NAME

perlrebackslash - Perl Regular Expression Backslash Sequences and Escapes

DESCRIPTION

The top level documentation about Perl regular expressions is found in *perlre*.

This document describes all backslash and escape sequences. After explaining the role of the backslash, it lists all the sequences that have a special meaning in Perl regular expressions (in alphabetical order), then describes each of them.

Most sequences are described in detail in different documents; the primary purpose of this document is to have a quick reference guide describing all backslash and escape sequences.

The backslash

In a regular expression, the backslash can perform one of two tasks: it either takes away the special meaning of the character following it (for instance, `\|` matches a vertical bar, it's not an alternation), or it is the start of a backslash or escape sequence.

The rules determining what it is are quite simple: if the character following the backslash is an ASCII punctuation (non-word) character (that is, anything that is not a letter, digit or underscore), then the backslash just takes away the special meaning (if any) of the character following it.

If the character following the backslash is an ASCII letter or an ASCII digit, then the sequence may be special; if so, it's listed below. A few letters have not been used yet, so escaping them with a backslash doesn't change them to be special. A future version of Perl may assign a special meaning to them, so if you have warnings turned on, Perl will issue a warning if you use such a sequence. [1].

It is however guaranteed that backslash or escape sequences never have a punctuation character following the backslash, not now, and not in a future version of Perl 5. So it is safe to put a backslash in front of a non-word character.

Note that the backslash itself is special; if you want to match a backslash, you have to escape the backslash with a backslash: `/\\/` matches a single backslash.

[1]

There is one exception. If you use an alphanumerical character as the delimiter of your pattern (which you probably shouldn't do for readability reasons), you will have to escape the delimiter if you want to match it. Perl won't warn then. See also *"Gory details of parsing quoted constructs" in perlop*.

All the sequences and escapes

Those not usable within a bracketed character class (like `[\da-z]`) are marked as `Not in []`.

```
\000              Octal escape sequence.
\1                Absolute backreference.  Not in [].
\a                Alarm or bell.
\A                Beginning of string.  Not in [].
\b                Word/non-word boundary. (Backspace in []).
\B                Not a word/non-word boundary.  Not in [].
\cX               Control-X
\C                Single octet, even under UTF-8.  Not in [].
\d                Character class for digits.
\D                Character class for non-digits.
\e                Escape character.
\E                Turn off \Q, \L and \U processing.  Not in [].
\f                Form feed.
\g{}, \g1         Named, absolute or relative backreference.  Not in [].
\G                Pos assertion.  Not in [].
```

```
\h                Character class for horizontal whitespace.
\H                Character class for non horizontal whitespace.
\k{}, \k<>, \k''  Named backreference.  Not in [].
\K                Keep the stuff left of \K.  Not in [].
\l                Lowercase next character.  Not in [].
\L                Lowercase till \E.  Not in [].
\n                (Logical) newline character.
\N                Any character but newline.  Experimental.  Not in [].
\N{}              Named or numbered (Unicode) character.
\p{}, \pP         Character with the given Unicode property.
\P{}, \PP         Character without the given Unicode property.
\Q                Quotemeta till \E.  Not in [].
\r                Return character.
\R                Generic new line.  Not in [].
\s                Character class for whitespace.
\S                Character class for non whitespace.
\t                Tab character.
\u                Titlecase next character.  Not in [].
\U                Uppercase till \E.  Not in [].
\v                Character class for vertical whitespace.
\V                Character class for non vertical whitespace.
\w                Character class for word characters.
\W                Character class for non-word characters.
\x{}, \x00        Hexadecimal escape sequence.
\X                Unicode "extended grapheme cluster".  Not in [].
\z                End of string.  Not in [].
\Z                End of string.  Not in [].
```

Character Escapes

Fixed characters

A handful of characters have a dedicated *character escape*. The following table shows them, along with their ASCII code points (in decimal and hex), their ASCII name, the control escape on ASCII platforms and a short description. (For EBCDIC platforms, see *"OPERATOR DIFFERENCES" in perlebcdic*.)

Seq.	Code Point Dec	Hex	ASCII	Cntrl	Description.
\a	7	07	BEL	\cG	alarm or bell
\b	8	08	BS	\cH	backspace [1]
\e	27	1B	ESC	\c[	escape character
\f	12	0C	FF	\cL	form feed
\n	10	0A	LF	\cJ	line feed [2]
\r	13	0D	CR	\cM	carriage return
\t	9	09	TAB	\cI	tab

[1]

\b is the backspace character only inside a character class. Outside a character class, \b is a word/non-word boundary.

[2]

\n matches a logical newline. Perl will convert between \n and your OS's native newline character when reading from or writing to text files.

Example

```
$str =~ /\t/;   # Matches if $str contains a (horizontal) tab.
```

Control characters

`\c` is used to denote a control character; the character following `\c` determines the value of the construct. For example the value of `\cA` is `chr(1)`, and the value of `\cb` is `chr(2)`, etc. The gory details are in *"Regexp Quote-Like Operators" in perlop*. A complete list of what `chr(1)`, etc. means for ASCII and EBCDIC platforms is in *"OPERATOR DIFFERENCES" in perlebcdic*.

Note that `\c\` alone at the end of a regular expression (or doubled-quoted string) is not valid. The backslash must be followed by another character. That is, `\c\`*X* means `chr(28) . 'X'` for all characters *X*.

To write platform-independent code, you must use `\N{`*NAME*`}` instead, like `\N{ESCAPE}` or `\N{U+001B}`, see *charnames*.

Mnemonic: *c*ontrol character.

Example

```
$str =~ /\cK/;  # Matches if $str contains a vertical tab (control-K).
```

Named or numbered characters

Unicode characters have a Unicode name and numeric ordinal value. Use the `\N{}` construct to specify a character by either of these values.

To specify by name, the name of the character goes between the curly braces. In this case, you have to `use charnames` to load the Unicode names of the characters, otherwise Perl will complain.

To specify by Unicode ordinal number, use the form `\N{U+`*wide hex character*`}`, where *wide hex character* is a number in hexadecimal that gives the ordinal number that Unicode has assigned to the desired character. It is customary (but not required) to use leading zeros to pad the number to 4 digits. Thus `\N{U+0041}` means `Latin Capital Letter A`, and you will rarely see it written without the two leading zeros. `\N{U+0041}` means "A" even on EBCDIC machines (where the ordinal value of "A" is not 0x41).

It is even possible to give your own names to characters, and even to short sequences of characters. For details, see *charnames*.

(There is an expanded internal form that you may see in debug output: `\N{U+`*wide hex character*`.`*wide hex character*`...}`. The `...` means any number of these *wide hex character* s separated by dots. This represents the sequence formed by the characters. This is an internal form only, subject to change, and you should not try to use it yourself.)

Mnemonic: *N*amed character.

Note that a character that is expressed as a named or numbered character is considered as a character without special meaning by the regex engine, and will match "as is".

Example

```
use charnames ':full';               # Loads the Unicode names.
$str =~ /\N{THAI CHARACTER SO SO}/;  # Matches the Thai SO SO character

use charnames 'Cyrillic';            # Loads Cyrillic names.
$str =~ /\N{ZHE}\N{KA}/;             # Match "ZHE" followed by "KA".
```

Octal escapes

Octal escapes consist of a backslash followed by two or three octal digits matching the code point of the character you want to use. This allows for 512 characters (`\00` up to `\777`) that can be expressed this way (but anything above `\377` is deprecated). Enough in pre-Unicode days, but most Unicode characters cannot be escaped this way.

Note that a character that is expressed as an octal escape is considered as a character without special meaning by the regex engine, and will match "as is".

Examples (assuming an ASCII platform)

```
$str = "Perl";
$str =~ /\120/;    # Match, "\120" is "P".
$str =~ /\120+/;   # Match, "\120" is "P", it is repeated at least once.
$str =~ /P\053/;   # No match, "\053" is "+" and taken literally.
```

Caveat

Octal escapes potentially clash with backreferences. They both consist of a backslash followed by numbers. So Perl has to use heuristics to determine whether it is a backreference or an octal escape. Perl uses the following rules:

1 If the backslash is followed by a single digit, it's a backreference.

2 If the first digit following the backslash is a 0, it's an octal escape.

3 If the number following the backslash is N (in decimal), and Perl already has seen N capture groups, Perl will consider this to be a backreference. Otherwise, it will consider it to be an octal escape. Note that if N has more than three digits, Perl only takes the first three for the octal escape; the rest are matched as is.

```
my $pat  = "(" x 999;
   $pat .= "a";
   $pat .= ")" x 999;
/^($pat)\1000$/;   #  Matches 'aa'; there are 1000 capture groups.
/^$pat\1000$/;     #  Matches 'a@0'; there are 999 capture groups
                   #    and \1000 is seen as \100 (a '@') and a '0'.
```

Hexadecimal escapes

Hexadecimal escapes start with `\x` and are then either followed by a two digit hexadecimal number, or a hexadecimal number of arbitrary length surrounded by curly braces. The hexadecimal number is the code point of the character you want to express.

Note that a character that is expressed as a hexadecimal escape is considered as a character without special meaning by the regex engine, and will match "as is".

Mnemonic: hexadecimal.

Examples (assuming an ASCII platform)

```
$str = "Perl";
$str =~ /\x50/;    # Match, "\x50" is "P".
$str =~ /\x50+/;   # Match, "\x50" is "P", it is repeated at least once.
$str =~ /P\x2B/;   # No match, "\x2B" is "+" and taken literally.

/\x{2603}\x{2602}/ # Snowman with an umbrella.
                   # The Unicode character 2603 is a snowman,
                   # the Unicode character 2602 is an umbrella.
/\x{263B}/         # Black smiling face.
/\x{263b}/         # Same, the hex digits A - F are case insensitive.
```

Modifiers

A number of backslash sequences have to do with changing the character, or characters following them. `\l` will lowercase the character following it, while `\u` will uppercase (or, more accurately, titlecase) the character following it. (They perform similar functionality as the functions `lcfirst` and `ucfirst`).

To uppercase or lowercase several characters, one might want to use `\L` or `\U`, which will lowercase/uppercase all characters following them, until either the end of the pattern, or the next occurrence of `\E`, whatever comes first. They perform similar functionality as the functions `lc` and `uc` do.

`\Q` is used to escape all characters following, up to the next `\E` or the end of the pattern. `\Q` adds a backslash to any character that isn't a letter, digit or underscore. This will ensure that any character between `\Q` and `\E` is matched literally, and will not be interpreted by the regexp engine.

Mnemonic: *L*owercase, *U*ppercase, *Q*uotemeta, *E*nd.

Examples

```
$sid     = "sid";
$greg    = "GrEg";
$miranda = "(Miranda)";
$str     =~ /\u$sid/;         # Matches 'Sid'
$str     =~ /\L$greg/;        # Matches 'greg'
$str     =~ /\Q$miranda\E/;   # Matches '(Miranda)', as if the pattern
                              #   had been written as /\(Miranda\)/
```

Character classes

Perl regular expressions have a large range of character classes. Some of the character classes are written as a backslash sequence. We will briefly discuss those here; full details of character classes can be found in *perlrecharclass*.

`\w` is a character class that matches any single *word* character (letters, digits, underscore). `\d` is a character class that matches any decimal digit, while the character class `\s` matches any whitespace character. New in perl 5.10.0 are the classes `\h` and `\v` which match horizontal and vertical whitespace characters.

The uppercase variants (`\W`, `\D`, `\S`, `\H`, and `\V`) are character classes that match any character that isn't a word character, digit, whitespace, horizontal whitespace nor vertical whitespace.

Mnemonics: *w*ord, *d*igit, *s*pace, *h*orizontal, *v*ertical.

Unicode classes

`\pP` (where `P` is a single letter) and `\p{Property}` are used to match a character that matches the given Unicode property; properties include things like "letter", or "thai character". Capitalizing the sequence to `\PP` and `\P{Property}` make the sequence match a character that doesn't match the given Unicode property. For more details, see *"Backslash sequences" in perlrecharclass* and *"Unicode Character Properties" in perlunicode*.

Mnemonic: *p*roperty.

Referencing

If capturing parenthesis are used in a regular expression, we can refer to the part of the source string that was matched, and match exactly the same thing. There are three ways of referring to such *backreference*: absolutely, relatively, and by name.

Absolute referencing

A backslash sequence that starts with a backslash and is followed by a number is an absolute reference (but be aware of the caveat mentioned above). If the number is *N*, it refers to the Nth set of parentheses - whatever has been matched by that set of parenthesis has to be matched by the `\N` as well.

Examples

```
/(\w+) \1/;    # Finds a duplicated word, (e.g. "cat cat").
/(.)(.)\2\1/;  # Match a four letter palindrome (e.g. "ABBA").
```

Relative referencing

New in perl 5.10.0 is a different way of referring to capture buffers: `\g`. `\g` takes a number as argument, with the number in curly braces (the braces are optional). If the number (N) does not have a sign, it's a reference to the Nth capture group (so `\g{2}` is equivalent to `\2` - except that `\g` always refers to a capture group and will never be seen as an octal escape). If the number is negative, the reference is relative, referring to the Nth group before the `\g{-N}`.

The big advantage of `\g{-N}` is that it makes it much easier to write patterns with references that can be interpolated in larger patterns, even if the larger pattern also contains capture groups.

Mnemonic: *g*roup.

Examples

```
/(A)        # Buffer 1
 (          # Buffer 2
   (B)      # Buffer 3
   \g{-1}   # Refers to buffer 3 (B)
   \g{-3}   # Refers to buffer 1 (A)
 )
/x;         # Matches "ABBA".

my $qr = qr /(.)(.)\g{-2}\g{-1}/;  # Matches 'abab', 'cdcd', etc.
/$qr$qr/                           # Matches 'ababcdcd'.
```

Named referencing

Also new in perl 5.10.0 is the use of named capture buffers, which can be referred to by name. This is done with `\g{name}`, which is a backreference to the capture buffer with the name *name*.

To be compatible with .Net regular expressions, `\g{name}` may also be written as `\k{name}`, `\k<name>` or `\k'name'`.

Note that `\g{}` has the potential to be ambiguous, as it could be a named reference, or an absolute or relative reference (if its argument is numeric). However, names are not allowed to start with digits, nor are they allowed to contain a hyphen, so there is no ambiguity.

Examples

```
/(?<word>\w+) \g{word}/ # Finds duplicated word, (e.g. "cat cat")
/(?<word>\w+) \k{word}/ # Same.
/(?<word>\w+) \k<word>/ # Same.
/(?<letter1>.)(?<letter2>.)\g{letter2}\g{letter1}/
                        # Match a four letter palindrome (e.g. "ABBA")
```

Assertions

Assertions are conditions that have to be true; they don't actually match parts of the substring. There are six assertions that are written as backslash sequences.

\A

\A only matches at the beginning of the string. If the /m modifier isn't used, then /\A/ is equivalent with /^/. However, if the /m modifier is used, then /^/ matches internal newlines, but the meaning of /\A/ isn't changed by the /m modifier. \A matches at the beginning of the string regardless whether the /m modifier is used.

\z, \Z

\z and \Z match at the end of the string. If the /m modifier isn't used, then /\Z/ is equivalent with /$/, that is, it matches at the end of the string, or before the newline at the end of the string. If the /m modifier is used, then /$/ matches at internal newlines, but the meaning of /\Z/ isn't changed by the /m modifier. \Z matches at the end of the string (or just before a trailing newline) regardless whether the /m modifier is used.

\z is just like \Z, except that it will not match before a trailing newline. \z will only match at the end of the string - regardless of the modifiers used, and not before a newline.

\G

\G is usually only used in combination with the /g modifier. If the /g modifier is used (and the match is done in scalar context), Perl will remember where in the source string the last match ended, and the next time, it will start the match from where it ended the previous time.

\G matches the point where the previous match ended, or the beginning of the string if there was no previous match.

Mnemonic: *G*lobal.

\b, \B

\b matches at any place between a word and a non-word character; \B matches at any place between characters where \b doesn't match. \b and \B assume there's a non-word character before the beginning and after the end of the source string; so \b will match at the beginning (or end) of the source string if the source string begins (or ends) with a word character. Otherwise, \B will match.

Mnemonic: *b*oundary.

Examples

```
"cat"   =~ /\Acat/;     # Match.
"cat"   =~ /cat\Z/;     # Match.
"cat\n" =~ /cat\Z/;     # Match.
"cat\n" =~ /cat\z/;     # No match.


"cat"   =~ /\bcat\b/;   # Matches.
"cats"  =~ /\bcat\b/;   # No match.
"cat"   =~ /\bcat\B/;   # No match.
"cats"  =~ /\bcat\B/;   # Match.


while ("cat dog" =~ /(\w+)/g) {
    print $1;           # Prints 'catdog'
}
while ("cat dog" =~ /\G(\w+)/g) {
    print $1;           # Prints 'cat'
}
```

Misc

Here we document the backslash sequences that don't fall in one of the categories above. They are:

\C

`\C` always matches a single octet, even if the source string is encoded in UTF-8 format, and the character to be matched is a multi-octet character. `\C` was introduced in perl 5.6.

Mnemonic: o*C*tet.

\K

This is new in perl 5.10.0. Anything that is matched left of `\K` is not included in `$&` - and will not be replaced if the pattern is used in a substitution. This will allow you to write `s/PAT1 \K PAT2/REPL/x` instead of `s/(PAT1) PAT2/${1}REPL/x` or `s/(?<=PAT1) PAT2/REPL/x`.

Mnemonic: *K*eep.

\N

This is a new experimental feature in perl 5.12.0. It matches any character that is not a newline. It is a short-hand for writing `[^\n]`, and is identical to the `.` metasymbol, except under the `/s` flag, which changes the meaning of `.`, but not `\N`.

Note that `\N{...}` can mean a *named or numbered character*.

Mnemonic: Complement of *n*.

\R

`\R` matches a *generic newline*, that is, anything that is considered a newline by Unicode. This includes all characters matched by `\v` (vertical whitespace), and the multi character sequence `"\x0D\x0A"` (carriage return followed by a line feed, aka the network newline, or the newline used in Windows text files). `\R` is equivalent to `(?>\x0D\x0A|\v)`. Since `\R` can match a sequence of more than one character, it cannot be put inside a bracketed character class; `/[\R]/` is an error; use `\v` instead. `\R` was introduced in perl 5.10.0.

Mnemonic: none really. `\R` was picked because PCRE already uses `\R`, and more importantly because Unicode recommends such a regular expression metacharacter, and suggests `\R` as the notation.

\X

This matches a Unicode *extended grapheme cluster*.

`\X` matches quite well what normal (non-Unicode-programmer) usage would consider a single character. As an example, consider a G with some sort of diacritic mark, such as an arrow. There is no such single character in Unicode, but one can be composed by using a G followed by a Unicode "COMBINING UPWARDS ARROW BELOW", and would be displayed by Unicode-aware software as if it were a single character.

Mnemonic: e*X*tended Unicode character.

Examples

```
"\x{256}" =~ /^\C\C$/;    # Match as chr (256) takes 2 octets in UTF-8.

$str =~ s/foo\Kbar/baz/g; # Change any 'bar' following a 'foo' to 'baz'.
$str =~ s/(.)\K\1//g;     # Delete duplicated characters.

"\n"   =~ /^\R$/;         # Match, \n   is a generic newline.
"\r"   =~ /^\R$/;         # Match, \r   is a generic newline.
"\r\n" =~ /^\R$/;         # Match, \r\n is a generic newline.

"P\x{0307}" =~ /^\X$/     # \X matches a P with a dot above.
```