NAME

perlrecharclass - Perl Regular Expression Character Classes

DESCRIPTION

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

This manual page discusses the syntax and use of character classes in Perl regular expressions.

A character class is a way of denoting a set of characters in such a way that one character of the set is matched. It's important to remember that: matching a character class consumes exactly one character in the source string. (The source string is the string the regular expression is matched against.)

There are three types of character classes in Perl regular expressions: the dot, backslash sequences, and the form enclosed in square brackets. Keep in mind, though, that often the term "character class" is used to mean just the bracketed form. Certainly, most Perl documentation does that.

The dot

The dot (or period), `.` is probably the most used, and certainly the most well-known character class. By default, a dot matches any character, except for the newline. That default can be changed to add matching the newline by using the *single line* modifier: for the entire regular expression with the `/s` modifier, or locally with `(?s)` (and even globally within the scope of `use re '/s'`). (The `\N` backslash sequence, described below, matches any character except newline without regard to the *single line* modifier.)

Here are some examples:

```
"a"  =~  /./       # Match
"."  =~  /./       # Match
""   =~  /./       # No match (dot has to match a character)
"\n" =~  /./       # No match (dot does not match a newline)
"\n" =~  /./s      # Match (global 'single line' modifier)
"\n" =~  /(?s:.)/  # Match (local 'single line' modifier)
"ab" =~  /^.$/     # No match (dot matches one character)
```

Backslash sequences

A backslash sequence is a sequence of characters, the first one of which is a backslash. Perl ascribes special meaning to many such sequences, and some of these are character classes. That is, they match a single character each, provided that the character belongs to the specific set of characters defined by the sequence.

Here's a list of the backslash sequences that are character classes. They are discussed in more detail below. (For the backslash sequences that aren't character classes, see *perlrebackslash*.)

```
\d            Match a decimal digit character.
\D            Match a non-decimal-digit character.
\w            Match a "word" character.
\W            Match a non-"word" character.
\s            Match a whitespace character.
\S            Match a non-whitespace character.
\h            Match a horizontal whitespace character.
\H            Match a character that isn't horizontal whitespace.
\v            Match a vertical whitespace character.
\V            Match a character that isn't vertical whitespace.
\N            Match a character that isn't a newline.
\pP, \p{Prop} Match a character that has the given Unicode property.
\PP, \P{Prop} Match a character that doesn't have the Unicode property
```

\N

`\N`, available starting in v5.12, like the dot, matches any character that is not a newline. The difference is that `\N` is not influenced by the *single line* regular expression modifier (see *The dot* above). Note that the form `\N{...}` may mean something completely different. When the `{...}` is a *quantifier*, it means to match a non-newline character that many times. For example, `\N{3}` means to match 3 non-newlines; `\N{5,}` means to match 5 or more non-newlines. But if `{...}` is not a legal quantifier, it is presumed to be a named character. See *charnames* for those. For example, none of `\N{COLON}`, `\N{4F}`, and `\N{F4}` contain legal quantifiers, so Perl will try to find characters whose names are respectively `COLON`, `4F`, and `F4`.

Digits

`\d` matches a single character considered to be a decimal *digit*. If the `/a` regular expression modifier is in effect, it matches [0-9]. Otherwise, it matches anything that is matched by `\p{Digit}`, which includes [0-9]. (An unlikely possible exception is that under locale matching rules, the current locale might not have `[0-9]` matched by `\d`, and/or might match other characters whose code point is less than 256. The only such locale definitions that are legal would be to match `[0-9]` plus another set of 10 consecutive digit characters; anything else would be in violation of the C language standard, but Perl doesn't currently assume anything in regard to this.)

What this means is that unless the `/a` modifier is in effect `\d` not only matches the digits '0' - '9', but also Arabic, Devanagari, and digits from other languages. This may cause some confusion, and some security issues.

Some digits that `\d` matches look like some of the [0-9] ones, but have different values. For example, BENGALI DIGIT FOUR (U+09EA) looks very much like an ASCII DIGIT EIGHT (U+0038). An application that is expecting only the ASCII digits might be misled, or if the match is `\d+`, the matched string might contain a mixture of digits from different writing systems that look like they signify a number different than they actually do. *"num()" in Unicode::UCD* can be used to safely calculate the value, returning `undef` if the input string contains such a mixture.

What `\p{Digit}` means (and hence `\d` except under the `/a` modifier) is `\p{General_Category=Decimal_Number}`, or synonymously, `\p{General_Category=Digit}`. Starting with Unicode version 4.1, this is the same set of characters matched by `\p{Numeric_Type=Decimal}`. But Unicode also has a different property with a similar name, `\p{Numeric_Type=Digit}`, which matches a completely different set of characters. These characters are things such as `CIRCLED DIGIT ONE` or subscripts, or are from writing systems that lack all ten digits.

The design intent is for `\d` to exactly match the set of characters that can safely be used with "normal" big-endian positional decimal syntax, where, for example 123 means one 'hundred', plus two 'tens', plus three 'ones'. This positional notation does not necessarily apply to characters that match the other type of "digit", `\p{Numeric_Type=Digit}`, and so `\d` doesn't match them.

The Tamil digits (U+0BE6 - U+0BEF) can also legally be used in old-style Tamil numbers in which they would appear no more than one in a row, separated by characters that mean "times 10", "times 100", etc. (See *http://www.unicode.org/notes/tn21*.)

Any character not matched by `\d` is matched by `\D`.

Word characters

A `\w` matches a single alphanumeric character (an alphabetic character, or a decimal digit); or a connecting punctuation character, such as an underscore ("_"); or a "mark" character (like some sort of accent) that attaches to one of those. It does not match a whole word. To match a whole word, use `\w+`. This isn't the same thing as matching an English word, but in the ASCII range it is the same as a string of Perl-identifier characters.

If the `/a` modifier is in effect ...

`\w` matches the 63 characters [a-zA-Z0-9_].

otherwise ...

For code points above 255 ...

`\w` matches the same as `\p{Word}` matches in this range. That is, it matches Thai letters, Greek letters, etc. This includes connector punctuation (like the underscore) which connect two words together, or diacritics, such as a `COMBINING TILDE` and the modifier letters, which are generally used to add auxiliary markings to letters.

For code points below 256 ...

if locale rules are in effect ...

`\w` matches the platform's native underscore character plus whatever the locale considers to be alphanumeric.

if, instead, Unicode rules are in effect ...

`\w` matches exactly what `\p{Word}` matches.

otherwise ...

`\w` matches [a-zA-Z0-9_].

Which rules apply are determined as described in *"Which character set modifier is in effect?" in perlre*.

There are a number of security issues with the full Unicode list of word characters. See *http://unicode.org/reports/tr36*.

Also, for a somewhat finer-grained set of characters that are in programming language identifiers beyond the ASCII range, you may wish to instead use the more customized *Unicode Properties*, `\p{ID_Start}`, `\p{ID_Continue}`, `\p{XID_Start}`, and `\p{XID_Continue}`. See *http://unicode.org/reports/tr31*.

Any character not matched by `\w` is matched by `\W`.

Whitespace

`\s` matches any single character considered whitespace.

If the `/a` modifier is in effect ...

In all Perl versions, `\s` matches the 5 characters [\t\n\f\r]; that is, the horizontal tab, the newline, the form feed, the carriage return, and the space. Starting in Perl v5.18, it also matches the vertical tab, `\cK`. See note `[1]` below for a discussion of this.

otherwise ...

For code points above 255 ...

`\s` matches exactly the code points above 255 shown with an "s" column in the table below.

For code points below 256 ...

if locale rules are in effect ...

`\s` matches whatever the locale considers to be whitespace.

if, instead, Unicode rules are in effect ...

`\s` matches exactly the characters shown with an "s" column in the table below.

otherwise ...

`\s` matches [\t\n\f\r] and, starting in Perl v5.18, the vertical tab, `\cK`. (See note `[1]` below for a discussion of this.) Note that this list doesn't include the non-breaking space.

Which rules apply are determined as described in *"Which character set modifier is in effect?" in perlre*.

Any character not matched by `\s` is matched by `\S`.

`\h` matches any character considered horizontal whitespace; this includes the platform's space and tab characters and several others listed in the table below. `\H` matches any character not considered horizontal whitespace. They use the platform's native character set, and do not consider any locale that may otherwise be in use.

`\v` matches any character considered vertical whitespace; this includes the platform's carriage return and line feed characters (newline) plus several other characters, all listed in the table below. `\V` matches any character not considered vertical whitespace. They use the platform's native character set, and do not consider any locale that may otherwise be in use.

`\R` matches anything that can be considered a newline under Unicode rules. It can match a multi-character sequence. It cannot be used inside a bracketed character class; use `\v` instead (vertical whitespace). It uses the platform's native character set, and does not consider any locale that may otherwise be in use. Details are discussed in *perlrebackslash*.

Note that unlike `\s` (and `\d` and `\w`), `\h` and `\v` always match the same characters, without regard to other factors, such as the active locale or whether the source string is in UTF-8 format.

One might think that `\s` is equivalent to `[\h\v]`. This is indeed true starting in Perl v5.18, but prior to that, the sole difference was that the vertical tab (`"\cK"`) was not matched by `\s`.

The following table is a complete listing of characters matched by `\s`, `\h` and `\v` as of Unicode 6.3.

The first column gives the Unicode code point of the character (in hex format), the second column gives the (Unicode) name. The third column indicates by which class(es) the character is matched (assuming no locale is in effect that changes the `\s` matching).

```
0x0009         CHARACTER TABULATION   h s
0x000a               LINE FEED (LF)    vs
0x000b              LINE TABULATION    vs  [1]
0x000c               FORM FEED (FF)    vs
0x000d         CARRIAGE RETURN (CR)    vs
0x0020                        SPACE   h s
0x0085              NEXT LINE (NEL)    vs  [2]
0x00a0               NO-BREAK SPACE   h s  [2]
0x1680            OGHAM SPACE MARK   h s
0x2000                     EN QUAD   h s
0x2001                     EM QUAD   h s
0x2002                    EN SPACE   h s
0x2003                    EM SPACE   h s
0x2004          THREE-PER-EM SPACE   h s
0x2005           FOUR-PER-EM SPACE   h s
0x2006            SIX-PER-EM SPACE   h s
0x2007                FIGURE SPACE   h s
0x2008           PUNCTUATION SPACE   h s
0x2009                  THIN SPACE   h s
0x200a                  HAIR SPACE   h s
0x2028              LINE SEPARATOR    vs
0x2029         PARAGRAPH SEPARATOR    vs
0x202f       NARROW NO-BREAK SPACE   h s
0x205f   MEDIUM MATHEMATICAL SPACE   h s
0x3000           IDEOGRAPHIC SPACE   h s
```

[1]

Prior to Perl v5.18, `\s` did not match the vertical tab. `[^\S\cK]` (obscurely) matches what `\s` traditionally did.

[2]

NEXT LINE and NO-BREAK SPACE may or may not match `\s` depending on the rules in effect. See *the beginning of this section*.

Unicode Properties

`\pP` and `\p{Prop}` are character classes to match characters that fit given Unicode properties. One letter property names can be used in the `\pP` form, with the property name following the `\p`, otherwise, braces are required. When using braces, there is a single form, which is just the property name enclosed in the braces, and a compound form which looks like `\p{name=value}`, which means to match if the property "name" for the character has that particular "value". For instance, a match for a number can be written as `/\pN/` or as `/\p{Number}/`, or as `/\p{Number=True}/`. Lowercase letters are matched by the property *Lowercase_Letter* which has the short form *Ll*. They need the braces, so are written as `/\p{Ll}/` or `/\p{Lowercase_Letter}/`, or `/\p{General_Category=Lowercase_Letter}/` (the underscores are optional). `/\pLl/` is valid, but means something different. It matches a two character string: a letter (Unicode property `\pL`), followed by a lowercase `l`.

If locale rules are not in effect, the use of a Unicode property will force the regular expression into using Unicode rules, if it isn't already.

Note that almost all properties are immune to case-insensitive matching. That is, adding a `/i` regular expression modifier does not change what they match. There are two sets that are affected. The first set is `Uppercase_Letter`, `Lowercase_Letter`, and `Titlecase_Letter`, all of which match `Cased_Letter` under `/i` matching. The second set is `Uppercase`, `Lowercase`, and `Titlecase`, all of which match `Cased` under `/i` matching. (The difference between these sets is that some things, such as Roman numerals, come in both upper and lower case, so they are `Cased`, but aren't considered to be letters, so they aren't `Cased_Letters`. They're actually `Letter_Numbers`.) This set also includes its subsets `PosixUpper` and `PosixLower`, both of which under `/i` match `PosixAlpha`.

For more details on Unicode properties, see *"Unicode Character Properties" in perlunicode*; for a complete list of possible properties, see *"Properties accessible through \p{} and \P{}" in perluniprops*, which notes all forms that have `/i` differences. It is also possible to define your own properties. This is discussed in *"User-Defined Character Properties" in perlunicode*.

Unicode properties are defined (surprise!) only on Unicode code points. Starting in v5.20, when matching against `\p` and `\P`, Perl treats non-Unicode code points (those above the legal Unicode maximum of 0x10FFFF) as if they were typical unassigned Unicode code points.

Prior to v5.20, Perl raised a warning and made all matches fail on non-Unicode code points. This could be somewhat surprising:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}     # Fails on Perls < v5.20.
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}    # Also fails on Perls
                                              # < v5.20
```

Even though these two matches might be thought of as complements, until v5.20 they were so only on Unicode code points.

Examples

```
"a"  =~  /\w/      # Match, "a" is a 'word' character.
"7"  =~  /\w/      # Match, "7" is a 'word' character as well.
"a"  =~  /\d/      # No match, "a" isn't a digit.
"7"  =~  /\d/      # Match, "7" is a digit.
" "  =~  /\s/      # Match, a space is whitespace.
```

```
 "a"  =~  /\D/      # Match, "a" is a non-digit.
 "7"  =~  /\D/      # No match, "7" is not a non-digit.
 " "  =~  /\S/      # No match, a space is not non-whitespace.

 " "  =~  /\h/      # Match, space is horizontal whitespace.
 " "  =~  /\v/      # No match, space is not vertical whitespace.
 "\r" =~  /\v/      # Match, a return is vertical whitespace.

 "a"  =~  /\pL/     # Match, "a" is a letter.
 "a"  =~  /\p{Lu}/  # No match, /\p{Lu}/ matches upper case letters.

 "\x{0e0b}" =~ /\p{Thai}/  # Match, \x{0e0b} is the character
                           # 'THAI CHARACTER SO SO', and that's in
                           # Thai Unicode class.
 "a"  =~  /\P{Lao}/ # Match, as "a" is not a Laotian character.
```

It is worth emphasizing that `\d`, `\w`, etc, match single characters, not complete numbers or words. To match a number (that consists of digits), use `\d+`; to match a word, use `\w+`. But be aware of the security considerations in doing so, as mentioned above.

Bracketed Character Classes

The third form of character class you can use in Perl regular expressions is the bracketed character class. In its simplest form, it lists the characters that may be matched, surrounded by square brackets, like this: `[aeiou]`. This matches one of `a`, `e`, `i`, `o` or `u`. Like the other character classes, exactly one character is matched.* To match a longer string consisting of characters mentioned in the character class, follow the character class with a *quantifier*. For instance, `[aeiou]+` matches one or more lowercase English vowels.

Repeating a character in a character class has no effect; it's considered to be in the set only once.

Examples:

```
 "e"  =~  /[aeiou]/        # Match, as "e" is listed in the class.
 "p"  =~  /[aeiou]/        # No match, "p" is not listed in the class.
 "ae" =~  /^[aeiou]$/      # No match, a character class only matches
                           # a single character.
 "ae" =~  /^[aeiou]+$/     # Match, due to the quantifier.

 -------
```

* There are two exceptions to a bracketed character class matching a single character only. Each requires special handling by Perl to make things work:

- When the class is to match caselessly under `/i` matching rules, and a character that is explicitly mentioned inside the class matches a multiple-character sequence caselessly under Unicode rules, the class will also match that sequence. For example, Unicode says that the letter `LATIN SMALL LETTER SHARP S` should match the sequence `ss` under `/i` rules. Thus,

  ```
  'ss' =~ /\A\N{LATIN SMALL LETTER SHARP S}\z/i             # Matches
  'ss' =~ /\A[aeioust\N{LATIN SMALL LETTER SHARP S}]\z/i    # Matches
  ```

  For this to happen, the class must not be inverted (see *Negation*) and the character must be explicitly specified, and not be part of a multi-character range (not even as one of its endpoints). (*Character Ranges* will be explained shortly.) Therefore,

  ```
  'ss' =~ /\A[\0-\x{ff}]\z/ui       # Doesn't match
  ```

```
'ss' =~ /\A[\0-\N{LATIN SMALL LETTER SHARP S}]\z/ui   # No match
'ss' =~ /\A[\xDF-\xDF]\z/ui   # Matches on ASCII platforms, since
                              # \xDF is LATIN SMALL LETTER SHARP S,
                              # and the range is just a single
                              # element
```

Note that it isn't a good idea to specify these types of ranges anyway.

- Some names known to `\N{...}` refer to a sequence of multiple characters, instead of the usual single character. When one of these is included in the class, the entire sequence is matched. For example,

  ```
  "\N{TAMIL LETTER KA}\N{TAMIL VOWEL SIGN AU}"
                               =~ / ^ [\N{TAMIL SYLLABLE KAU}]  $ /x;
  ```

  matches, because `\N{TAMIL SYLLABLE KAU}` is a named sequence consisting of the two characters matched against. Like the other instance where a bracketed class can match multiple characters, and for similar reasons, the class must not be inverted, and the named sequence may not appear in a range, even one where it is both endpoints. If these happen, it is a fatal error if the character class is within the scope of *`use re 'strict`*, or within an extended *`(?[...])`* class; otherwise only the first code point is used (with a `regexp`-type warning raised).

Special Characters Inside a Bracketed Character Class

Most characters that are meta characters in regular expressions (that is, characters that carry a special meaning like `.`, `*`, or `()` lose their special meaning and can be used inside a character class without the need to escape them. For instance, `[()]` matches either an opening parenthesis, or a closing parenthesis, and the parens inside the character class don't group or capture. Be aware that, unless the pattern is evaluated in single-quotish context, variable interpolation will take place before the bracketed class is parsed:

```
$, = "\t| ";
$a =~ m'[$,]';        # single-quotish: matches '$' or ','
$a =~ q{[$,]}'        # same
$a =~ m/[$,]/;        # double-quotish: matches "\t", "|", or " "
```

Characters that may carry a special meaning inside a character class are: `\`, `^`, `-`, `[` and `]`, and are discussed below. They can be escaped with a backslash, although this is sometimes not needed, in which case the backslash may be omitted.

The sequence `\b` is special inside a bracketed character class. While outside the character class, `\b` is an assertion indicating a point that does not have either two word characters or two non-word characters on either side, inside a bracketed character class, `\b` matches a backspace character.

The sequences `\a`, `\c`, `\e`, `\f`, `\n`, `\N{`*`NAME`*`}`, `\N{U+`*`hex char`*`}`, `\r`, `\t`, and `\x` are also special and have the same meanings as they do outside a bracketed character class.

Also, a backslash followed by two or three octal digits is considered an octal number.

A `[` is not special inside a character class, unless it's the start of a POSIX character class (see *POSIX Character Classes* below). It normally does not need escaping.

A `]` is normally either the end of a POSIX character class (see *POSIX Character Classes* below), or it signals the end of the bracketed character class. If you want to include a `]` in the set of characters, you must generally escape it.

However, if the `]` is the *first* (or the second if the first character is a caret) character of a bracketed character class, it does not denote the end of the class (as you cannot have an empty class) and is considered part of the set of characters that can be matched without escaping.

Examples:

```
 "+"   =~ /[+?*]/     #  Match, "+" in a character class is not special.
 "\cH" =~ /[\b]/      #  Match, \b inside in a character class
                      #  is equivalent to a backspace.
 "]"   =~ /[][]/      #  Match, as the character class contains
                      #  both [ and ].
 "[]"  =~ /[[]]/      #  Match, the pattern contains a character class
                      #  containing just [, and the character class is
                      #  followed by a ].
```

Bracketed Character Classes and the /xx pattern modifier

Normally SPACE and TAB characters have no special meaning inside a bracketed character class; they are just added to the list of characters matched by the class. But if the */xx* pattern modifier is in effect, they are generally ignored and can be added to improve readability. They can't be added in the middle of a single construct:

```
 / [ \x{10 FFFF} ] /xx  # WRONG!
```

The SPACE in the middle of the hex constant is illegal.

To specify a literal SPACE character, you can escape it with a backslash, like:

```
 /[ a e i o u \  ]/xx
```

This matches the English vowels plus the SPACE character.

For clarity, you should already have been using `\t` to specify a literal tab, and `\t` is unaffected by `/xx`.

Character Ranges

It is not uncommon to want to match a range of characters. Luckily, instead of listing all characters in the range, one may use the hyphen (`-`). If inside a bracketed character class you have two characters separated by a hyphen, it's treated as if all characters between the two were in the class. For instance, `[0-9]` matches any ASCII digit, and `[a-m]` matches any lowercase letter from the first half of the ASCII alphabet.

Note that the two characters on either side of the hyphen are not necessarily both letters or both digits. Any character is possible, although not advisable. `['-?]` contains a range of characters, but most people will not know which characters that means. Furthermore, such ranges may lead to portability problems if the code has to run on a platform that uses a different character set, such as EBCDIC.

If a hyphen in a character class cannot syntactically be part of a range, for instance because it is the first or the last character of the character class, or if it immediately follows a range, the hyphen isn't special, and so is considered a character to be matched literally. If you want a hyphen in your set of characters to be matched and its position in the class is such that it could be considered part of a range, you must escape that hyphen with a backslash.

Examples:

```
 [a-z]       #  Matches a character that is a lower case ASCII letter.
 [a-fz]      #  Matches any letter between 'a' and 'f' (inclusive) or
             #  the letter 'z'.
 [-z]        #  Matches either a hyphen ('-') or the letter 'z'.
 [a-f-m]     #  Matches any letter between 'a' and 'f' (inclusive), the
             #  hyphen ('-'), or the letter 'm'.
 ['-?]       #  Matches any of the characters  '()*+,-./0123456789:;<=>?
```

```
            #  (But not on an EBCDIC platform).
[\N{APOSTROPHE}-\N{QUESTION MARK}]
            #  Matches any of the characters  '()*+,-./0123456789:;<=>?
            #  even on an EBCDIC platform.
[\N{U+27}-\N{U+3F}] # Same. (U+27 is "'", and U+3F is "?")
```

As the final two examples above show, you can achieve portablity to non-ASCII platforms by using the `\N{...}` form for the range endpoints. These indicate that the specified range is to be interpreted using Unicode values, so `[\N{U+27}-\N{U+3F}]` means to match `\N{U+27}`, `\N{U+28}`, `\N{U+29}`, ..., `\N{U+3D}`, `\N{U+3E}`, and `\N{U+3F}`, whatever the native code point versions for those are. These are called "Unicode" ranges. If either end is of the `\N{...}` form, the range is considered Unicode. A `regexp` warning is raised under `"use re 'strict'"` if the other endpoint is specified non-portably:

```
[\N{U+00}-\x09]    # Warning under re 'strict'; \x09 is non-portable
[\N{U+00}-\t]      # No warning;
```

Both of the above match the characters `\N{U+00}` `\N{U+01}`, ... `\N{U+08}`, `\N{U+09}`, but the `\x09` looks like it could be a mistake so the warning is raised (under `re 'strict'`) for it.

Perl also guarantees that the ranges `A-Z`, `a-z`, `0-9`, and any subranges of these match what an English-only speaker would expect them to match on any platform. That is, `[A-Z]` matches the 26 ASCII uppercase letters; `[a-z]` matches the 26 lowercase letters; and `[0-9]` matches the 10 digits. Subranges, like `[h-k]`, match correspondingly, in this case just the four letters `"h"`, `"i"`, `"j"`, and `"k"`. This is the natural behavior on ASCII platforms where the code points (ordinal values) for `"h"` through `"k"` are consecutive integers (0x68 through 0x6B). But special handling to achieve this may be needed on platforms with a non-ASCII native character set. For example, on EBCDIC platforms, the code point for `"h"` is 0x88, `"i"` is 0x89, `"j"` is 0x91, and `"k"` is 0x92. Perl specially treats `[h-k]` to exclude the seven code points in the gap: 0x8A through 0x90. This special handling is only invoked when the range is a subrange of one of the ASCII uppercase, lowercase, and digit ranges, AND each end of the range is expressed either as a literal, like `"A"`, or as a named character (`\N{...}`, including the `\N{U+...` form).

EBCDIC Examples:

```
[i-j]               #  Matches either "i" or "j"
[i-\N{LATIN SMALL LETTER J}]  # Same
[i-\N{U+6A}]        #  Same
[\N{U+69}-\N{U+6A}] #  Same
[\x{89}-\x{91}]     #  Matches 0x89 ("i"), 0x8A .. 0x90, 0x91 ("j")
[i-\x{91}]          #  Same
[\x{89}-j]          #  Same
[i-J]               #  Matches, 0x89 ("i") .. 0xC1 ("J"); special
                    #  handling doesn't apply because range is mixed
                    #  case
```

Negation

It is also possible to instead list the characters you do not want to match. You can do so by using a caret (^) as the first character in the character class. For instance, `[^a-z]` matches any character that is not a lowercase ASCII letter, which therefore includes more than a million Unicode code points. The class is said to be "negated" or "inverted".

This syntax make the caret a special character inside a bracketed character class, but only if it is the first character of the class. So if you want the caret as one of the characters to match, either escape the caret or else don't list it first.

In inverted bracketed character classes, Perl ignores the Unicode rules that normally say that named

sequence, and certain characters should match a sequence of multiple characters use under caseless `/i` matching. Following those rules could lead to highly confusing situations:

```
"ss" =~ /^[^\xDF]+$/ui;   # Matches!
```

This should match any sequences of characters that aren't `\xDF` nor what `\xDF` matches under `/i`. `"s"` isn't `\xDF`, but Unicode says that `"ss"` is what `\xDF` matches under `/i`. So which one "wins"? Do you fail the match because the string has `ss` or accept it because it has an `s` followed by another `s`? Perl has chosen the latter. (See note in *Bracketed Character Classes* above.)

Examples:

```
"e"  =~  /[^aeiou]/   #  No match, the 'e' is listed.
"x"  =~  /[^aeiou]/   #  Match, as 'x' isn't a lowercase vowel.
"^"  =~  /[^^]/       #  No match, matches anything that isn't a caret.
"^"  =~  /[x^]/       #  Match, caret is not special here.
```

Backslash Sequences

You can put any backslash sequence character class (with the exception of `\N` and `\R`) inside a bracketed character class, and it will act just as if you had put all characters matched by the backslash sequence inside the character class. For instance, `[a-f\d]` matches any decimal digit, or any of the lowercase letters between 'a' and 'f' inclusive.

`\N` within a bracketed character class must be of the forms `\N{`*`name`*`}` or `\N{U+`*`hex char`*`}`, and NOT be the form that matches non-newlines, for the same reason that a dot `.` inside a bracketed character class loses its special meaning: it matches nearly anything, which generally isn't what you want to happen.

Examples:

```
/[\p{Thai}\d]/     # Matches a character that is either a Thai
                   # character, or a digit.
/[^\p{Arabic}()]/  # Matches a character that is neither an Arabic
                   # character, nor a parenthesis.
```

Backslash sequence character classes cannot form one of the endpoints of a range. Thus, you can't say:

```
/[\p{Thai}-\d]/     # Wrong!
```

POSIX Character Classes

POSIX character classes have the form `[:class:]`, where *class* is the name, and the `[:` and `:]` delimiters. POSIX character classes only appear *inside* bracketed character classes, and are a convenient and descriptive way of listing a group of characters.

Be careful about the syntax,

```
# Correct:
$string =~ /[[:alpha:]]/

# Incorrect (will warn):
$string =~ /[:alpha:]/
```

The latter pattern would be a character class consisting of a colon, and the letters `a`, `l`, `p` and `h`.

POSIX character classes can be part of a larger bracketed character class. For example,

```
[01[:alpha:]%]
```

is valid and matches '0', '1', any alphabetic character, and the percent sign.

Perl recognizes the following POSIX character classes:

```
alpha  Any alphabetical character ("[A-Za-z]").
alnum  Any alphanumeric character ("[A-Za-z0-9]").
ascii  Any character in the ASCII character set.
blank  A GNU extension, equal to a space or a horizontal tab ("\t").
cntrl  Any control character.  See Note [2] below.
digit  Any decimal digit ("[0-9]"), equivalent to "\d".
graph  Any printable character, excluding a space.  See Note [3] below.
lower  Any lowercase character ("[a-z]").
print  Any printable character, including a space.  See Note [4] below.
punct  Any graphical character excluding "word" characters.  Note [5].
space  Any whitespace character. "\s" including the vertical tab
       ("\cK").
upper  Any uppercase character ("[A-Z]").
word   A Perl extension ("[A-Za-z0-9_]"), equivalent to "\w".
xdigit Any hexadecimal digit ("[0-9a-fA-F]").
```

Like the *Unicode properties*, most of the POSIX properties match the same regardless of whether case-insensitive (`/i`) matching is in effect or not. The two exceptions are `[:upper:]` and `[:lower:]`. Under `/i`, they each match the union of `[:upper:]` and `[:lower:]`.

Most POSIX character classes have two Unicode-style `\p` property counterparts. (They are not official Unicode properties, but Perl extensions derived from official Unicode properties.) The table below shows the relation between POSIX character classes and these counterparts.

One counterpart, in the column labelled "ASCII-range Unicode" in the table, matches only characters in the ASCII character set.

The other counterpart, in the column labelled "Full-range Unicode", matches any appropriate characters in the full Unicode character set. For example, `\p{Alpha}` matches not just the ASCII alphabetic characters, but any character in the entire Unicode character set considered alphabetic. An entry in the column labelled "backslash sequence" is a (short) equivalent.

```
[[:...:]]      ASCII-range          Full-range  backslash  Note
                Unicode              Unicode     sequence
-----------------------------------------------------------
  alpha      \p{PosixAlpha}       \p{XPosixAlpha}
  alnum      \p{PosixAlnum}       \p{XPosixAlnum}
  ascii      \p{ASCII}
  blank      \p{PosixBlank}       \p{XPosixBlank}  \h      [1]
                                  or \p{HorizSpace}        [1]
  cntrl      \p{PosixCntrl}       \p{XPosixCntrl}          [2]
  digit      \p{PosixDigit}       \p{XPosixDigit}  \d
  graph      \p{PosixGraph}       \p{XPosixGraph}          [3]
  lower      \p{PosixLower}       \p{XPosixLower}
  print      \p{PosixPrint}       \p{XPosixPrint}          [4]
  punct      \p{PosixPunct}       \p{XPosixPunct}          [5]
             \p{PerlSpace}        \p{XPerlSpace}   \s      [6]
  space      \p{PosixSpace}       \p{XPosixSpace}          [6]
  upper      \p{PosixUpper}       \p{XPosixUpper}
  word       \p{PosixWord}        \p{XPosixWord}   \w
  xdigit     \p{PosixXDigit}      \p{XPosixXDigit}
```

[1]

`\p{Blank}` and `\p{HorizSpace}` are synonyms.

[2]

Control characters don't produce output as such, but instead usually control the terminal somehow: for example, newline and backspace are control characters. On ASCII platforms, in the ASCII range, characters whose code points are between 0 and 31 inclusive, plus 127 (`DEL`) are control characters; on EBCDIC platforms, their counterparts are control characters.

[3]

Any character that is *graphical*, that is, visible. This class consists of all alphanumeric characters and all punctuation characters.

[4]

All printable characters, which is the set of all graphical characters plus those whitespace characters which are not also controls.

[5]

`\p{PosixPunct}` and `[[:punct:]]` in the ASCII range match all non-controls, non-alphanumeric, non-space characters: `[-!"#$%&'()*+,./:;<=>?@[\\\]^_`{|}~]` (although if a locale is in effect, it could alter the behavior of `[[:punct:]]`).

The similarly named property, `\p{Punct}`, matches a somewhat different set in the ASCII range, namely `[-!"#%&'()*,./:;?@[\\\]_{}]`. That is, it is missing the nine characters `[$+<=>^`|~]`. This is because Unicode splits what POSIX considers to be punctuation into two categories, Punctuation and Symbols.

`\p{XPosixPunct}` and (under Unicode rules) `[[:punct:]]`, match what `\p{PosixPunct}` matches in the ASCII range, plus what `\p{Punct}` matches. This is different than strictly matching according to `\p{Punct}`. Another way to say it is that if Unicode rules are in effect, `[[:punct:]]` matches all characters that Unicode considers punctuation, plus all ASCII-range characters that Unicode considers symbols.

[6]

`\p{XPerlSpace}` and `\p{Space}` match identically starting with Perl v5.18. In earlier versions, these differ only in that in non-locale matching, `\p{XPerlSpace}` did not match the vertical tab, `\cK`. Same for the two ASCII-only range forms.

There are various other synonyms that can be used besides the names listed in the table. For example, `\p{XPosixAlpha}` can be written as `\p{Alpha}`. All are listed in *"Properties accessible through \p{} and \P{}" in perluniprops*.

Both the `\p` counterparts always assume Unicode rules are in effect. On ASCII platforms, this means they assume that the code points from 128 to 255 are Latin-1, and that means that using them under locale rules is unwise unless the locale is guaranteed to be Latin-1 or UTF-8. In contrast, the POSIX character classes are useful under locale rules. They are affected by the actual rules in effect, as follows:

If the `/a` modifier, is in effect ...

Each of the POSIX classes matches exactly the same as their ASCII-range counterparts.

otherwise ...

For code points above 255 ...

The POSIX class matches the same as its Full-range counterpart.

For code points below 256 ...

if locale rules are in effect ...

The POSIX class matches according to the locale, except:

`word`

also includes the platform's native underscore character, no matter what the locale is.

`ascii`

on platforms that don't have the POSIX `ascii` extension, this matches just the platform's native ASCII-range characters.

`blank`

on platforms that don't have the POSIX `blank` extension, this matches just the platform's native tab and space characters.

if, instead, Unicode rules are in effect ...

The POSIX class matches the same as the Full-range counterpart.

otherwise ...

The POSIX class matches the same as the ASCII range counterpart.

Which rules apply are determined as described in *"Which character set modifier is in effect?" in perlre*.

It is proposed to change this behavior in a future release of Perl so that whether or not Unicode rules are in effect would not change the behavior: Outside of locale, the POSIX classes would behave like their ASCII-range counterparts. If you wish to comment on this proposal, send email to `perl5-porters@perl.org`.

Negation of POSIX character classes

A Perl extension to the POSIX character class is the ability to negate it. This is done by prefixing the class name with a caret (^). Some examples:

```
     POSIX         ASCII-range     Full-range  backslash
                    Unicode         Unicode    sequence
-----------------------------------------------------------
[[:^digit:]]   \P{PosixDigit}  \P{XPosixDigit}   \D
[[:^space:]]   \P{PosixSpace}  \P{XPosixSpace}
               \P{PerlSpace}   \P{XPerlSpace}    \S
[[:^word:]]    \P{PerlWord}    \P{XPosixWord}    \W
```

The backslash sequence can mean either ASCII- or Full-range Unicode, depending on various factors as described in *"Which character set modifier is in effect?" in perlre*.

[= =] and [. .]

Perl recognizes the POSIX character classes `[=class=]` and `[.class.]`, but does not (yet?) support them. Any attempt to use either construct raises an exception.

Examples

```
 /[[:digit:]]/           # Matches a character that is a digit.
 /[01[:lower:]]/         # Matches a character that is either a
                         # lowercase letter, or '0' or '1'.
 /[[:digit:][:^xdigit:]]/ # Matches a character that can be anything
                         # except the letters 'a' to 'f' and 'A' to
                         # 'F'.  This is because the main character
                         # class is composed of two POSIX character
                         # classes that are ORed together, one that
                         # matches any digit, and the other that
```

```
# matches anything that isn't a hex digit.
# The OR adds the digits, leaving only the
# letters 'a' to 'f' and 'A' to 'F' excluded.
```

Extended Bracketed Character Classes

This is a fancy bracketed character class that can be used for more readable and less error-prone classes, and to perform set operations, such as intersection. An example is

```
/(?[ \p{Thai} & \p{Digit} ])/
```

This will match all the digit characters that are in the Thai script.

This is an experimental feature available starting in 5.18, and is subject to change as we gain field experience with it. Any attempt to use it will raise a warning, unless disabled via

```
no warnings "experimental::regex_sets";
```

Comments on this feature are welcome; send email to `perl5-porters@perl.org`.

The rules used by *`use re 'strict`* apply to this construct.

We can extend the example above:

```
/(?[ ( \p{Thai} + \p{Lao} ) & \p{Digit} ])/
```

This matches digits that are in either the Thai or Laotian scripts.

Notice the white space in these examples. This construct always has the `/xx` modifier turned on within it.

The available binary operators are:

```
&    intersection
+    union
|    another name for '+', hence means union
-    subtraction (the result matches the set consisting of those
     code points matched by the first operand, excluding any that
     are also matched by the second operand)
^    symmetric difference (the union minus the intersection).  This
     is like an exclusive or, in that the result is the set of code
     points that are matched by either, but not both, of the
     operands.
```

There is one unary operator:

```
!    complement
```

All the binary operators left associate; "`&`" is higher precedence than the others, which all have equal precedence. The unary operator right associates, and has highest precedence. Thus this follows the normal Perl precedence rules for logical operators. Use parentheses to override the default precedence and associativity.

The main restriction is that everything is a metacharacter. Thus, you cannot refer to single characters by doing something like this:

```
/(?[ a + b ])/ # Syntax error!
```

The easiest way to specify an individual typable character is to enclose it in brackets:

```
/(?[ [a] + [b] ])/
```

(This is the same thing as `[ab]`.) You could also have said the equivalent:

```
/(?[[ a b ]])/
```

(You can, of course, specify single characters by using, `\x{...}`, `\N{...}`, etc.)

This last example shows the use of this construct to specify an ordinary bracketed character class without additional set operations. Note the white space within it. This is allowed because `/xx` is automatically turned on within this construct.

All the other escapes accepted by normal bracketed character classes are accepted here as well.

Because this construct compiles under *`use re 'strict`*, unrecognized escapes that generate warnings in normal classes are fatal errors here, as well as all other warnings from these class elements, as well as some practices that don't currently warn outside `re 'strict'`. For example you cannot say

```
/(?[ [ \xF ] ])/     # Syntax error!
```

You have to have two hex digits after a braceless `\x` (use a leading zero to make two). These restrictions are to lower the incidence of typos causing the class to not match what you thought it would.

If a regular bracketed character class contains a `\p{}` or `\P{}` and is matched against a non-Unicode code point, a warning may be raised, as the result is not Unicode-defined. No such warning will come when using this extended form.

The final difference between regular bracketed character classes and these, is that it is not possible to get these to match a multi-character fold. Thus,

```
/(?[ [\xDF] ])/iu
```

does not match the string `ss`.

You don't have to enclose POSIX class names inside double brackets, hence both of the following work:

```
/(?[ [:word:] - [:lower:] ])/
/(?[ [[:word:]] - [[:lower:]] ])/
```

Any contained POSIX character classes, including things like `\w` and `\D` respect the `/a` (and `/aa`) modifiers.

`(?[ ])` is a regex-compile-time construct. Any attempt to use something which isn't knowable at the time the containing regular expression is compiled is a fatal error. In practice, this means just three limitations:

1 When compiled within the scope of `use locale` (or the `/l` regex modifier), this construct assumes that the execution-time locale will be a UTF-8 one, and the generated pattern always uses Unicode rules. What gets matched or not thus isn't dependent on the actual runtime locale, so tainting is not enabled. But a `locale` category warning is raised if the runtime locale turns out to not be UTF-8.

2 Any *user-defined property* used must be already defined by the time the regular expression is compiled (but note that this construct can be used instead of such properties).

3 A regular expression that otherwise would compile using `/d` rules, and which uses this construct will instead use `/u`. Thus this construct tells Perl that you don't want `/d` rules for the entire regular expression containing it.

Note that skipping white space applies only to the interior of this construct. There must not be any space between any of the characters that form the initial `(?[`. Nor may there be space between the closing `])` characters.

Just as in all regular expressions, the pattern can be built up by including variables that are interpolated at regex compilation time. Care must be taken to ensure that you are getting what you expect. For example:

```
my $thai_or_lao = '\p{Thai} + \p{Lao}';
...
qr/(?[ \p{Digit} & $thai_or_lao ])/;
```

compiles to

```
qr/(?[ \p{Digit} & \p{Thai} + \p{Lao} ])/;
```

But this does not have the effect that someone reading the code would likely expect, as the intersection applies just to `\p{Thai}`, excluding the Laotian. Pitfalls like this can be avoided by parenthesizing the component pieces:

```
my $thai_or_lao = '( \p{Thai} + \p{Lao} )';
```

But any modifiers will still apply to all the components:

```
my $lower = '\p{Lower} + \p{Digit}';
qr/(?[ \p{Greek} & $lower ])/i;
```

matches upper case things. You can avoid surprises by making the components into instances of this construct by compiling them:

```
my $thai_or_lao = qr/(?[ \p{Thai} + \p{Lao} ])/;
my $lower = qr/(?[ \p{Lower} + \p{Digit} ])/;
```

When these are embedded in another pattern, what they match does not change, regardless of parenthesization or what modifiers are in effect in that outer pattern.

Due to the way that Perl parses things, your parentheses and brackets may need to be balanced, even including comments. If you run into any examples, please send them to `perlbug@perl.org`, so that we can have a concrete example for this man page.

We may change it so that things that remain legal uses in normal bracketed character classes might become illegal within this experimental construct. One proposal, for example, is to forbid adjacent uses of the same character, as in `(?[ [aa] ])`. The motivation for such a change is that this usage is likely a typo, as the second "a" adds nothing.