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 any special meaning 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 issues 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 alphanumeric character as the delimiter of your pattern (which you probably shouldn't do for readability reasons), you 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. See also \o{}.
\1	Absolute backreference. Not in [].
\a	Alarm or bell.
\A	Beginning of string. Not in [].
\b{}, \b	Boundary. (\b is a backspace in []).
\B{}, \B	Not a boundary. Not in [].
\cX	Control-X.
\d	Match any digit character.
\D	Match any character that isn't a digit.
\e	Escape character.
\E	Turn off \Q, \L and \U processing. Not in [].
\f	Form feed.
\F	Foldcase till \E. Not in [].
\g{}, \g1	Named, absolute or relative backreference. Not in [].
\G	Pos assertion. Not in [].
\h	Match any horizontal whitespace character.
\H	Match any character that isn't 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	Match any character but newline. Not in [].
\N{}	Named or numbered (Unicode) character or sequence.
\o{}	Octal escape sequence.
\p{}, \pP	Match any character with the given Unicode property.
\P{}, \PP	Match any character without the given property.
\Q	Quote (disable) pattern metacharacters till \E. Not in [].
\r	Return character.
\R	Generic new line. Not in [].
\s	Match any whitespace character.
\S	Match any character that isn't a whitespace.
\t	Tab character.
\u	Titlecase next character. Not in [].
\U	Uppercase till \E. Not in [].
\v	Match any vertical whitespace character.

\V	Match any character that isn't vertical whitespace
\w	Match any word character.
\W	Match any character that isn't a word character.
\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" alone is a word-character/non-word-character boundary, and "\b{}" is some other type of boundary.

[2] "\n" matches a logical newline. Perl converts 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 and character sequences

Unicode characters have a Unicode name and numeric code point (ordinal) value. Use the "\N{}" construct to specify a character by either of these values. Certain sequences of characters also have names.

To specify by name, the name of the character or character sequence goes between the curly braces.

To specify a character by Unicode code point, use the form "\N{U+*code point*}", where *code point* is a number in hexadecimal that gives the code point 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).

Blanks may freely be inserted adjacent to but within the braces enclosing the name or code point. So "\N{ U+0041 }" is perfectly legal.

It is even possible to give your own names to characters and character sequences by using the charnames module. These custom names are lexically scoped, and so a given code point may have different names in different scopes. The name used is what is in effect at the time the "\N{}" is expanded. For patterns in double-quotish context, that means at the time the pattern is parsed. But for patterns that are delimitted by single quotes, the expansion is deferred until pattern compilation time, which may very well have a different "charnames" translator in effect.

(There is an expanded internal form that you may see in debug output: "\N{U+*code point*.*code point*...}". The "..." means any number of these *code point*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 or character sequence expressed as a named or numbered character is considered a character without special meaning by the regex engine, and will match "as is".

Example

```
 $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

There are two forms of octal escapes. Each is used to specify a character by its code point specified in base 8.

One form, available starting in Perl 5.14 looks like "\o{...}", where the dots represent one or more octal digits. It can be used for any Unicode character.

It was introduced to avoid the potential problems with the other form, available in all Perls. That form consists of a backslash followed by three octal digits. One problem with this form is that it can look exactly like an old-style backreference (see "Disambiguation rules between old-style octal escapes and backreferences" below.) You can avoid this by making the first of the three digits always a zero, but that makes \077 the largest code point specifiable.

In some contexts, a backslash followed by two or even one octal digits may be interpreted as an octal escape, sometimes with a warning, and because of some bugs, sometimes with surprising results. Also, if you are creating a regex out of smaller snippets concatenated together, and you use fewer than three digits, the beginning of one snippet may be interpreted as adding digits to the ending of the snippet before it. See "Absolute referencing" for more discussion and examples of the snippet problem.

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

To summarize, the "\o{}" form is always safe to use, and the other form is safe to use for code points through \077 when you use exactly three digits to specify them.

Mnemonic: *0*ctal or *o*ctal.

Examples (assuming an ASCII platform)

```
$str = "Perl";
$str =~ /\o{120}/;  # Match, "\120" is "P".
$str =~ /\120/;     # Same.
$str =~ /\o{120}+/; # Match, "\120" is "P",
                    # it's repeated at least once.
$str =~ /\120+/;    # Same.
$str =~ /P\053/;    # No match, "\053" is "+" and taken literally.
/\o{23073}/         # Black foreground, white background smiling face.
/\o{4801234567}/    # Raises a warning, and yields chr(4).
/\o{ 400}/          # LATIN CAPITAL LETTER A WITH MACRON
/\o{ 400 }/         # Same. These show blanks are allowed adjacent to
                    # the braces
```

Disambiguation rules between old-style octal escapes and backreferences

Octal escapes of the "\000" form outside of bracketed character classes potentially clash with old-style backreferences (see "Absolute referencing" below). 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 to disambiguate:

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 considers this a backreference. Otherwise, it considers it an octal escape. If N has more than three digits, Perl takes only 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'.
```

You can force a backreference interpretation always by using the "\g{...}" form. You can the force an octal interpretation always by using the "\o{...}" form, or for numbers up through \077 (= 63 decimal), by using three digits, beginning with a "0".

Hexadecimal escapes

Like octal escapes, there are two forms of hexadecimal escapes, but both start with the sequence "\x". This is followed by either exactly two hexadecimal digits forming a 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 expressed as one of these escapes is considered 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.
 /\x{ 263b }/       # Same, showing optional blanks adjacent to the
                    # braces
```

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 provide functionality similar to 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", whichever comes first. They provide functionality similar to what the functions "lc" and "uc" provide.

"\Q" is used to quote (disable) pattern metacharacters, up to the next "\E" or the end of the pattern. "\Q" adds a backslash to any character that could have special meaning to Perl. In the ASCII range, it quotes every character that isn't a letter, digit, or underscore. See "quotemeta" in perlfunc for details on what gets quoted for non-ASCII code points. Using this ensures that any character between "\Q" and "\E" will be matched literally, not interpreted as a metacharacter by the regex engine.

"\F" can be used to casefold all characters following, up to the next "\E" or the end of the pattern. It provides the functionality similar to the "fc" function.

Mnemonic: *L*owercase, *U*ppercase, *F*old-case, *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, Unicode marks, and connector punctuation (like the 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 exact set of characters matched by "\d", "\s", and "\w" varies depending on various pragma and regular expression modifiers. It is possible to restrict the match to the ASCII range by using the "/a" regular expression modifier. See perlrecharclass.

The uppercase variants ("\W", "\D", "\S", "\H", and "\V") are character classes that match, respectively, any character that isn't a word character, digit, whitespace, horizontal whitespace, or 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

Either "\g*N*" (starting in Perl 5.10.0), or "*N*" (old-style) where *N* is a positive (unsigned) decimal number of any length is an absolute reference to a capturing group.

N refers to the Nth set of parentheses, so "\g*N*" refers to whatever has been matched by that set of parentheses. Thus "\g1" refers to the first capture group in the regex.

The "\g*N*" form can be equivalently written as "\g{*N*}" which avoids ambiguity when building a regex by concatenating shorter strings. Otherwise if you had a regex "qr/ab/", and $a contained "\g1", and $b contained "37", you would get "/\g137/" which is probably not what you intended.

In the "*N*" form, *N* must not begin with a "0", and there must be at least *N* capturing groups, or else *N* is considered an octal escape (but something like "\18" is the same as "\0018"; that is, the octal escape "\001" followed by a literal digit "8").

Mnemonic: *g*roup.

Examples

```
/(\w+) \g1/;    # Finds a duplicated word, (e.g. "cat cat").
/(\w+) \1/;     # Same thing; written old-style.
/(\w+) \g{1}/;  # Same, using the safer braced notation
/(\w+) \g{ 1 }/;# Same, showing optional blanks adjacent to the braces
/(.)(.)\g2\g1/; # Match a four letter palindrome (e.g. "ABBA").
```

Relative referencing

"\g-*N*" (starting in Perl 5.10.0) is used for relative addressing. (It can be written as "\g{-*N*}".) It refers to the *N*th group before the "\g{-*N*}".

The big advantage of this form 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.

Examples

```
/(A)       # Group 1
 (         # Group 2
   (B)     # Group 3
   \g{-1}  # Refers to group 3 (B)
   \g{-3}  # Refers to group 1 (A)
   \g{ -3 } # Same, showing optional blanks adjacent to the braces
 )
/x;        # Matches "ABBA".

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

Named referencing

"\g{*name*}" (starting in Perl 5.10.0) can be used to back refer to a named capture group, dispensing completely with having to think about capture buffer positions.

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

To prevent any ambiguity, *name* must not start with a digit nor contain a hyphen.

Examples

```
/(?<word>\w+) \g{word}/   # Finds duplicated word, (e.g. "cat cat")
/(?<word>\w+) \k{word}/   # Same.
/(?<word>\w+) \g{ word }/ # Same, showing optional blanks adjacent to
                          # the braces
/(?<word>\w+) \k{ word }/ # Same.
/(?<word>\w+) \k<word>/   # Same.  There are no braces, so no blanks
                          # are permitted
/(?<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 to "/^/". 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 to "/$/"; that is, it matches at the end of the string, or one before the newline at the end of the string. If the "/m" modifier is used, then "/$/" 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 does not match before a trailing newline. "\z" matches at the end of the string only, regardless of the modifiers used, and not just before a newline. It is how to anchor the match to the true end of the string under all conditions.

\G "\G" is usually used only in combination with the "/g" modifier. If the "/g" modifier is used and the match is done in scalar context, Perl remembers 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 on that string ended, or the beginning of that string if there was no previous match.

Mnemonic: *G*lobal.

\b{}, \b, \B{}, \B

"\b{...}", available starting in v5.22, matches a boundary (between two characters, or before the first character of the string, or after the final character of the string) based on the Unicode rules for the boundary type specified inside the braces. The boundary types are given a few paragraphs below. "\B{...}" matches at any place between characters where "\b{...}" of the same type doesn't match.

"\b" when not immediately followed by a "{" is available in all Perls. It matches at any place between a word (something matched by "\w") and a non-word character ("\W"); "\B" when not immediately followed by a "{" matches at any place between characters where "\b" doesn't match. To get better word matching of natural language text, see "\b{wb}" below.

"\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.

Do not use something like "\b=head\d\b" and expect it to match the beginning of a line. It can't, because for there to be a boundary before the non-word "=", there must be a word character immediately previous. All plain "\b" and "\B" boundary determinations look for word characters alone, not for non-word characters nor for string ends. It may help to understand how "\b" and "\B" work by equating them as follows:

```
\b  really means   (?:(?<=\w)(?!\w)|(?<!\w)(?=\w))
\B  really means   (?:(?<=\w)(?=\w)|(?<!\w)(?!\w))
```

In contrast, "\b{...}" and "\B{...}" may or may not match at the beginning and end of the line, depending on the boundary type. These implement the Unicode default boundaries, specified in <https://www.unicode.org/reports/tr14/> and <https://www.unicode.org/reports/tr29/>. The boundary types are:

"\b{gcb}" or "\b{g}"

This matches a Unicode "Grapheme Cluster Boundary". (Actually Perl always uses the improved "extended" grapheme cluster"). These are explained below under "\X". In fact, "\X" is another way to get the same functionality. It is equivalent to "/.+?\b{gcb}/". Use whichever is most convenient for your situation.

"\b{lb}"

This matches according to the default Unicode Line Breaking Algorithm (<https://www.unicode.org/reports/tr14/>), as customized in that document (Example 7 of revision 35 <https://www.unicode.org/reports/tr14/tr14-35.html#Example7>) for better

handling of numeric expressions.

This is suitable for many purposes, but the Unicode::LineBreak module is available on CPAN that provides many more features, including customization.

"\b{sb}"
: This matches a Unicode "Sentence Boundary". This is an aid to parsing natural language sentences. It gives good, but imperfect results. For example, it thinks that "Mr. Smith" is two sentences. More details are at <https://www.unicode.org/reports/tr29/>. Note also that it thinks that anything matching "\R" (except form feed and vertical tab) is a sentence boundary. "\b{sb}" works with text designed for word-processors which wrap lines automatically for display, but hard-coded line boundaries are considered to be essentially the ends of text blocks (paragraphs really), and hence the ends of sentences. "\b{sb}" doesn't do well with text containing embedded newlines, like the source text of the document you are reading. Such text needs to be preprocessed to get rid of the line separators before looking for sentence boundaries. Some people view this as a bug in the Unicode standard, and this behavior is quite subject to change in future Perl versions.

"\b{wb}"
: This matches a Unicode "Word Boundary", but tailored to Perl expectations. This gives better (though not perfect) results for natural language processing than plain "\b" (without braces) does. For example, it understands that apostrophes can be in the middle of words and that parentheses aren't (see the examples below). More details are at <https://www.unicode.org/reports/tr29/>.

  The current Unicode definition of a Word Boundary matches between every white space character. Perl tailors this, starting in version 5.24, to generally not break up spans of white space, just as plain "\b" has always functioned. This allows "\b{wb}" to be a drop-in replacement for "\b", but with generally better results for natural language processing. (The exception to this tailoring is when a span of white space is immediately followed by something like U+0303, COMBINING TILDE. If the final space character in the span is a horizontal white space, it is broken out so that it attaches instead to the combining character. To be precise, if a span of white space that ends in a horizontal space has the character immediately following it have any of the Word Boundary property values "Extend", "Format" or "ZWJ", the boundary between the final horizontal space character and the rest of the span matches "\b{wb}". In all other cases the boundary between two white space characters matches "\B{wb}".)

It is important to realize when you use these Unicode boundaries, that you are taking a risk that a future version of Perl which contains a later version of the Unicode Standard will not work

precisely the same way as it did when your code was written. These rules are not considered stable and have been somewhat more subject to change than the rest of the Standard. Unicode reserves the right to change them at will, and Perl reserves the right to update its implementation to Unicode's new rules. In the past, some changes have been because new characters have been added to the Standard which have different characteristics than all previous characters, so new rules are formulated for handling them. These should not cause any backward compatibility issues. But some changes have changed the treatment of existing characters because the Unicode Technical Committee has decided that the change is warranted for whatever reason. This could be to fix a bug, or because they think better results are obtained with the new rule.

It is also important to realize that these are default boundary definitions, and that implementations may wish to tailor the results for particular purposes and locales. For example, some languages, such as Japanese and Thai, require dictionary lookup to accurately determine word boundaries.

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'
 }

 my $s = "He said, \"Is pi 3.14? (I'm not sure).\"";
 print join("|", $s =~ m/ ( .+? \b     ) /xg), "\n";
 print join("|", $s =~ m/ ( .+? \b{wb} ) /xg), "\n";
prints
 He| |said|, "|Is| |pi| |3|.|14|? (|I|'|m| |not| |sure
```

He| |said|,| |"|Is| |pi| |3.14|?| |(|I'm| |not| |sure|)|.|"

Misc

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

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

Mnemonic: *K*eep.

\N This feature, available starting in v5.12, 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 considered a linebreak sequence 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, sometimes called the network newline; it's the end of line sequence used in Microsoft text files opened in binary mode). "\R" is equivalent to "(?>\x0D\x0A|\v)". (The reason it doesn't backtrack is that the sequence is considered inseparable. That means that

```
"\x0D\x0A" =~ /^\R\x0A$/   # No match
```

fails, because the "\R" matches the entire string, and won't backtrack to match just the "\x0D".) 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.

Note that this does not respect any locale that might be in effect; it matches according to the platform's native character set.

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 its 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.

The match is greedy and non-backtracking, so that the cluster is never broken up into smaller components.

See also "\b{gcb}".

Mnemonic: e*X*tended Unicode character.

Examples

```
$str =~ s/foo\Kbar/baz/g; # Change any ’bar’ following a ’foo’ to ’baz’
$str =~ s/(.)\K\g1//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{307}" =~ /^\X$/     # \X matches a P with a dot above.
```