



## Regex Basics & Metacharacters

### Basic Matching

<code>literal</code>	Matches the literal character sequence. Example: <code>abc</code> matches 'abc'.
<code>.</code> (dot)	Matches any single character except newline. Example: <code>a.c</code> matches 'abc', 'adc', 'aec', etc.
<code>^</code>	Matches the beginning of the string. Example: <code>^abc</code> matches 'abcdef', but not 'defabc'.
<code>\$</code>	Matches the end of the string. Example: <code>abc\$</code> matches 'defabc', but not 'abcdef'.
<code>[]</code>	Character class: Matches any single character within the brackets. Example: <code>[abc]</code> matches 'a', 'b', or 'c'.
<code>[^]</code>	Negated character class: Matches any single character <i>not</i> within the brackets. Example: <code>[^abc]</code> matches any character except 'a', 'b', or 'c'.
<code> </code>	Alternation: Matches either the expression before or after the <code> </code> . Example: <code>cat dog</code> matches 'cat' or 'dog'.

### Quantifiers

<code>*</code>	Matches the preceding character or group zero or more times. Example: <code>ab*c</code> matches 'ac', 'abc', 'abbc', 'abbbc', etc.
<code>+</code>	Matches the preceding character or group one or more times. Example: <code>ab+c</code> matches 'abc', 'abbc', 'abbbc', etc., but not 'ac'.
<code>?</code>	Matches the preceding character or group zero or one time. Example: <code>ab?c</code> matches 'ac' or 'abc'.
<code>{n}</code>	Matches the preceding character or group exactly <code>n</code> times. Example: <code>ab{2}c</code> matches 'abbc'.
<code>{n,}</code>	Matches the preceding character or group <code>n</code> or more times. Example: <code>ab{2,}c</code> matches 'abbc', 'abbbc', 'abbbbc', etc.
<code>{n,m}</code>	Matches the preceding character or group between <code>n</code> and <code>m</code> times (inclusive). Example: <code>ab{2,4}c</code> matches 'abbc', 'abbbc', and 'abbbbc'.

### Character Classes

<code>\d</code>	Matches any digit (0-9). Equivalent to <code>[0-9]</code> .
<code>\D</code>	Matches any non-digit character. Equivalent to <code>[^0-9]</code> .
<code>\w</code>	Matches any word character (alphanumeric and underscore). Equivalent to <code>[a-zA-Z0-9_]</code> .
<code>\W</code>	Matches any non-word character. Equivalent to <code>[^a-zA-Z0-9_]</code> .
<code>\s</code>	Matches any whitespace character (space, tab, newline, etc.).
<code>\S</code>	Matches any non-whitespace character.

## Grouping and Backreferences

### Grouping

<code>( )</code>	Groups the enclosed pattern. Allows you to apply quantifiers or alternations to the entire group. Also captures the matched group for backreferencing.
<code>(?: )</code>	Non-capturing group. Groups the pattern but does <i>not</i> capture the matched group. Useful for performance or when you don't need the captured value.

### Backreferences

<code>\1</code> , <code>\2</code> , etc.	Refers to the first, second, etc. captured group in the regex. Example: <code>(.)\2\1</code> matches 'abba'.
<code>\$1</code> , <code>\$2</code> , etc. (in replacement strings)	Refers to the first, second, etc. captured group in the replacement string of a substitution operation.

### Examples

Match a date in <code>YYYY-MM-DD</code> format: <code>\d{4}-\d{2}-\d{2}</code>
Match an email address (simplified): <code>\w+@\w+\.\w+</code>
Match HTML tags: <code>&lt;[&gt;]+&gt;</code>

## Anchors and Lookarounds

### Anchors

<code>^</code>	Matches the beginning of the string (or line, in multiline mode).
<code>\$</code>	Matches the end of the string (or line, in multiline mode).
<code>\b</code>	Matches a word boundary (the position between a word character and a non-word character).
<code>\B</code>	Matches a non-word boundary.

### Lookarounds

<code>(?=pattern)</code>	Positive lookahead: Asserts that the pattern <i>follows</i> the current position, but does not consume the characters. Example: <code>\w+(?= \d)</code> matches 'abc' in 'abc123', but not 'abc' in 'abc def'.
<code>(?!pattern)</code>	Negative lookahead: Asserts that the pattern does <i>not</i> follow the current position. Example: <code>\w+(?! \d)</code> matches 'abc' in 'abc def', but not 'abc' in 'abc123'.
<code>(?&lt;=pattern)</code>	Positive lookbehind: Asserts that the pattern <i>precedes</i> the current position, but does not consume the characters. Example: <code>(?&lt; \d)\w+</code> matches 'abc' in '123abc', but not 'abc' in 'abc def'.
<code>(?&lt;!pattern)</code>	Negative lookbehind: Asserts that the pattern does <i>not</i> precede the current position. Example: <code>(?&lt;! \d)\w+</code> matches 'abc' in 'abc def', but not 'abc' in '123abc'.

# Flags/Modifiers

## Common Flags

i	Case-insensitive matching. Example: <code>/abc/i</code> matches 'abc', 'ABC', 'aBc', etc.
g	Global matching. Finds all matches instead of stopping after the first.
m	Multiline mode. <code>^</code> and <code>\$</code> match the beginning and end of each line (delimited by <code>\n</code> ).
s	Dotall mode. Allows the <code>.</code> to match newline characters as well.
x	Verbose mode. Allows whitespace and comments in the regex pattern for better readability. Whitespace is ignored, and comments start with <code>#</code> .

## Using Flags (Examples)

In Python:

```
import re

pattern = re.compile('abc', re.IGNORECASE) # Case-insensitive
matches = pattern.findall('aBcAbc')

print(matches) # Output: ['aBc', 'Abc']
```

In JavaScript:

```
const regex = /abc/i; // Case-insensitive
const matches = 'aBcAbc'.match(regex);
console.log(matches); // Output: ['aBc', index: 0, input: 'aBcAbc', groups: undefined]

const regexGlobal = /abc/gi; // Global and case-insensitive
const allMatches = 'aBcAbc'.match(regexGlobal);
console.log(allMatches); // Output: [ 'aBc', 'Abc' ]
```

In Ruby:

```
pattern = /abc/i # Case-insensitive
matches = 'aBcAbc'.scan(pattern)

puts matches # Output: aBc
puts matches.count # Output: 2
```