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. The default can be changed to add matching the newline by using the *single line* modifier: either for the entire regular expression with the `/s` modifier, or locally with `(?s)`. (The experimental `\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.  Experimental.
\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
```

Digits

`\d` matches a single character that is considered to be a decimal *digit*. What is considered a decimal digit depends on the internal encoding of the source string and the locale that is in effect. If the source string is in UTF-8 format, `\d` not only matches the digits '0' - '9', but also Arabic, Devanagari and digits from other languages. Otherwise, if there is a locale in effect, it will match whatever characters the locale considers decimal digits. Without a locale, `\d` matches just the digits '0' to '9'. See *Locale, EBCDIC, Unicode and UTF-8*.

Unicode digits may cause some confusion, and some security issues. In UTF-8 strings, `\d` matches the same characters matched by `\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 subscripts.

The design intent is for `\d` to match all the digits (and no other characters) that can be used with "normal" big-endian positional decimal syntax, whereby a sequence of such digits {N0, N1, N2, ...Nn} has the numeric value (...(N0 * 10 + N1) * 10 + N2) * 10 ... + Nn). In Unicode 5.2, 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*.)

Some of the non-European digits that `\d` matches look like European ones, but have different values. For example, BENGALI DIGIT FOUR (U+09A) looks very much like an ASCII DIGIT EIGHT (U+0038).

It may be useful for security purposes for an application to require that all digits in a row be from the same script. See *"charscript()" in Unicode::UCD*.

Any character that isn't matched by `\d` will be matched by `\D`.

Word characters

A `\w` matches a single alphanumeric character (an alphabetic character, or a decimal digit) or an underscore (_), not a whole word. To match a whole word, use `\w+`. This isn't the same thing as matching an English word, but is the same as a string of Perl-identifier characters. What is considered a word character depends on the internal encoding of the string and the locale or EBCDIC code page that is in effect. If it's in UTF-8 format, `\w` matches those characters that are considered word characters in the Unicode database. That is, it not only matches ASCII letters, but also Thai letters, Greek letters, etc. If the source string isn't in UTF-8 format, `\w` matches those characters that are considered word characters by the current locale or EBCDIC code page. Without a locale or EBCDIC code page, `\w` matches the ASCII letters, digits and the underscore. See *Locale, EBCDIC, Unicode and UTF-8*.

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, "ID_Start", ID_Continue", "XID_Start", and "XID_Continue". See *http://unicode.org/reports/tr31*.

Any character that isn't matched by `\w` will be matched by `\W`.

Whitespace

`\s` matches any single character that is considered whitespace. The exact set of characters matched by `\s` depends on whether the source string is in UTF-8 format and the locale or EBCDIC code page that is in effect. If it's in UTF-8 format, `\s` matches what is considered whitespace in the Unicode database; the complete list is in the table below. Otherwise, if there is a locale or EBCDIC code page in effect, `\s` matches whatever is considered whitespace by the current locale or EBCDIC code page.

Without a locale or EBCDIC code page, `\s` matches the horizontal tab (`\t`), the newline (`\n`), the form feed (`\f`), the carriage return (`\r`), and the space. (Note that it doesn't match the vertical tab, `\cK`.) Perhaps the most notable possible surprise is that `\s` matches a non-breaking space only if the non-breaking space is in a UTF-8 encoded string or the locale or EBCDIC code page that is in effect has that character. See *Locale, EBCDIC, Unicode and UTF-8*.

Any character that isn't matched by `\s` will be matched by `\S`.

`\h` will match any character that is considered horizontal whitespace; this includes the space and the tab characters and a number other characters, all of which are listed in the table below. `\H` will match any character that is not considered horizontal whitespace.

`\v` will match any character that is considered vertical whitespace; this includes the carriage return and line feed characters (newline) plus several other characters, all listed in the table below. `\V` will match any character that is not considered vertical whitespace.

`\R` matches anything that can be considered a newline under Unicode rules. It's not a character class, as it can match a multi-character sequence. Therefore, it cannot be used inside a bracketed character class; use `\v` instead (vertical whitespace). Details are discussed in *perlrebackslash*.

Note that unlike `\s`, `\d` and `\w`, `\h` and `\v` always match the same characters, regardless whether the source string is in UTF-8 format or not. The set of characters they match is also not influenced by locale nor EBCDIC code page.

One might think that `\s` is equivalent to `[\h\v]`. This is not true. The vertical tab (`"\x0b"`) is not matched by `\s`, it is however considered vertical whitespace. Furthermore, if the source string is not in UTF-8 format, and any locale or EBCDIC code page that is in effect doesn't include them, the next line (ASCII-platform `"\x85"`) and the no-break space (ASCII-platform `"\xA0"`) characters are not matched by `\s`, but are by `\v` and `\h` respectively. If the source string is in UTF-8 format, both the next line and the no-break space are matched by `\s`.

The following table is a complete listing of characters matched by `\s`, `\h` and `\v` as of Unicode 5.2.

The first column gives the 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 or EBCDIC code page is in effect that changes the `\s` matching).

```
0x00009         CHARACTER TABULATION   h s
0x0000a               LINE FEED (LF)    vs
0x0000b              LINE TABULATION    v
0x0000c               FORM FEED (FF)    vs
0x0000d         CARRIAGE RETURN (CR)    vs
0x00020                        SPACE   h s
0x00085              NEXT LINE (NEL)    vs  [1]
0x000a0               NO-BREAK SPACE   h s  [1]
0x01680             OGHAM SPACE MARK   h s
0x0180e   MONGOLIAN VOWEL SEPARATOR   h s
0x02000                      EN QUAD   h s
0x02001                      EM QUAD   h s
0x02002                     EN SPACE   h s
0x02003                     EM SPACE   h s
0x02004           THREE-PER-EM SPACE   h s
0x02005            FOUR-PER-EM SPACE   h s
0x02006             SIX-PER-EM SPACE   h s
0x02007                 FIGURE SPACE   h s
0x02008            PUNCTUATION SPACE   h s
0x02009                   THIN SPACE   h s
0x0200a                   HAIR SPACE   h s
0x02028               LINE SEPARATOR    vs
```

```
0x02029         PARAGRAPH SEPARATOR    vs
0x0202f       NARROW NO-BREAK SPACE    h s
0x0205f   MEDIUM MATHEMATICAL SPACE    h s
0x03000            IDEOGRAPHIC SPACE   h s
```

[1]

NEXT LINE and NO-BREAK SPACE only match `\s` if the source string is in UTF-8 format, or the locale or EBCDIC code page that is in effect includes them.

It is worth noting that `\d`, `\w`, etc, match single characters, not complete numbers or words. To match a number (that consists of integers), use `\d+`; to match a word, use `\w+`.

\N

`\N` is new in 5.12, and is experimental. It, like the dot, will match 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`.

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 the 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 as 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`.

For more details, see *"Unicode Character Properties" in perlunicode*; for a complete list of possible properties, see *"Properties accessible through \p{} and \P{}" in perluniprops*. It is also possible to define your own properties. This is discussed in *"User-Defined Character Properties" in perlunicode*.

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.
```

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 will be 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 a string of 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.
```

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.

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+`*`wide 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 ].
```

Character Ranges

It is not uncommon to want to match a range of characters. Luckily, instead of listing all the 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 the characters between the two are 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 necessary 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 will be. 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 will be considered a character that is to be matched literally. You have to escape the hyphen with a backslash if you want to have 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.

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).
```

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 a character that is not a lowercase ASCII letter.

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 to have the caret as one of the characters you want to match, you either have to escape the caret, or not list it first.

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`) inside a bracketed character class, and it will act just as if you put all the characters matched by the backslash sequence inside the character class. For instance, `[a-f\d]` will match 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+`*wide 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 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, though they currently suffer from portability issues (see below and *Locale, EBCDIC, Unicode and UTF-8*).

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 alphanumerical 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" plus 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]").
```

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, will only match 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}` will match not just the ASCII alphabetic characters, but any character in the entire Unicode character set that is considered to be alphabetic.

(Each of the counterparts has various synonyms as well. *"Properties accessible through \p{} and \P{}" in perluniprops* lists all the synonyms, plus all the characters matched by each of the ASCII-range properties. For example `\p{AHex}` is a synonym for `\p{ASCII_Hex_Digit}`, and any `\p` property name can be prefixed with "Is" such as `\p{IsAlpha}`.)

Both the `\p` forms are unaffected by any locale that is in effect, or whether the string is in UTF-8 format or not, or whether the platform is EBCDIC or not. In contrast, the POSIX character classes are affected. If the source string is in UTF-8 format, the POSIX classes (with the exception of `[[:punct:]]`, see Note [5] below) behave like their "Full-range" Unicode counterparts. If the source string is not in UTF-8 format, and no locale is in effect, and the platform is not EBCDIC, all the POSIX classes behave like their ASCII-range counterparts. Otherwise, they behave based on the rules of the locale or EBCDIC code page.

It is proposed to change this behavior in a future release of Perl so that the the UTF8ness of the source string will be irrelevant to the behavior of the POSIX character classes. This means they will always behave in strict accordance with the official POSIX standard. That is, if either locale or EBCDIC code page is present, they will behave in accordance with those; if absent, the classes will match only their ASCII-range counterparts. If you disagree with this proposal, send email to `perl5-porters@perl.org`.

`[[:...:]]`	ASCII-range Unicode	Full-range Unicode	backslash sequence	Note
alpha	\p{PosixAlpha}	\p{Alpha}		
alnum	\p{PosixAlnum}	\p{Alnum}		
ascii	\p{ASCII}			
blank	\p{PosixBlank}	\p{Blank} =		[1]
		\p{HorizSpace}	\h	[1]
cntrl	\p{PosixCntrl}	\p{Cntrl}		[2]
digit	\p{PosixDigit}	\p{Digit}	\d	
graph	\p{PosixGraph}	\p{Graph}		[3]
lower	\p{PosixLower}	\p{Lower}		
print	\p{PosixPrint}	\p{Print}		[4]
punct	\p{PosixPunct}	\p{Punct}		[5]
	\p{PerlSpace}	\p{SpacePerl}	\s	[6]
space	\p{PosixSpace}	\p{Space}		[6]
upper	\p{PosixUpper}	\p{Upper}		
word	\p{PerlWord}	\p{Word}	\w	
xdigit	\p{ASCII_Hex_Digit}	\p{XDigit}		

[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. In the ASCII range, characters whose ordinals are between 0 and 31 inclusive, plus 127 (`DEL`) are control

characters.On EBCDIC platforms, it is likely that the code page will define `[[:cntrl:]]` to be the EBCDIC equivalents of the ASCII controls, plus the controls that in Unicode have ordinals from 128 through 159.

[3]

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

[4]

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

[5] (punct)

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

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

When the matching string is in UTF-8 format, `[[:punct:]]` matches what it 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 for a UTF-8 string, `[[:punct:]]` matches all the characters that Unicode considers to be punctuation, plus all the ASCII-range characters that Unicode considers to be symbols.

[6]

`\p{SpacePerl}` and `\p{Space}` differ only in that `\p{Space}` additionally matches the vertical tab, `\cK`. Same for the two ASCII-only range forms.

Negation

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{Digit}      \D
[[:^space:]]   \P{PosixSpace}     \P{Space}
               \P{PerlSpace}      \P{SpacePerl}  \S
[[:^word:]]    \P{PerlWord}       \P{Word}       \W
```

[= =] and [. .]

Perl will recognize the POSIX character classes `[=class=]`, and `[.class.]`, but does not (yet?) support them. Use of such a construct will lead to an error.

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'.  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 result matches all
    # characters except the letters 'a' to 'f' and
    # 'A' to 'F'.
```

Locale, EBCDIC, Unicode and UTF-8

Some of the character classes have a somewhat different behaviour depending on the internal encoding of the source string, and the locale that is in effect, and if the program is running on an EBCDIC platform.

`\w`, `\d`, `\s` and the POSIX character classes (and their negations, including `\W`, `\D`, `\S`) suffer from this behaviour. (Since the backslash sequences `\b` and `\B` are defined in terms of `\w` and `\W`, they also are affected.)

The rule is that if the source string is in UTF-8 format, the character classes match according to the Unicode properties. If the source string isn't, then the character classes match according to whatever locale or EBCDIC code page is in effect. If there is no locale nor EBCDIC, they match the ASCII defaults (0 to 9 for `\d`; 52 letters, 10 digits and underscore for `\w`; etc.).

This usually means that if you are matching against characters whose `ord()` values are between 128 and 255 inclusive, your character class may match or not depending on the current locale or EBCDIC code page, and whether the source string is in UTF-8 format. The string will be in UTF-8 format if it contains characters whose `ord()` value exceeds 255. But a string may be in UTF-8 format without it having such characters. See *"The "Unicode Bug"" in perlunicode*.

For portability reasons, it may be better to not use `\w`, `\d`, `\s` or the POSIX character classes, and use the Unicode properties instead.

Examples

```
$str =  "\xDF";      # $str is not in UTF-8 format.
$str =~ /^\w/;       # No match, as $str isn't in UTF-8 format.
$str .= "\x{0e0b}";  # Now $str is in UTF-8 format.
$str =~ /^\w/;       # Match! $str is now in UTF-8 format.
chop $str;
$str =~ /^\w/;       # Still a match! $str remains in UTF-8 format.
```