Regular Expressions Quick Start

This quick start gets you up to speed quickly with regular expressions. Obviously, this brief introduction cannot explain everything there is to know about regular expressions. For detailed information, consult the regular expressions tutorial. Each topic in the quick start corresponds with a topic in the tutorial, so you can easily go back and forth between the two.

Many applications and programming languages have their own implementation of regular expressions, often with slight and sometimes with significant differences from other implementations. When two applications use a different implementation of regular expressions, we say that they use different “regular expression flavors”. This quick start explains the syntax supported by the most popular regular expression flavors.

Text Patterns and Matches

A regular expression, or regex for short, is a pattern describing a certain amount of text. On this website, regular expressions are highlighted in red as `regex`. This is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text `regex`. Matches are highlighted in blue on this site. We use the term “string” to indicate the text that the regular expression is applied to. Strings are highlighted in `green`.

Characters with special meanings in regular expressions are highlighted in various different colors. The regex `(?x)([Rr]egexp?)\?` shows meta tokens in purple, grouping in green, character classes in orange, quantifiers and other special tokens in blue, and escaped characters in gray.

Literal Characters

The most basic regular expression consists of a single literal character, such as `a`. It matches the first occurrence of that character in the string. If the string is `Jack is a boy`, it matches the `a` after the `J`.

This regex can match the second `a` too. It only does so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its “Find Next” or “Search Forward” function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Twelve characters have special meanings in regular expressions: the backslash `\`, the caret `^`, the dollar sign `$`, the period or dot `.`, the vertical bar or pipe symbol `|`, the question mark `?`, the asterisk or star `*`, the plus sign `+`, the opening parenthesis `(`, the closing parenthesis `)`, the opening square bracket `[`, and the opening curly brace `{`. These special characters are often called “metacharacters”. Most of them are errors when used alone.

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match `1+1=2`, the correct regex is `1\+1=2`. Otherwise, the plus sign has a special meaning.

Learn more about literal characters

Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. Use `\t` to match a tab character (ASCII 0x09), `\r` for carriage return (0x0D) and `\n` for line feed (0x0A). More exotic non-

printables are `\a` (bell, 0x07), `\e` (escape, 0x1B), `\f` (form feed, 0x0C) and `\v` (vertical tab, 0x0B). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

If your application supports Unicode, use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. `\u20AC` or `\x{20AC}` matches the euro currency sign.

If your application does not support Unicode, use `\xFF` to match a specific character by its hexadecimal index in the character set. `\xA9` matches the copyright symbol in the Latin-1 character set.

All non-printable characters can be used directly in the regular expression, or as part of a character class.

Learn more about non-printable characters

Character Classes or Character Sets

A “character class” matches only one out of several characters. To match an a or an e, use `[ae]`. You could use this in `gr[ae]y` to match either `gray` or `grey`. A character class matches only a single character. `gr[ae]y` does not match `graay`, `graey` or any such thing. The order of the characters inside a character class does not matter.

You can use a hyphen inside a character class to specify a range of characters. `[0-9]` matches a *single* digit between 0 and 9. You can use more than one range. `[0-9a-fA-F]` matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. `[0-9a-fxA-FX]` matches a hexadecimal digit or the letter X.

Typing a caret after the opening square bracket negates the character class. The result is that the character class matches any character that is *not* in the character class. `q[^x]` matches `qu` in `question`. It does *not* match `Iraq` since there is no character after the q for the negated character class to match.

Learn more about character classes

Shorthand Character Classes

`\d` matches a single character that is a digit, `\w` matches a “word character” (alphanumeric characters plus underscore), and `\s` matches a whitespace character (includes tabs and line breaks). The actual characters matched by the shorthands depends on the software you’re using. In modern applications, they include non-English letters and numbers.

Learn more about shorthand character classes

The Dot Matches (Almost) Any Character

The dot matches a single character, except line break characters. Most applications have a “dot matches all” or “single line” mode that makes the dot match any single character, including line breaks.

`gr.y` matches `gray`, `grey`, `gr%y`, etc. Use the dot sparingly. Often, a character class or negated character class is faster and more precise.

Learn more about the dot

Anchors

Anchors do not match any characters. They match a position. `^` matches at the start of the string, and `$` matches at the end of the string. Most regex engines have a "multi-line" mode that makes `^` match after any line break, and `$` before any line break. E.g. `^b` matches only the first `b` in `bob`.

`\b` matches at a word boundary. A word boundary is a position between a character that can be matched by `\w` and a character that cannot be matched by `\w`. `\b` also matches at the start and/or end of the string if the first and/or last characters in the string are word characters. `\B` matches at every position where `\b` cannot match.

Learn more about anchors

Alternation

Alternation is the regular expression equivalent of "or". `cat|dog` matches `cat` in `About cats and dogs`. If the regex is applied again, it matches `dog`. You can add as many alternatives as you want: `cat|dog|mouse|fish`.

Alternation has the lowest precedence of all regex operators. `cat|dog food` matches `cat` or `dog food`. To create a regex that matches `cat food` or `dog food`, you need to group the alternatives: `(cat|dog) food`.

Learn more about alternation

Repetition

The question mark makes the preceding token in the regular expression optional. `colou?r` matches `colour` or `color`.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<[A-Za-z][A-Za-z0-9]*>` matches an HTML tag without any attributes. `<[A-Za-z0-9]+>` is easier to write but matches invalid tags such as `<1>`.

Use curly braces to specify a specific amount of repetition. Use `\b[1-9][0-9]{3}\b` to match a number between 1000 and 9999. `\b[1-9][0-9]{2,4}\b` matches a number between 100 and 99999.

Learn more about quantifiers

Greedy and Lazy Repetition

The repetition operators or quantifiers are greedy. They expand the match as far as they can, and only give back if they must to satisfy the remainder of the regex. The regex `<.+>` matches `<EM>first</EM>` in `This is a <EM>first</EM> test`.

Place a question mark after the quantifier to make it lazy. `<.+?>` matches `<EM>` in the above string.

A better solution is to follow my advice to use the dot sparingly. Use `<[^<>]+>` to quickly match an HTML tag without regard to attributes. The negated character class is more specific than the dot, which helps the regex engine find matches quickly.

Learn more about greedy and lazy quantifiers

Grouping and Capturing

Place parentheses around multiple tokens to group them together. You can then apply a quantifier to the group. E.g. `Set(Value)?` matches `Set` or `SetValue`.

Parentheses create a capturing group. The above example has one group. After the match, group number one contains nothing if `Set` was matched. It contains `Value` if `SetValue` was matched. How to access the group's contents depends on the software or programming language you're using. Group zero always contains the entire regex match.

Use the special syntax `Set(?:Value)?` to group tokens without creating a capturing group. This is more efficient if you don't plan to use the group's contents. Do not confuse the question mark in the non-capturing group syntax with the quantifier.

Learn more about grouping and capturing

Backreferences

Within the regular expression, you can use the backreference `\1` to match the same text that was matched by the capturing group. `([abc])=\1` matches `a=a`, `b=b`, and `c=c`. It does not match anything else. If your regex has multiple capturing groups, they are numbered counting their opening parentheses from left to right.

Learn more about backreferences

Named Groups and Backreferences

If your regex has many groups, keeping track of their numbers can get cumbersome. Make your regexes easier to read by naming your groups. `(?<mygroup>[abc])=\k<mygroup>` is identical to `([abc])=\1`, except that you can refer to the group by its name.

Learn more about named groups

Unicode Properties

`\p{L}` matches a single character that is in the given Unicode category. `L` stands for letter. `\P{L}` matches a single character that is not in the given Unicode category. You can find a complete list of Unicode categories in the tutorial.

Learn more about Unicode regular expressions

Lookaround

Lookaround is a special kind of group. The tokens inside the group are matched normally, but then the regex engine makes the group give up its match and keeps only the result. Lookaround matches a position, just like anchors. It does not expand the regex match.

`q(?=u)` matches the `q` in `question`, but not in `Iraq`. This is positive lookahead. The `u` is not part of the overall regex match. The lookahead matches at each position in the string before a `u`.

`q(?!u)` matches `q` in `Iraq` but not in `question`. This is negative lookahead. The tokens inside the lookahead are attempted, their match is discarded, and the result is inverted.

To look backwards, use lookbehind. `(?<=a)b` matches the `b` in `abc`. This is positive lookbehind. `(?<!a)b` fails to match `abc`.

You can use a full-fledged regular expression inside lookahead. Most applications only allow fixed-length expressions in lookbehind.

Learn more about lookaround

Free-Spacing Syntax

Many application have an option that may be labeled “free-spacing” or “ignore whitespace” or “comments” that makes the regular expression engine ignore unescaped spaces and line breaks and that makes the # character start a comment that runs until the end of the line. This allows you to use whitespace to format your regular expression in a way that makes it easier for humans to read and thus makes it easier to maintain.

Learn more about free-spacing

Make a Donation

Did this website just save you a trip to the bookstore? Please make a donation to support this site, and you'll get a **lifetime of advertisement-free access** to this site!

Page URL: https://www.regular-expressions.info/quickstart.html
Page last updated: 22 November 2019
Site last updated: 08 April 2020