Character Classes or Character Sets

With a “character class”, also called “character set”, you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use `[ae]`. You could use this in `gr[ae]y` to match either `gray` or `grey`. Very useful if you do not know whether the document you are searching through is written in American or British English.

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. The results are identical.

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. Again, the order of the characters and the ranges does not matter.

Character classes are one of the most commonly used features of regular expressions. You can find a word, even if it is misspelled, such as `sep[ae]r[ae]te` or `li[cs]en[cs]e`. You can find an identifier in a programming language with `[A-Za-z_][A-Za-z_0-9]*`. You can find a C-style hexadecimal number with `0[xX][A-Fa-f0-9]+`.

Negated Character Classes

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. Unlike the dot, negated character classes also match (invisible) line break characters. If you don’t want a negated character class to match line breaks, you need to include the line break characters in the class. `[^0-9\r\n]` matches any character that is not a digit or a line break.

It is important to remember that a negated character class still must match a character. `q[^u]` does *not* mean: “a q not followed by a u”. It means: “a q followed by a character that is not a u”. It does not match the q in the string `Iraq`. It does match the q and the space after the q in `Iraq is a country`. Indeed: the space becomes part of the overall match, because it is the “character that is not a u” that is matched by the negated character class in the above regexp. If you want the regex to match the q, and only the q, in both strings, you need to use negative lookahead: `q(?!u)`. But we will get to that later.

Metacharacters Inside Character Classes

In most regex flavors, the only special characters or metacharacters inside a character class are the closing bracket `]`, the backslash `\`, the caret `^`, and the hyphen `-`. The usual metacharacters are normal characters inside a character class, and do not need to be escaped by a backslash. To search for a star or plus, use `[+*]`. Your regex will work fine if you escape the regular metacharacters inside a character class, but doing so significantly reduces readability.

To include a backslash as a character without any special meaning inside a character class, you have to escape it with another backslash. `[\\x]` matches a backslash or an x. The closing bracket `]`, the caret `^` and the hyphen `-` can be included by escaping them with a backslash, or by placing them in a position where they do not take on their special meaning. The POSIX and GNU flavors are an exception. They treat backslashes in character classes as literal characters. So with these flavors, you can’t escape anything in character classes.

To include an unescaped caret as a literal, place it anywhere except right after the opening bracket. `[x^]` matches an x or a caret. This works with all flavors discussed in this tutorial.

You can include an unescaped closing bracket by placing it right after the opening bracket, or right after the negating caret. `[]x]` matches a closing bracket or an x. `[^]x]` matches any character that is not a closing bracket or an x. This does not work in JavaScript, which treats `[]` as an empty character class that always fails to match, and `[^]` as a negated empty character class that matches any single character. Ruby treats empty character classes as an error. So both JavaScript and Ruby require closing brackets to be escaped with a backslash to include them as literals in a character class.

The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both `[-x]` and `[x-]` match an x or a hyphen. `[^-x]` and `[^x-]` match any character that is not an x or a hyphen. This works in all flavors discussed in this tutorial. Hyphens at other positions in character classes where they can't form a range may be interpreted as literals or as errors. Regex flavors are quite inconsistent about this.

Many regex tokens that work outside character classes can also be used inside character classes. This includes character escapes, octal escapes, and hexadecimal escapes for non-printable characters. For flavors that support Unicode, it also includes Unicode character escapes and Unicode properties. `[$\u20AC]` matches a dollar or euro sign, assuming your regex flavor supports Unicode escapes.

Repeating Character Classes

If you repeat a character class by using the `?`, `*` or `+` operators, you're repeating the entire character class. You're not repeating just the character that it matched. The regex `[0-9]+` can match `837` as well as `222`.

If you want to repeat the matched character, rather than the class, you need to use backreferences. `([0-9])\1+` matches `222` but not `837`. When applied to the string `833337`, it matches `3333` in the middle of this string. If you do not want that, you need to use lookaround.

Looking Inside The Regex Engine

As was mentioned earlier: the order of the characters inside a character class does not matter. `gr[ae]y` matches `grey` in `Is his hair grey or gray?`, because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Now we'll see how it applies a regex that has more than one permutation. That is: `gr[ae]y` can match both `gray` and `grey`.

Nothing noteworthy happens for the first twelve characters in the string. The engine fails to match `g` at every step, and continues with the next character in the string. When the engine arrives at the 13th character, `g` is matched. The engine then tries to match the remainder of the regex with the text. The next token in the regex is the literal `r`, which matches the next character in the text. So the third token, `[ae]` is attempted at the next character in the text (`e`). The character class gives the engine two options: match `a` or match `e`. It first attempts to match `a`, and fails.

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it continues with the other option, and finds that `e` matches `e`. The last regex token is `y`, which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It returns `grey` as the match result, and looks no further. Again, the *leftmost match* is returned, even though we put the `a` first in the character class, and `gray` could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it. `gray` is only matched if you tell the regex engine to continue looking for a second match in the remainder of the subject string after the first match.

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!
