**Engineer**

# Regulärer Ausdruck

Muster, das festlegt, welche Zeichenfolgen passen

## Was es ist

Ein regulärer Ausdruck ist ein kurzes Muster, das eine Menge von Zeichenfolgen beschreibt. Ein Programm kann damit diejenigen auswählen, die dazugehören. Der von The Open Group und IEEE veröffentlichte POSIX-Standard eröffnet sein Kapitel dazu mit der Beschreibung regulärer Ausdrücke als Werkzeug, um bestimmte Zeichenfolgen aus einer Menge von Zeichenfolgen auszuwählen. Im Alltag und in vielen Funktions- und Bibliotheksnamen wird der Begriff zu Regex oder regexp verkürzt. Letztere Schreibweise findet sich im RegExp-Objekt von JavaScript und im regexp-Paket von Go.

Die meisten Zeichen eines Musters stehen für sich selbst: Ein `a` im Muster passt zu einem `a` im Text. Einige haben eine Sonderbedeutung. Ein Punkt passt zu jedem Zeichen, `*` wiederholt das Vorhergehende nullmal oder öfter, ein Klammerausdruck wie `[0-9]` passt zu einem Zeichen aus einer Menge, `^` und `$` binden das Muster an Anfang und Ende. Ein Intervall wie `{4}` verlangt eine genaue Zahl von Wiederholungen. `^[0-9]{4}$` beschreibt daher jede Zeichenfolge aus genau vier Ziffern ohne weitere Zeichen davor oder danach. Das eigene Intervallbeispiel des Standards ist `c{3}` auf `abababccccccd` : Es passt zum siebten bis neunten Zeichen, den ersten drei der sechs c.

Es gibt mehrere Dialekte. POSIX selbst definiert zwei: einfache reguläre Ausdrücke (BRE), die die meisten seiner Werkzeuge verwenden, und erweiterte reguläre Ausdrücke (ERE), die einige stattdessen verwenden. Die C-Funktionen `regcomp()` und `regexec()` bieten beide. Der Unterschied ist nicht kosmetisch. In einem ERE bedeutet `a+` ein oder mehrere `a` . In einem BRE ist `+` ein gewöhnliches Zeichen. Ob `\+` ein oder mehrere oder ein wörtliches Pluszeichen bedeutet, überlässt der Standard der jeweiligen Implementierung.

Die andere große Familie stammt von Perl ab. Laut ECMAScript-Spezifikation sind Form und Funktion der regulären Ausdrücke von JavaScript der entsprechenden Funktionalität von Perl 5 nachgebildet. Das regexp-Paket von Go gibt an, im Wesentlichen dieselbe Syntax wie Perl, Python und andere Sprachen zu akzeptieren. Pythons Modul `re` ist ein drittes Mitglied. Diese Engines teilen den Großteil ihrer Notation, aber nicht jedes Detail. Die Kurzklassen zeigen das: `\d` für eine Ziffer und `\w` für ein Wortzeichen sehen überall gleich aus. Was sie umfassen, hängt jedoch von Engine und Optionen ab, wie der Praxisteil zeigt.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein regulärer Ausdruck?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Ein Team führt eine Idee von der Skizze bis zu einem Produkt, bei dem Ihre Kunden sich anmelden können, statt sie zwischen drei Dienstleistern weiterzureichen.

**Unsere Leistungen entdecken** Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird.
GIS-Kartenprodukt zu 10-fachem Umsatz geführt — 10x Umsatzwachstum durch die Leitung des Kartenprodukts

**GIS-Kartenprodukt zu 10-fachem Umsatz geführt** Kontakt aufnehmen →

## Warum es wichtig ist

Reguläre Ausdrücke stecken in viel Software, die nicht nach Textverarbeitung aussieht: Formularvalidierung, Filter zum Auslesen von Werten aus Logzeilen, Routingregeln in Webservern, Suchen und Ersetzen in Texteditoren. Jedes Muster ist kurz. Dadurch wird leicht überlesen, was es wirklich akzeptiert. Zwei Fragen entscheiden das Ergebnis, und beide Antworten hängen von der Engine ab: Welcher Treffer wird bei mehreren Möglichkeiten gewählt, und wie lange kann die Suche danach dauern?

POSIX wählt den längsten der am weitesten links beginnenden Treffer: Unter allen Treffern mit dem frühesten Beginn gewinnt der längste. Die Beispiele des Standards sind der BRE `bb*` , der zum zweiten bis vierten Zeichen von `abbbc` passt, und der ERE `(wee|week)(knights|night)` , der alle zehn Zeichen von `weeknights` erfasst. Die Perl-Familie arbeitet anders. Sie probiert Alternativen in ihrer geschriebenen Reihenfolge und nimmt die erste, die einen Treffer ergibt. Gos Dokumentation nennt das leftmost-first. Dasselbe Muster auf demselben Text kann daher zwei Antworten ergeben:

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

Go bietet beide Verhaltensweisen. `Compile` verwendet leftmost-first. `CompilePOSIX` beschränkt die Syntax auf POSIX-ERE und wechselt zu leftmost-longest. Selbst dann behält das Paket eine eigene Regel zur Auswahl von Teiltreffern. Die Dokumentation erklärt den Grund: Die POSIX-Regel sei zu teuer zu berechnen und nicht einmal klar definiert.

Die zweite Frage betrifft die Zeit. Pythons `re` arbeitet wie die meisten Engines der Perl-Familie mit Backtracking. Kann ein Teil des Musters nicht fortfahren, geht die Engine zurück und versucht, das Vorhergehende anders zuzuordnen. Die Python-Dokumentation zeigt das an `a*a` für `aaaa` : `a*` nimmt zuerst alle vier Buchstaben und gibt dann einen zurück, damit das abschließende `a` passen kann. Bei den meisten Mustern fällt der Aufwand nicht auf. Bei einigen wächst die Zahl der Versuche exponentiell mit der Eingabelänge. Eine Zeichenfolge von wenigen Dutzend Zeichen kann einen Prozessor minutenlang beschäftigen. MITRE führt dies als CWE-1333, ineffiziente Komplexität regulärer Ausdrücke, und nennt ReDoS, eine Dienstverweigerung durch reguläre Ausdrücke, sowie katastrophales Backtracking als weitere Namen. Gos regexp-Paket wurde um das entgegengesetzte Versprechen herum entworfen. Seine Dokumentation sagt, die Suche laufe „garantiert in einer zur Eingabegröße linearen Zeit“, und ergänzt, dass die meisten Open-Source-Implementierungen keine solche Garantie geben.

## Wie es in der Praxis aussieht

Nehmen wir ein Registrierungsformular für Kunden in Dänemark und der Ukraine mit einem Vornamensfeld, das nur Buchstaben akzeptieren soll. Das naheliegende Muster ist `[A-Za-z]+` . Unter Python 3.15 verhält es sich so:

```
>>> 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")
>>>
```

Ein Aufruf, der `None` zurückgibt, druckt in der interaktiven Shell überhaupt keine Zeile. Die ASCII-Klasse weist deshalb Søren und Олена (Olena) ab, zwei völlig gewöhnliche Namen in Kopenhagen und Kyjiw. Mit `\w` werden sie zugelassen. In Python umfasst `\w` jedoch jedes Zeichen, für das `str.isalnum()` wahr ist, sowie den Unterstrich. Es lässt also auch `Søren_42` zu. Die Klasse `[^\W\d_]` schließt Nichtwortzeichen, Dezimalziffern und den Unterstrich aus. Sie erlaubt aber einige nichtdezimale numerische Zeichen und ist daher kein strenger Test auf ausschließlich Buchstaben. Für mehrteilige Namen akzeptiert `[^\W\d_]+([^\W\d_]+)*` sowohl `Anne Sofie Østergaard` als auch `Олена Коваль`. Das mit Bindestrich geschriebene `Anne-Marie` wird weiterhin abgewiesen. Deshalb sollte ein Namensfeld so offen bleiben, wie es die fachlichen Anforderungen erlauben. POSIX-Werkzeuge haben dasselbe Problem in anderer Form: Der Standard lässt die Bedeutung eines Bereichs wie `A-Z` außerhalb der POSIX-Locale offen und bietet die Klasse `[[:alpha:]]`, deren Mitglieder die aktuelle Locale bestimmt.

Das richtig zu lösen ist über ein einzelnes Formular hinaus wichtig. Eine Leistung dieser Website, Internationalisierung und Lokalisierung, beschreibt ihr Ziel als Software, die sich in jedem Markt eines Kunden natürlich liest, bis hin zur Sortierung von Namen. Ein Namensfeld, das Søren ablehnt, verfehlt dieses Ziel lange vor jeder Sortierung.

Ziffern stellen die umgekehrte Frage. Eine dänische Postleitzahl hat vier Ziffern, also sieht `\d{4}` wie eine sichere Prüfung für `2200` aus. In Python passt `\d` in einem Textmuster jedoch zu jeder Unicode-Dezimalziffer, nicht nur zu 0 bis 9. Es akzeptiert dieselbe Postleitzahl auch in arabisch-indischen Ziffern:

```
>>> 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 entscheidet anders. Die ECMAScript-Spezifikation definiert `\d` als genau die zehn Zeichen 0 bis 9. Deshalb liefert `/^\d{4}$/.test(s)` in Node.js für dieselbe Zeichenfolge `false`, während `/^\p{Nd}{4}$/u` `true` liefert. Ein mit der einen Regel im Browser und der anderen auf dem Server geprüftes Formular urteilt über dieselbe Eingabe unterschiedlich. Pythons `int()` liest sie als 2.200. Eine Datenbankspalte, eine Nachschlagetabelle oder ein anderer Dienst tut das möglicherweise nicht. Entscheiden Sie, welche Ziffern gemeint sind: `[0-9]` oder die Option `re.ASCII`, wenn ausschließlich ASCII-Ziffern zulässig sind, und `\p{Nd}` mit der Option `u` in JavaScript, wenn jede Dezimalziffer willkommen ist.

## Wo es schiefgeht

**Ein Muster verdoppelt seine Laufzeit mit jedem Buchstaben.** Ein Namensmuster für durch einzelne Leerzeichen getrennte Wörter wird leicht als `([^\W\d_]+ ?)+` geschrieben. Bei einem gültigen Namen ist es schnell. Bei einer Buchstabenfolge mit einem unzulässigen Zeichen am Ende probiert die Engine vor dem Aufgeben jede mögliche Aufteilung der Buchstaben in Wörter. Die Zahl der Möglichkeiten verdoppelt sich mit jedem Buchstaben. Auf einem Rechner mit Python 3.15 ergab der schnellste von fünf Läufen je Länge diese Zeiten für `fullmatch` auf einer Buchstabenfolge mit abschließendem `!`:

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

Jeder Buchstabe verdoppelt die Zeit, zwei weitere vervierfachen sie also. Aus diesem Faktor hochgerechnet würden dreißig Buchstaben ungefähr eine Minute und zweiunddreißig mehr als vier Minuten dauern. Das ist der von CWE-1333 beschriebene Fall: Die Backtracking-Versuche wachsen exponentiell mit der Eingabe, diese braucht nicht zum Muster zu passen und kann lang genug sein. Die Umformulierung `[^\W\d_]+( [^\W\d_]+)*` beschreibt dieselben Namen ohne verschachtelten Quantifizierer. Sie weist die Folge von 24 Buchstaben in ungefähr einer Mikrosekunde und eine Million Buchstaben mit abschließendem `!` in ungefähr dreißig Millisekunden ab. Entfernen Sie verschachtelte Quantifizierer, begrenzen Sie die Eingabelänge vor der Prüfung und verwenden Sie für Muster mit nicht vertrauenswürdigen Eingaben eine Engine mit garantierter linearer Laufzeit wie die von Go.

**Suchen, obwohl der ganze Wert passen muss.** In Python akzeptiert `re.search` einen Treffer an jeder Stelle der Zeichenfolge. `re.match` bindet ihn nur an den Anfang. Python 3.15 nennt es klarer `prefixmatch` und rät vom alten Namen in Form einer Soft-Deprecation ab. Dabei weist es darauf hin, dass andere Sprachen match für das verwenden, was Python schon immer search nennt. `re.prefixmatch(r"\d{4}", "22000")` trifft deshalb die ersten vier Ziffern, und `re.search(r"\d{4}", "DK-2200")` findet die Postleitzahl innerhalb der Zeichenfolge. Sogar `re.search(r"^\d{4}$", "2200\n")` trifft, weil `$` auch unmittelbar vor einem abschließenden Zeilenumbruch passt. Das seit Python 3.4 verfügbare `re.fullmatch` weist alle drei ab. Verwenden Sie `fullmatch`, wenn ein Muster den gesamten Wert beschreibt, oder binden Sie es mit `\A` und `\Z`, das Python 3.14 auch `\z` schreibt.

**Eine Zulassungsliste mit einem nicht maskierten Punkt.** Eine Prüfung, die Weiterleitungen nur zur eigenen Unternehmenswebsite zulässt, könnte als `^https://www.example.com` geschrieben sein. Sie akzeptiert `https://www.example.com.evil.example/`, weil das Muster vor dem Ende des Hostnamens endet. Jeder nicht maskierte Punkt passt außerdem zu jedem Zeichen. Deshalb kommt auch ein Host durch, der sich anstelle eines Punkts durch ein einziges anderes Zeichen vom echten unterscheidet. Die Punkte zu maskieren, wie es `re.escape("www.example.com")` tut, und das Muster mit `/` abzuschließen schließt beide Lücken. Noch besser ist, die URL zu parsen und ihren vollständigen Hostnamen mit dem erwarteten zu vergleichen.

**Ein Muster wird zwischen Engines kopiert.** Ein mit `grep -E` geprüftes und in Python eingefügtes Muster kann einen anderen Treffer wählen, wie `a|ab` zeigt. Ein in einem BRE-Werkzeug mit `\+` funktionierendes Muster kann in einem anderen ein wörtliches Plus bedeuten. `\d` akzeptiert arabisch-indische Ziffern in Python und weist sie in JavaScript ab. Prüfen Sie jedes Muster in der Engine, die es später ausführt, und mit den dabei verwendeten Optionen.

**Eine globale Option erinnert sich.** In JavaScript speichert ein regulärer Ausdruck mit der Option `g` oder `y` das Ende seines letzten Treffers in `lastIndex` und beginnt die nächste Suche dort. Nach einem fehlgeschlagenen Versuch setzt er `lastIndex` auf 0 zurück. Ein in einer Konstanten gehaltener und zur Validierung wiederverwendeter Ausdruck antwortet dann bei aufeinanderfolgenden Aufrufen abwechselnd unterschiedlich:

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

Lassen Sie die Option `g` bei Mustern für `test()` weg oder setzen Sie `lastIndex` vor jedem Aufruf auf 0.

**Text sieht gleich aus, ist es aber nicht.** Der Name André kann mit einem einzelnen vorkombinierten é oder einem einfachen e mit folgendem kombinierendem Akzent gespeichert sein. Beide sehen auf dem Bildschirm gleich aus. POSIX sagt jedoch, dass die Zuordnung dem zur Zeichenkodierung verwendeten Bitmuster folgt, nicht der grafischen Darstellung. Pythons `re.search("André", "Andre\u0301")` gibt `None` zurück. Nach `unicodedata.normalize("NFC", ...)` auf dem Text findet es den Namen. Normalisieren Sie Muster und Text vor der Suche auf dieselbe Form, mit `unicodedata.normalize` in Python oder `String.prototype.normalize` in JavaScript.

## Weiterführende Lektüre

- The Open Group Base Specifications, Ausgabe 8: reguläre Ausdrücke — die POSIX-Definition: BRE und ERE, Klammerausdrücke, Zeichenklassen und Locales sowie die Regel des längsten links beginnenden Treffers.
- regexp-Paket in Go — die Garantie linearer Laufzeit, leftmost-first gegenüber leftmost-longest und weshalb CompilePOSIX bei Teiltreffern von POSIX abweicht.
- re: Operationen mit regulären Ausdrücken in Python — Syntax, Unicode- und ASCII-Klassen, Backtracking und possessive Quantifizierer sowie search, prefixmatch und fullmatch.
- ECMAScript: Textverarbeitung, laufender Entwurf ECMAScript 2027 — reguläre Ausdrücke in JavaScript: Herkunft von Perl 5, Definition von `\d`, Escape-Sequenzen für Unicode-Eigenschaften und Verwendung von lastIndex.
- CWE-1333: ineffiziente Komplexität regulärer Ausdrücke — katastrophales Backtracking und ReDoS: Bedingungen für ausnutzbare Muster, beobachtete Beispiele und Gegenmaßnahmen.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

**Preise ansehen** Kontakt aufnehmen →

## Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein regulärer Ausdruck?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

**Unsere Leistungen entdecken**

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/regular-expression/