**Engineer**

# expresión regular

Patrón que define qué cadenas cuentan como coincidencia

## Qué es

Una expresión regular es un patrón breve que describe un conjunto de cadenas para que un programa pueda seleccionar las que pertenecen a él. La norma POSIX, publicada por The Open Group y el IEEE, abre su capítulo sobre el tema describiendo las expresiones regulares como la herramienta para seleccionar cadenas concretas de un conjunto de cadenas de caracteres. En el habla cotidiana y en muchos nombres de funciones y bibliotecas, el término se abrevia como regex o regexp, la grafía del objeto RegExp de JavaScript y del paquete regexp de Go.

La mayoría de los caracteres de un patrón se representan a sí mismos: una `a` en el patrón coincide con una `a` en el texto. Algunos son especiales. Un punto coincide con cualquier carácter, `*` repite lo que lo precede cero o más veces, una expresión entre corchetes como `[0-9]` coincide con un carácter de un conjunto, `^` y `$` anclan el patrón al principio y al final, y un intervalo como `{4}` pide un número exacto de repeticiones. Así, `^[0-9]{4}$` describe todas las cadenas formadas por exactamente cuatro dígitos, sin ningún otro carácter antes o después. El ejemplo de intervalo de la propia norma es `c{3}` aplicado a `abababccccccd`: coincide con los caracteres séptimo a noveno, las tres primeras de las seis letras c.

Hay más de un dialecto. El propio POSIX define dos: expresiones regulares básicas (BRE), que usa la mayoría de sus utilidades, y expresiones regulares extendidas (ERE), que algunas usan en su lugar. Las funciones de C `regcomp()` y `regexec()` ofrecen ambas. La diferencia no es superficial. En una ERE, `a+` significa una o más `a`. En una BRE, `+` es un carácter normal, y la norma deja a cada implementación decidir si `\+` significa una o más o un signo más literal.

La otra gran familia desciende de Perl. La especificación ECMAScript afirma que la forma y la funcionalidad de las expresiones regulares de JavaScript toman como modelo las de Perl 5. El paquete regexp de Go indica que acepta, a grandes rasgos, la misma sintaxis que Perl, Python y otros lenguajes, y el módulo `re` de Python es un tercer miembro. Estos motores comparten la mayor parte de su notación, pero no todos los detalles. Las clases abreviadas son un ejemplo: `\d` (un dígito) y `\w` (un carácter de palabra) parecen iguales en todas partes, pero su alcance depende del motor y sus indicadores, como muestra la sección práctica.

Utilice el concepto en una decisión

Aplique «¿Qué es una expresión regular?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Un equipo que lleva una idea del boceto a un producto al que sus clientes pueden acceder, sin pasarla entre tres proveedores.

**Explore nuestros servicios** Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Ingresos multiplicados por 10 con mapas GIS — 10x crecimiento de ingresos impulsado desde el producto cartográfico

**Ingresos multiplicados por 10 con mapas GIS** Póngase en contacto →

## Por qué importa

Las expresiones regulares aparecen en mucho software que no parece dedicado al procesamiento de texto: validación de campos de formularios, filtros que extraen valores de líneas de registro, reglas de enrutamiento de servidores web, búsqueda y sustitución en editores de texto. Cada patrón es breve, por lo que es fácil pasar por alto lo que realmente acepta. Dos preguntas deciden el resultado, y ambas respuestas dependen del motor: qué coincidencia se elige si hay varias posibles y cuánto puede tardarse en encontrar una.

POSIX elige la coincidencia situada más a la izquierda y más larga: de todas las que empiezan antes en el texto, gana la más larga. Los ejemplos de la norma son la BRE `bb*`, que coincide con los caracteres segundo a cuarto de `abbbc`, y la ERE `(wee|week)(knights|night)`, que coincide con los diez caracteres de `weeknights`. La familia Perl funciona de otra manera. Prueba las alternativas en el orden en que están escritas y toma la primera que produce una coincidencia, lo que la documentación de Go llama leftmost-first: la primera coincidencia situada más a la izquierda. El mismo patrón sobre el mismo texto puede, por tanto, dar dos respuestas:

```
$ echo ab | grep -oE 'a|ab'
ab
>>> re.search("a|ab", "ab").group()
'a'
> "ab".match(/a|ab/)[0]
'a'
```

Go ofrece ambos comportamientos. Su función `Compile` usa la semántica leftmost-first, y `CompilePOSIX` restringe la sintaxis a ERE de POSIX y cambia a la coincidencia más a la izquierda y más larga. Aun así, el paquete mantiene su propia regla para elegir subcoincidencias. Su documentación explica el motivo: la regla POSIX, afirma, resultaría demasiado costosa de calcular y ni siquiera está definida con claridad.

La segunda pregunta es el tiempo. El módulo `re` de Python, como la mayoría de los motores de la familia Perl, usa retroceso. Cuando una parte de un patrón no puede continuar, el motor retrocede y prueba otra manera de hacer coincidir lo anterior. La documentación de Python ilustra la idea con `a*a` frente a `aaaa`: `a*` toma primero las cuatro letras y luego devuelve una para que coincida la `a` final. En la mayoría de los patrones, el coste es imperceptible. En algunos, el número de posibilidades crece exponencialmente con la longitud de la entrada, y una cadena de unas pocas decenas de caracteres puede mantener ocupado un procesador durante minutos. MITRE lo cataloga como CWE-1333, complejidad ineficiente de expresiones regulares, y enumera ReDoS (denegación de servicio mediante expresiones regulares) y retroceso catastrófico como otros nombres. El paquete regexp de Go se diseñó en torno a la promesa contraria. Su documentación afirma que la búsqueda de coincidencias «garantiza un tiempo lineal respecto al tamaño de la entrada» y añade que la mayoría de las implementaciones de código abierto no ofrecen esa garantía.

## Cómo se aplica en la práctica

Tomemos un formulario de registro para clientes de Dinamarca y Ucrania con un campo de nombre de pila que debería aceptar solo letras. El patrón evidente es `[A-Za-z]+`, y en Python 3.15 se comporta así:

```
>>> import re
>>> re.fullmatch(r"[A-Za-z]+", "Mia")
<re.Match object; span=(0, 3), match='Mia'>
>>> re.fullmatch(r"[A-Za-z]+", "Søren")
>>> re.fullmatch(r"[A-Za-z]+", "Олена")
```

```
>>> re.fullmatch(r"\w+", "Søren_42")
<re.Match object; span=(0, 8), match='Søren_42'>
>>> re.fullmatch(r"[^\W\d_]+", "Олена")
<re.Match object; span=(0, 5), match='Олена'>
>>> re.fullmatch(r"[^\W\d_]+", "Søren_42")
>>>
```

Una llamada que devuelve `None` no imprime ninguna línea en la consola interactiva. Por eso, la clase ASCII rechaza Søren y Олена (Olena), dos nombres totalmente normales en Copenhague y Kyiv. Pasar a `\w` los acepta, pero en Python `\w` incluye todos los caracteres para los que `str.isalnum()` es verdadero, además del guion bajo, por lo que también acepta `Søren_42`. La clase `[^\W\d_]` excluye caracteres que no sean de palabra, dígitos decimales y el guion bajo. También permite algunos caracteres numéricos no decimales, por lo que no es una prueba estricta de solo letras. Para nombres de más de una palabra, `[^\W\d_]+( [^\W\d_]+)*` acepta tanto `Anne Sofie Østergaard` como `Олена Коваль`. Sigue rechazando `Anne-Marie` por el guion, un motivo para que el campo de nombre sea tan permisivo como permita el negocio. Las herramientas POSIX presentan el mismo problema de otra forma: la norma deja sin especificar el significado de un rango como `A-Z` fuera de la configuración regional POSIX y ofrece la clase `[[:alpha:]]`, cuyos miembros define la configuración regional actual.

Acertar con esto importa mucho más allá de un formulario. Uno de los servicios de este sitio, internacionalización y localización, describe su objetivo como software que se lea con naturalidad en todos los mercados donde venda el cliente, hasta en la forma de ordenar los nombres. Un campo de nombre que rechaza Søren incumple ese objetivo mucho antes de ordenar nada.

Los dígitos plantean la pregunta opuesta. Un código postal danés tiene cuatro dígitos, por lo que `\d{4}` parece una comprobación segura para `2200`. Sin embargo, en Python, `\d` en un patrón de texto coincide con cualquier dígito decimal Unicode, no solo del 0 al 9, y acepta el mismo código postal escrito con dígitos arábigo-índicos:

```
>>> s = "\u0662\u0662\u0660\u0660"
>>> print(s, int(s))
٢٢٠٠ 2200
>>> re.fullmatch(r"\d{4}", s)
<re.Match object; span=(0, 4), match='٢٢٠٠'>
```

```
>>> re.fullmatch(r"\d{4}", s, re.ASCII)
>>> re.fullmatch(r"[0-9]{4}", s)
>>>
```

JavaScript adopta la opción contraria. La especificación ECMAScript define `\d` como exactamente los diez caracteres del 0 al 9, por lo que `/^\d{4}$/.test(s)` devuelve `false` en Node.js para la misma cadena, mientras que `/^\p{Nd}{4}$/u` devuelve `true`. Un formulario comprobado con una regla en el navegador y con la otra en el servidor discrepará sobre la misma entrada. El `int()` de Python lee la cadena como 2200, pero una columna de base de datos, una tabla de consulta u otro servicio puede no hacerlo. Decida qué dígitos se admiten: `[0-9]` o el indicador `re.ASCII` si solo sirven dígitos ASCII, y `\p{Nd}` con el indicador `u` en JavaScript si se admite cualquier dígito decimal.

## Dónde surgen los problemas

**Un patrón duplica su tiempo de ejecución con cada letra.** Un patrón de nombres que pretenda permitir palabras separadas por un solo espacio se escribe fácilmente como `([^\W\d_]+ ?)+`. Con un nombre válido es rápido. Con una secuencia de letras seguida de un carácter que no puede aceptar, el motor prueba todas las formas de dividir las letras en palabras antes de rendirse, y el número de posibilidades se duplica con cada letra. En una máquina con Python 3.15, la más rápida de cinco ejecuciones para cada longitud dio estos tiempos de `fullmatch` sobre una cadena de letras seguida de `!`:

```
letters   seconds
     18     0.017
     20     0.067
     22     0.27
     24     1.0
     26     4.1
```

Cada letra duplica el tiempo, por lo que dos letras más lo multiplican por cuatro. Extrapolando ese factor, treinta letras tardarían alrededor de un minuto y treinta y dos, más de cuatro minutos. Es el caso que describe CWE-1333: los intentos de retroceso crecen exponencialmente con la entrada, esta no tiene por qué coincidir y puede ser lo bastante larga. La reformulación `[^\W\d_]+( [^\W\d_]+)*` describe los mismos nombres sin un cuantificador anidado, rechaza la cadena de 24 letras en aproximadamente un microsegundo y un millón de letras seguidas de `!` en unos treinta milisegundos. Elimine los cuantificadores anidados, limite

la longitud de la entrada antes de buscar coincidencias y ejecute los patrones que reciban entradas no fiables en un motor con garantía de tiempo lineal, como el de Go.

**Buscar donde debe coincidir el valor completo.** En Python, `re.search` acepta una coincidencia en cualquier parte de la cadena, y `re.match` solo la ancla al principio. Python 3.15 le da el nombre más claro `prefixmatch` y marca el anterior como desaconsejado («soft-deprecated»), señalando que otros lenguajes usan match para lo que Python siempre ha llamado search. Así, `re.prefixmatch(r"\d{4}", "22000")` tiene éxito con los cuatro primeros dígitos, y `re.search(r"\d{4}", "DK-2200")` encuentra el código postal dentro de la cadena. Incluso `re.search(r"^\d{4}$", "2200\n")` tiene éxito, porque `$` también coincide justo antes de un salto de línea al final. `re.fullmatch`, disponible desde Python 3.4, rechaza los tres casos. Use `fullmatch` cuando un patrón describa el valor completo, o ánclelo con `\A` y `\Z`, que Python 3.14 también escribe `\z`.

**Una lista de permitidos con un punto demasiado amplio.** Una comprobación que solo permita redirecciones al propio sitio de la empresa podría escribirse como `^https://www.example.com`. Acepta `https://www.example.com.evil.example/` porque el patrón termina antes que el nombre del host, y cada punto sin escapar coincide con cualquier carácter. Por tanto, también pasa un host que difiera del real en un solo carácter en lugar de un punto. Escapar los puntos, como hace `re.escape("www.example.com")`, y terminar el patrón con `/` cierra ambos huecos. Mejor aún: analice la URL y compare su nombre de host con el esperado como cadena completa.

**Un patrón copiado entre motores.** Un patrón probado con `grep -E` y pegado en Python puede elegir otra coincidencia, como muestra el ejemplo `a|ab`, y uno que funcione con `\+` en una herramienta BRE puede significar un signo más literal en otra. El `\d` que acepta dígitos arábigo-índicos en Python los rechaza en JavaScript. Pruebe cada patrón en el motor que lo ejecutará y con los indicadores con los que se ejecutará.

**Un indicador global que recuerda.** En JavaScript, una expresión regular con el indicador `g` o `y` guarda en `lastIndex` la posición donde terminó su última coincidencia y empieza allí la siguiente búsqueda. Después de un intento fallido, restablece `lastIndex` a 0. Un patrón guardado en una constante y reutilizado para validar responde entonces de manera distinta en llamadas alternas:

```
> const re = /^\d{4}$/g;
undefined
> re.test("2200")
true
> re.test("2200")
false
> re.test("2200")
true
```

Omita el indicador `g` en los patrones usados con `test()` o establezca `lastIndex` en 0 antes de cada llamada.

**Texto que parece igual, pero no lo es.** El nombre André puede almacenarse con una única é precompuesta o con una e simple seguida de un acento combinante. Ambos se ven iguales en pantalla, pero POSIX establece que la coincidencia sigue el patrón de bits usado para codificar un carácter, no su representación gráfica, y `re.search("André", "Andre\u0301")` en Python devuelve `None`. Después de aplicar `unicodedata.normalize("NFC", ...)` al texto, encuentra el nombre. Normalice el patrón y el texto a la misma forma antes de buscar coincidencias, con `unicodedata.normalize` en Python o `String.prototype.normalize` en JavaScript.

## Lecturas adicionales

- Especificaciones básicas de The Open Group, edición 8: expresiones regulares — definición POSIX: BRE y ERE, expresiones entre corchetes, clases de caracteres y configuraciones regionales, y la regla de coincidencia más a la izquierda y más larga.
- Paquete regexp de Go — garantía de tiempo lineal, elección de la primera coincidencia o de la más larga entre las situadas más a la izquierda, y por qué CompilePOSIX se aparta de POSIX en las subcoincidencias.
- re: operaciones con expresiones regulares en Python — sintaxis, clases Unicode y ASCII, retroceso y cuantificadores posesivos, y search, prefixmatch y fullmatch.
- ECMAScript: procesamiento de texto (borrador vivo, ECMAScript 2027) — expresiones regulares de JavaScript: herencia de Perl 5, definición de `\d`, escapes de propiedades Unicode y uso de lastIndex.
- CWE-1333: complejidad ineficiente de expresiones regulares — retroceso catastrófico y ReDoS: condiciones que hacen explotable un patrón, ejemplos observados y mitigaciones.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

**Vea cuánto cuesta** Póngase en contacto →

# Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es una expresión regular?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

**Explore nuestros servicios**

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

**Creado con esmero y cariño por D.**

**CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N**

**Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.**

**Impreso desde https://engineer.company/es/glossary/regular-expression/**