**Engineer**

# Regular expression

a pattern that describes which strings count as a match

## What it is

A regular expression is a short pattern that describes a set of strings, so that a program can pick out the ones that belong to it. The POSIX standard, published by The Open Group and the IEEE, opens its chapter on the subject by describing regular expressions as the tool for picking particular strings out of a set of character strings. In everyday speech, and in many function and library names, the term shrinks to regex, or to regexp, the spelling of JavaScript's RegExp object and of Go's regexp package.

Most characters in a pattern stand for themselves: an `a` in the pattern matches an `a` in the text. A few are special. A period matches any character, `*` repeats what comes before it zero or more times, a bracket expression such as `[0-9]` matches one character from a set, `^` and `$` tie the pattern to the start and the end, and an interval such as `{4}` asks for an exact number of repetitions. So `^[0-9]{4}$` describes every string made of exactly four digits, with no other characters before or after. The standard's own example of an interval is `c{3}` applied to `abababccccccd` : it matches the seventh to ninth characters, the first three of the six c's.

There is more than one dialect. POSIX itself defines two: basic regular expressions (BRE), which most of its utilities use, and extended regular expressions (ERE), which some use instead. The C functions `regcomp()` and `regexec()` offer both. The difference is not cosmetic. In an ERE, `a+` means one or more `a` . In a BRE, `+` is an ordinary character, and the standard leaves it to each implementation whether `\+` means one or more or a literal plus sign.

The other large family descends from Perl. The ECMAScript specification states that the form and functionality of JavaScript's regular expressions are modelled on the regular expression facility of Perl 5. Go's regexp package says it accepts

broadly the same syntax as Perl, Python and other languages, and Python's `re` module is a third member. These engines share most of their notation, but not every detail. The shorthand classes are a case in point: `\d` (a digit) and `\w` (a word character) look the same everywhere, but what they cover depends on the engine and its flags, as the practice section below shows.

Use the concept in a decision

Put "What is a regular expression?" to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

**Explore our services** Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

**Led a GIS map application to 10x revenue growth** Get in touch →

## Why it matters

Regular expressions sit inside a great deal of software that does not look like text processing: validation of form fields, filters that pull values out of log lines, routing rules in web servers, search and replace in text editors. Each pattern is short, which makes it easy to read past what it actually accepts. Two questions decide the result, and the answer to both depends on the engine: which match is chosen when several are possible, and how long it may take to find one.

POSIX chooses the leftmost-longest match: of all the matches that start earliest in the text, the longest wins. The standard's examples are the BRE `bb*` , which matches the second to fourth characters of `abbbc` , and the ERE `(wee|week)(knights|night)` , which matches all ten characters of `weeknights` . The Perl family works differently. It tries the alternatives in the order they are written and takes the first one that leads to a match, which Go's documentation calls leftmost-first. The same pattern on the same text can therefore give two answers:

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

Go offers both behaviours. Its `Compile` function uses leftmost-first semantics, and `CompilePOSIX` restricts the syntax to POSIX ERE and switches to leftmost-longest. Even then the package keeps its own rule for choosing submatches, and its documentation explains why: the POSIX rule, it says, would cost too much to compute and is not even clearly defined.

The second question is time. Python's `re`, like most engines in the Perl family, backtracks. When part of a pattern cannot go on, the engine steps back and tries another way of matching what came before. Python's documentation shows the idea with `a*a` against `aaaa`: `a*` first takes all four letters, then gives one back so that the final `a` can match. For most patterns the cost is invisible. For some, the number of ways to try grows exponentially with the length of the input, and a string of a few dozen characters can keep a processor busy for minutes. MITRE catalogues this as CWE-1333, Inefficient Regular Expression Complexity, and lists ReDoS (regular expression denial of service) and catastrophic backtracking as other names for it. Go's regexp package was designed around the opposite promise. Its documentation states that matching is "guaranteed to run in time linear in the size of the input", and adds that most open-source implementations make no such guarantee.

## What it looks like in practice

Take a sign-up form for customers in Denmark and Ukraine, and a first-name field that should accept letters only. The obvious pattern is `[A-Za-z]+`, and in Python 3.15 it behaves like this:

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

A call that returns `None` prints no line at all in the interactive shell, so the ASCII class turns away Søren and Олена (Olena), two entirely ordinary names in Copenhagen and Kyiv. Switching to `\w` lets them in, but in Python `\w` covers

every character for which `str.isalnum()` is true, plus the underscore, so it also lets in `Søren_42`. The class `[^\W\d_]` excludes non-word characters, decimal digits and the underscore. It also permits some non-decimal numeric characters, so it is not a strict letters-only test. For names of more than one word `[^\W\d_]+([^\W\d_]+)*` accepts `Anne Sofie Østergaard` and `Олена Коваль` alike. It still turns away a hyphenated `Anne-Marie`, which is a reason to keep a name field as permissive as the business allows. POSIX tools have the same problem in another form: the standard leaves the meaning of a range such as `A-Z` unspecified outside the POSIX locale, and offers the class `[[:alpha:]]`, whose members are defined by the current locale.

Getting this right matters well beyond one form. One of this site's services, internationalization and localization, describes its aim as software that reads naturally in every market a client sells in, down to the way names sort. A name field that refuses Søren falls short of that aim long before any sorting happens.

Digits raise the opposite question. A Danish postcode is four digits, so `\d{4}` looks like a safe check for `2200`. In Python, though, `\d` in a text pattern matches any Unicode decimal digit, not just 0 to 9, and accepts the same postcode written in Arabic-Indic digits:

```
>>> 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 takes the other side. The ECMAScript specification defines `\d` as exactly the ten characters 0 to 9, so `/^\d{4}$/.test(s)` returns `false` in Node.js for the same string, while `/^\p{Nd}{4}$/u` returns `true`. A form checked in the browser with one rule and on the server with the other will disagree about the same input. Python's `int()` reads the string as 2200, but a database column, a lookup table or another service may not. Decide which digits are meant: `[0-9]`, or the `re.ASCII` flag, when only ASCII digits will do, and `\p{Nd}` with the `u` flag in JavaScript when any decimal digit is welcome.

# Where it goes wrong

**A pattern that doubles its running time with every letter.** A name pattern meant to allow words separated by single spaces is easily written as `([^\W\d_]+ ?)+` . On a valid name it is fast. On a run of letters followed by a character it cannot accept, the engine tries every way of dividing the letters into words before it gives up, and the number of ways doubles with each letter. On one machine with Python 3.15, the fastest of five runs at each length gave these times for `fullmatch` on a string of letters followed by `!` :

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

Each letter doubles the time, so two more letters multiply it by four. Extrapolating from that factor, thirty letters would take about a minute and thirty-two more than four minutes. This is the case CWE-1333 describes: the backtracking attempts grow exponentially with the input, the input does not have to match, and the input can be long enough. The rewrite `[^\W\d_]+( [^\W\d_]+)*` describes the same names without a nested quantifier, rejects the 24-letter string in about a microsecond and a million letters followed by `!` in about thirty milliseconds. Remove nested quantifiers, cap the length of the input before matching it, and run patterns that see untrusted input on an engine with a linear-time guarantee, such as Go’s.

**Searching where the whole value must match.** In Python, `re.search` accepts a match anywhere in the string, and `re.match` anchors only at the start; Python 3.15 gives it the clearer name `prefixmatch` and soft-deprecates the old one, noting that other languages use match for what Python has always called search. So `re.prefixmatch(r"\d{4}", "22000")` succeeds on the first four digits, and `re.search(r"\d{4}", "DK-2200")` finds the postcode inside the string. Even `re.search(r"^\d{4}$", "2200\n")` succeeds, because `$` also matches just before a newline at the end. `re.fullmatch` , available since Python 3.4, refuses all three. Use `fullmatch` whenever a pattern describes the whole value, or anchor with `\A` and `\Z` , which Python 3.14 also spells `\z` .

**An allowlist with a loose dot.** A check that only lets redirects go to the company's own site might be written as `^https://www.example.com`. It accepts `https://www.example.com.evil.example/`, because the pattern stops before the host name does, and each unescaped dot matches any character, so a host that differs from the real one by a single character in place of a dot passes too. Escaping the dots, as `re.escape("www.example.com")` does, and ending the pattern with `/` closes both holes. Better still, parse the URL and compare its host name with the expected one as a whole string.

**A pattern copied between engines.** A pattern tested with `grep -E` and pasted into Python can pick a different match, as the `a|ab` example shows, and one that works in one BRE tool with `\+` can mean a literal plus sign in another. The `\d` that accepts Arabic-Indic digits in Python rejects them in JavaScript. Test every pattern in the engine that will run it, with the flags it will run with.

**A global flag that remembers.** In JavaScript, a regular expression with the `g` or `y` flag stores the position where its last match ended in `lastIndex` and starts the next search there; after a failed attempt it resets `lastIndex` to 0. A pattern kept in a constant and reused for validation then answers differently on alternate calls:

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

Leave the `g` flag off patterns used with `test()`, or set `lastIndex` to 0 before every call.

**Text that looks the same but is not.** The name André can be stored with a single precomposed é or with a plain e followed by a combining accent. Both look identical on screen, but POSIX states that matching follows the bit pattern used to encode a character, not its graphic representation, and Python's `re.search("André", "Andre\u0301")` returns `None`. After `unicodedata.normalize("NFC",`

`...)` on the text it finds the name. Normalise the pattern and the text to the same form before matching, with `unicodedata.normalize` in Python or `String.prototype.normalize` in JavaScript.

## Further reading

- The Open Group Base Specifications Issue 8: Regular Expressions — the POSIX definition: BRE and ERE, bracket expressions, character classes and locales, and the leftmost-longest rule.
- regexp package (Go) — the linear-time guarantee, leftmost-first against leftmost-longest matching, and why CompilePOSIX departs from POSIX on submatches.
- re — Regular expression operations (Python) — the syntax, Unicode and ASCII classes, backtracking and possessive quantifiers, and search, prefixmatch and fullmatch.
- ECMAScript: Text Processing (living draft, ECMAScript 2027) — JavaScript's regular expressions: the Perl 5 heritage, the definition of `\d`, Unicode property escapes and how lastIndex is used.
- CWE-1333: Inefficient Regular Expression Complexity — catastrophic backtracking and ReDoS: the conditions that make a pattern exploitable, observed examples and mitigations.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

**See what it costs** Get in touch →

# Make the terminology practical

Ask how “What is a regular expression?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

**Explore our services**

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/regular-expression/