Regular Expression Language - Quick Reference

Article01/25/202212 minutes to read21 contributors

In this article

A regular expression is a pattern that the regular expression engine attempts to match in input text. A pattern consists of one or more character literals, operators, or constructs. For a brief introduction, see .NET Regular Expressions.

Each section in this quick reference lists a particular category of characters, operators, and constructs that you can use to define regular expressions.

We've also provided this information in two formats that you can download and print for easy reference:

- Download in Word (.docx) format
- Download in PDF (.pdf) format

Character Escapes

The backslash character (\) in a regular expression indicates that the character that follows it either is a special character (as shown in the following table), or should be interpreted literally. For more information, see Character Escapes.

Escaped character	Description	Pattern	Matches
`\a`	Matches a bell character, \u0007.	`\a`	`"\u0007"` in `"Error!" + '\u0007'`
`\b`	In a character class, matches a backspace, \u0008.	`[\b]{3,}`	`"\b\b\b\b"` in `"\b\b\b\b"`
`\t`	Matches a tab, \u0009.	`(\w+)\t`	`"item1\t", "item2\t"` in `"item1\titem2\t"`

Escaped character	Description	Pattern	Matches
`\r`	Matches a carriage return, \u000D. (`\r` is not equivalent to the newline character, `\n`.)	`\r\n(\w+)`	`"\r\nThese"` in `"\r\nThese are\ntwo lines."`
`\v`	Matches a vertical tab, \u000B.	`[\v]{2,}`	`"\v\v\v"` in `"\v\v\v"`
`\f`	Matches a form feed, \u000C.	`[\f]{2,}`	`"\f\f\f"` in `"\f\f\f"`
`\n`	Matches a new line, \u000A.	`\r\n(\w+)`	`"\r\nThese"` in `"\r\nThese are\ntwo lines."`
`\e`	Matches an escape, \u001B.	`\e`	`"\x001B"` in `"\x001B"`
`\` *nnn*	Uses octal representation to specify a character (*nnn* consists of two or three digits).	`\w\040\w`	`"a b"`, `"c d"` in `"a bc d"`
`\x` *nn*	Uses hexadecimal representation to specify a character (*nn* consists of exactly two digits).	`\w\x20\w`	`"a b"`, `"c d"` in `"a bc d"`

Escaped character	Description	Pattern	Matches
`\c` *X* `\c` *x*	Matches the ASCII control character that is specified by *X* or *x*, where *X* or *x* is the letter of the control character.	`\cC`	`"\x0003"` in `"\x0003"` (Ctrl-C)
`\u` *nnnn*	Matches a Unicode character by using hexadecimal representation (exactly four digits, as represented by *nnnn*).	`\w\u0020\w`	`"a b"`, `"c d"` in `"a bc d"`

Escaped character	Description	Pattern	Matches
`\`	When followed by a character that is not recognized as an escaped character in this and other tables in this topic, matches that character. For example, `\*` is the same as `\x2A`, and `\.` is the same as `\x2E`. This allows the regular expression engine to disambiguate language elements (such as * or ?) and character literals (represented by `\*` or `\?`).	`\d+[\+-x\*]\d+`	`"2+2"` and `"3*9"` in `"(2+2) * 3*9"`

Character Classes

A character class matches any one of a set of characters. Character classes include the language elements listed in the following table. For more information, see Character Classes.

Character class	Description	Pattern	Matches

`[` *character_group* `]`	Matches any single character in *character_group*. By default, the match is case-sensitive.	`[ae]`	`"a"` in `"gray"`
			
			`"a"`, `"e"` in `"lane"`

Character class	Description	Pattern	Matches
`[^` *character_group* `]`	Negation: Matches any single character that is not in *character_group*. By default, characters in *character_group* are case-sensitive.	`[^aei]`	`"r"`, `"g"`, `"n"` in `"reign"`
`[` *first* `-` *last* `]`	Character range: Matches any single character in the range from *first* to *last*.	`[A-Z]`	`"A"`, `"B"` in `"AB123"`

.	Wildcard: Matches any single character except \n.
	
	To match a literal period character (. or `\u002E`), you must precede it with the escape character (`\.`).
		`a.e`	`"ave"` in `"nave"`
			
			`"ate"` in `"water"`

`\p{` *name* `}`	Matches any single character in the Unicode general category or named block specified by *name*.	`\p{Lu}`
		
		`\p{IsCyrillic}`
			`"C"`, `"L"` in `"City Lights"`
			
			`"Д"`, `"Ж"` in `"ДЖem"`

`\P{` *name* `}`	Matches any single character that is not in the Unicode general category or named block specified by *name*.	`\P{Lu}`
		
		`\P{IsCyrillic}`
			`"i"`, `"t"`, `"y"` in `"City"`
			
			`"e"`, `"m"` in `"ДЖem"`

`\w`	Matches any word character.	`\w`	`"I"`, `"D"`, `"A"`, `"1"`, `"3"` in `"ID A1.3"`
`\W`	Matches any non-word character.	`\W`	`" "`, `"."` in `"ID A1.3"`
`\s`	Matches any white-space character.	`\w\s`	`"D "` in `"ID A1.3"`

Character class	Description	Pattern	Matches
`\S`	Matches any non-white-space character.	`\s\S`	`" _"` in `"int __ctr"`
`\d`	Matches any decimal digit.	`\d`	`"4"` in `"4 = IV"`
`\D`	Matches any character other than a decimal digit.	`\D`	`" ", "=", " ", "I", "V"` in `"4 = IV"`

Anchors

Anchors, or atomic zero-width assertions, cause a match to succeed or fail depending on the current position in the string, but they do not cause the engine to advance through the string or consume characters. The metacharacters listed in the following table are anchors. For more information, see Anchors.

Assertion	Description	Pattern	Matches
`^`	By default, the match must start at the beginning of the string; in multiline mode, it must start at the beginning of the line.	`^\d{3}`	`"901"` in `"901-333-"`
`$`	By default, the match must occur at the end of the string or before `\n` at the end of the string; in multiline mode, it must occur before the end of the line or before `\n` at the end of the line.	`-\d{3}$`	`"-333"` in `"-901-333"`
`\A`	The match must occur at the start of the string.	`\A\d{3}`	`"901"` in `"901-333-"`
`\Z`	The match must occur at the end of the string or before `\n` at the end of the string.	`-\d{3}\Z`	`"-333"` in `"-901-333"`
`\z`	The match must occur at the end of the string.	`-\d{3}\z`	`"-333"` in `"-901-333"`

Assertion	Description	Pattern	Matches
`\G`	The match must occur at the point where the previous match ended.	`\G\(\d\)`	`"(1)", "(3)", "(5)"` in `"(1)(3)(5)[7](9)"`
`\b`	The match must occur on a boundary between a `\w` (alphanumeric) and a `\W` (nonalphanumeric) character.	`\b\w+\s\w+\b`	`"them theme", "them them"` in `"them theme them them"`
`\B`	The match must not occur on a `\b` boundary.	`\Bend\w*\b`	`"ends", "ender"` in `"end sends endure lender"`

Grouping Constructs

Grouping constructs delineate subexpressions of a regular expression and typically capture substrings of an input string. Grouping constructs include the language elements listed in the following table. For more information, see Grouping Constructs.

Grouping construct	Description	Pattern	Matches
`(` *subexpression* `)`	Captures the matched subexpression and assigns it a one-based ordinal number.	`(\w)\1`	`"ee"` in `"deep"`
`(?<` *name* `>` *subexpression* `)` or `(?'` *name* `'` *subexpression* `)`	Captures the matched subexpression into a named group.	`(?<double>\w)\k<double>`	`"ee"` in `"deep"`
`(?<` *name1* `-` *name2* `>` *subexpression* `)` or `(?'` *name1* `-` *name2* `'` *subexpression* `)`	Defines a balancing group definition. For more information, see the "Balancing Group Definition" section in Grouping Constructs.	`(((?'Open'\()[^\(\)]*)+((?'Close-Open'\))[^\(\)]*)+)*(?(Open)(?!))$`	`"((1-3)*(3-1))"` in `"3+2^((1-3)*(3-1))"`
`(?:` *subexpression* `)`	Defines a noncapturing group.	`Write(?:Line)?`	`"WriteLine"` in `"Console.WriteLine()"` `"Write"` in `"Console.Write(value)"`

Grouping construct	Description	Pattern	Matches
`(?imnsx-imnsx:` *subexpression* `)`	Applies or disables the specified options within *subexpression*. For more information, see Regular Expression Options.	`A\d{2}(?i:\w+)\b`	`"A12xl"`, `"A12XL"` in `"A12xl A12XL a12xl"`
`(?=` *subexpression* `)`	Zero-width positive lookahead assertion.	`\b\w+\b(?=.+and.+)`	`"cats"`, `"dogs"` in `"cats, dogs and some mice."`
`(?!` *subexpression* `)`	Zero-width negative lookahead assertion.	`\b\w+\b(?!.+and.+)`	`"and"`, `"some"`, `"mice"` in `"cats, dogs and some mice."`
`(?<=` *subexpression* `)`	Zero-width positive lookbehind assertion.	`\b\w+\b(?<=.+and.+)`	`"some"`, `"mice"` in `"cats, dogs and some mice."`
		`\b\w+\b(?<=.+and.*)`	`"and"`, `"some"`, `"mice"` in `"cats, dogs and some mice."`
`(?<!` *subexpression* `)`	Zero-width negative lookbehind assertion.	`\b\w+\b(?<!.+and.+)`	`"cats"`, `"dogs"`, `"and"` in `"cats, dogs and some mice."`
		`\b\w+\b(?<!.+and.*)`	`"cats"`, `"dogs"` in `"cats, dogs and some mice."`
`(?>` *subexpression* `)`	Atomic group.	`(?>a|ab)c`	`"ac"` in `"ac"` *nothing* in `"abc"`

Lookarounds at a glance

When the regular expression engine hits a **lookaround expression**, it takes a substring reaching from the current position to the start (lookbehind) or end (lookahead) of the original string, and then runs Regex.IsMatch on that substring using the lookaround pattern. Success of this subexpression's result is then determined by whether it's a positive or negative assertion.

Lookaround	Name	Function

Lookaround	Name	Function
`(?=check)`	Positive Lookahead	Asserts that what immediately follows the current position in the string is "check"
`(?<=check)`	Positive Lookbehind	Asserts that what immediately precedes the current position in the string is "check"
`(?!check)`	Negative Lookahead	Asserts that what immediately follows the current position in the string is not "check"
`(?<!check)`	Negative Lookbehind	Asserts that what immediately precedes the current position in the string is not "check"

Once they have matched, **atomic groups** won't be re-evaluated again, even when the remainder of the pattern fails due to the match. This can significantly improve performance when quantifiers occur within the atomic group or the remainder of the pattern.

Quantifiers

A quantifier specifies how many instances of the previous element (which can be a character, a group, or a character class) must be present in the input string for a match to occur. Quantifiers include the language elements listed in the following table. For more information, see Quantifiers.

Quantifier	Description	Pattern	Matches

Quantifier	Description	Pattern	Matches
`*`	Matches the previous element zero or more times.	`\d*\.\d`	`".0", "19.9", "219.9"`
`+`	Matches the previous element one or more times.	`"be+"`	`"bee"` in `"been", "be"` in `"bent"`
`?`	Matches the previous element zero or one time.	`"rai?n"`	`"ran", "rain"`
`{ n }`	Matches the previous element exactly *n* times.	`",\d{3}"`	`",043"` in `"1,043.6", ",876", ",543",` and `",210"` in `"9,876,543,210"`
`{ n ,}`	Matches the previous element at least *n* times.	`"\d{2,}"`	`"166", "29", "1930"`
`{ n , m }`	Matches the previous element at least *n* times, but no more than *m* times.	`"\d{3,5}"`	`"166", "17668"` `"19302"` in `"193024"`
`*?`	Matches the previous element zero or more times, but as few times as possible.	`\d*?\.\d`	`".0", "19.9", "219.9"`
`+?`	Matches the previous element one or more times, but as few times as possible.	`"be+?"`	`"be"` in `"been", "be"` in `"bent"`
`??`	Matches the previous element zero or one time, but as few times as possible.	`"rai??n"`	`"ran", "rain"`
`{ n }?`	Matches the preceding element exactly *n* times.	`",\d{3}?"`	`",043"` in `"1,043.6", ",876", ",543",` and `",210"` in `"9,876,543,210"`
`{ n ,}?`	Matches the previous element at least *n* times, but as few times as possible.	`"\d{2,}?"`	`"166", "29", "1930"`

Quantifier	Description	Pattern	Matches
`{ n , m }?`	Matches the previous element between *n* and *m* times, but as few times as possible.	`"\d{3,5}?"`	`"166"`, `"17668"` `"193"`, `"024"` in `"193024"`

Backreference Constructs

A backreference allows a previously matched subexpression to be identified subsequently in the same regular expression. The following table lists the backreference constructs supported by regular expressions in .NET. For more information, see Backreference Constructs.

Backreference construct	Description	Pattern	Matches
`\` *number*	Backreference. Matches the value of a numbered subexpression.	`(\w)\1`	`"ee"` in `"seek"`
`\k<` *name* `>`	Named backreference. Matches the value of a named expression.	`(?<char>\w)\k<char>`	`"ee"` in `"seek"`

Alternation Constructs

Alternation constructs modify a regular expression to enable either/or matching. These constructs include the language elements listed in the following table. For more information, see Alternation Constructs.

Alternation construct	Description	Pattern	Matches
|	Matches any one element separated by the vertical bar (`|`) character.	`th(e|is|at)`	`"the"`, `"this"` in `"this is the day."`

Alternation construct	Description	Pattern	Matches
`(?(` *expression* `)` *yes* `|` *no* `)` or `(?(` *expression* `)` *yes* `)`	Matches *yes* if the regular expression pattern designated by *expression* matches; otherwise, matches the optional *no* part. *expression* is interpreted as a zero-width assertion. To avoid ambiguity with a named or numbered capturing group, you can optionally use an explicit assertion, like this: `(?(` `(?=` *expression* `)` `)` *yes* `|` *no* `)`	`(?(A)A\d{2}\b|\b\d{3}\b)`	`"A10"`, `"910"` in `"A10 C103 910"`
`(?(` *name* `)` *yes* `|` *no* `)` or `(?(` *name* `)` *yes* `)`	Matches *yes* if *name*, a named or numbered capturing group, has a match; otherwise, matches the optional *no*.	`(?<quoted>")?(?(quoted).+?"|\S+\s)`	`"Dogs.jpg "`, `"\"Yiska playing.jpg\""` in `"Dogs.jpg \"Yiska playing.jpg\""`

Substitutions

Substitutions are regular expression language elements that are supported in replacement patterns. For more information, see Substitutions. The metacharacters listed in the following table are atomic zero-width assertions.

Character	Description	Pattern	Replacement pattern	Input string	Result string
`$` *number*	Substitutes the substring matched by group *number*.	`\b(\w+)(\s)(\w+)\b`	`$3$2$1`	`"one two"`	`"two one"`
`${` *name* `}`	Substitutes the substring matched by the named group *name*.	`\b(?<word1>\w+)(\s)(?<word2>\w+)\b`	`${word2} ${word1}`	`"one two"`	`"two one"`
`$$`	Substitutes a literal "$".	`\b(\d+)\s?USD`	`$$1`	`"103 USD"`	`"$103"`

Character	Description	Pattern	Replacement pattern	Input string	Result string
`$&`	Substitutes a copy of the whole match.	`\$?\d*\.?\d+`	`**$&**`	`"$1.30"`	`"**$1.30**"`
`$``	Substitutes all the text of the input string before the match.	`B+`	`$``	`"AABBCC"`	`"AAAACC"`
`$'`	Substitutes all the text of the input string after the match.	`B+`	`$'`	`"AABBCC"`	`"AACCCC"`
`$+`	Substitutes the last group that was captured.	`B+(C+)`	`$+`	`"AABBCCDD"`	`"AACCDD"`
`$_`	Substitutes the entire input string.	`B+`	`$_`	`"AABBCC"`	`"AAAABBCCCC"`

Regular Expression Options

You can specify options that control how the regular expression engine interprets a regular expression pattern. Many of these options can be specified either inline (in the regular expression pattern) or as one or more RegexOptions constants. This quick reference lists only inline options. For more information about inline and RegexOptions options, see the article Regular Expression Options.

You can specify an inline option in two ways:

- By using the miscellaneous construct `(?imnsx-imnsx)`, where a minus sign (-) before an option or set of options turns those options off. For example, `(?i-mn)` turns case-insensitive matching (`i`) on, turns multiline mode (`m`) off, and turns unnamed group captures (`n`) off. The option applies to the regular expression pattern from the point at which the option is defined, and is effective either to the end of the pattern or to the point where another construct reverses the option.
- By using the grouping construct `(?imnsx-imnsx:` *subexpression* `)`, which defines options for the specified group only.

The .NET regular expression engine supports the following inline options:

Option	Description	Pattern	Matches
`i`	Use case-insensitive matching.	`\b(?i)a(?-i)a\w+\b`	`"aardvark"`, `"aaaAuto"` in `"aardvark AAAuto aaaAuto Adam breakfast"`

Option	Description	Pattern	Matches
`m`	Use multiline mode. `^` and `$` match the beginning and end of a line, instead of the beginning and end of a string.	For an example, see the "Multiline Mode" section in Regular Expression Options.	
`n`	Do not capture unnamed groups.	For an example, see the "Explicit Captures Only" section in Regular Expression Options.	
`s`	Use single-line mode.	For an example, see the "Single-line Mode" section in Regular Expression Options.	
`x`	Ignore unescaped white space in the regular expression pattern.	`\b(?x) \d+ \s \w+`	`"1 aardvark", "2 cats"` in `"1 aardvark 2 cats IV centurions"`

Miscellaneous Constructs

Miscellaneous constructs either modify a regular expression pattern or provide information about it. The following table lists the miscellaneous constructs supported by .NET. For more information, see Miscellaneous Constructs.

Construct	Definition	Example

Construct	Definition	Example
`(?imnsx-imnsx)`	Sets or disables options such as case insensitivity in the middle of a pattern.For more information, see Regular Expression Options.	`\bA(?i)b\w+\b` matches `"ABA"`, `"Able"` in `"ABA Able Act"`
`(?# comment )`	Inline comment. The comment ends at the first closing parenthesis.	`\bA(?#Matches words starting with A)\w+\b`
`#` [to end of line]	X-mode comment. The comment starts at an unescaped `#` and continues to the end of the line.	`(?x)\bA\w+\b#Matches words starting with A`