Regular Expressions

Version ES2025 · Python 3.15 · .NET 10 · verified 2026-11-02

Regex syntax common to JavaScript, Python and .NET, the dialect differences that bite, and how to keep a pattern from backtracking forever.

Contents

Mental model 2

Literals and character classes 2

Quantifiers and greediness 4

Anchors and boundaries 5

Groups and backreferences 6

Lookaround 7

Flags and Unicode 8

Replacement syntax 10

Catastrophic backtracking 12

Dialects and testing 13

Further reading 15

Mental model

A regex is a small program run by a backtracking engine: it tries the pattern from each start position, and when a step fails it rewinds to the last choice point and tries the next option. Most surprises follow from that: greedy quantifiers grab too much, unanchored patterns match inside strings, and nested choices can explode into billions of retries. Write patterns that are anchored and fail fast.

Literals and character classes

Syntax	Matches
`abc`	The literal text `abc`
`.`	Any char except newline
`[abc]`, `[a-z]`	One char from the set
`[^0-9]`	One char not in the set
`\d`, `\w`, `\s`	Digit, word char, whitespace
`\D`, `\W`, `\S`	Their negations

Escape the metacharacters `. * + ? ^ $ ( ) [ ] { } | \` with a backslash to match them literally. Inside a class most of them are literal already; only `]`, `\`, `^` (first) and `-` (between chars) need care.

```
/gr[ae]y/.test("grey");          // true
/\d\d:\d\d/.exec("at 09:45")[0]; // "09:45"
/3.14/.test("3x14");             // true: . is any
/3\.14/.test("3x14");            // false
RegExp(RegExp.escape("1+1")).test("1+1"); // true
```

> **Tip:** Never build a pattern from user input by concatenation. Use `RegExp.escape()` (ES2025, Node 24+), `re.escape()` in Python or `Regex.Escape()` in .NET.

Quantifiers and greediness

Syntax	Means	Lazy form
`*`	0 or more	`*?`
`+`	1 or more	`+?`
`?`	0 or 1	`??`
`{3}`	Exactly 3	—
`{2,5}`	2 to 5	`{2,5}?`
`{2,}`	2 or more	`{2,}?`

Quantifiers are greedy: they take as much as possible, then give back one character at a time until the rest of the pattern fits. The lazy form takes as little as possible and grows. A negated class is often clearer and faster than either.

```
const s = "<b>bold</b>";
s.match(/<.+>/)[0];     // "<b>bold</b>" greedy
s.match(/<.+?>/)[0];    // "<b>"         lazy
s.match(/<[^>]+>/)[0];  // "<b>"  no backtracking
"aaa".match(/a{2}/)[0]; // "aa"
```

> **Gotcha:** `.*` stops at a newline, so it silently fails on multi-line input. Add the `s` flag, or match `[\s\S]*`.

Anchors and boundaries

Syntax	Matches at
`^`	Start of input (line with `m`)
`$`	End of input (line with `m`)
`\b`	Word/non-word boundary
`\B`	Not a word boundary
`\A`, `\z`	Absolute start/end (Python, .NET)

Without anchors, `test` succeeds if the pattern matches anywhere in the string, which is almost never what validation means. Anchor both ends.

```
/\d{4}/.test("abc12345");        // true!
/^\d{4}$/.test("abc12345");      // false
/^\d{4}$/.test("2026");          // true
"cat scat".match(/\bcat\b/g);    // ["cat"]
"a\nb".match(/^b$/m)?.[0];       // "b"
```

> **Gotcha:** In Python and .NET, `$` also matches before a trailing newline, so `"42\n"` passes `^\d+$`. Use `\z` (Python 3.14+, .NET) or Python's `re.fullmatch()` to reject it.

Groups and backreferences

Syntax	Does
`(abc)`	Capture group 1
`(?:abc)`	Group without capturing
`(?<y>\d+)`	Named group (Python: `(?P<y>…)`)
`\1`, `\k<y>`	Match the same text again
`a|b`	Alternation, lowest precedence

Alternation binds loosest: `^cat|dog$` means `^cat` or `dog$`. Wrap it in a non-capturing group to scope it. Named groups survive edits that renumber positional ones.

```
const re = /(?<y>\d{4})-(?<m>\d{2})/;
const { y, m } = "on 2026-11".match(re).groups;
[y, m];                              // ["2026",
"11"]
/^(?:cat|dog)$/.test("catdog");      // false
/^cat|dog$/.test("catdog");          // true!
/(\w)\1/.exec("hello")[0];           // "ll"
```

> **Gotcha:** Python's `re` rejects `(?<y>…)` with "unknown extension". It needs `(?P<y>…)`, and `(?P=y)` to refer back to it.

Lookaround

Syntax	Asserts the text ahead/behind
`(?=x)`	Is followed by `x`
`(?!x)`	Is not followed by `x`
`(?<=x)`	Is preceded by `x`
`(?<!x)`	Is not preceded by `x`

Lookarounds check a condition without consuming characters, so the match itself excludes them. They are the clean way to express “every rule must hold” for things like password policies.

```
"$42 and €7".match(/(?<=\$)\d+/)[0];  // "42"
"price: 100px".match(/\d+(?=px)/)[0]; // "100"
const strong =
  /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{12,}$/;
strong.test("Correct8Horse");          // true
strong.test("correcthorse");           // false
```

> **Gotcha:** Python’s lookbehind must have a fixed width: `(?<=\$|USD)` raises `re.error`. JavaScript and .NET accept variable-length lookbehind.

Flags and Unicode

JS flag	Python / .NET	Effect
`i`	`re.I` / `IgnoreCase`	Case-insensitive

JS flag	Python / .NET	Effect
`m`	`re.M` / `Multiline`	`^` / `$` per line
`s`	`re.S` / `Singleline`	`.` matches newline
`g`	`findall` / `Matches`	Find every match
`u`, `v`	always on for Python `str`	Code points, `\p{…}`
`(?i:x)`	same inline syntax	Flag for a group only

Without `u` or `v`, JavaScript sees UTF-16 code units, so an emoji is two “characters”. `v` (ES2024) adds set operations to `u`. Inline modifiers like `(?i:x)` are ES2025 and Baseline since 2025.

```
/^.$/.test("𝌆");            // false: 2 units
/^.$/u.test("𝌆");           // true
/\p{L}+/u.exec("añb1")[0];   // "añb"
/[\p{L}--[a-z]]/v.test("q"); // false
/(?i:a)b/.test("Ab");        // true
```

> **Gotcha:** A `g` regex keeps `lastIndex` between calls, so the same object alternates: `re.test("a")` is `true`, then `false`. Drop `g` for `test`, or create the regex per call.

Replacement syntax

Insert	JavaScript	Python / .NET
Group 1	`$1`	`\1` / `$1`
Named group	`$<y>`	`\g<y>` / `${y}`
Whole match	`$&`	`\g<0>` / `$&`
Literal `$`	`$$`	`$` / `$$`
Computed	callback `(m, …g) =>`	function / `MatchEvaluator`

Replacement strings have their own mini-language, separate from the pattern. Pass a function when the replacement depends on the match.

```
"2026-11-02".replace(
  /(\d+)-(\d+)-(\d+)/, "$3.$2.$1"); //
"02.11.2026"
"a-b-c".replace(/-/g, "+");          // "a+b+c"
"x1y22".replace(/\d+/g, (d) => d * 2); // "x2y44"
"a.b".replaceAll(".", "!");          // "a!b"
```

```
import re
re.sub(r"(?P<y>\d{4})-(\d\d)",
       r"\2/\g<y>", "2026-11")  # '11/2026'
re.fullmatch(r"\d+", "42x")     # None
```

> **Gotcha:** JavaScript's `replace` with a string or a regex without `g` changes only the first match. `replaceAll` with a regex that lacks `g` throws a `TypeError`.

Catastrophic backtracking

Risky shape	Why	Safer
`(a+)+`	Nested quantifiers	`a+`
`(a|a)*`	Overlapping choices	`a*`
`(\w+\s?)+$`	Ambiguous split	Atomic or possessive
`.*x.*y` on long input	Many retry points	`[^x]*x[^y]*y`

When a match fails, the engine retries every way the input could be split among nested or overlapping quantifiers: exponential in input length. One crafted string can pin a CPU core (ReDoS). Python 3.11+ has atomic groups `(?>…)` and possessive `*+`; .NET has atomic groups and `RegexOptions.NonBacktracking`; JavaScript has neither.

```
const bad = /^(a+)+$/;
const s = "a".repeat(26) + "!";
bad.test(s);  // ~0.5 s; doubles per extra "a"
/^a+$/.test(s); // false, instantly
```

> **Warning:** Run untrusted input only against linear-shaped patterns, and set a timeout where the engine supports one. .NET's default `matchTimeout` is infinite.

Dialects and testing

Feature	JavaScript	Python `re` · .NET
`\d` matches	ASCII 0-9	Any Unicode digit · same
`\p{L}`	With `u` or `v`	No (`regex` pkg) · yes
Variable lookbehind	Yes	No · yes
Atomic `(?>…)`	No	3.11+ · yes

Feature	JavaScript	Python `re` · .NET
Whole-string match	`^…$`	`fullmatch` · `^…\z`

Test a pattern against inputs that should fail, not just ones that should pass: empty string, trailing newline, non-ASCII digits, and a long near-miss for backtracking. Keep those cases in a unit test next to the pattern.

```
import re
re.search(r"^\d+$", "٤٢")          # matches!
re.search(r"^\d+$", "٤٢", re.A)    # None
re.prefixmatch(r"ab", "abc")       # 3.15 name
```

> **Note:** Python 3.15 adds `re.prefixmatch()` as the clearer name for `re.match()`, which only matches at the start of the string; `match` stays as a soft-deprecated alias.

Further reading

- MDN — Regular expressions guide
- ECMAScript — RegExp objects
- Python — re module
- .NET — Regular expression language quick reference
- .NET — Backtracking in regular expressions