Regular expression syntax cheatsheet

This page provides an overall cheat sheet of all the capabilities of `RegExp` syntax by aggregating the content of the articles in the `RegExp` guide. If you need more information on a specific topic, please follow the link on the corresponding heading to access the full article or head to the guide.

Character classes

Characters	Meaning
.	Has one of the following meanings: • Matches any single character *except* line terminators: `\n`, `\r`, `\u2028` or `\u2029`. For example, `/.y/` matches "my" and "ay", but not "yes", in "yes make my day". • Inside a character set, the dot loses its special meaning and matches a literal dot. Note that the `m` multiline flag doesn't change the dot behavior. So to match a pattern across multiple lines, the character set `[^]` can be used — it will match any character including newlines. ES2018 added the `s` "dotAll" flag, which allows the dot to also match line terminators.
`\d`	Matches any digit (Arabic numeral). Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches "2" in "B2 is the suite number".

Characters	Meaning
`\D`	Matches any character that is not a digit (Arabic numeral). Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches "B" in "B2 is the suite number".
`\w`	Matches any alphanumeric character from the basic Latin alphabet, including the underscore. Equivalent to `[A-Za-z0-9_]`. For example, `/\w/` matches "a" in "apple", "5" in "$5.28", and "3" in "3D".
`\W`	Matches any character that is not a word character from the basic Latin alphabet. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^A-Za-z0-9_]/` matches "%" in "50%".
`\s`	Matches a single white space character, including space, tab, form feed, line feed, and other Unicode spaces. Equivalent to `[\f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\s\w*/` matches " bar" in "foo bar".
`\S`	Matches a single character other than white space. Equivalent to `[^\f\n\r\t\v\u00a0\u1680\u2000-\u200a\u2028\u2029\u202f\u205f\u3000\ufeff]`. For example, `/\S\w*/` matches "foo" in "foo bar".
`\t`	Matches a horizontal tab.
`\r`	Matches a carriage return.
`\n`	Matches a linefeed.
`\v`	Matches a vertical tab.
`\f`	Matches a form-feed.
`[\b]`	Matches a backspace. If you're looking for the word-boundary character (`\b`), see Boundaries.
`\0`	Matches a NUL character. Do not follow this with another digit.

Characters	Meaning
`\cX`	Matches a control character using caret notation, where "X" is a letter from A–Z (corresponding to codepoints `U+0001`–`U+001F`). For example, `/\cM/` matches "\r" in "\r\n".
`\xhh`	Matches the character with the code *hh* (two hexadecimal digits).
`\uhhhh`	Matches a UTF-16 code-unit with the value *hhhh* (four hexadecimal digits).
`\u{hhhh}` or `\u{hhhhh}`	(Only when the `u` flag is set.) Matches the character with the Unicode value `U+`*hhhh* or `U+`*hhhhh* (hexadecimal digits).
`\`	Indicates that the following character should be treated specially, or "escaped". It behaves one of two ways. • For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character "b". By placing a backslash in front of "b", that is by using `/\b/`, the character becomes special to mean match a word boundary. • For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, "*" is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more "a"s. To match `*` literally, precede it with a backslash; for example, `/a\*/` matches "a*". Note that some characters like `:`, `-`, `@`, etc. neither have a special meaning when escaped nor when unescaped. Escape sequences like `\:`, `\-`, `\@` will be equivalent to their literal, unescaped character equivalents in regular expressions. However, in regular expressions with the unicode flag, these will cause an *invalid identity escape* error. This is done to ensure backward compatibility with existing code that uses new escape sequences like `\p` or `\k`. > To match this character literally, escape it with itself. In other words to search for `\` use `/\\/`.

Assertions

Boundary-type assertions

Characters	Meaning
^	Matches the beginning of input. If the multiline flag is set to true, also matches immediately after a line break character. For example, `/^A/` does not match the "A" in "an A", but does match the first "A" in "An A". This character has a different meaning when it appears at the start of a group.
$	Matches the end of input. If the multiline flag is set to true, also matches immediately before a line break character. For example, `/t$/` does not match the "t" in "eater", but does match it in "eat".
\b	Matches a word boundary. This is the position where a word character is not followed or preceded by another word-character, such as between a letter and a space. Note that a matched word boundary is not included in the match. In other words, the length of a matched word boundary is zero. Examples: • `/\bm/` matches the "m" in "moon". • `/oo\b/` does not match the "oo" in "moon", because "oo" is followed by "n" which is a word character. • `/oon\b/` matches the "oon" in "moon", because "oon" is the end of the string, thus not followed by a word character. • `/\w\b\w/` will never match anything, because a word character can never be followed by both a non-word and a word character. To match a backspace character (`[\b]`), see Character Classes.

Characters	Meaning
`\B`	Matches a non-word boundary. This is a position where the previous and next character are of the same type: Either both must be words, or both must be non-words, for example between two letters or between two spaces. The beginning and end of a string are considered non-words. Same as the matched word boundary, the matched non-word boundary is also not included in the match. For example, `/\Bon/` matches "on" in "at noon", and `/ye\B/` matches "ye" in "possibly yesterday".

Other assertions

> **Note**: The `?` character may also be used as a quantifier.

Characters	Meaning
`x(?=y)`	**Lookahead assertion:** Matches "x" only if "x" is followed by "y". For example, `/Jack(?=Sprat)/` matches "Jack" only if it is followed by "Sprat". `/Jack(?=Sprat|Frost)/` matches "Jack" only if it is followed by "Sprat" or "Frost". However, neither "Sprat" nor "Frost" is part of the match results.
`x(?!y)`	**Negative lookahead assertion:** Matches "x" only if "x" is not followed by "y". For example, `/\d+(?!\.)/` matches a number only if it is not followed by a decimal point. `/\d+(?!\.)/.exec('3.141')` matches "141" but not "3".
`(?<=y)x`	**Lookbehind assertion:** Matches "x" only if "x" is preceded by "y". For example, `/(?<=Jack)Sprat/` matches "Sprat" only if it is preceded by "Jack". `/(?<=Jack|Tom)Sprat/` matches "Sprat" only if it is preceded by "Jack" or "Tom". However, neither "Jack" nor "Tom" is part of the match results.
`(?<!y)x`	**Negative lookbehind assertion:** Matches "x" only if "x" is not preceded by "y". For example, `/(?<!-)\d+/` matches a number only if it is not preceded by a minus sign. `/(?<!-)\d+/.exec('3')` matches "3". `/(?<!-)\d+/.exec('-3')` match is not found because the number is preceded by the minus sign.

Groups and ranges

Characters	Meaning
`x|y`	Matches either "x" or "y". For example, `/green|red/` matches "green" in "green apple" and "red" in "red apple".
`[xyz]` `[a-c]`	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character set as a normal character. It is also possible to include a character class in a character set. For example, `[abcd]` is the same as `[a-d]`. They match the "b" in "brisket", and the "c" in "chop". For example, `[abcd-]` and `[-abcd]` match the "b" in "brisket", the "c" in "chop", and the "-" (hyphen) in "non-profit". For example, `[\w-]` is the same as `[A-Za-z0-9_-]`. They both match the "b" in "brisket", the "c" in "chop", and the "n" in "non-profit".
`[^xyz]` `[^a-c]`	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen, but if the hyphen appears as the first or last character enclosed in the square brackets it is taken as a literal hyphen to be included in the character set as a normal character. For example, `[^abc]` is the same as `[^a-c]`. They initially match "o" in "bacon" and "h" in "chop". > The ^ character may also indicate the beginning of input.

Characters	Meaning
`(x)`	**Capturing group:** Matches *x* and remembers the match. For example, `/(foo)/` matches and remembers "foo" in "foo bar". A regular expression may have multiple capturing groups. In results, matches to capturing groups typically in an array whose members are in the same order as the left parentheses in the capturing group. This is usually just the order of the capturing groups themselves. This becomes important when capturing groups are nested. Matches are accessed using the index of the the result's elements (`[1], ..., [n]`) or from the predefined `RegExp` object's properties (`$1, ..., $9`). Capturing groups have a performance penalty. If you don't need the matched substring to be recalled, prefer non-capturing parentheses (see below). `String.match()` won't return groups if the `/.../g` flag is set. However, you can still use `String.matchAll()` to get all matches.
`\n`	Where "n" is a positive integer. A back reference to the last substring matching the n parenthetical in the regular expression (counting left parentheses). For example, `/apple(,)\sorange\1/` matches "apple, orange," in "apple, orange, cherry, peach".
\k<Name>	A back reference to the last substring matching the **Named capture group** specified by `<Name>`. For example, `/(?<title>\w+), yes \k<title>/` matches "Sir, yes Sir" in "Do you copy? Sir, yes Sir!". > `\k` is used literally here to indicate the beginning of a back reference to a Named capture group.

Characters	Meaning
`(?<Name>x)`	**Named capturing group:** Matches "x" and stores it on the groups property of the returned matches under the name specified by `<Name>`. The angle brackets (`<` and `>`) are required for group name. For example, to extract the United States area code from a phone number, we could use `/\((?<area>\d\d\d)\)/`. The resulting number would appear under `matches.groups.area`.
`(?:x)`	**Non-capturing group:** Matches "x" but does not remember the match. The matched substring cannot be recalled from the resulting array's elements (`[1], ..., [n]`) or from the predefined `RegExp` object's properties (`$1, ..., $9`).

Quantifiers

> **Note:** In the following, *item* refers not only to singular characters, but also includes character classes, Unicode property escapes, groups and ranges.

Characters	Meaning
x`*`	Matches the preceding item "x" 0 or more times. For example, `/bo*/` matches "boooo" in "A ghost booooed" and "b" in "A bird warbled", but nothing in "A goat grunted".
x`+`	Matches the preceding item "x" 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the "a" in "candy" and all the "a"'s in "caaaaaaandy".
x`?`	Matches the preceding item "x" 0 or 1 times. For example, `/e?le?/` matches the "el" in "angel" and the "le" in "angle." If used immediately after any of the quantifiers `*`, `+`, `?`, or `{}`, makes the quantifier non-greedy (matching the minimum number of times), as opposed to the default, which is greedy (matching the maximum number of times).

Characters	Meaning
`x{n}`	Where "n" is a positive integer, matches exactly "n" occurrences of the preceding item "x". For example, `/a{2}/` doesn't match the "a" in "candy", but it matches all of the "a"'s in "caandy", and the first two "a"'s in "caaandy".
`x{n,}`	Where "n" is a positive integer, matches at least "n" occurrences of the preceding item "x". For example, `/a{2,}/` doesn't match the "a" in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy".
`x{n,m}`	Where "n" is 0 or a positive integer, "m" is a positive integer, and $m > n$, matches at least "n" and at most "m" occurrences of the preceding item "x". For example, `/a{1,3}/` matches nothing in "cndy", the "a" in "candy", the two "a"'s in "caandy", and the first three "a"'s in "caaaaaaandy". Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more "a"s in it.

`x*?`
`x+?`
`x??`
`x{n}?`
`x{n,}?`
`x{n,m}?`
	By default quantifiers like `*` and `+` are "greedy", meaning that they try to match as much of the string as possible. The `?` character after the quantifier makes the quantifier "non-greedy": meaning that it will stop as soon as it finds a match. For example, given a string like "some <foo> <bar> new </bar> </foo> thing":
	• `/<.*>/` will match "<foo> <bar> new </bar> </foo>"
	• `/<.*?>/` will match "<foo>"

Unicode property escapes

```
// Non-binary values
\p{UnicodePropertyValue}
\p{UnicodePropertyName=UnicodePropertyValue}

// Binary and non-binary values
\p{UnicodeBinaryPropertyName}

// Negation: \P is negated \p
```

> `\P{`*`UnicodePropertyValue`*`}`
> `\P{`*`UnicodeBinaryPropertyName`*`}`

UnicodeBinaryPropertyName

The name of a binary property. E.g.: `ASCII`, `Alpha`, `Math`, `Diacritic`, `Emoji`, `Hex_Digit`, `Math`, `White_space`, etc. See Unicode Data PropList.txt for more info.

UnicodePropertyName

The name of a non-binary property:

- General_Category (`gc`)
- Script (`sc`)
- Script_Extensions (`scx`)

See also PropertyValueAliases.txt

UnicodePropertyValue

One of the tokens listed in the Values section, below. Many values have aliases or shorthand (e.g. the value `Decimal_Number` for the `General_Category` property may be written `Nd`, `digit`, or `Decimal_Number`). For most values, the *`UnicodePropertyName`* part and equals sign may be omitted. If a *`UnicodePropertyName`* is specified, the value must correspond to the property type given.

> **Note:** As there are many properties and values available, we will not describe them exhaustively here but rather provide various examples

---

Last modified: Aug 1, 2020, by MDN contributors

Related Topics

JavaScript

Tutorials:

- ▶ Complete beginners
- ▼ JavaScript Guide

- Introduction
- Grammar and types
- Control flow and error handling
- Loops and iteration
- Functions
- Expressions and operators
- Numbers and dates
- Text formatting
- Regular expressions
- Indexed collections
- Keyed collections
- Working with objects
- Details of the object model
- Using promises
- Iterators and generators
- Meta programming
- JavaScript modules

- Intermediate
- Advanced

References:

- Built-in objects
- Expressions & operators
- Statements & declarations
- Functions
- Classes
- Errors
- Misc