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: 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
```

\N

`\N` is new in 5.12, and is experimental. It, 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. Such a locale definition 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 ("_"). 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 Unicode rules are in effect or if on an EBCDIC platform ...

\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 ...

\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. (Note that it doesn't match the vertical tab, \cK on ASCII platforms.)

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. Note that this is likely to include the vertical space, unlike non-locale \s matching.

if Unicode rules are in effect or if on an EBCDIC platform ...

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

otherwise ...

\s matches [\t\n\f\r]. 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'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). 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 not true. The difference is that the vertical tab ("\x0b") is not matched by \s; it is however considered vertical whitespace.

The following table is a complete listing of characters matched by \s, \h and \v as of Unicode 6.0.

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 or EBCDIC code page is in effect that changes the \s matching).

```
0x0009         CHARACTER TABULATION   h s
0x000a               LINE FEED (LF)    vs
0x000b              LINE TABULATION    v
0x000c               FORM FEED (FF)    vs
0x000d         CARRIAGE RETURN (CR)    vs
0x0020                        SPACE   h s
0x0085              NEXT LINE (NEL)    vs  [1]
0x00a0               NO-BREAK SPACE   h s  [1]
0x1680            OGHAM SPACE MARK   h s
0x180e   MONGOLIAN VOWEL SEPARATOR   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]

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 neither the `/a` modifier nor locale rules are in effect, the use of a Unicode property will force the regular expression into using Unicode rules.

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. A warning is raised and all matches fail on non-Unicode code points (those above the legal Unicode maximum of 0x10FFFF). This can be somewhat surprising,

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}      # Fails.
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}     # Also fails!
```

Even though these two matches might be thought of as complements, they are 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 is an exception to a bracketed character class matching a single character only. When the class is to match caselessly under `/i` matching rules, and a character inside the class matches a multiple-character sequence caselessly under Unicode rules, the class (when not *inverted*) 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
```

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+`*`hex char`*`}`, `\r`, `\t`, and `\x` are also special

and have the same meanings as they do outside a bracketed character class. (However, inside a bracketed character class, if `\N{`*NAME*`}` expands to a sequence of characters, only the first one in the sequence is used, with a warning.)

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 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).
```

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 certain characters should match a sequence of multiple characters 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.

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 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" 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, 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. In the ASCII range, characters whose code points 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 code pointss from 128 through 159.

[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{SpacePerl} and \p{Space} differ only in that in non-locale matching, \p{Space} additionally matches 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{PosixAlpha} can be written as \p{Alpha}. All are listed in *"Properties accessible through \p{} and \P{}" in perluniprops*, plus all characters matched by each ASCII-range property.

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 that `word` uses the platform's native underscore character, no matter what the locale is.

if Unicode rules are in effect or if on an EBCDIC platform ...

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 or an EBCDIC code page, 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'.  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'.
```