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), and `\f` (form feed, 0x0C). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

In some flavors, `\v` matches the vertical tab (ASCII 0x0B). In other flavors, `\v` is a shorthand that matches any vertical whitespace character. That includes the vertical tab, form feed, and all line break characters. Perl 5.10, PCRE 7.2, PHP 5.2.4, R, Delphi XE, and later versions treat it as a shorthand. Earlier versions treated it as a needlessly escaped literal v. The JGsoft flavor originally matched only the vertical tab with `\v`. JGsoft V2 matches any vertical whitespace with `\v`.

Many regex flavors also support the tokens `\cA` through `\cZ` to insert ASCII control characters. The letter after the backslash is always a lowercase c. The second letter is an uppercase letter A through Z, to indicate Control+A through Control+Z. These are equivalent to `\x01` through `\x1A` (26 decimal). E.g. `\cM` matches a carriage return, just like `\r`, `\x0D`, and `\u000D`. Most flavors allow the second letter to be lowercase, with no difference in meaning. Only Java requires the A to Z to be uppercase.

Using characters other than letters after\cis not recommended because the behavior is inconsistent between applications. Some allow any character after\cwhile other allow ASCII characters. The application may take the last 5 bits that character index in the code page or its Unicode code point to form an ASCII control character. Or the application may just flip bit 0x40. Either way `\c@` through `\c_` would match control characters 0x00 through 0x1F. But `\c*` might match a line feed or the letter `j`. The asterisk is character 0x2A in the ASCII table, so the lower 5 bits are 0x0A while flipping bit 0x40 gives 0x6A. Metacharacters indeed lose their meaning immediately after\cin applications that support `\cA` through `\cZ` for matching control characters. The original JGsoft flavor, .NET, and XRegExp are more sensible. They treat anything other than a letter after\cas an error.

In XML Schema regular expressions and XPath, `\c` is a shorthand character class that matches any character allowed in an XML name.

The JGsoft flavor originally treated `\cA` through `\cZ` as control characters. But JGsoft V2 treats `\c` as an XML shorthand.

If your regular expression engine supports Unicode, you can use `\uFFFF` or `\x{FFFF}` to insert a Unicode character. The euro currency sign occupies Unicode code point U+20AC. If you cannot type it on your keyboard, you can insert it into a regular expression with `\u20AC` or `\x{20AC}`. See the tutorial section on Unicode for more details on matching Unicode code points.

If your regex engine works with 8-bit code pages instead of Unicode, then you can include any character in your regular expression if you know its position in the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use `\xA9`. Another way to search for a tab is to use `\x09`. Note that the leading zero is required. In Tcl 8.5 and prior you have to be careful with this syntax, because Tcl used to eat up all hexadecimal characters after `\x` and treat the last 4 as a Unicode code point. So `\xA9ABC20AC` would match the euro symbol. Tcl 8.6 only takes the first two hexadecimal digits as part of the `\x`, as all other regex flavors do, so `\xA9ABC20AC` matches `©ABC20AC`.

Line Breaks

`\R` is a special escape that matches any line break, including Unicode line breaks. What makes it special is that it treats CRLF pairs as indivisible. If the match attempt of `\R` begins before a CRLF pair in the string, then a single `\R` matches the whole CRLF pair. `\R` will not backtrack to match only the CR in a CRLF pair. So while `\R` can match a lone CR or a lone LF, `\R{2}` or `\R\R` cannot match a single CRLF pair. The first `\R` matches the whole CRLF pair, leaving nothing for the second one to match.

Or at least, that is how `\R` should work. It works like that in JGsoft V2, Ruby 2.0 and later, Java 8, and PCRE 8.13 and later. Java 9 introduced a bug that allows `\R\R` to match a single CRLF pair. PCRE 7.0 through 8.12 had a bug that allows `\R{2}` to match a single CRLF pair. Perl has a different bug with the same result.

Note that `\R` only looks forward to match CRLF pairs. The regex `\r\R` can match a single CRLF pair. After `\r` has consumed the CR, the remaining lone LF is a valid line break for `\R` to match. This behavior is consistent across all flavors.

Octal Escapes

Many applications also support octal escapes in the form of `\0377` or `\377`, where 377 is the octal representation of the character's position in the character set (255 decimal in this case). There is a lot of variation between regex flavors as to the number of octal digits allowed or required after the backslash, whether the leading zero is required or not allowed, and whether `\0` without additional digits matches a NULL byte. In some flavors this causes complications as `\1` to `\77` can be octal escapes 1 to 63 (decimal) or backreferences 1 to 77 (decimal), depending on how many capturing groups there are in the regex. Therefore, using these octal escapes in regexes is strongly discouraged. Use hexadecimal escapes instead.

Perl 5.14, PCRE 8.34, PHP 5.5.10, and R 3.0.3 support a new syntax `\o{377}` for octal escapes. You can have any number of octal digits between the curly braces, with or without leading zero. There is no confusion with backreferences and literal digits that follow are cleanly separated by the closing curly brace. Do be careful to only put octal digits between the curly braces. In Perl, `\o{whatever}` is not an error but matches a NULL byte.

The JGsoft flavor originally supported octal escapes in the form of `\0377`. JGsoft V2 supports `\o{377}` and treats `\0377` as an error.

Regex Syntax versus String Syntax

Many programming languages support similar escapes for non-printable characters in their syntax for literal strings in source code. Then such escapes are translated by the compiler into their actual characters before the string is passed to the regex engine. If the regex engine does not support the same escapes, this can cause an apparent difference in behavior when a regex is specified as a literal string in source code compared with a regex that is read from a file or received from user input. For example, POSIX regular expressions do not support any of these escapes. But the C programming language does support escapes like `\n` and `\x0A` in string literals. So when developing an application in C using the POSIX library, `\n` is only interpreted as a newline when you add the regex as a string literal to your source code. Then the compiler interprets \n and the regex engine sees an actual newline character. If your code reads the same regex from a file, then the regex engine sees `\n`. Depending on the implementation, the POSIX library interprets this as a literal `n` or as an error. The actual POSIX standard states that the behavior of an "ordinary" character preceded by a backslash is "undefined".

A similar issue exists in Python 3.2 and prior with the Unicode escape `\uFFFF`. Python has supported this syntax as part of (Unicode) string literals ever since Unicode support was added to Python. But Python's re module only supports `\uFFFF` starting with Python 3.3. In Python 3.2 and earlier, `\uFFFF` works when you add your regex as a literal (Unicode) string to your Python code. But when your Python 3.2 script reads the regex from a file or user input, `\uFFFF` matches `uFFFF` literally as the regex engine sees `\u` as an escaped literal `u`.

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!