Regex cheat sheet

Every token, construct, flag, and gotcha — 12 pages. Print it. Pin it.

CHARACTER CLASSES

`.`	Any character except newline (any with /s)
`\d`	Digit [0-9] (Unicode-aware in some engines)
`\D`	Non-digit
`\w`	Word char [A-Za-z0-9_]
`\W`	Non-word char
`\s`	Whitespace (space, tab, newline, etc.)
`\S`	Non-whitespace
`\h \v`	Horizontal / vertical whitespace (PCRE)
`[abc]`	Any of a, b, or c
`[^abc]`	NOT a, b, or c (negated class)
`[a-z]`	Range: lowercase letters
`[a-zA-Z0-9]`	Combined ranges
`\p{L}`	Any Unicode letter (with /u flag)
`\p{N}`	Any Unicode number
`\P{L}`	Anything that is NOT a letter
`[[:alpha:]]`	POSIX class (letter; in PCRE/Ruby)

ANCHORS & BOUNDARIES

`^`	Start of string (start of line with /m)
`$`	End of string (end of line with /m)
`\A`	Absolute start of input (PCRE/Python/Java)
`\z`	Absolute end of input
`\Z`	End (before optional trailing \n)
`\b`	Word boundary (between \w and \W)
`\B`	Non-word boundary
`\G`	Match start position (PCRE/Java)
`\K`	Reset match start (PCRE; "keep" what follows)

QUANTIFIERS

`*`	0 or more (greedy)
`+`	1 or more (greedy)
`?`	0 or 1 (optional)
`{n}`	Exactly n times
`{n,}`	n or more times
`{n,m}`	Between n and m times
`*? +? ??`	Lazy: match minimum
`*+ ++ ?+`	Possessive: never give back

GROUPS & ALTERNATION

`(abc)`	Capture group
`(?:abc)`	Non-capturing group
`(?<x>abc)`	Named group (JS/.NET/Java/PCRE)
`(?P<x>abc)`	Named group (Python)
`\1 \2`	Backref to group 1, 2
`\k<x>`	Named backref
`(?>abc)`	Atomic group (no backtrack)
`(?|...)`	Branch reset (PCRE)
`a|b|c`	Alternation: a OR b OR c

ESCAPE SEQUENCES

`\n \r \t`	Newline, CR, tab
`\v \f \0`	Vertical tab, form feed, null
`\xFF`	Hex byte (FF)
`\uFFFF`	Unicode code point
`\u{1F600}`	Code point (any size; needs /u)
`\cA`	Control char (Ctrl-A)
`\Q...\E`	Literal-text block (PCRE/Java)

FLAGS

`g`	Global (find all matches)
`i`	Case-insensitive
`m`	Multiline (^$ match line boundaries)
`s`	Dotall (. matches newlines)
`u`	Unicode (JS, full code-point matching)
`y`	Sticky (JS, anchored to lastIndex)
`d`	Has indices (JS, ES2022)
`x`	Verbose (whitespace + # comments)
`A`	ASCII-only \w \d (Python)

TIP Paste any regex into regexguide.com/index.html#tool for a token-by-token plain-English breakdown.

Advanced syntax

Lookarounds, backreferences, conditional patterns, modes.

LOOKAROUNDS (ZERO-WIDTH)

`(?=foo)`	Positive lookahead — followed by foo
`(?!foo)`	Negative lookahead — NOT followed by foo
`(?<=foo)`	Positive lookbehind — preceded by foo
`(?<!foo)`	Negative lookbehind — NOT preceded by foo

Lookarounds match positions, not characters; they consume zero input. JS got lookbehind in ES2018. Go (RE2) has no lookarounds at all. Python's re module requires fixed-width lookbehind; the regex package supports variable-width.

CONDITIONAL PATTERNS

`(?(1)yes|no)`	If group 1 matched, match yes, else no
`(?(<x>)yes|no)`	Conditional on named group
`(?(?=...)yes|no)`	Conditional on lookahead

Conditionals: PCRE, .NET, Boost. Not in JS, Python re, Go, or stdlib Ruby.

RECURSION

`(?R)  (?0)`	Recurse entire pattern (PCRE)
`(?1)  (?2)`	Recurse into group 1, 2 (PCRE)
`(?&name)`	Recurse into named group
`(?(DEFINE)...)`	Define subroutines (PCRE)

INLINE MODE MODIFIERS

`(?i)`	Turn on case-insensitive
`(?-i)`	Turn off case-insensitive
`(?im)`	Multiple flags on
`(?i:abc)`	Apply flags to group only
`(?x)`	Verbose: ignore whitespace + #comments

REPLACEMENT SYNTAX

`$&  \0`	Entire match
`$1  $2`	Capture group 1, 2 (JS/PCRE/.NET)
`\1  \2`	Capture group (Python re.sub, sed)
`$<x>  ${x}`	Named group (JS / .NET)
`\g<x>`	Named group (Python)
`$$  \\`	Literal $ / backslash
`$` $'`	Text before / after match
`\u  \l  \U  \L`	Upper/lower next char or block
`\E`	End \U or \L block

ATOMIC & POSSESSIVE

`(?>abc)`	Atomic group — refuse to backtrack
`a++  a*+  a?+`	Possessive quantifiers
`a{n,m}+`	Possessive {n,m}

Prevent catastrophic backtracking on adversarial input. Available: Java, PCRE, .NET, Ruby, JS ES2024. Not in Python re or Go.

MUST-ESCAPE (OUTSIDE [...])

`.  *  +  ?  ^  $  |  (  )  [  ]  {  }  \`

Inside [...]: only] ^ - \ stay special. Inside [...]: dot, plus, star, etc. are literal.

COMMENTS

`(?#comment)`	Inline comment
`# in /x mode`	Comment to end of line

Language-specific syntax

Differences across JavaScript, Python, Java, .NET, Go, Ruby, PHP/PCRE.

FEATURE SUPPORT MATRIX

Feature	JS	Python	Java	.NET	Go (RE2)	Ruby	PHP/PCRE
Lookahead (?=)	✓	✓	✓	✓	✗	✓	✓
Lookbehind (?<=)	ES2018	fixed	✓	✓ var	✗	✓	✓
Named groups	(?<x>)	(?P<x>)	(?<x>)	(?<x>)	(?P<x>)	(?<x>)	(?<x>)
Atomic group (?>)	ES2024	✗	✓	✓	✗	✓	✓
Possessive a++	ES2024	✗	✓	✓	✗	✓	✓
Unicode \p{L}	/u	regex pkg	✓	✓	utf8 pkg	✓	✓ /u
Recursion (?R)	✗	✗	✗	✓ bal.	✗	✗	✓
Verbose /x	✗	✓	✓	✓	✗	✓	✓
Conditional (?(1)y|n)	✗	✗	✗	✓	✗	✗	✓
Backrefs in lookbehind	✓	✗	✓	✓	✗	✓	✓
Match timeout	✗	✗	✗	✓	use ctx	✗	✗
Linear-time guarantee	✗	✗	✗	NonBT	✓ RE2	✗	✗

QUICK START — TEST IF STRING MATCHES

JavaScript: `/^\d+$/.test("123")  →  true`

Python: `re.match(r"^\d+$", "123")  →  Match object`

Java: `Pattern.compile("^\\d+$").matcher("123").matches()  →  true`

C# / .NET: `Regex.IsMatch("123", @"^\d+$")  →  true`

Go: `regexp.MustCompile(`^\d+$`).MatchString("123")  →  true`

Ruby: `"123" =~ /^\d+$/  →  0   (truthy in Ruby)`

PHP: `preg_match('/^\d+$/', "123")  →  1`

Rust: `Regex::new(r"^\d+$").unwrap().is_match("123")  →  true`

Swift: `"123".range(of: "^\\d+$", options: .regularExpression) != nil`

STRING-ESCAPE PITFALL

When the regex is in a string literal (not a regex literal), every backslash is interpreted twice — first by the string parser, then by the regex engine. To match \d you typically write "\\d" in source. Use raw strings to skip the first layer: Python r"\d", C# @"\d", Go `\d`, Rust r"\d". JS regex literals /\d/ also skip it; new RegExp("\\d") does not.

Regex in editors & on the command line

grep, ripgrep, sed, awk, Vim, VS Code, IntelliJ — what each flavor expects.

CLI TOOLS — REGEX FLAVORS

grep	BRE by default. -E for ERE, -P for PCRE (GNU). -F = fixed strings (faster, no regex).
egrep	Deprecated; same as grep -E.
ripgrep (rg)	Rust regex crate (similar to RE2). Fast, Unicode-aware. -P for PCRE2 fallback.
ag (silver)	PCRE. -Q for fixed strings.
ack	Perl regex (PCRE-like).
sed	BRE by default. -E for ERE (most modern seds). For Perl-style, use perl -pe instead.
awk	ERE. /regex/ in patterns, gsub("re",rep,str) for replace.
perl -pe	Perl regex — closest to PCRE. The grandparent of modern regex.
find -regex	GNU find: emacs regex by default; -regextype posix-extended for ERE.

BRE VS ERE — ESCAPING

BRE	() {} ? + | need backslash: \(\) \{ \} \? \+
ERE	() {} ? + | work without backslash
BRE	\b \w \s (GNU extensions, not POSIX)
PCRE	lookarounds, named groups, possessive, etc.

EDITOR FLAVORS

VS Code
Custom (similar to JS). Has lookarounds, named groups. Use $1 in replace.

Sublime Text
Boost.Regex (PCRE-style).

IntelliJ / WebStorm
Java's Pattern engine. Has atomic + possessive.

Vim
Idiosyncratic. \v = "very magic" (closer to PCRE). \(\) capture; \1 backref.

Emacs
Own flavor. \(\) capture; \b \w supported. M-x query-replace-regexp.

Notepad++
Boost.Regex (PCRE).

TextMate / BBEdit
Oniguruma (Ruby's engine).

Atom (legacy)
JS regex via Oniguruma extension.

JetBrains (RubyMine etc)
Java's Pattern engine across all JetBrains IDEs.

DATABASES

PostgreSQL	POSIX ERE. ~ matches; ~* case-insensitive.
MySQL	ICU regex (8.0+). REGEXP_LIKE, REGEXP_REPLACE.
SQLite	No regex by default. Use REGEXP extension.
Oracle	POSIX-ish. REGEXP_LIKE, REGEXP_SUBSTR.
MongoDB	PCRE-compatible.
Elasticsearch	Lucene flavor — atomic groups not supported.

WARNING Don't copy a regex blindly between tools. Test it in the target flavor first. Atomic groups (?>), recursion (?R), and conditionals are common breakage points.

Common patterns

Battle-tested patterns for the data you match most often.

IDENTITY & CONTACT

Email (practical)
```
^[\w.+-]+@[\w-]+\.[\w.-]+$
```

Phone US (loose)
```
^\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}$
```

Phone E.164
```
^\+[1-9]\d{1,14}$
```

URL (http/https)
```
^https?://[^\s/$.?#].\S*$
```

IPv4 strict
```
^((25[0-5]|2[0-4]\d|[01]?\d\d?)\.)
  {3}(25[0-5]|2[0-4]\d|[01]?\d\d?)$
```

IPv6
```
^([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$
```

MAC address
```
^([0-9A-Fa-f]{2}[:-]){5}[0-9A-Fa-f]{2}$
```

UUID v4
```
^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]
  {3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$
```

UUID (any version)
```
^[0-9a-f]{8}-[0-9a-f]{4}-[1-5]
  [0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$
```

DATES & NUMBERS

ISO date YYYY-MM-DD
```
^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])$
```

ISO datetime
```
^\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d+)?Z?$
```

US date MM/DD/YYYY
```
^(0[1-9]|1[0-2])/(0[1-9]|[12]\d|3[01])/\d{4}$
```

24-hour time HH:MM
```
^([01]\d|2[0-3]):[0-5]\d$
```

Unix timestamp (s)
```
^\d{10}$
```

Unix timestamp (ms)
```
^\d{13}$
```

Hex color
```
^#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})$
```

Integer (signed)
```
^-?\d+$
```

Decimal (signed)
```
^-?\d+(\.\d+)?$
```

Scientific notation
```
^-?\d+(\.\d+)?[eE][-+]?\d+$
```

Currency (USD)
```
^\$?-?\d{1,3}(,\d{3})*(\.\d{2})?$
```

CODES & DEV

Slug (URL-safe)
```
^[a-z0-9]+(-[a-z0-9]+)*$
```

Username (alphanum+_)
```
^[A-Za-z0-9_]{3,20}$
```

Strong password
```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*\W).{8,}$
```

Semver
```
^\d+\.\d+\.\d+(-[A-Za-z0-9.-]+)?(\+[A-Za-z0-9.-]+)?$
```

JWT
```
^[A-Za-z0-9_-]+\.[A-Za-z0-9_-]+\.[A-Za-z0-9_-]+$
```

Base64
```
^[A-Za-z0-9+/]+={0,2}$
```

Base64 URL-safe
```
^[A-Za-z0-9_-]+={0,2}$
```

Hex (any length)
```
^[0-9a-fA-F]+$
```

SHA-256
```
^[a-fA-F0-9]{64}$
```

Credit card (loose)
```
^\d{13,19}$  then run Luhn check
```

IBAN
```
^[A-Z]{2}\d{2}[A-Z0-9]{1,30}$
```

SWIFT/BIC
```
^[A-Z]{6}[A-Z0-9]{2}([A-Z0-9]{3})?$
```

Regional ID patterns

United States, United Kingdom, and European Union identifiers.

UNITED STATES

ZIP code (5)
```
^\d{5}$
```

ZIP+4
```
^\d{5}-\d{4}$
```

SSN
```
^\d{3}-\d{2}-\d{4}$
```

EIN
```
^\d{2}-\d{7}$
```

Phone US/Canada
```
^\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}$
```

State abbrev
```
^[A-Z]{2}$  validate against list
```

Passport (US)
```
^[A-Z0-9]{6,9}$
```

UNITED KINGDOM

Postcode (full)
```
^[A-Z]{1,2}\d[A-Z\d]? ?\d[A-Z]{2}$
```

NI number
```
^[A-CEGHJ-PR-TW-Z]{2}\d{6}[A-D]$
```

VAT number
```
^GB(\d{9}|\d{12}|GD\d{3}|HA\d{3})$
```

UTR (tax ref)
```
^\d{10}$
```

Phone (loose)
```
^(\+44|0)\d{10}$
```

Sort code
```
^\d{2}-\d{2}-\d{2}$
```

Bank account
```
^\d{8}$
```

EUROPEAN UNION

IBAN (any EU country)
```
^[A-Z]{2}\d{2}[A-Z0-9]{1,30}$
```

VAT (general EU)
```
^[A-Z]{2}[A-Z0-9]{2,12}$
```

VAT Germany
```
^DE\d{9}$
```

VAT France
```
^FR[A-HJ-NP-Z0-9]{2}\d{9}$
```

VAT Italy
```
^IT\d{11}$
```

VAT Spain
```
^ES[A-Z0-9]\d{7}[A-Z0-9]$
```

Postcode Germany
```
^\d{5}$
```

Postcode France
```
^\d{5}$
```

Postcode Netherlands
```
^\d{4} ?[A-Z]{2}$
```

Postcode Italy
```
^\d{5}$
```

OTHER REGIONS

Canada postal
```
^[A-Z]\d[A-Z] ?\d[A-Z]\d$
```

Australia post
```
^\d{4}$
```

Japan postal
```
^\d{3}-\d{4}$
```

Brazil CEP
```
^\d{5}-\d{3}$
```

Brazil CPF
```
^\d{3}\.\d{3}\.\d{3}-\d{2}$
```

Russia INN (10)
```
^\d{10}$
```

China ID (18)
```
^\d{17}[\dX]$
```

South Korea RRN
```
^\d{6}-\d{7}$
```

NOTE These patterns validate format only. For tax IDs especially, verify against the issuing authority (HMRC, VIES, etc.). Many include checksums regex can't enforce.

India-specific patterns

IDs, tax, banking, vehicle, and telecom formats used across India.

GOVERNMENT IDS

Aadhaar (12-digit UID)
```
^[2-9]\d{11}$
```

PAN
```
^[A-Z]{5}\d{4}[A-Z]$
```

TAN
```
^[A-Z]{4}\d{5}[A-Z]$
```

Voter ID (EPIC)
```
^[A-Z]{3}\d{7}$
```

Passport
```
^[A-PR-WYa-pr-wy][0-9]{7}$
```

Driving Licence
```
^[A-Z]{2}\d{2}[A-Z0-9]{11,13}$
```

Ration card
```
^[A-Z]{3}-?\d{2}-?\d{3}-?\d{6}$
```

TAX & BUSINESS

GSTIN
```
^\d{2}[A-Z]{5}\d{4}[A-Z][1-9A-Z]Z[0-9A-Z]$
```

HSN code
```
^\d{4,8}$
```

SAC code
```
^99\d{4}$
```

CIN
```
^[LU]\d{5}[A-Z]{2}\d{4}[A-Z]{3}\d{6}$
```

DIN
```
^\d{8}$
```

Udyam (MSME)
```
^UDYAM-[A-Z]{2}-\d{2}-\d{7}$
```

LLPIN
```
^[A-Z]{3}-\d{4}$
```

BANKING & PAYMENTS

IFSC
```
^[A-Z]{4}0[A-Z0-9]{6}$
```

UPI ID
```
^[\w.-]+@[\w.-]+$
```

MICR
```
^\d{9}$
```

Bank A/c (loose)
```
^\d{9,18}$
```

RTGS UTR
```
^[A-Z]{4}[A-Z0-9]{6}\d{12}$
```

Cheque MICR line
```
^\d{6} \d{9} \d{6} \d{2}$
```

PHONE & ADDRESS

Mobile (10-digit)
```
^[6-9]\d{9}$
```

Mobile with +91
```
^\+91[\s-]?[6-9]\d{9}$
```

Landline w/ STD
```
^(\+91|0)?\d{2,4}[\s-]?\d{6,8}$
```

PIN code
```
^[1-9]\d{5}$
```

Rupee amount
```
^₹\s?-?\d{1,3}(,\d{3})*(\.\d{1,2})?$
```

Lakh format
```
^\d{1,2}(,\d{2})*,\d{3}$
```

VEHICLE PLATES

Generic (any state)
```
^[A-Z]{2}\d{1,2}[A-Z]{1,3}\d{4}$
```

Maharashtra (MH)
```
^MH\d{1,2}[A-Z]{1,3}\d{4}$
```

Delhi (DL)
```
^DL\d{1,2}[A-Z]{1,3}\d{4}$
```

Bharat (BH series)
```
^\d{2}BH\d{4}[A-Z]{2}$
```

PAN ENTITY TYPES (4TH LETTER)

Individual
```
P  — fourth letter for personal PAN
```

Company
```
C  — used for companies
```

HUF
```
H  — Hindu Undivided Family
```

Firm / Trust
```
F (firm) / T (trust)
```

Other
```
A / B / G / J / L
```

CHECKSUMS Aadhaar uses Verhoeff. IFSC: 5th char is always 0. GSTIN: 14th = Z, 15th is checksum. Run the algorithm after regex.

Unicode in regex

Properties, scripts, blocks, code points, and grapheme clusters.

PROPERTY CLASSES

`\p{L}`	Any letter (any script)
`\p{Ll}`	Lowercase letter
`\p{Lu}`	Uppercase letter
`\p{Lt}`	Title-case letter
`\p{N}`	Any number
`\p{Nd}`	Decimal digit
`\p{Nl}`	Letter-like number (e.g. Roman)
`\p{P}`	Punctuation
`\p{S}`	Symbol
`\p{Z}`	Separator (spaces)
`\p{M}`	Mark (combining accents)
`\p{C}`	Other (control, format, unassigned)
`\P{L}`	Negated — NOT a letter

SCRIPT PROPERTIES

`\p{Latin}`	Latin alphabet
`\p{Greek}`	Greek letters
`\p{Cyrillic}`	Russian etc.
`\p{Arabic}`	Arabic
`\p{Hebrew}`	Hebrew
`\p{Han}`	Chinese characters
`\p{Hiragana}`	Japanese hiragana
`\p{Katakana}`	Japanese katakana
`\p{Devanagari}`	Hindi, Marathi, Sanskrit
`\p{Script=Tamil}`	Tamil (full prefix syntax)

CODE POINTS & SURROGATES

`\uFFFF`	4-hex BMP code point
`\u{1F600}`	Any code point (needs /u in JS)
`\x{1F600}`	PCRE/Perl syntax
`\N{U+1F600}`	.NET / Boost.Regex
`\N{GRINNING FACE}`	By Unicode name (PCRE)

Emoji often span two UTF-16 code units (surrogate pair). Without /u, JavaScript regex sees each half separately — . will match half an emoji.

EMOJI BLOCKS (COMMON)

`U+1F300-1F5FF`	Misc symbols & pictographs
`U+1F600-1F64F`	Emoticons
`U+1F680-1F6FF`	Transport & map
`U+1F900-1F9FF`	Supplemental symbols
`U+2600-26FF`	Misc symbols
`U+2700-27BF`	Dingbats

GRAPHEME CLUSTERS

A user-perceived "character" can be many code points: emoji + skin tone + ZWJ. Regex matches code points, not graphemes.

`\X`	Match one grapheme cluster (PCRE, Java, Ruby)
`\p{Grapheme_Base}`	Base character (with properties)

CASE FOLDING

Unicode case is complex: German ß uppercases to SS (two letters); Turkish i has dotless variants. /i flag handles simple casing; locale-aware folding may need ICU.

JAVASCRIPT Without /u, \p{L} is a literal "p{L}" string (Node/Chrome). Always use /u with Unicode property escapes. Without /u, surrogate pairs count as 2 chars.

Email regex — annotated walkthrough

A token-by-token breakdown of the pragmatic email pattern.

```
^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$
```

`^`	Start of string. Requires matching from the first character.
`[A-Za-z0-9._%+-]`	Local-part char class — letters, digits, plus +._%- which RFC 5321 allows in the unquoted local part. Excludes spaces, @, /, etc.
`+`	One or more local-part characters. Minimum length 1. No upper bound (could constrain with {1,64} for the RFC limit).
`@`	Literal at-sign separator between local part and domain.
`[A-Za-z0-9.-]`	Domain label characters: letters, digits, dot, hyphen. Doesn't enforce that dots can't be consecutive or that hyphens can't be at edges.
`+`	One or more domain characters.
`\.`	Escaped dot — the literal "." separator before the TLD. Without the backslash, . would match any character.
`[A-Za-z]`	TLD must be alphabetic.
`{2,}`	TLD must be at least 2 characters. No upper bound (modern TLDs like .museum are valid).
`$`	End of string. Rejects trailing whitespace, comments, or extra text.

WHAT THIS REGEX DELIBERATELY DOESN'T DO

It doesn't enforce RFC 5322 (which allows quoted strings, comments, and IP-literal domains — almost nothing rejects an email like "foo bar"@example.com). It doesn't check that the domain exists or that the mailbox accepts mail (use DNS MX + SMTP probe). It doesn't reject consecutive dots in the local part (illegal in RFC 5321). Pragmatically: use this to catch obvious typos, then verify deliverability with a real email service.

Performance & ReDoS

Catastrophic backtracking, anti-patterns, and how to fix them.

VULNERABLE PATTERNS

`(a+)+`	Nested + on the same char — classic
`(a*)*`	Same idea with star
`(.*)*`	"Wildcard times wildcard"
`(\w+)*`	Word chars wrapped in *
`(a|a?)+`	Optional inside +
`(a|aa)+`	Overlapping alternatives
`^(\w+\s?)*$`	Real-world example — common parser

FIXES

`a+`	Just use one quantifier
`(?>a+)`	Atomic group: never backtrack
`a++`	Possessive: never give back
`[^a]+a`	Negative class — avoid lookahead loop
`Use RE2`	Go / re2 / .NET NonBacktracking
`Set timeout`	.NET Regex(pattern, opt, TimeSpan)

HOW TO TEST FOR REDOS

Take your regex, identify any nested quantifiers or alternations with overlap, then construct input that almost matches: e.g. "a" × N + "X". A safe regex takes time roughly proportional to N; a vulnerable one takes exponential time. Try N=10, 20, 30 — if 30 chars takes seconds, you have ReDoS.

```
# JavaScript test
const re = /^(a+)+$/;
console.log(performance.now());
re.test('a'.repeat(30) + 'X');   // ~4s
console.log(performance.now());
```

OPTIMIZATION TIPS

- Anchor your regex (^...$) — searching unbounded text is slower.
- Prefer negated character classes [^x]* over .*?x — less backtracking.
- Compile once, reuse — every language has a way to cache compiled regex.
- Use possessive or atomic groups for parts that won't need to backtrack.
- Profile in production with a representative input set — synthetic benchmarks lie.
- If you control the engine choice, RE2-family engines guarantee linear time.
- Replace `(?=.*A)(?=.*B)` (lookahead chains) with set logic when possible.

DANGER CVE-2017-16002 (Express.js), CVE-2018-3737 (sshpk) — both real exploits caused by vulnerable regex on user-controlled input. Always set a timeout or use a linear-time engine when matching untrusted input.

Testing & debugging regex

How to be confident your regex works — and how to fix it when it doesn't.

TEST CASES CHECKLIST

☐ **Empty string**
Does "" match or not? Often a bug.

☐ **Single char**
Does "a" match when you expect 1+?

☐ **Maximum size**
Hit your upper limit — {n,m} edge.

☐ **Just whitespace**
" " " " "\t\n" — usually shouldn't match.

☐ **Leading / trailing space**
" abc" / "abc "

☐ **Unicode chars**
café, □□, Δ, □

☐ **Mixed case**
AbC, aBC, abc — does /i matter?

☐ **Special chars**
<>&"' inside the input.

☐ **Newlines inside**
"a\nb" — does . match \n?

☐ **Adversarial**
What you fear: SQL injection, XSS, long input.

COMMON BUGS & FIXES

Pattern matches "" unexpectedly
You have a * (zero or more) somewhere — try +.

Greedy match goes too far
Change .* to .*?, or use a negated class [^x]*.

Anchors don't work as expected
Check the /m flag. ^ becomes per-line.

Backref doesn't fire
Group is non-capturing (?:) — remove the ?:

Lookbehind fails (Python)
Variable-width not supported in re; use regex module.

Pattern is way too slow
Catastrophic backtracking — see Performance page.

Unicode chars don't match
Add /u (JS) or use \p{L} for letters.

. doesn't match newline
Add /s flag (dotall) or use [\s\S].

Match across multiple lines
Set /m flag and use ^/$ per line.

DEBUG STRATEGIES

1. Use a visualizer: regexguide.com's explainer shows what each token does — fastest way to spot a misunderstanding.
2. Simplify in steps: comment out parts (with /x mode or by editing) until the bug appears or disappears.
3. Test both directions: confirm what should match does, AND that what shouldn't match doesn't.
4. Print the actual match: log m.group(0), m.start(), m.end() to see what regex thinks it matched.
5. Reduce greediness: convert greedy quantifiers to lazy and see if behavior changes.
6. Check flags: case, multiline, dotall, Unicode — many bugs come from flag mismatch.

Recipes, gotchas & glossary

One-liners you'll write hundreds of times, plus terminology.

COMMON RECIPES

Strip HTML tags
```
s.replace(/<[^>]*>/g, "")
```

Collapse whitespace
```
s.replace(/\s+/g, " ").trim()
```

Extract URLs
```
s.match(/https?:\/\/\S+/g)
```

Extract numbers
```
s.match(/-?\d+(\.\d+)?/g)
```

Find @mentions
```
s.match(/@\w+/g)
```

Find #hashtags
```
s.match(/#\w+/g)
```

Remove emojis
```
s.replace(/\p{Emoji}/gu, "")
```

Mask credit card
```
s.replace(/\d(?=\d{4})/g, "*")
```

Duplicate words
```
/\b(\w+)\s+\1\b/gi
```

Trim each line
```
s.replace(/^[ \t]+|[ \t]+$/gm, "")
```

Between two markers
```
/\[\[([\s\S]*?)\]\]/
```

Camel → snake_case
```
s.replace(/([A-Z])/g, "_$1").toLowerCase()
```

Strip ANSI codes
```
s.replace(/\x1b\[[0-9;]*m/g, "")
```

Validate then capture
```
/^(\d{4})-(\d{2})-(\d{2})$/
```

GLOSSARY

Anchor
Zero-width token matching a position, not a character (^, $, \b).

Atom
A single regex unit — a char, class, group, or escape.

Atomic group
Group that refuses to give up matches once made — prevents backtracking.

Backreference
Matching the text captured by a previous group: \1, \k<name>.

Backtracking
Engine's "try again with less greedy" behavior when a match fails.

Boundary
Position between two character types — \b at word/non-word transitions.

Capture group
(parens) — saved match available as backref or in result.

Character class
[set] of chars to match. Shorthand: \d \w \s.

DFA
Deterministic finite automaton — linear-time engine (RE2, awk).

Flag
Modifier that changes engine behavior: g, i, m, s, u, x.

Greedy
Default — quantifier matches as much as possible.

Lazy
?-suffixed quantifier — matches as little as possible.

Lookaround
Zero-width assertion: (?=...) (?!...) (?<=...) (?<!...)

NFA
Non-deterministic FA — most common engines; supports backreferences.

Possessive
a++ — match and never give back; faster than atomic group.

Quantifier
How many times: * + ? {n} {n,m}

ReDoS
Regex Denial of Service — exponential time on crafted input.

SHOULD I USE REGEX?

Validating shape (email, phone, date)	**YES**	— but follow with a real validator for semantics.
Parsing structured data (JSON, XML, HTML)	**NO**	— use a real parser. Regex can't do recursion well.
Extracting one specific token from text	**YES**	— regex is perfect for this.
Splitting on a complex delimiter	**YES**	— most languages' .split() takes a regex.
Matching balanced brackets / quotes	**MOSTLY NO**	— only PCRE/.NET handle recursion.
Searching code (logs, source, configs)	**YES**	— grep/ripgrep are unbeatable for this.

KEEP THIS HANDY For interactive practice, paste any regex into regexguide.com/index.html#tool. Browse 300+ ready-made patterns at regexguide.com/patterns.html.