo
bar
foobar
barfoo
```

- la `bar` expresión regular coincidirá con las cuatro cadenas,
- `\bbar\b` solo coincidirá con el 2do,
- `bar\b` podrá hacer coincidir las cadenas 2 y 3, y
- `\bbar` coincidirá con las cadenas 2 y 4.

Límites de palabras

El metacarácter `\b`

Para que sea más fácil encontrar palabras completas, podemos usar el metacarácter `\b` . Marca el **comienzo** *y* el **final** de una secuencia alfanumérica *. Además, como solo sirve para marcar estas ubicaciones, en realidad no coincide con ningún personaje por sí solo.

**: Es común llamar palabra a una secuencia alfanumérica, ya que podemos capturar sus caracteres con una `\w` (la clase de caracteres de la palabra). Sin embargo, esto puede ser engañoso, ya que `\w` también incluye números y, en la mayoría de los casos, el guión bajo.*

Ejemplos:

Regex	Entrada	¿Partidos?
`\bstack\b`	`stackoverflow`	**No** , ya que no hay ocurrencia de **toda la** `stack` palabras
`\bstack\b`	`foo stack bar`	**Sí** , ya que no hay nada antes ni después de la `stack`
`\bstack\b`	`stack!overflow`	**Sí** : no hay nada antes de `stack` y `!` no es un caracter de palabra
`\bstack`	`stackoverflow`	**Sí** , ya que no hay nada antes de `stack`

Regex	Entrada	¿Partidos?
`overflow\b`	`stackoverflow`	**Sí** , ya que no hay nada después del `overflow`

El metacarácter `\B`

Esto es lo opuesto a `\b` , que coincide con la ubicación de cada carácter no delimitador. Me gusta `\b` , ya que coincide con ubicaciones, no coincide con ningún personaje por sí solo. Es útil para encontrar palabras *no* completas.

Ejemplos:

Regex	Entrada	¿Partidos?
`\Bb\B`	`abc`	**Sí** , ya que `b` no está rodeado por límites de palabras.
`\Ba\B`	`abc`	**No** , `a` tiene un límite de palabra en su lado izquierdo.
`a\B`	`abc`	**Sí** , `a` no tiene un límite de palabra en su lado derecho.
`\B,\B`	`a,,,b`	**Sí** , coincide con la segunda coma porque `\B` también coincidirá con el espacio entre dos caracteres que no son palabras (se debe tener en cuenta que hay un límite de palabra a la izquierda de la primera coma ya la derecha de la segunda).

Hacer el texto más corto pero no romper la última palabra

Para hacer que el texto largo tenga un máximo de N caracteres pero deje la última palabra intacta, use el patrón `.{0,N}\b` :

```
^(.{0,N})\b.*
```

Lea Límite de palabra en línea: https://riptutorial.com/es/regex/topic/1539/limite-de-palabra

Capítulo 13: Lookahead y Lookbehind

Sintaxis

- **Lookahead positivo:** `(?=pattern)`
- **Lookahead negativo:** `(?!pattern)`
- **Mirada positiva detrás de** : `(?<=pattern)`
- **Mirada negativa detrás de** : `(?<!pattern)`

Observaciones

No es compatible con todos los motores de expresiones regulares.

Además, muchos motores de expresiones regulares limitan los patrones en el interior de las cuerdas de longitud fija. Por ejemplo, el patrón `(?<=a+)b` debe coincidir con la `b` en `aaab` pero arroja un error en Python.

Los grupos de captura están permitidos y funcionan como se espera, incluidas las referencias inversas. Sin embargo, el lookahead / lookbehind no es un grupo de captura.

Examples

Lo esencial

Un **lookahead positivo** `(?=123)` afirma que el texto es seguido por el patrón dado, sin incluir el patrón en la coincidencia. De manera similar, una **mirada positiva detrás de** `(?<=123)` afirma que el texto está precedido por el patrón dado. Sustituyendo el `=` con `!` Niega la afirmación.

Entrada : `123456`

- `123(?=456)` coincide con `123` (*lookahead positivo*)
- `(?<=123)456` coincidencias `456` (*aspecto positivo detrás de*)
- `123(?!456)` falla (*lookahead negativo*)
- `(?<!123)456` falla (*aspecto negativo detrás de*)

Entrada : `456`

- `123(?=456)` falla
- `(?<=123)456` falla
- `123(?!456)` falla
- `(?<!123)456` partidos `456`

Usando lookbehind para probar finales.

Se puede usar una mirada detrás del final de un patrón para asegurar que termine o no de cierta manera.

`([az ]+|[AZ ]+)(?<! )` coincide con secuencias de solo letras en minúsculas o mayúsculas, excluyendo los espacios en blanco finales.

Simulando la apariencia de longitud variable detrás de \ K

Algunos sabores de expresiones regulares (Perl, PCRE, Oniguruma, Boost) solo son compatibles con el aspecto de longitud fija, pero ofrecen la función `\K` , que se puede usar para simular el aspecto de longitud variable al inicio de un patrón. Al encontrar un `\K` , el texto coincidente hasta este punto se descarta, y solo el texto que coincide con la parte del patrón que *sigue a* `\K` se mantiene en el resultado final.

```
ab+\Kc
```

Es equivalente a:

```
(?<=ab+)c
```

En general, un patrón de la forma:

```
(subpattern A)\K(subpattern B)
```

Termina siendo similar a:

```
(?<=subpattern A)(subpattern B)
```

Excepto cuando el subpatrón B puede coincidir con el mismo texto que el subpatrón A, podría terminar con resultados sutilmente diferentes, porque el subpatrón A todavía consume el texto, a diferencia de lo que se ve por detrás.

Lea Lookahead y Lookbehind en línea: https://riptutorial.com/es/regex/topic/639/lookahead-y-lookbehind

Capítulo 14: Modificadores de expresiones regulares (banderas)

Introducción

Los patrones de expresión regular a menudo se usan con *modificadores* (también llamados *indicadores*) que redefinen el comportamiento de expresiones regulares. Los modificadores Regex pueden ser *regulares* (por ejemplo, `/abc/i`) y en *línea* (o *incrustados*) (por ejemplo, `(?i)abc`). Los modificadores más comunes son los modificadores globales, que no distinguen entre mayúsculas y minúsculas, multilínea y dotall. Sin embargo, los tipos de expresión regular difieren en el número de modificadores de expresión regular admitidos y sus tipos.

Observaciones

Modificadores PCRE

Modificador	En línea	Descripción
PCRE_CASELESS	(?yo)	Coincidencia insensible al caso
PCRE_MULTILINE	(?metro)	Coincidencia de líneas múltiples
PCRE_DOTALL	(? s)	. une nuevas lineas
PCRE_ANCHORED	(?UNA)	Meta-carácter `^` coincide solo al inicio
PCRE_EXTENDED	(?X)	Los espacios en blanco son ignorados
PCRE_DOLLAR_ENDONLY	n / A	Meta-carácter `$` coincide solo al final
PCRE_EXTRA	(?X)	Estricto escape de análisis
PCRE_UTF8		Maneja caracteres `UTF-8`
PCRE_UTF16		Maneja caracteres `UTF-16`
PCRE_UTF32		Maneja caracteres `UTF-32`
PCRE_UNGREEDY	(? U)	Configura el motor para una coincidencia perezosa.
PCRE_NO_AUTO_CAPTURE	(? :)	Desactiva los grupos de auto captura

Modificadores de Java

Modificador (`Pattern.###`)	Valor	Descripción
UNIX_LINES	1	Habilita el modo de líneas Unix .
CASE_INSENSITIVE	2	Permite la coincidencia entre mayúsculas y minúsculas.
COMENTARIOS	4	Permite espacios en blanco y comentarios en un patrón.
MULTILINE	8	Habilita el modo multilínea.
LITERAL	dieciséis	Habilita el análisis literal del patrón.
DOTALL	32	Habilita el modo dotall.
UNICODE_CASE	64	Habilita el plegado de casos conscientes de Unicode.
CANON_EQ	128	Habilita la equivalencia canónica.
UNICODE_CHARACTER_CLASS	256	Habilita la versión Unicode de clases de caracteres predefinidas y clases de caracteres POSIX.

Examples

Modificador DOTALL

Un patrón de expresiones regulares donde un modificador DOTALL (en la mayoría de los sabores de expresiones regulares expresado con `s`) cambia el comportamiento de `.` habilitándolo para que coincida con un símbolo de nueva línea (LF):

```
/cat (.*?) dog/s
```

Esta expresión regular al estilo de Perl coincidirá con una cadena como `"cat fled from\na dog"` capturando `"fled from\na"` al Grupo 1.

Una versión en línea: `(?s)` (por ejemplo, `(?s)cat (.*?) dog`)

Nota : En Ruby, el equivalente del modificador DOTALL es `m` , `Regexp::MULTILINE` modificador `Regexp::MULTILINE` (por ejemplo, `/a.*b/m`).

Nota : JavaScript no proporciona un modificador DOTALL, por lo que a `.` nunca se puede permitir

que coincida con un carácter de nueva línea. Para lograr el mismo efecto, una solución es necesaria, por ejemplo, sustituyendo todos los `.` s con una clase de caracteres comodín como `[\S\s]` , o una clase de carácter *no nada* `[^]` (sin embargo, esta construcción será tratada como un error por todos los otros motores, y por lo tanto no es portátil).

Modificador MULTILINE

Otro ejemplo es un modificador MULTILINE (generalmente expresado con el indicador `m` (no en Oniguruma (por ejemplo, Ruby) que usa `m` para indicar un modificador DOTALL)) que hace que los anclajes de `^` y `$` coincidan con el inicio / final de una *línea* , no el inicio / final de toda la cadena.

```
/^My Line \d+$/gm
```

encontrará todas las *líneas* que comienzan con `My Line` , luego contendrá un espacio y 1+ dígitos hasta el final de la línea.

Una versión en línea: `(?m)` (por ejemplo, `(?m)^My Line \d+$`)

NOTA : En Oniguruma (por ejemplo, en Ruby), y también en casi todos los editores de texto que admiten expresiones regulares, los anclajes `^` y `$` indican las posiciones de inicio / final de *línea de forma predeterminada* . Debe usar `\A` para definir todo el comienzo de cadena / documento y `\z` para denotar el final de cadena / documento. La diferencia entre `\Z` y `\z` es que la primera puede coincidir antes del símbolo de nueva línea final (LF) al final de la cadena (por ejemplo, `/\Astring\Z/` encontrará una coincidencia en `"string\n"`) (excepto Python, donde el comportamiento de `\Z` es igual a `\z` no se admite el anclaje `\z`).

Modificador IGNORE CASE

El modificador común para ignorar el caso es `i` :

```
/fog/i
```

coincidirá con `Fog` , `foG` , etc.

La versión en línea del modificador se parece a `(?i)` .

Notas:

En Java, de forma predeterminada, la *coincidencia que no distingue entre mayúsculas y minúsculas asume que solo se comparan los caracteres en el conjunto de caracteres US-ASCII. La coincidencia no sensible a mayúsculas y minúsculas se puede habilitar especificando el indicador* `UNICODE_CASE` *junto con este* `CASE_INSENSITIVE` *(* `CASE_INSENSITIVE` *) .* (por ejemplo, `Pattern p = Pattern.compile("YOUR_REGEX", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);`). Se puede encontrar algo más sobre esto en *Coincidencia insensible a mayúsculas y minúsculas en RegEx de Java* . Además, `UNICODE_CHARACTER_CLASS` se puede usar para hacer que Unicode concuerde.

VERBOSE / COMMENT / IgnorePatternWhitespace modifier

El modificador que permite usar espacios en blanco dentro de algunas partes del patrón para formatearlo para una mejor legibilidad y para permitir comentarios que comiencen con `#` :

```
/(?x)^           # start of string
  (?=\D*\d)      # the string should contain at least 1 digit
  (?!\d+$)       # the string cannot consist of digits only
  \#             # the string starts with a hash symbol
  [a-zA-Z0-9]+ # the string should have 1 or more alphanumeric symbols
  $              # end of string
/
```

Ejemplo de una cadena: `#word1here` . Tenga en cuenta que el símbolo `#` se escapa para denotar un `#` literal que forma parte de un patrón.

El espacio en blanco que no se ha escapado en el patrón de expresión regular se ignora, escápelo para que forme parte del patrón.

Generalmente, el espacio en blanco dentro de las clases de caracteres (`[...]`) se trata como un espacio en blanco literal, excepto en Java.

Además, vale la pena mencionar que en PCRE, .NET, Python, Ruby Oniguruma, ICU, Boost regex, los sabores se pueden usar `(?#:...)` dentro del patrón regex.

Modificador explícito de captura

Este es un modificador específico de expresiones regulares de .NET expresado con `n` . Cuando se utilizan, los grupos sin nombre (como `(\d+)`) no se capturan. Solo las capturas válidas son grupos con nombres explícitos (por ejemplo, `(?<name> subexpression)`).

```
(?n)(\d+)-(\w+)-(?<id>\w+)
```

coincidirá con todo `123-1_abc-00098` , pero `(\d+)` y `(\w+)` no crearán grupos en el objeto coincidente resultante. El único grupo será `${id}` . Ver demo .

Modificador UNICODE

El modificador UNICODE, generalmente expresado como `u` (PHP, Python) o `U` (Java), hace que el motor de expresiones regulares trate el patrón y la cadena de entrada como cadenas y patrones de Unicode, hace que las clases abreviadas del patrón como `\w` , `\d` , `\s` , etc. Unicode-conscientes.

```
/\A\p{L}+\z/u
```

es una expresión regular de PHP para hacer coincidir cadenas que constan de 1 o más letras Unicode. Ver la demo regex .

Tenga en cuenta que en **PHP** , el modificador `/u` permite al motor PCRE manejar cadenas como

UTF8 (activando el verbo `PCRE_UTF8`) y hacer que las clases de caracteres abreviadas en el patrón Unicode sean conscientes (al habilitar el verbo `PCRE_UCP` , consulte más en *pcre.org*) .

> **Los patrones y las cadenas de sujetos se tratan como UTF-8.** Este modificador está disponible desde PHP 4.1.0 o superior en Unix y desde PHP 4.2.3 en win32. La validez UTF-8 del patrón y del sujeto se verifica desde PHP 4.3.5. Un sujeto no válido hará que la función preg_ * no coincida con nada; un patrón no válido activará un error de nivel E_WARNING. Las secuencias de UTF-8 de cinco y seis octetos se consideran inválidas desde PHP 5.3.4 (resp. PCRE 7.3 2007-08-28); anteriormente aquellos han sido considerados como válidos UTF-8.

En Python 2.x, `re.UNICODE` solo afecta al patrón: *Make* `\w` , `\W` , `\b` , `\B` , `\d` , `\D` , `\s` *y* `\S` *dependientes de la base de datos de propiedades de caracteres Unicode.*

Una versión en línea: `(?u)` en Python, `(?U)` en Java. Por ejemplo:

```
print(re.findall(ur"(?u)\w+", u"Dąb")) # [u'D\u0105b']
print(re.findall(r"\w+", u"Dąb"))      # [u'D', u'b']

System.out.println("Dąb".matches("(?U)\\w+")); // true
System.out.println("Dąb".matches("\\w+"));     // false
```

PCRE_DOLLAR_ENDONLY modificador

El modificador *PCRE_DOLLAR_ENDONLY* compatible con *PCRE* que hace que el ancla `$` coincida al *final de la cadena* (excluyendo la posición antes de la nueva línea final en la cadena).

```
/^\d+$/D
```

es igual a

```
/^\d+\z/
```

y coincide con una cadena completa que consta de 1 o más dígitos y no coincidirá con `"123\n"` , pero coincidirá con `"123"` .

Modificador PCRE_ANCHORED

Otro modificador compatible con PCRE expresado con `/A` modificador. Si se establece este modificador, el patrón se obliga a "anclarse", es decir, se limita a coincidir solo al comienzo de la cadena que se está buscando (la "cadena de asunto"). Este efecto también puede lograrse mediante construcciones apropiadas en el propio patrón, que es la única forma de hacerlo en Perl.

```
/man/A
```

es lo mismo que

```
/^man/
```

Modificador PCRE_UNGREEDY

El indicador PCRE_UNGREEDY compatible con PCRE expresado con `/U` Cambia la codicia dentro de un patrón: `/a.*?b/U` = `/a.*b/` y viceversa.

Modificador PCRE_INFO_JCHANGED

Un modificador PCRE más que permite el uso de grupos nombrados duplicados.

NOTA : solo se admite la versión en *línea* - `(?J)` , y se debe colocar al inicio del patrón.

Si utiliza

```
/(?J)\w+-(?:new-(?<val>\w+)|\d+-empty-(?<val>[^-]+)-collection)/
```

los valores de grupo "val" nunca estarán vacíos (siempre se establecerán). Sin embargo, se puede lograr un efecto similar con el reinicio de rama.

Modificador PCRE_EXTRA

Un modificador de PCRE que causa un error si una barra invertida en un patrón es seguida por una letra que no tiene un significado especial. De forma predeterminada, una barra invertida seguida de una letra sin un significado especial se trata como un literal.

P.ej

```
/big\y/
```

coincidirá `bigy` , pero

```
/big\y/X
```

lanzará una excepción.

Versión en línea: `(?X)`

Lea Modificadores de expresiones regulares (banderas) en línea: https://riptutorial.com/es/regex/topic/5138/modificadores-de-expresiones-regulares--banderas-

Capítulo 15: Patrones simples a juego

Examples

Haga coincidir un carácter de un solo dígito usando [0-9] o \ d (Java)

`[0-9]` y `\d` son patrones equivalentes (a menos que su motor Regex sea compatible con Unicode y `\d` también coincida con cosas como). Ambos coincidirán con un carácter de un solo dígito para que pueda usar la notación que encuentre más legible.

Crea una cadena del patrón que deseas hacer coincidir. Si usa la notación \ d, deberá agregar una segunda barra diagonal inversa para escapar de la primera barra diagonal inversa.

```
String pattern = "\\d";
```

Crear un objeto de patrón. Pase la cadena de patrón en el método compile ().

```
Pattern p = Pattern.compile(pattern);
```

Crea un objeto Matcher. Pase la cadena que está buscando para encontrar el patrón en el método matcher (). Compruebe si se encuentra el patrón.

```
Matcher m1 = p.matcher("0");
m1.matches(); //will return true

Matcher m2 = p.matcher("5");
m2.matches(); //will return true

Matcher m3 = p.matcher("12345");
m3.matches(); //will return false since your pattern is only for a single integer
```

Coincidencia de varios números

`[ab]` donde ayb son dígitos en el rango de `0` a `9`

```
[3-7] will match a single digit in the range 3 to 7.
```

Coincidencia de varios dígitos

```
\d\d       will match 2 consecutive digits
\d+        will match 1 or more consecutive digits
\d*        will match 0 or more consecutive digits
\d{3}      will match 3 consecutive digits
\d{3,6}    will match 3 to 6 consecutive digits
\d{3,}     will match 3 or more consecutive digits
```

El `\d` en los ejemplos anteriores se puede reemplazar con un rango de números:

```
[3-7][3-7]    will match 2 consecutive digits that are in the range 3 to 7
[3-7]+        will match 1 or more consecutive digits that are in the range 3 to 7
[3-7]*        will match 0 or more consecutive digits that are in the range 3 to 7
[3-7]{3}      will match 3 consecutive digits that are in the range 3 to 7
[3-7]{3,6}    will match 3 to 6 consecutive digits that are in the range 3 to 7
[3-7]{3,}     will match 3 or more consecutive digits that are in the range 3 to 7
```

También puede seleccionar dígitos específicos:

```
[13579]       will only match "odd" digits
[02468]       will only match "even" digits
1|3|5|7|9     another way of matching "odd" digits - the | symbol means OR
```

Números coincidentes en rangos que contienen más de un dígito:

```
\d|10        matches 0 to 10    single digit OR 10.  The | symbol means OR
[1-9]|10     matches 1 to 10    digit in range 1 to 9 OR 10
[1-9]|1[0-5] matches 1 to 15    digit in range 1 to 9 OR 1 followed by digit 1 to 5
\d{1,2}|100  matches 0 to 100   one to two digits OR 100
```

Números coincidentes que se dividen por otros números:

```
\d*0         matches any number that divides by 10  - any number ending in 0
\d*00        matches any number that divides by 100 - any number ending in 00
\d*[05]      matches any number that divides by 5   - any number ending in 0 or 5
\d*[02468]   matches any number that divides by 2   - any number ending in 0,2,4,6 or 8
```

números coincidentes que se dividen por 4: cualquier número que sea 0, 4 u 8 o que finalice en 00, 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60 64, 68, 72, 76, 80, 84, 88, 92 o 96

```
[048]|\d*(00|04|08|12|16|20|24|28|32|36|40|44|48|52|56|60|64|68|72|76|80|84|88|92|96)
```

Esto se puede acortar. Por ejemplo, en lugar de usar `20|24|28` podemos usar `2[048]` . Además, como los años 40, 60 y 80 tienen el mismo patrón, podemos incluirlos: `[02468][048]` y los otros también tienen un patrón `[13579][26]` . Entonces toda la secuencia puede reducirse a:

```
[048]|\d*([02468][048]|[13579][26])    - numbers divisible by 4
```

Los números coincidentes que no tienen un patrón como los divisibles por 2,4,5,10, etc. no siempre se pueden hacer de manera sucinta y, por lo general, hay que recurrir a un rango de números. Por ejemplo, hacer coincidir todos los números que se dividen por 7 dentro del rango de 1 a 50 se puede hacer de manera simple al enumerar todos esos números:

```
7|14|21|28|35|42|49

or you could do it this way

7|14|2[18]|35|4[29]
```

Emparejando espacios en blanco iniciales / finales

Espacios finales

`\s*$` : Esto coincidirá con cualquier (`*`) espacio en blanco (`\s`) al final (`$`) del texto

Espacios principales

`^\s*` : Esto coincidirá con cualquier (`*`) espacio en blanco (`\s`) al principio (`^`) del texto

Observaciones

`\s` es un metacarácter común para varios motores RegExp, y está destinado a capturar caracteres de espacios en blanco (espacios, nuevas líneas y pestañas, por ejemplo). **Nota** : probablemente *no* capturará todos los caracteres de espacio Unicode . Revise la documentación de su motor para estar seguro de esto.

Empareja cualquier flotador

```
[\+\-]?\d+(\.\d*)?
```

Esto coincidirá con cualquier flotante firmado, si no quiere signos o está analizando una ecuación, elimine `[\+\-]?` entonces tienes `\d+(\.\d+)?`

Explicación:

- `\d+` coincide con cualquier entero
- `()?` significa que los contenidos de los paréntesis son opcionales pero siempre tienen que aparecer juntos
- '\.' Partidos '.', tenemos que escapar de esto ya que '. normalmente coincide con cualquier personaje

Así que esta expresión coincidirá

```
5
+5
-5
5.5
+5.5
-5.5
```

Seleccionar una línea determinada de una lista basada en una palabra en cierta ubicación

Tengo la siguiente lista:

```
1. Alon Cohen
```

```
2. Elad Yaron
3. Yaron Amrani
4. Yogev Yaron
```

Quiero seleccionar el primer nombre de los chicos con el apellido Yaron.

Ya que no me importa qué número es, solo lo pondré como el dígito que sea y el punto y el espacio que le siguen desde el principio de la línea, como este: `^[\d]+\.\s` .

Ahora tendremos que coincidir con el espacio y el primer nombre, ya que no podemos saber si es mayúscula o minúscula, solo coincidiremos con ambos: `[a-zA-Z]+\s` o `[aZ]+\s` y también puede ser `[\w]+\s` .

Ahora especificaremos el apellido requerido para obtener solo las líneas que contienen Yaron como apellido (al final de la línea): `\sYaron$` .

Poniendo todo esto junto `^[\d]+\.\s[\w]+\sYaron$` .

Ejemplo en vivo: https://regex101.com/r/nW4fH8/1

Lea Patrones simples a juego en línea: https://riptutorial.com/es/regex/topic/343/patrones-simples-a-juego

Capítulo 16: Personajes ancla: Caret (^)

Observaciones

Terminología

Los siguientes términos también hacen referencia al carácter Caret (^):

- sombrero
- controlar
- estrecho
- cheurón
- acento circunflejo

Uso

Tiene dos usos en expresiones regulares:

- Para denotar el inicio de la línea.
- Si se usa inmediatamente después de un corchete (`[^`), niega el conjunto de caracteres permitidos (es decir, `[123]` significa que se permite el carácter 1, 2 o 3, mientras que la declaración `[^123]` significa cualquier carácter que no sea 1 , 2, o 3 está permitido.

Escape de personajes

Para expresar un cursor sin un significado especial, debe escaparse precediéndolo con una barra invertida; es decir `\^` .

Examples

Comienzo de linea

Cuando el modificador de multilínea `(?m)` está desactivado , `^` coincide solo con el principio de la cadena de entrada:

Para el regex

```
^He
```

Las siguientes cadenas de entrada coinciden:

- `Hedgehog\nFirst line\nLast line`
- `Help me, please`
- `He`

Y las siguientes cadenas de entrada **no** coinciden:

- `First line\nHedgehog\nLast line`
- `IHedgehog`
- `Hedgehog` (por espacios en blanco)

Cuando se activa el modificador multilínea `(?m)` , `^` coincide con el comienzo de cada línea:

```
^He
```

Lo anterior coincidiría con cualquier cadena de entrada que contenga una línea que comience con `He` .

Considerando `\n` como el nuevo carácter de línea, las siguientes líneas coinciden:

- `Hello`
- `First line\nHedgehog\nLast line` (solo segunda línea)
- `My\nText\nIs\nHere` (solo la última línea)

Y las siguientes cadenas de entrada **no** coinciden:

- `Camden Hells Brewery`
- `Helmet` (debido a espacios en blanco)

Coincidencia de líneas vacías usando `^`

Otro caso de uso típico de caret es hacer coincidir líneas vacías (o una cadena vacía si el modificador multilínea está desactivado).

Para hacer coincidir una línea vacía (línea múltiple **en**), se utiliza un símbolo de intercalación junto a `$` que es otro carácter de ancla que representa la posición al final de la línea (Caracteres de Ancla: Dólar ($)). Por lo tanto, la siguiente expresión regular coincidirá con una línea vacía:

```
^$
```

Escapar del personaje de caret

Si necesita usar el carácter `^` en una clase de caracteres (clases de caracteres), póngalo en un lugar que no sea el principio de la clase:

```
[12^3]
```

O escapar de la `^` usando una barra invertida `\` :

```
[\^123]
```

Si quieres hacer coincidir el personaje de caret en sí mismo fuera de una clase de personaje, necesitas escapar de él:

```
\^
```

Esto evita que `^` se interprete como el carácter de anclaje que representa el comienzo de la cadena / línea.

Comparación de inicio de línea de anclaje y comienzo de cadena de anclaje

Si bien muchas personas piensan que `^` significa el inicio de una cadena, en realidad significa el inicio de una línea. Para un inicio real de uso de anclaje de cadena, `\A`

La cadena `hello\nworld` (o más claramente)

```
hello
world
```

Coincidiría con las expresiones regulares `^h` , `^w` y `\Ah` pero no con `\Aw`

Modificador multilínea

Por defecto, caret `^` metacharacter coincide con la **posición** antes del primer carácter en la cadena.

Dada la cadena " **charsequence** " aplicada contra los siguientes patrones: `/^char/` & `/^sequence/` , el motor intentará coincidir de la siguiente manera:

- `/^char/`
    - **^** - charsequence
    - **c** - `c` harsequence
    - **h** - `ch` arsequence
    - **a** - `cha` rsequence
    - **r** - `char` secuencia

  Coincidencia encontrada

- `/^sequence/`
    - **^** - charsequence
    - **s** - charsequence

  Partido no encontrado

El mismo comportamiento se aplicará incluso si la cadena contiene *terminadores de línea* , como `\r?\n` Solo se emparejará la posición al comienzo de la cadena.

Por ejemplo:

```
/^/g
```

\Char \ r \ n
\ r \ n
secuencia

Sin embargo, si necesita hacer coincidir después de cada terminador de línea, tendrá que establecer el modo **multilínea** (`//m` , `(?m)`) dentro de su patrón. Al hacerlo, el carácter de intercalación `^` coincidirá con "el comienzo de cada línea", que corresponde a la posición al comienzo de la cadena y las posiciones **inmediatamente después de** [1] los terminadores de línea.

[1] En algunos tipos (Java, PCRE, ...), `^` no coincidirá después del terminador de línea, si el terminador de línea es el último de la cadena.

Por ejemplo:

```
/^/gm
```

\Char \ r \ n
\ r \ n
secuencia

Algunos de los motores de expresiones regulares que admiten el modificador multilínea:

- Java

```
Pattern pattern = Pattern.compile("(?m)^abc");
Pattern pattern = Pattern.compile("^abc", Pattern.MULTILINE);
```

- .RED

```
var abcRegex = new Regex("(?m)^abc");
var abdRegex = new Regex("^abc", RegexOptions.Multiline)
```

- PCRE

```
/(?m)^abc/
/^abc/m
```

- Python 2 y 3 (módulo de `re` incorporado)

```
abc_regex = re.compile("(?m)^abc");
abc_regex = re.compile("^abc", re.MULTILINE);
```

Lea Personajes ancla: Caret (^) en línea: https://riptutorial.com/es/regex/topic/452/personajes-ancla--caret----

Capítulo 17: Recursion

Observaciones

La recursión está disponible principalmente en sabores compatibles con Perl, tales como:

- Perl
- PCRE
- Oniguruma
- Aumentar

Examples

Repetir todo el patrón

La construcción `(?R)` es equivalente a `(?0)` (o `\g<0>`) - te permite repetir todo el patrón:

```
<(?>[^<>]+|(?R))+>
```

Esto hará coincidir los corchetes angulares correctamente equilibrados con cualquier texto entre los corchetes, como `<a<b>c<d>e>` .

Reclamar en un subpatrón

Puede recurrir a un subpatrón usando las siguientes construcciones (según el tipo), asumiendo que `n` es un número de grupo de captura y `name` el nombre de un grupo de captura.

- `(?n)`
- `\g<n>`
- `\g'0'`
- `(?&name)`
- `\g<name>`
- `\g'name'`
- `(?P>name)`

El siguiente patrón:

```
\[(?<angle><(?&angle)*+>)*\]
```

Coincidirá con el texto como: `[<<><>><>]` - corchetes angulares bien equilibrados entre corchetes. La recursión se utiliza a menudo para la construcción equilibrada de construcciones.

Definiciones de subpattern

La construcción `(?(DEFINE)...)` permite definir subpatrones a los que puede hacer referencia más adelante mediante recursión. Cuando se encuentre en el patrón, *no* se comparará con.

Este grupo debe contener definiciones de subpatrones con nombre, a las que solo se podrá acceder mediante recursión. Puedes definir las gramáticas de esta manera:

```
(?x) # ignore pattern whitespace
(?(DEFINE)
  (?<string> ".*?" )
  (?<number> \d+ )
  (?<value>
    \s* (?:
        (?&string)
      | (?&number)
      | (?&list)
    ) \s*
  )
  (?<list> \[ (?&value) (?: , (?&value) )* \] )
)
^(?&value)$
```

Este patrón validará texto como el siguiente:

```
[42, "abc", ["foo", "bar"], 10]
```

Observe cómo una lista puede contener uno o más valores, y un valor puede ser una lista.

Referencias de grupo relativas

Se puede hacer referencia a los subpatrones con su número de grupo *relativo* :

- `(?-1)` entrará en el grupo *anterior*
- `(?+1)` retrocederá en el *siguiente* grupo

También se puede utilizar con la sintaxis `\g<N>` .

Referencias en recursiones (PCRE)

En PCRE, los grupos emparejados utilizados para las referencias inversas antes de una recursión se mantienen en la recursión. Pero después de la recursión, todos se restablecen a lo que eran antes de entrar. En otras palabras, los grupos emparejados en la recursión son todos olvidados.

Por ejemplo:

```
(?J)(?(DEFINE)(\g{a}(?<a>b)\g{a}))(?<a>a)\g{a}(?1)\g{a}
```

partidos

```
aaabba
```

Las recursiones son atómicas (PCRE)

En PCRE, no hace un trackback después de que se encuentra la primera coincidencia para una

recursión. Asi que

```
(?(DEFINE)(aaa|aa|a))(?1)ab
```

no coincide

```
aab
```

porque después de coincidir con `aa` en la recursión, nunca volverá a intentar coincidir con solo `a` .

Lea Recursion en línea: https://riptutorial.com/es/regex/topic/739/recursion

Capítulo 18: Referencia posterior

Examples

Lo esencial

Las referencias anteriores se utilizan para hacer coincidir el mismo texto que un grupo de captura coincidió previamente. Esto ayuda tanto a reutilizar partes anteriores de su patrón como a asegurar que dos partes de una cadena coinciden.

Por ejemplo, si está intentando verificar que una cadena tiene un dígito de cero a nueve, un separador, como guiones, barras o incluso espacios, una letra minúscula, otro separador, luego otro dígito de cero a nueve, podría use un regex como este:

```
[0-9][-/ ][a-z][-/ ][0-9]
```

Esto coincidiría con `1-a-4` , pero *también* coincidiría con `1-a/4` o `1 a-4` . Si queremos que los separadores coincidan, podemos usar un grupo de captura y una referencia posterior. La referencia posterior verá la coincidencia encontrada en el grupo de captura indicado y asegurará que la ubicación de la referencia posterior coincida exactamente.

Usando nuestro mismo ejemplo, la expresión regular se convertiría en:

```
[0-9]([-/ ])[a-z]\1[0-9]
```

El `\1` denota el primer grupo de captura en el patrón. Con este pequeño cambio, la expresión regular ahora combina `1-a-4` o `1 a 4` pero no `1 a-4` o `1-a/4` .

El número a usar para su referencia de respaldo depende de la ubicación de su grupo de captura. El número puede ser de uno a nueve y se puede encontrar contando sus grupos de captura.

```
([0-9])([-/ ])[a-z][-/ ]([0-9])
|--1--||--2--|           |--3--|
```

Los grupos de captura anidados cambian este conteo ligeramente. Primero cuenta el grupo de captura exterior, luego el siguiente nivel y continúa hasta que abandonas el nido:

```
(([0-9])([-/ ]))([a-z])
 |--2--||--3--|
|-------1------||--4--|
```

Referencias ambiguas

Problema: necesita hacer coincidir el texto de un determinado formato, por ejemplo:

```
1-a-0
```

```
6/p/0
4 g 0
```

Eso es un dígito, un separador (uno de `-` , `/` , o un espacio), una letra, el mismo separador y un cero.

Solución ingenua: adaptando la expresión regular del ejemplo de Conceptos básicos , puede encontrar esta expresión regular:

```
[0-9]([-/ ])[a-z]\10
```

Pero eso probablemente no funcionará. La mayoría de los tipos de expresiones regulares admiten más de nueve grupos de captura, y muy pocos de ellos son lo suficientemente inteligentes como para darse cuenta de que, dado que solo hay un grupo de captura, `\10` debe ser una referencia inversa al grupo 1 seguido de un `0` literal. La mayoría de los sabores lo tratarán como una referencia inversa al grupo 10. Algunos de ellos lanzarán una excepción porque no hay un grupo 10; el resto simplemente no coincidirá.

Hay varias formas de evitar este problema. Una es usar grupos con nombre (y referencias con nombre):

```
[0-9](?<sep>[-/ ])[a-z]\k<sep>0
```

Si su lenguaje de expresiones regulares lo admite, el formato `\g{n}` (donde `n` es un número) puede incluir el número de referencia inversa entre paréntesis para separarlo de cualquier dígito después de él:

```
[0-9]([-/ ])[a-z]\g{1}0
```

Otra forma es usar el formato de expresiones regulares extendido, separando los elementos con espacios en blanco insignificantes (en Java, tendrá que escapar del espacio entre paréntesis):

```
(?x) [0-9] ([-/ ]) [a-z] \1 0
```

Si su versión de expresiones regulares no es compatible con esas características, puede agregar una sintaxis innecesaria pero inofensiva, como un grupo que no captura:

```
[0-9]([-/ ])[a-z](?:\1)0
```

... o un cuantificador ficticio (esta es posiblemente la única circunstancia en la que `{1}` es útil):

```
[0-9]([-/ ])[a-z]\1{1}0
```

Lea Referencia posterior en línea: https://riptutorial.com/es/regex/topic/4072/referencia-posterior

Capítulo 19: Restablecer partido: \ K

Observaciones

Regex101 define la funcionalidad \ K como:

> `\K` restablece el punto de partida de la coincidencia informada. Cualquier carácter consumido previamente ya no se incluye en la partida final

La secuencia de escape `\K` es compatible con varios motores, idiomas o herramientas, tales como:

- impulsar (desde ???)
- grep -P ← *usa PCRE*
- Oniguruma (desde 5.13.3)
- PCRE (desde 7.2)
- Perl (desde la 5.10.0)
- PHP (desde 5.2.4)
- Ruby (desde 2.0.0)

... y (hasta ahora) no soportado por:

- .RED
- awk
- golpetazo
- ÑU
- UCI
- Java
- Javascript
- Bloc de notas ++
- C objetivo
- POSIX
- Pitón
- Qt / QRegExp
- sed
- Tcl
- empuje
- XML
- XPath

Examples

Buscar y reemplazar utilizando el operador \ K

Dado el texto:

foo: bar

Me gustaría reemplazar cualquier cosa después de "foo:" con "baz", pero quiero mantener "foo:". Esto se podría hacer con un grupo de captura como este:

```
s/(foo: ).*/$1baz/
```

Que resulta en el texto:

foo: baz

Ejemplo 1

o podríamos usar `\K` , que "olvida" todo lo que ha igualado previamente, con un patrón como este:

```
s/foo: \K.*/baz/
```

La expresión regular coincide con "foo:" y luego se encuentra con la `\K` , los caracteres coincidentes previamente se dan por sentados y la expresión regular los deja, lo que significa que solo la cadena que coincide con `.*` Se reemplazará por "baz", lo que da como resultado el texto:

foo: baz

Ejemplo 2

Lea Restablecer partido: \ K en línea: https://riptutorial.com/es/regex/topic/1338/restablecer-partido----k

Capítulo 20: Retroceso

Examples

¿Qué causa el Backtracking?

Para encontrar una coincidencia, el motor de expresiones regulares consumirá caracteres uno por uno. Cuando comienza una coincidencia parcial, el motor recordará la posición de inicio para que pueda retroceder en caso de que los siguientes caracteres no completen la coincidencia.

- Si la coincidencia es completa, no hay retroceso.
- Si la coincidencia no está completa, el motor retrocederá la cadena (como cuando rebobinó una cinta vieja) para tratar de encontrar una coincidencia completa.

Por ejemplo: `\d{3}[az]{2}` contra la cadena `abc123def` buscará como tal:

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does match \d (third one)
abc123def
      ^ Does match [a-z] (first one)
abc123def
       ^ Does match [a-z] (second one)
           MATCH FOUND
```

Ahora cambiemos la expresión regular a `\d{2}[az]{2}` contra la misma cadena (`abc123def`):

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does not match [a-z]
abc123def
    ^ BACKTRACK to catch \d{2} => (23)
abc123def
      ^ Does match [a-z] (first one)
abc123def
```

```
      ^ Does match [a-z] (second one)
        MATCH FOUND
```

¿Por qué el retroceso puede ser una trampa?

El retroceso puede ser causado por cuantificadores opcionales o construcciones de alternancia, porque el motor de expresiones regulares intentará explorar cada ruta. Si ejecuta la expresión regular `a+b` contra `aaaaaaaaaaaaa` no hay coincidencia y el motor lo encontrará bastante rápido.

Pero si cambia la expresión regular a `(aa*)+b` la cantidad de combinaciones crecerá bastante rápido, y la mayoría de los motores (no optimizados) intentarán explorar todos los caminos y tomarán una eternidad para tratar de encontrar una coincidencia o lanzar una excepción de tiempo de espera. Esto se llama **retroceso catastrófico** .

Por supuesto, `(aa*)+b` parece un error de novato, pero está aquí para ilustrar el punto y, a veces, terminará con el mismo problema pero con patrones más complicados.

Un caso más extremo de retroceso catastrófico ocurre con la expresión regular `(x+x+)+y` (probablemente lo haya visto antes aquí y aquí), que necesita un tiempo exponencial para descubrir que una cadena que contiene `x` s y nada más (por ejemplo, `xxxxxxxxxxxxxxxxxx`) no coinciden.

¿Cómo evitarlo?

Sea lo más específico posible, reduzca lo más posible los caminos posibles. Tenga en cuenta que algunos comparadores de expresiones regulares no son vulnerables al retroceso, como los incluidos en `awk` o `grep` porque se basan en Thompson NFA .

Lea Retroceso en línea: https://riptutorial.com/es/regex/topic/977/retroceso

Capítulo 21: Sustituciones con expresiones regulares

Parámetros

En línea	Descripción
$ numero	Sustituye la subcadena emparejada por número de grupo.
${nombre}	Sustituye la subcadena que coincide con un nombre de grupo nombrado .
$$	Se escapó el carácter '$' en la cadena de resultado (reemplazo).
$ Y *($ 0)*	Reemplaza con toda la cadena emparejada.
$ + *($ &)*	Sustituye el texto coincidente al último grupo capturado.
$ `	Sustituye todo el texto coincidente con cada texto no coincidente antes de la coincidencia.
PS	Sustituye todo el texto coincidente con cada texto no coincidente después de la coincidencia.
PS	Sustituye todo el texto coincidente a la cadena completa.
Nota:	*Los* términos en *cursiva* significan que las cadenas son volátiles (puede variar dependiendo de su sabor regex).

Examples

Conceptos básicos de la sustitución

Una de las formas más comunes y útiles para reemplazar texto con expresiones regulares es mediante el uso de grupos de captura .
O incluso un grupo de captura con nombre , como referencia para almacenar o reemplazar los datos.

Hay dos términos bastante parecidos en los documentos de expresiones regulares, por lo que puede ser importante nunca mezclar **sustituciones** (es decir, `$1`) con **referencias** (es decir, `\1`). Los términos de sustitución se utilizan en un texto de reemplazo; Backreferences, en la expresión Regex pura. Aunque algunos lenguajes de programación aceptan ambas sustituciones, no es alentador.

Digamos que tenemos esta expresión regular: `/hello(\s+)world/i` . Cuando se hace referencia a

`$number` (en este caso, `$1`), los espacios en blanco que coincidan con `\s+` serán reemplazados en su lugar.
El mismo resultado se expondrá con la expresión regular: `/hello(?<spaces>\s+)world/i` . Y como tenemos un grupo nombrado aquí, también podemos usar `${spaces}` .

En este mismo ejemplo, también podemos usar `$0` o `$&` (**Nota:** `$&` puede usarse como `$+` lugar, lo que significa recuperar el **último** grupo de captura en otros motores de expresiones regulares), dependiendo del sabor de expresiones regulares con el que esté trabajando, para obtener todo el texto emparejado. (Es decir, `$&` devolverá `hEllo woRld` para la cadena: `hEllo woRld of Regex!`)

Eche un vistazo a este sencillo ejemplo de sustitución utilizando la cita adaptada de John Lennon utilizando el `$number` y la sintaxis `${name}` :

Ejemplo de grupo de captura simple:

```
/(Happy)\./g
```

Test String

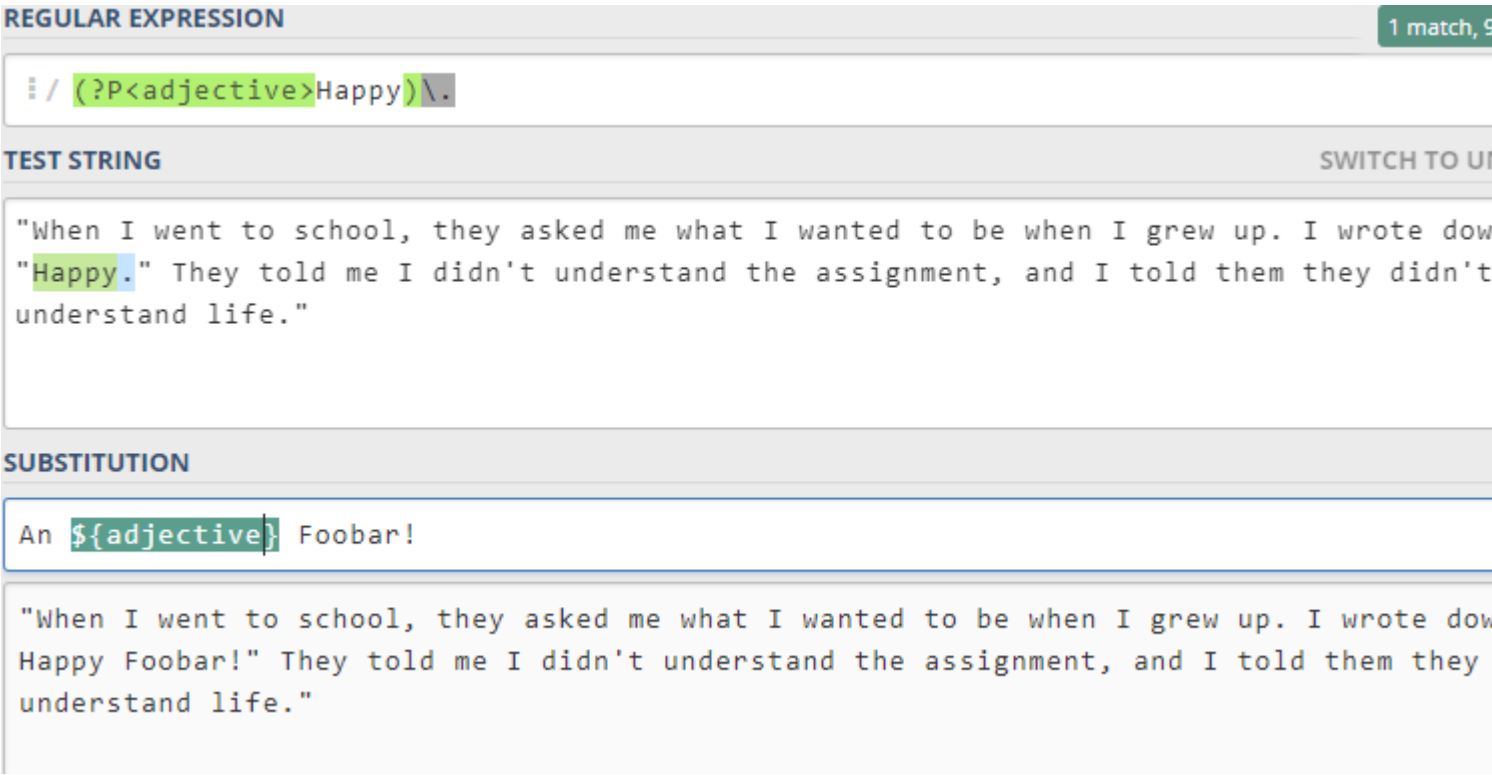
```
"When I went to school, they asked me what I wanted to be when I grew up. I w
me I didn't understand the assignment, and I told them they didn't understand
```

Substitution

```
An $1 Foobar!
```

```
"When I went to school, they asked me what I wanted to be when I grew up. I w
They told me I didn't understand the assignment, and I told them they didn't
```

Ejemplo de grupo de captura con nombre:



Reemplazo avanzado

Algunos lenguajes de programación tienen sus propias peculiaridades Regex, por ejemplo, el término `$+` (en C #, Perl, VB, etc.) que reemplaza el texto coincidente con el último grupo capturado.

Ejemplo:

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//       The dog jumped over the fence.
```

Ejemplo de la red de desarrolladores de Microsoft Official [1]

Otros términos de sustitución raros son `$`` y `$'` :

`$`` = Reemplaza coincidencias con el texto **antes de** la cadena correspondiente
`$'` = Reemplaza coincidencias con el texto **después de** la cadena correspondiente

Debido a este hecho, estas cadenas de reemplazo deben hacer su trabajo así:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$`"
Output: "part1part1part3" //Note that part2 was replaced with part1, due &` term
--------------------------------------------------------------------------------
Regex: /part2/
Input: "part1part2part3"
Replacement: "$'"
Output: "part1part3part3" //Note that part2 was replaced with part3, due &' term
```

Aquí hay un ejemplo de estas sustituciones trabajando en una pieza de javascript:

```
var rgx = /middle/;
var text = "Your story must have a beginning, middle, and end"
console.log(text.replace(rgx, "$`"));
//Logs: "Your story must have a beginning, Your story must have a beginning, , and end"
console.log(text.replace(rgx, "$'"))
//Logs: "Your story must have a beginning, , and end, and end"
```

También está el término `$_` que recupera todo el texto coincidente en su lugar:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$_"
Output: "part1part1part2part3part3" //Note that part2 was replaced with part1part2part3,
                                                                    // due $_ term
```

Convertir esto a VB nos daría esto:

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "ABC123DEF456"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$_"
      Console.WriteLine("Original string:          {0}", input)
      Console.WriteLine("String with substitution: {0}", _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

Ejemplo de la red de desarrolladores de Microsoft Official [2]

Y el último término de sustitución, pero no menos importante, es `$$` , que traducido a una expresión de expresión regular sería el mismo que `\$` (una versión escapada del **$** literal).

Si desea hacer coincidir una cadena como esta: `USD: $3.99` por ejemplo, y desea almacenar la `3.99` , pero reemplazarla por `$3.99` con una sola expresión regular, puede usar:

```
Regex: /USD:\s+\$([\d.]+)/
Input: "USD: $3.99"
Replacement: "$$$1"
To Store: "$1"
Output: "$3.99"
Stored: "3.99"
```

Si desea probar esto con Javascript, puede usar el código:

```
var rgx = /USD:\s+\$([\d.]+)/;
var text = "USD: $3.99";
var stored = parseFloat(rgx.exec(text)[1]);
console.log(stored); //Logs 3.99
console.log(text.replace(rgx, "$$$1")); //Logs $3.99
```

Referencias

[1]: Sustituyendo al último grupo capturado
[2]: Sustituyendo toda la cadena de entrada

Lea Sustituciones con expresiones regulares en línea:
https://riptutorial.com/es/regex/topic/9852/sustituciones-con-expresiones-regulares

Capítulo 22: Tipos de motores de expresiones regulares

Examples

NFA

Un motor NFA (automatización finita no determinista) es *impulsado por el patrón* .

Principio

El patrón regex se analiza en un árbol.

El puntero de *posición actual* se establece en el inicio de la cadena de entrada y se intenta una coincidencia en esta posición. Si la coincidencia es fais, la posición se incrementa al siguiente carácter en la cadena y se intenta otra coincidencia desde esta posición. Este proceso se repite hasta que se encuentra una coincidencia o se llega al final de la cadena de entrada.

Para cada intento de partido

El algoritmo funciona realizando un recorrido del árbol de patrones para una posición inicial dada. A medida que avanza a través del árbol, actualiza la *posición de entrada actual* al consumir caracteres coincidentes.

Si el algoritmo encuentra un nodo de árbol que no coincide con la cadena de entrada en la posición actual, tendrá que *retroceder* . Esto se realiza volviendo al nodo principal en el árbol, restableciendo la posición de entrada actual al valor que tenía al ingresar al nodo principal e intentando la siguiente rama alternativa.

Si el algoritmo logra salir del árbol, informa una coincidencia exitosa. De lo contrario, cuando se han probado todas las posibilidades, la coincidencia falla.

Optimizaciones

Los motores Regex suelen aplicar algunas optimizaciones para un mejor rendimiento. Por ejemplo, si determinan que una coincidencia debe comenzar con un carácter dado, intentarán una coincidencia solo en aquellas posiciones en la cadena de entrada donde aparece ese carácter.

Ejemplo

`abeacab` coincidir `a(b|c)a` con la cadena de entrada `abeacab` :

El árbol de patrones podría ser algo así como:

```
CONCATENATION
    EXACT: a
    ALTERNATION
        EXACT: b
        EXACT: c
    EXACT: a
```

El partido se procesa de la siguiente manera:

```
a(b|c)a      abeacab
^            ^
```

`a` se encuentra en la cadena de entrada, consúmala y continúe con el siguiente elemento del árbol de patrones: la alternancia. Prueba la primera posibilidad: una `b` exacta.

```
a(b|c)a      abeacab
  ^           ^
```

Se encuentra `b` , por lo que la alternancia tiene éxito, consúmala y continúe con el siguiente elemento en la concatenación: una exacta `a` :

```
a(b|c)a      abeacab
      ^        ^
```

`a` *no se* encuentra en la posición esperada. Retroceda a la alternancia, restablezca la posición de entrada al valor que tenía al ingresar la alternancia por primera vez y pruebe la *segunda* alternativa:

```
a(b|c)a      abeacab
    ^         ^
```

`c` *no se* encuentra en esta posición. Retroceder a la concatenación. No hay otras posibilidades para probar en este punto, por lo que no hay coincidencia al comienzo de la cadena.

Intente un segundo partido en la siguiente posición de entrada:

```
a(b|c)a      abeacab
^             ^
```

`a` *no* coincide allí. Intenta otro partido en la siguiente posición:

```
a(b|c)a      abeacab
^              ^
```

Tampoco hay suerte. Avanzar a la siguiente posición.

```
a(b|c)a      abeacab
^               ^
```

`a` coincide, así que consúmelo y entra en la alternancia:

```
a(b|c)a      abeacab
  ^               ^
```

`b` no coincide. Intenta la segunda alternativa:

```
a(b|c)a      abeacab
    ^           ^
```

`c` coincidencias, así que consuma y avance al siguiente elemento en la concatenación:

```
a(b|c)a      abeacab
      ^           ^
```

`a` coincidencia, y el final del árbol se ha alcanzado. Reportar una coincidencia exitosa:

```
a(b|c)a      abeacab
               \_/
```

DFA

Un motor DFA (Deterministic Finite Automaton) es *impulsado por la entrada* .

Principio

El algoritmo escanea a través de la cadena de entrada *una vez* , y recuerda todas las rutas posibles en la expresión regular que podrían coincidir. Por ejemplo, cuando se encuentra una alternancia en el patrón, se crean dos nuevas rutas y se intentan de forma independiente. Cuando una ruta dada no coincide, se elimina de las posibilidades establecidas.

Trascendencia

El tiempo coincidente está limitado por el tamaño de la cadena de entrada. No hay retroceso, y el motor puede encontrar varias coincidencias simultáneamente, incluso coincidencias superpuestas.

El principal inconveniente de este método es el reducido conjunto de características que puede soportar el motor, en comparación con el tipo de motor NFA.

Ejemplo

Empareja `a(b|c)a` contra `abadaca` :

```
abadaca      a(b|c)a
^            ^        Attempt 1      ==> CONTINUE

abadaca      a(b|c)a
 ^           ^        Attempt 2      ==> FAIL
               ^      Attempt 1.1    ==> CONTINUE
                  ^   Attempt 1.2    ==> FAIL

abadaca      a(b|c)a
  ^          ^        Attempt 3      ==> CONTINUE
                    ^  Attempt 1.1    ==> MATCH

abadaca      a(b|c)a
   ^         ^        Attempt 4      ==> FAIL
               ^      Attempt 3.1    ==> FAIL
                  ^   Attempt 3.2    ==> FAIL

abadaca      a(b|c)a
    ^        ^        Attempt 5      ==> CONTINUE

abadaca      a(b|c)a
     ^       ^        Attempt 6      ==> FAIL
               ^      Attempt 5.1    ==> FAIL
                  ^   Attempt 5.2    ==> CONTINUE

abadaca      a(b|c)a
      ^      ^        Attempt 7      ==> CONTINUE
                    ^  Attempt 5.2    ==> MATCH

abadaca      a(b|c)a
       ^       ^      Attempt 7.1    ==> FAIL
                  ^   Attempt 7.2    ==> FAIL
```

Lea Tipos de motores de expresiones regulares en línea:
https://riptutorial.com/es/regex/topic/2861/tipos-de-motores-de-expresiones-regulares

Capítulo 23: Trampas Regex

Examples

¿Por qué el punto (.) No coincide con el carácter de nueva línea ("\ n")?

`.*` en regex básicamente significa "capturar **todo** hasta el final de la entrada".

Entonces, para cadenas simples, como `hello world .*` Funciona perfectamente. Pero si tiene una cadena que representa, por ejemplo, líneas en un archivo, estas líneas estarán separadas por un *separador de línea* , como `\n` (nueva línea) en sistemas similares a Unix y `\r\n` (retorno de carro y nueva línea) en Windows

Por defecto en la mayoría de los motores de expresiones regulares, `.` **no** coincide con los caracteres de nueva línea, por lo que la coincidencia se detiene al final de cada *línea lógica* . Si quieres `.` para hacer coincidir **realmente** todo, incluidas las nuevas líneas, debe habilitar el modo "punto-coincidencias-todo" en el motor de `re.DOTALL` regulares de su elección (por ejemplo, agregue `re.DOTALL` flag en Python, o `/s` en PCRE).

¿Por qué una expresión regular omite algunos paréntesis / paréntesis de cierre y los combina después?

Considera este ejemplo:

> Entró en el café "Dostoevski" y dijo: "Buenas noches".

Aquí tenemos dos conjuntos de citas. Asumamos que queremos hacer coincidir ambas, de modo que nuestra expresión regular coincida con `"Dostoevski"` **y** `"Good evening."`

Al principio, podrías estar tentado a hacerlo simple:

```
".*"  # matches a quote, then any characters until the next quote
```

Pero no funciona: coincide con la primera cita en `"Dostoevski"` y **hasta** la cita de cierre en `"Good evening."` , incluyendo el `and said:` parte. Demo regex101

¿Por qué sucedió?

Esto sucede porque el motor de expresiones regulares, cuando se encuentra `.*` , "Consume" toda la entrada hasta el final. Luego, debe coincidir con la final `"` . Por lo tanto," retrocede "desde el final de la coincidencia, soltando el texto coincidente hasta que se encuentre el primer `"` y, por supuesto, es el último `"` en la coincidencia , al final de la parte `"Good evening."` .

¿Cómo evitar esto y coincidir exactamente con las primeras

citas?

Use `[^"]*` . No come toda la entrada, solo hasta la primera `"` , según sea necesario. Demo regex101

Lea Trampas Regex en línea: https://riptutorial.com/es/regex/topic/10747/trampas-regex

Capítulo 24: Útil escaparate Regex

Examples

Emparejar una fecha

Debe recordar que la expresión regular fue diseñada para coincidir con una fecha (o no). Decir que una fecha es *válida* es una lucha mucho más complicada, ya que requerirá una gran cantidad de manejo de excepciones (consulte las condiciones del año bisiesto).

Comencemos por hacer coincidir el mes (1 - 12) con un 0 inicial opcional:

```
0?[1-9]|1[0-2]
```

Para emparejar el día, también con un 0 inicial opcional:

```
0?[1-9]|[12][0-9]|3[01]
```

Y para coincidir con el año (asumamos el rango 1900 - 2999):

```
(?:19|20)[0-9]{2}
```

El separador puede ser un espacio, un guión, una barra oblicua, un espacio vacío, etc. Siéntase libre de agregar cualquier cosa que crea que pueda usarse como separador:

```
[-\\/ ]?
```

Ahora concatena todo y obtienes:

```
(0?[1-9]|1[0-2])[-\\/ ]?(0?[1-9]|[12][0-9]|3[01])[-/ ]?(?:19|20)[0-9]{2} // MMDDYYYY
(0?[1-9]|[12][0-9]|3[01])[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Si desea ser un poco más pedante, puede usar una referencia posterior para asegurarse de que los dos separadores sean iguales:

```
(0?[1-9]|1[0-2])([-\\/ ]?)(0?[1-9]|[12][0-9]|3[01])\2(?:19|20)[0-9]{2} // MMDDYYYY
                                                   ^ refer to [-/ ]
(0?[1-9]|[12][0-9]|3[01])([-\\/ ]?)(0?[1-9]|1[0-2])\2(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}([-\\/ ]?)(0?[1-9]|1[0-2])\2(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Coincidir con una dirección de correo electrónico

> Hacer coincidir una dirección de correo electrónico dentro de una cadena es una tarea difícil, porque la especificación que la define, el RFC2822 , es compleja, lo que dificulta su implementación como una expresión regular. Para obtener más detalles sobre por

> qué no es una buena idea hacer coincidir un correo electrónico con una expresión regular, consulte el ejemplo antipattern cuando no use una expresión regular: para correos electrónicos coincidentes . El mejor consejo para tomar nota de esa página es usar una biblioteca revisada por pares y ampliamente en su idioma favorito para implementar esto.

Validar un formato de dirección de correo electrónico

Cuando necesite validar rápidamente una entrada para asegurarse de que se *vea como* un correo electrónico, la mejor opción es que sea sencillo:

```
^\S{1,}@\S{2,}\.\S{2,}$
```

Esa expresión regular comprobará que la dirección de correo es una secuencia de caracteres de longitud superior a uno sin espacios separados por espacios, seguida de una `@` , seguida de dos secuencias de caracteres de espacios no espacios de dos o más caracteres separados por a `.` . No es perfecto y puede validar direcciones no válidas (según el formato), pero lo más importante es que no invalida las direcciones válidas.

Verifica que la dirección exista

La única forma confiable de verificar que un correo electrónico es válido es verificar su existencia. Solía haber el `VRFY` SMTP que fue diseñado para ese propósito, pero lamentablemente, después de ser abusado por los spammers, ahora ya no está disponible .

Por lo tanto, la única forma de verificar que el correo es válido y existe es enviar un correo electrónico a esa dirección.

Enormes alternativas Regex

Sin embargo, no es imposible validar un correo electrónico de dirección utilizando una expresión regular. El único problema es que cuanto más cerca de la especificación estén esas expresiones regulares, más grandes serán y, por consiguiente, son increíblemente difíciles de leer y mantener. A continuación, encontrará ejemplos de expresiones regulares más precisas que se utilizan en algunas bibliotecas.

> Las siguientes **expresiones** regulares se proporcionan para fines de documentación y aprendizaje, y copiarlas en su código es una mala idea. En su lugar, use esa biblioteca directamente, de modo que puede confiar en el código de flujo ascendente y los desarrolladores pares para mantener actualizado y actualizado su código de análisis de correo electrónico.

Módulo de coincidencia de direcciones Perl

Los mejores ejemplos de expresiones regulares de este tipo se encuentran en las bibliotecas estándar de algunos idiomas. Por ejemplo, hay uno del módulo `RFC::RFC822::Address` en la

biblioteca de Perl que intenta ser lo más preciso posible de acuerdo con el RFC. Para su curiosidad, puede encontrar una versión de esa expresión regular en esta URL , que se ha generado a partir de la gramática, y si está tentado de copiarla, aquí hay una cita del autor de la expresión regular:

> " *No mantengo la expresión regular [vinculada]. Puede que haya errores que ya se hayan solucionado en el módulo Perl* " .

Módulo de coincidencia de direcciones .Net

Otra variante más corta es la que usa la biblioteca estándar .Net en el módulo `EmailAddressAttribute` :

```
^((([a-z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])+(\.([a-
z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])+)*)|((\x22)((((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(([\x01-\x08\x0b\x0c\x0e-
\x1f\x7f]|\x21|[\x23-\x5b]|[\x5d-\x7e]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(\\([\x01-
\x09\x0b\x0c\x0d-\x7f]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])))*(((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(\x22)))@((([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])|(([a-z]|\d|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-
|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])*([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])))\.)+(([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(([a-z]|[\u00A0-
\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])*([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])))\.?$
```

Pero incluso si es *más corto* , todavía es demasiado grande para ser legible y fácil de mantener.

Módulo de Ruby Address Match

En ruby, una composición de expresiones regulares se está utilizando en el módulo rfc822 para que coincida con una dirección. Esta es una buena idea, ya que en el caso de que se encuentren errores, será más fácil identificar la parte de expresiones regulares para cambiarla y corregirla.

Módulo de coincidencia de direcciones Python

Como ejemplo de contador, el módulo de análisis de correo electrónico de Python no está utilizando una expresión regular, sino que lo implementa utilizando un analizador.

Coincidir con un número de teléfono

Aquí le indicamos cómo hacer coincidir un código de prefijo (a `+` o (00), luego un número del 1 al 1939, con un espacio opcional):
Esto no busca un prefijo *válido* sino algo que podría ser un prefijo. Ver la lista completa de prefijos

```
(?:00|\+)?[0-9]{4}
```

Luego, como la longitud total del número de teléfono es, como máximo, 15, podemos buscar hasta 14 dígitos:
Se gasta al menos 1 dígito para el prefijo

```
[0-9]{1,14}
```

Los números pueden contener espacios, puntos o guiones y se pueden agrupar por 2 o 3.

```
(?:[ .-][0-9]{3}){1,5}
```

Con el prefijo opcional:

```
(?:(?:00|\+)?[0-9]{4})?(?:[ .-][0-9]{3}){1,5}
```

Si desea hacer coincidir un formato de país específico, puede utilizar esta consulta de búsqueda y agregar el país, la pregunta ya ha sido hecha.

Coincidir con una dirección IP

IPv4

Para coincidir con el formato de la dirección IPv4, debe verificar los números `[0-9]{1,3}` tres veces `{3}` separados por puntos `\.` y terminando con otro número.

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

Esta expresión regular es demasiado simple: si desea que sea precisa, debe verificar que los números estén entre `0` y `255` , con la expresión regular anterior que acepta `444` en cualquier posición. ¿Quiere verificar 250-255 con `25[0-5]` , o cualquier otro valor 200 `2[0-4][0-9]` , o cualquier valor 100 o menos con `[01]?[0-9][0-9]` . Desea comprobar que va seguido de un punto `\.` tres veces `{3}` y luego una vez sin un punto.

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

IPv6

Las direcciones IPv6 se presentan en forma de 8 palabras, hexagonales de 16 bits delimitados con los dos puntos (`:` carácter). En este caso, verificamos 7 palabras seguidas de dos puntos, seguidas de una que no lo sea. Si una palabra tiene ceros iniciales, se *pueden* truncar, lo que significa que cada palabra puede contener entre 1 y 4 dígitos hexadecimales.

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$
```

Esto, sin embargo, es insuficiente. Como las direcciones IPv6 pueden volverse bastante "complicadas", la norma especifica que las palabras de solo cero pueden reemplazarse por `::` . Esto solo se puede hacer una vez en una dirección (para cualquier lugar entre 1 y 7 palabras consecutivas), ya que de otro modo sería indeterminado. Esto produce una serie de variaciones (bastante desagradables):

```
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Ahora, al juntarlo todo (usando alternancia) se obtiene:

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$|
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Asegúrate de escribirlo en modo multilínea y con un montón de comentarios para que quienquiera que tenga la tarea inevitable de averiguar qué significa esto no te persiga con un objeto contundente.

Validar una cadena de tiempo de 12 horas y 24 horas

Para un formato de 12 horas se puede usar:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9]\s*[ap]m$
```

Dónde

- `(?:0?[0-9]|1[0-2])` es la hora
- `[-:]` es el separador, que puede ajustarse para adaptarse a sus necesidades
- `[0-5][0-9]` es el minuto
- `\s*[ap]m` siguió cualquier número de caracteres de espacio en blanco, y `am` o `pm`

Si necesitas los segundos:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9][-:][0-5][0-9]\s*[ap]m$
```

Para un formato de 24 horas:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9]$
```

Dónde:

- `(?:[01][0-9]|2[0-3])` es la hora
- `[-:h]` el separador, que puede ajustarse para adaptarse a sus necesidades

- `[0-5][0-9]` es el minuto

Con los segundos:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9][-:m][0-5][0-9]$
```

Donde `[-:m]` es un segundo separador, reemplazando la `h` por horas con una `m` por minutos, y `[0-5][0-9]` es el segundo.

Código postal del Reino Unido del partido

Regex para hacer coincidir los códigos postales en el Reino Unido

El formato es el siguiente, donde A significa una letra y 9 un dígito:

Formato	Cobertura	Ejemplo
Célula	Célula	
AA9A 9AA	Área de código postal de WC; EC1 – EC4, NW1W, SE1P, SW1	EC1A 1BB
A9A 9AA	E1W, N1C, N1P	W1A 0AX
A9 9AA, A99 9AA	B, E, G, L, M, N, S, W	M1 1AE, B33 8º
AA9 9AA, AA99 9AA	Todos los demás códigos postales	CR2 6XH, DN55 1PT

```
(GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY])))) [0-9][A-Z-[CIKMOV]]{2})
```

Donde primera parte:

```
 (GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY]))))
```

Segundo:

```
[0-9][A-Z-[CIKMOV]]{2})
```

Lea Útil escaparate Regex en línea: https://riptutorial.com/es/regex/topic/3605/util-escaparate-regex

Capítulo 25: Validación de contraseñas de expresiones regulares

Examples

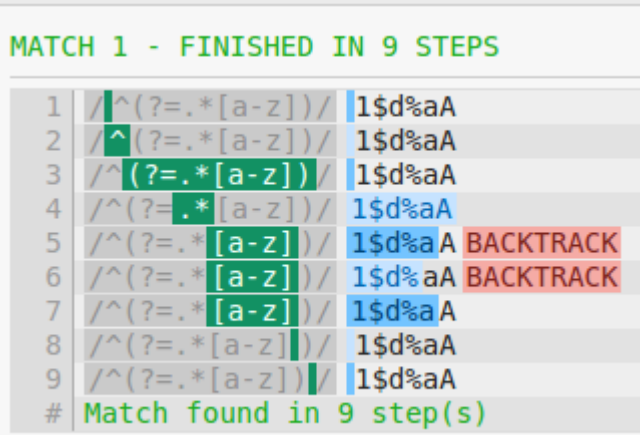
Una contraseña que contiene al menos 1 mayúscula, 1 minúscula, 1 dígito, 1 carácter especial y tiene una longitud de al menos 10

Como los caracteres / dígitos pueden estar en cualquier lugar dentro de la cadena, requerimos lookaheads. Lookaheads son de `zero width` lo que significa que no consumen ninguna cadena. En palabras simples, la posición de verificación se restablece a la posición original después de que se cumple cada condición de búsqueda anticipada.

Supuesto : - *Considerar los caracteres sin palabras como especiales*

```
^(?=.{10,}$)(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*\W).*$
```

Antes de continuar con la explicación, veamos cómo funciona la expresión regular `^(?=.*[az])` (la *longitud no se considera aquí*) en la cadena `1$d%aA`



Crédito de la imagen : - **https://regex101.com/**

Cosas para notar

- La comprobación se inicia desde el principio de la cadena debido a la etiqueta de anclaje `^` .
- La posición de verificación se restablece al inicio después de que se cumpla la condición de búsqueda anticipada.

Desglose regex

```
^ #Starting of string
 (?=.{10,}$) #Check there is at least 10 characters in the string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[a-z]) #Check if there is at least one lowercase in string.
```

```
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[A-Z]) #Check if there is at least one uppercase in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*[0-9]) #Check if there is at least one digit in string.
            #As this is lookahead the position of checking will reset to starting again
 (?=.*\W) #Check if there is at least one special character in string.
          #As this is lookahead the position of checking will reset to starting again
.*$ #Capture the entire string if all the condition of lookahead is met. This is not required
if only validation is needed
```

También podemos usar la versión *no codiciosa* de la expresión regular anterior.

```
^(?=.{10,}$)(?=.*?[a-z])(?=.*?[A-Z])(?=.*?[0-9])(?=.*?\W).*$
```

Una contraseña que contiene al menos 2 mayúsculas, 1 minúscula, 2 dígitos y tiene una longitud de al menos 10

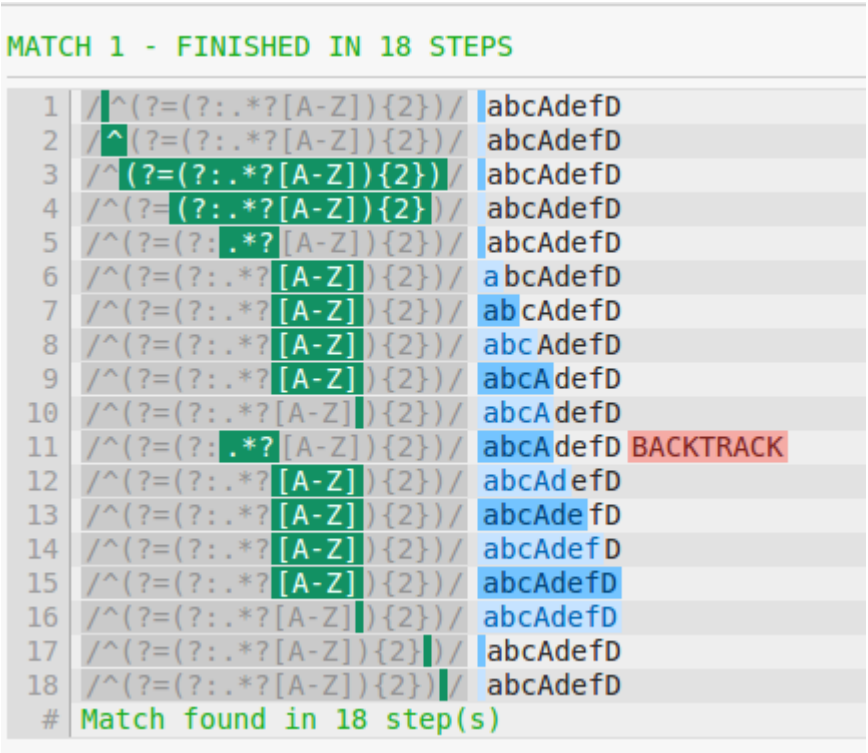
Esto se puede hacer con un poco de modificación en la expresión regular anterior.

```
 ^(?=.{10,}$)(?=(?:.*?[A-Z]){2})(?=.*?[a-z])(?=(?:.*?[0-9]){2}).*$
```

o

```
 ^(?=.{10,}$)(?=(?:.*[A-Z]){2})(?=.*[a-z])(?=(?:.*[0-9]){2}).*
```

Veamos cómo funciona una expresión regular `^(?=(?:.*?[AZ]){2})` en la cadena `abcAdefD`



Crédito de la imagen : - **https://regex101.com/**

Lea Validación de contraseñas de expresiones regulares en línea:

https://riptutorial.com/es/regex/topic/5340/validacion-de-contrasenas-de-expresiones-regulares

Creditos

S. No	Capítulos	Contributors
1	Empezando con Expresiones Regulares	0rkan, Addison, balpha, Community, Confiqure, Ibrahim, J F, JelmerS, JohnLBevan, Kendra, Laurel, Maria Deleva, Mariano, Mateus, mnoronha, Rudy M, Stephen Leppik, Tot Zam, TylerH, Wolf, Yaron, zmo
2	Agrupación atómica	OnlineCop
3	Caracteres del ancla: Dólar ($)	ArtOfCode, CPHPython, hjpotter92, Kendra, rubayet.R, Tom Lord, UNagaswamy, Wiktor Stribiżew
4	Clases de personajes	Acey, CPHPython, Dmitry Bychenko, HamZa, kdhp, Lucas Trzesniewski, Maria Deleva, RamenChef, rgoliveira, rock321987, Wiktor Stribiżew
5	Combinadores de UTF-8: letras, marcas, puntuación, etc.	mudasobwa
6	Cuando NO debes usar Expresiones Regulares	dorukayhan, Kendra, zmo
7	Cuantificadores Posesivos	Mark Hurd, Sebastian Lenartowicz
8	Cuantitativos codiciosos y perezosos	0rkan, Confiqure, David Knipe, GradientByte, Laurel, Mario, Mark Stewart, Nathan Arthur, nhahtdh, phatfingers, sweaver2112, Thomas Ayoub, Tim Pietzcker
9	Escapando	CPHPython, David Knipe, Laurel
10	Grupos de captura	Addison, Alan Moore, Lucas Trzesniewski, Tomalak, Vogel612
11	Grupos de captura con nombre	Thomas Ayoub
12	Límite de palabra	cdm, jonathanking, kdhp, Maria Deleva, Peter G, rgoliveira, Tushar
13	Lookahead y Lookbehind	BoppreH, hwnd, Lucas Trzesniewski, Maria Deleva, Wiktor Stribiżew

14	Modificadores de expresiones regulares (banderas)	Eder, Mateus, Tim Pietzcker, Wiktor Stribiżew
15	Patrones simples a juego	balpha, GradientByte, Graham, Joe, Mariano, rgoliveira, Tot Zam, Yaron
16	Personajes ancla: Caret (^)	CPHPython, Eder, J F, JohnLBevan, Jojodmo, knut, Mateus, Mike H-R, Mr. Deathless, nhahtdh, revo, rgoliveira, Tom Lord, zb226
17	Recursion	Keith Hall, Laurel, Lucas Trzesniewski, user23013
18	Referencia posterior	Alan Moore, Kendra, OnlineCop
19	Restablecer partido: \ K	nhahtdh, Wiktor Stribiżew, Will Barnwell
20	Retroceso	dorukayhan, Mike, Miljen Mikic, SQB, Thomas Ayoub, Vituel
21	Sustituciones con expresiones regulares	Mateus
22	Tipos de motores de expresiones regulares	Lucas Trzesniewski, Markus Jarderot
23	Trampas Regex	BrightOne
24	Útil escaparate Regex	depperm, Devid Farinelli, Echelon, Herb, Kendra, Matas Vaitkevicius, nhahtdh, Sebastian Lenartowicz, Steve Chambers, Thomas Ayoub, Tomasz Jakub Rup, zmo
25	Validación de contraseñas de expresiones regulares	rock321987