NAME

perlunicode - Unicode support in Perl

DESCRIPTION

If you haven't already, before reading this document, you should become familiar with both *perlunitut* and *perluniintro*.

Unicode aims to **UNI**-fy the en-**CODE**-ings of all the world's character sets into a single Standard. For quite a few of the various coding standards that existed when Unicode was first created, converting from each to Unicode essentially meant adding a constant to each code point in the original standard, and converting back meant just subtracting that same constant. For ASCII and ISO-8859-1, the constant is 0. For ISO-8859-5, (Cyrillic) the constant is 864; for Hebrew (ISO-8859-8), it's 1488; Thai (ISO-8859-11), 3424; and so forth. This made it easy to do the conversions, and facilitated the adoption of Unicode.

And it worked; nowadays, those legacy standards are rarely used. Most everyone uses Unicode.

Unicode is a comprehensive standard. It specifies many things outside the scope of Perl, such as how to display sequences of characters. For a full discussion of all aspects of Unicode, see *http://www.unicode.org*.

Important Caveats

Even though some of this section may not be understandable to you on first reading, we think it's important enough to highlight some of the gotchas before delving further, so here goes:

Unicode support is an extensive requirement. While Perl does not implement the Unicode standard or the accompanying technical reports from cover to cover, Perl does support many Unicode features.

Also, the use of Unicode may present security issues that aren't obvious. Read *Unicode Security Considerations*.

Safest if you `use feature 'unicode_strings'`

> In order to preserve backward compatibility, Perl does not turn on full internal Unicode support unless the pragma *`use feature 'unicode_strings'`* is specified. (This is automatically selected if you `use 5.012` or higher.) Failure to do this can trigger unexpected surprises. See *The "Unicode Bug"* below.
>
> This pragma doesn't affect I/O. Nor does it change the internal representation of strings, only their interpretation. There are still several places where Unicode isn't fully supported, such as in filenames.

Input and Output Layers

> Use the `:encoding(...)` layer to read from and write to filehandles using the specified encoding. (See *open*.)

You should convert your non-ASCII, non-UTF-8 Perl scripts to be UTF-8.

> See *encoding*.

`use utf8` still needed to enable *UTF-8* in scripts

> If your Perl script is itself encoded in *UTF-8*, the `use utf8` pragma must be explicitly included to enable recognition of that (in string or regular expression literals, or in identifier names). **This is the only time when an explicit `use utf8` is needed.** (See *utf8*).

`BOM`-marked scripts and *UTF-16* scripts autodetected

> However, if a Perl script begins with the Unicode `BOM` (UTF-16LE, UTF16-BE, or UTF-8), or if the script looks like non-`BOM`-marked UTF-16 of either endianness, Perl will correctly read in the script as the appropriate Unicode encoding. (`BOM`-less UTF-8 cannot be effectively recognized or differentiated from ISO 8859-1 or other eight-bit encodings.)

Byte and Character Semantics

Before Unicode, most encodings used 8 bits (a single byte) to encode each character. Thus a character was a byte, and a byte was a character, and there could be only 256 or fewer possible characters. "Byte Semantics" in the title of this section refers to this behavior. There was no need to distinguish between "Byte" and "Character".

Then along comes Unicode which has room for over a million characters (and Perl allows for even more). This means that a character may require more than a single byte to represent it, and so the two terms are no longer equivalent. What matter are the characters as whole entities, and not usually the bytes that comprise them. That's what the term "Character Semantics" in the title of this section refers to.

Perl had to change internally to decouple "bytes" from "characters". It is important that you too change your ideas, if you haven't already, so that "byte" and "character" no longer mean the same thing in your mind.

The basic building block of Perl strings has always been a "character". The changes basically come down to that the implementation no longer thinks that a character is always just a single byte.

There are various things to note:

- String handling functions, for the most part, continue to operate in terms of characters. `length()`, for example, returns the number of characters in a string, just as before. But that number no longer is necessarily the same as the number of bytes in the string (there may be more bytes than characters). The other such functions include `chop()`, `chomp()`, `substr()`, `pos()`, `index()`, `rindex()`, `sort()`, `sprintf()`, and `write()`.

  The exceptions are:

  - the bit-oriented `vec`
  - the byte-oriented `pack`/`unpack "C"` format

    However, the `W` specifier does operate on whole characters, as does the `U` specifier.
  - some operators that interact with the platform's operating system

    Operators dealing with filenames are examples.
  - when the functions are called from within the scope of the *`use bytes`* pragma

    Likely, you should use this only for debugging anyway.

- Strings--including hash keys--and regular expression patterns may contain characters that have ordinal values larger than 255.

  If you use a Unicode editor to edit your program, Unicode characters may occur directly within the literal strings in UTF-8 encoding, or UTF-16. (The former requires a `BOM` or `use utf8`, the latter requires a `BOM`.)

  "Creating Unicode" in perluniintro gives other ways to place non-ASCII characters in your strings.

- The `chr()` and `ord()` functions work on whole characters.
- Regular expressions match whole characters. For example, `"."` matches a whole character instead of only a single byte.
- The `tr///` operator translates whole characters. (Note that the `tr///CU` functionality has been removed. For similar functionality to that, see `pack('U0', ...)` and `pack('C0', ...)`).
- `scalar reverse()` reverses by character rather than by byte.

- The bit string operators, `& | ^ ~` and (starting in v5.22) `&. |. ^. ~.` can operate on characters that don't fit into a byte. However, the current behavior is likely to change. You should not use these operators on strings that are encoded in UTF-8. If you're not sure about the encoding of a string, downgrade it before using any of these operators; you can use *`utf8::utf8_downgrade()`*.

The bottom line is that Perl has always practiced "Character Semantics", but with the advent of Unicode, that is now different than "Byte Semantics".

ASCII Rules versus Unicode Rules

Before Unicode, when a character was a byte was a character, Perl knew only about the 128 characters defined by ASCII, code points 0 through 127 (except for under `use locale`). That left the code points 128 to 255 as unassigned, and available for whatever use a program might want. The only semantics they have is their ordinal numbers, and that they are members of none of the non-negative character classes. None are considered to match `\w` for example, but all match `\W`.

Unicode, of course, assigns each of those code points a particular meaning (along with ones above 255). To preserve backward compatibility, Perl only uses the Unicode meanings when there is some indication that Unicode is what is intended; otherwise the non-ASCII code points remain treated as if they are unassigned.

Here are the ways that Perl knows that a string should be treated as Unicode:

- Within the scope of `use utf8`

  If the whole program is Unicode (signified by using 8-bit **U**nicode **T**ransformation **F**ormat), then all strings within it must be Unicode.

- Within the scope of *`use feature 'unicode_strings'`*

  This pragma was created so you can explicitly tell Perl that operations executed within its scope are to use Unicode rules. More operations are affected with newer perls. See *The "Unicode Bug"*.

- Within the scope of `use 5.012` or higher

  This implicitly turns on `use feature 'unicode_strings'`.

- Within the scope of *`use locale 'not_characters'`*, or *`use locale`* and the current locale is a UTF-8 locale.

  The former is defined to imply Unicode handling; and the latter indicates a Unicode locale, hence a Unicode interpretation of all strings within it.

- When the string contains a Unicode-only code point

  Perl has never accepted code points above 255 without them being Unicode, so their use implies Unicode for the whole string.

- When the string contains a Unicode named code point `\N{...}`

  The `\N{...}` construct explicitly refers to a Unicode code point, even if it is one that is also in ASCII. Therefore the string containing it must be Unicode.

- When the string has come from an external source marked as Unicode

  The *`-C`* command line option can specify that certain inputs to the program are Unicode, and the values of this can be read by your Perl code, see *"${^UNICODE}" in perlvar*.

* When the string has been upgraded to UTF-8

  The function *`utf8::utf8_upgrade()`* can be explicitly used to permanently (unless a subsequent `utf8::utf8_downgrade()` is called) cause a string to be treated as Unicode.

* There are additional methods for regular expression patterns

A pattern that is compiled with the `/u` or `/a` modifiers is treated as Unicode (though there are some restrictions with `/a`). Under the `/d` and `/l` modifiers, there are several other indications for Unicode; see *"Character set modifiers" in perlre*.

Note that all of the above are overridden within the scope of *`use bytes`*; but you should be using this pragma only for debugging.

Note also that some interactions with the platform's operating system never use Unicode rules.

When Unicode rules are in effect:

- Case translation operators use the Unicode case translation tables.

  Note that `uc()`, or `\U` in interpolated strings, translates to uppercase, while `ucfirst`, or `\u` in interpolated strings, translates to titlecase in languages that make the distinction (which is equivalent to uppercase in languages without the distinction).

  There is a CPAN module, *`Unicode::Casing`*, which allows you to define your own mappings to be used in `lc()`, `lcfirst()`, `uc()`, `ucfirst()`, and `fc` (or their double-quoted string inlined versions such as `\U`). (Prior to Perl 5.16, this functionality was partially provided in the Perl core, but suffered from a number of insurmountable drawbacks, so the CPAN module was written instead.)

- Character classes in regular expressions match based on the character properties specified in the Unicode properties database.

  `\w` can be used to match a Japanese ideograph, for instance; and `[[:digit:]]` a Bengali number.

- Named Unicode properties, scripts, and block ranges may be used (like bracketed character classes) by using the `\p{}` "matches property" construct and the `\P{}` negation, "doesn't match property".

  See *Unicode Character Properties* for more details.

  You can define your own character properties and use them in the regular expression with the `\p{}` or `\P{}` construct. See *User-Defined Character Properties* for more details.

Extended Grapheme Clusters (Logical characters)

Consider a character, say `H`. It could appear with various marks around it, such as an acute accent, or a circumflex, or various hooks, circles, arrows, *etc.*, above, below, to one side or the other, *etc*. There are many possibilities among the world's languages. The number of combinations is astronomical, and if there were a character for each combination, it would soon exhaust Unicode's more than a million possible characters. So Unicode took a different approach: there is a character for the base `H`, and a character for each of the possible marks, and these can be variously combined to get a final logical character. So a logical character--what appears to be a single character--can be a sequence of more than one individual characters. The Unicode standard calls these "extended grapheme clusters" (which is an improved version of the no-longer much used "grapheme cluster"); Perl furnishes the `\X` regular expression construct to match such sequences in their entirety.

But Unicode's intent is to unify the existing character set standards and practices, and several pre-existing standards have single characters that mean the same thing as some of these combinations, like ISO-8859-1, which has quite a few of them. For example, `"LATIN CAPITAL LETTER E WITH ACUTE"` was already in this standard when Unicode came along. Unicode therefore added it to its repertoire as that single character. But this character is considered by Unicode to be equivalent to the sequence consisting of the character `"LATIN CAPITAL LETTER E"` followed by the character `"COMBINING ACUTE ACCENT"`.

`"LATIN CAPITAL LETTER E WITH ACUTE"` is called a "pre-composed" character, and its equivalence with the "E" and the "COMBINING ACCENT" sequence is called canonical equivalence. All pre-composed characters are said to have a decomposition (into the equivalent sequence), and the decomposition type is also called canonical. A string may be comprised as much as possible of

precomposed characters, or it may be comprised of entirely decomposed characters. Unicode calls these respectively, "Normalization Form Composed" (NFC) and "Normalization Form Decomposed". The *Unicode::Normalize* module contains functions that convert between the two. A string may also have both composed characters and decomposed characters; this module can be used to make it all one or the other.

You may be presented with strings in any of these equivalent forms. There is currently nothing in Perl 5 that ignores the differences. So you'll have to specially hanlde it. The usual advice is to convert your inputs to NFD before processing further.

For more detailed information, see *http://unicode.org/reports/tr15/*.

Unicode Character Properties

(The only time that Perl considers a sequence of individual code points as a single logical character is in the \X construct, already mentioned above. Therefore "character" in this discussion means a single Unicode code point.)

Very nearly all Unicode character properties are accessible through regular expressions by using the \p{} "matches property" construct and the \P{} "doesn't match property" for its negation.

For instance, \p{Uppercase} matches any single character with the Unicode "Uppercase" property, while \p{L} matches any character with a General_Category of "L" (letter) property (see *General_Category* below). Brackets are not required for single letter property names, so \p{L} is equivalent to \pL.

More formally, \p{Uppercase} matches any single character whose Unicode Uppercase property value is True, and \P{Uppercase} matches any character whose Uppercase property value is False, and they could have been written as \p{Uppercase=True} and \p{Uppercase=False}, respectively.

This formality is needed when properties are not binary; that is, if they can take on more values than just True and False. For example, the Bidi_Class property (see *Bidirectional Character Types* below), can take on several different values, such as Left, Right, Whitespace, and others. To match these, one needs to specify both the property name (Bidi_Class), AND the value being matched against (Left, Right, *etc.*). This is done, as in the examples above, by having the two components separated by an equal sign (or interchangeably, a colon), like \p{Bidi_Class: Left}.

All Unicode-defined character properties may be written in these compound forms of \p{*property*=*value*} or \p{*property*:*value*}, but Perl provides some additional properties that are written only in the single form, as well as single-form short-cuts for all binary properties and certain others described below, in which you may omit the property name and the equals or colon separator.

Most Unicode character properties have at least two synonyms (or aliases if you prefer): a short one that is easier to type and a longer one that is more descriptive and hence easier to understand. Thus the "L" and "Letter" properties above are equivalent and can be used interchangeably. Likewise, "Upper" is a synonym for "Uppercase", and we could have written \p{Uppercase} equivalently as \p{Upper}. Also, there are typically various synonyms for the values the property can be. For binary properties, "True" has 3 synonyms: "T", "Yes", and "Y"; and "False" has correspondingly "F", "No", and "N". But be careful. A short form of a value for one property may not mean the same thing as the same short form for another. Thus, for the *General_Category* property, "L" means "Letter", but for the *Bidi_Class* property, "L" means "Left". A complete list of properties and synonyms is in *perluniprops*.

Upper/lower case differences in property names and values are irrelevant; thus \p{Upper} means the same thing as \p{upper} or even \p{UpPeR}. Similarly, you can add or subtract underscores anywhere in the middle of a word, so that these are also equivalent to \p{U_p_p_e_r}. And white space is irrelevant adjacent to non-word characters, such as the braces and the equals or colon separators, so \p{ Upper } and \p{ Upper_case : Y } are equivalent to these as well. In

fact, white space and even hyphens can usually be added or deleted anywhere. So even `\p{ Up-per case = Yes}` is equivalent. All this is called "loose-matching" by Unicode. The few places where stricter matching is used is in the middle of numbers, and in the Perl extension properties that begin or end with an underscore. Stricter matching cares about white space (except adjacent to non-word characters), hyphens, and non-interior underscores.

You can also use negation in both `\p{}` and `\P{}` by introducing a caret (`^`) between the first brace and the property name: `\p{^Tamil}` is equal to `\P{Tamil}`.

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. And the second set is `Uppercase`, `Lowercase`, and `Titlecase`, all of which match `Cased` under `/i` matching. This set also includes its subsets `PosixUpper` and `PosixLower` both of which under `/i` match `PosixAlpha`. (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 letters, so they aren't `Cased_Letter`'s.)

See *Beyond Unicode code points* for special considerations when matching Unicode properties against non-Unicode code points.

General_Category

Every Unicode character is assigned a general category, which is the "most usual categorization of a character" (from *http://www.unicode.org/reports/tr44*).

The compound way of writing these is like `\p{General_Category=Number}` (short: `\p{gc:n}`). But Perl furnishes shortcuts in which everything up through the equal or colon separator is omitted. So you can instead just write `\pN`.

Here are the short and long forms of the values the `General Category` property can have:

```
Short       Long

L           Letter
LC, L&      Cased_Letter (that is: [\p{Ll}\p{Lu}\p{Lt}])
Lu          Uppercase_Letter
Ll          Lowercase_Letter
Lt          Titlecase_Letter
Lm          Modifier_Letter
Lo          Other_Letter

M           Mark
Mn          Nonspacing_Mark
Mc          Spacing_Mark
Me          Enclosing_Mark

N           Number
Nd          Decimal_Number (also Digit)
Nl          Letter_Number
No          Other_Number

P           Punctuation (also Punct)
Pc          Connector_Punctuation
Pd          Dash_Punctuation
Ps          Open_Punctuation
Pe          Close_Punctuation
Pi          Initial_Punctuation
```

```
                (may behave like Ps or Pe depending on usage)
Pf              Final_Punctuation
                (may behave like Ps or Pe depending on usage)
Po              Other_Punctuation

S               Symbol
Sm              Math_Symbol
Sc              Currency_Symbol
Sk              Modifier_Symbol
So              Other_Symbol

Z               Separator
Zs              Space_Separator
Zl              Line_Separator
Zp              Paragraph_Separator

C               Other
Cc              Control (also Cntrl)
Cf              Format
Cs              Surrogate
Co              Private_Use
Cn              Unassigned
```

Single-letter properties match all characters in any of the two-letter sub-properties starting with the same letter. `LC` and `L&` are special: both are aliases for the set consisting of everything matched by `Ll`, `Lu`, and `Lt`.

Bidirectional Character Types

Because scripts differ in their directionality (Hebrew and Arabic are written right to left, for example) Unicode supplies a `Bidi_Class` property. Some of the values this property can have are:

```
Value       Meaning

L           Left-to-Right
LRE         Left-to-Right Embedding
LRO         Left-to-Right Override
R           Right-to-Left
AL          Arabic Letter
RLE         Right-to-Left Embedding
RLO         Right-to-Left Override
PDF         Pop Directional Format
EN          European Number
ES          European Separator
ET          European Terminator
AN          Arabic Number
CS          Common Separator
NSM         Non-Spacing Mark
BN          Boundary Neutral
B           Paragraph Separator
S           Segment Separator
WS          Whitespace
ON          Other Neutrals
```

This property is always written in the compound form. For example, `\p{Bidi_Class:R}` matches

characters that are normally written right to left. Unlike the *General_Category* property, this property can have more values added in a future Unicode release. Those listed above comprised the complete set for many Unicode releases, but others were added in Unicode 6.3; you can always find what the current ones are in *perluniprops*. And *http://www.unicode.org/reports/tr9/* describes how to use them.

Scripts

The world's languages are written in many different scripts. This sentence (unless you're reading it in translation) is written in Latin, while Russian is written in Cyrillic, and Greek is written in, well, Greek; Japanese mainly in Hiragana or Katakana. There are many more.

The Unicode `Script` and `Script_Extensions` properties give what script a given character is in. Either property can be specified with the compound form like `\p{Script=Hebrew}` (short: `\p{sc=hebr}`), or `\p{Script_Extensions=Javanese}` (short: `\p{scx=java}`). In addition, Perl furnishes shortcuts for all `Script` property names. You can omit everything up through the equals (or colon), and simply write `\p{Latin}` or `\P{Cyrillic}`. (This is not true for `Script_Extensions`, which is required to be written in the compound form.)

The difference between these two properties involves characters that are used in multiple scripts. For example the digits '0' through '9' are used in many parts of the world. These are placed in a script named `Common`. Other characters are used in just a few scripts. For example, the `"KATAKANA-HIRAGANA DOUBLE HYPHEN"` is used in both Japanese scripts, Katakana and Hiragana, but nowhere else. The `Script` property places all characters that are used in multiple scripts in the `Common` script, while the `Script_Extensions` property places those that are used in only a few scripts into each of those scripts; while still using `Common` for those used in many scripts. Thus both these match:

```
"0" =~ /\p{sc=Common}/     # Matches
"0" =~ /\p{scx=Common}/    # Matches
```

and only the first of these match:

```
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{sc=Common}  # Matches
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{scx=Common} # No match
```

And only the last two of these match:

```
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{sc=Hiragana}  # No match
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{sc=Katakana}  # No match
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{scx=Hiragana} # Matches
"\N{KATAKANA-HIRAGANA DOUBLE HYPHEN}" =~ /\p{scx=Katakana} # Matches
```

`Script_Extensions` is thus an improved `Script`, in which there are fewer characters in the `Common` script, and correspondingly more in other scripts. It is new in Unicode version 6.0, and its data are likely to change significantly in later releases, as things get sorted out. New code should probably be using `Script_Extensions` and not plain `Script`.

(Actually, besides `Common`, the `Inherited` script, contains characters that are used in multiple scripts. These are modifier characters which inherit the script value of the controlling character. Some of these are used in many scripts, and so go into `Inherited` in both `Script` and `Script_Extensions`. Others are used in just a few scripts, so are in `Inherited` in `Script`, but not in `Script_Extensions`.)

It is worth stressing that there are several different sets of digits in Unicode that are equivalent to 0-9 and are matchable by `\d` in a regular expression. If they are used in a single language only, they are in that language's `Script` and `Script_Extension`. If they are used in more than one script, they will be in `sc=Common`, but only if they are used in many scripts should they be in `scx=Common`.

A complete list of scripts and their shortcuts is in *perluniprops*.

Use of the "Is" Prefix

For backward compatibility (with Perl 5.6), all properties writable without using the compound form mentioned so far may have `Is` or `Is_` prepended to their name, so `\P{Is_Lu}`, for example, is equal to `\P{Lu}`, and `\p{IsScript:Arabic}` is equal to `\p{Arabic}`.

Blocks

In addition to **scripts**, Unicode also defines **blocks** of characters. The difference between scripts and blocks is that the concept of scripts is closer to natural languages, while the concept of blocks is more of an artificial grouping based on groups of Unicode characters with consecutive ordinal values. For example, the `"Basic Latin"` block is all the characters whose ordinals are between 0 and 127, inclusive; in other words, the ASCII characters. The `"Latin"` script contains some letters from this as well as several other blocks, like `"Latin-1 Supplement"`, `"Latin Extended-A"`, *etc.*, but it does not contain all the characters from those blocks. It does not, for example, contain the digits 0-9, because those digits are shared across many scripts, and hence are in the `Common` script.

For more about scripts versus blocks, see UAX#24 "Unicode Script Property": *http://www.unicode.org/reports/tr24*

The `Script` or `Script_Extensions` properties are likely to be the ones you want to use when processing natural language; the `Block` property may occasionally be useful in working with the nuts and bolts of Unicode.

Block names are matched in the compound form, like `\p{Block: Arrows}` or `\p{Blk=Hebrew}`. Unlike most other properties, only a few block names have a Unicode-defined short name.

Perl also defines single form synonyms for the block property in cases where these do not conflict with something else. But don't use any of these, because they are unstable. Since these are Perl extensions, they are subordinate to official Unicode property names; Unicode doesn't know nor care about Perl's extensions. It may happen that a name that currently means the Perl extension will later be changed without warning to mean a different Unicode property in a future version of the perl interpreter that uses a later Unicode release, and your code would no longer work. The extensions are mentioned here for completeness: Take the block name and prefix it with one of: `In` (for example `\p{Blk=Arrows}` can currently be written as `\p{In_Arrows}`); or sometimes `Is` (like `\p{Is_Arrows}`); or sometimes no prefix at all (`\p{Arrows}`). As of this writing (Unicode 8.0) there are no conflicts with using the `In_` prefix, but there are plenty with the other two forms. For example, `\p{Is_Hebrew}` and `\p{Hebrew}` mean `\p{Script=Hebrew}` which is NOT the same thing as `\p{Blk=Hebrew}`. Our advice used to be to use the `In_` prefix as a single form way of specifying a block. But Unicode 8.0 added properties whose names begin with `In`, and it's now clear that it's only luck that's so far prevented a conflict. Using `In` is only marginally less typing than `Blk:`, and the latter's meaning is clearer anyway, and guaranteed to never conflict. So don't take chances. Use `\p{Blk=foo}` for new code. And be sure that block is what you really really want to do. In most cases scripts are what you want instead.

A complete list of blocks is in *perluniprops*.

Other Properties

There are many more properties than the very basic ones described here. A complete list is in *perluniprops*.

Unicode defines all its properties in the compound form, so all single-form properties are Perl extensions. Most of these are just synonyms for the Unicode ones, but some are genuine extensions, including several that are in the compound form. And quite a few of these are actually recommended by Unicode (in *http://www.unicode.org/reports/tr18*).

This section gives some details on all extensions that aren't just synonyms for compound-form Unicode properties (for those properties, you'll have to refer to the *Unicode Standard*.

\p{All}

This matches every possible code point. It is equivalent to `qr/./s`. Unlike all the other non-user-defined `\p{}` property matches, no warning is ever generated if this is property is matched against a non-Unicode code point (see *Beyond Unicode code points* below).

\p{Alnum}

This matches any `\p{Alphabetic}` or `\p{Decimal_Number}` character.

\p{Any}

This matches any of the 1_114_112 Unicode code points. It is a synonym for `\p{Unicode}`.

\p{ASCII}

This matches any of the 128 characters in the US-ASCII character set, which is a subset of Unicode.

\p{Assigned}

This matches any assigned code point; that is, any code point whose *general category* is not `Unassigned` (or equivalently, not `Cn`).

\p{Blank}

This is the same as `\h` and `\p{HorizSpace}`: A character that changes the spacing horizontally.

\p{Decomposition_Type: Non_Canonical} (Short: `\p{Dt=NonCanon}`)

Matches a character that has a non-canonical decomposition.

The *Extended Grapheme Clusters (Logical characters)* section above talked about canonical decompositions. However, many more characters have a different type of decomposition, a "compatible" or "non-canonical" decomposition. The sequences that form these decompositions are not considered canonically equivalent to the pre-composed character. An example is the `"SUPERSCRIPT ONE"`. It is somewhat like a regular digit 1, but not exactly; its decomposition into the digit 1 is called a "compatible" decomposition, specifically a "super" decomposition. There are several such compatibility decompositions (see *http://www.unicode.org/reports/tr44*), including one called "compat", which means some miscellaneous type of decomposition that doesn't fit into the other decomposition categories that Unicode has chosen.

Note that most Unicode characters don't have a decomposition, so their decomposition type is `"None"`.

For your convenience, Perl has added the `Non_Canonical` decomposition type to mean any of the several compatibility decompositions.

\p{Graph}

Matches any character that is graphic. Theoretically, this means a character that on a printer would cause ink to be used.

\p{HorizSpace}

This is the same as `\h` and `\p{Blank}`: a character that changes the spacing horizontally.

\p{In=*}

This is a synonym for `\p{Present_In=*}`

\p{PerlSpace}

This is the same as `\s`, restricted to ASCII, namely `[ \f\n\r\t]` and starting in Perl v5.18, a vertical tab.

Mnemonic: Perl's (original) space

\p{PerlWord}

This is the same as \w, restricted to ASCII, namely [A-Za-z0-9_]

Mnemonic: Perl's (original) word.

\p{Posix...}

There are several of these, which are equivalents, using the \p{} notation, for Posix classes and are described in *"POSIX Character Classes" in perlrecharclass*.

\p{Present_In: *} (Short: \p{In=*})

This property is used when you need to know in what Unicode version(s) a character is.

The "*" above stands for some two digit Unicode version number, such as 1.1 or 4.0; or the "*" can also be Unassigned. This property will match the code points whose final disposition has been settled as of the Unicode release given by the version number; \p{Present_In: Unassigned} will match those code points whose meaning has yet to be assigned.

For example, U+0041 "LATIN CAPITAL LETTER A" was present in the very first Unicode release available, which is 1.1, so this property is true for all valid "*" versions. On the other hand, U+1EFF was not assigned until version 5.1 when it became "LATIN SMALL LETTER Y WITH LOOP", so the only "*" that would match it are 5.1, 5.2, and later.

Unicode furnishes the Age property from which this is derived. The problem with Age is that a strict interpretation of it (which Perl takes) has it matching the precise release a code point's meaning is introduced in. Thus U+0041 would match only 1.1; and U+1EFF only 5.1. This is not usually what you want.

Some non-Perl implementations of the Age property may change its meaning to be the same as the Perl Present_In property; just be aware of that.

Another confusion with both these properties is that the definition is not that the code point has been *assigned*, but that the meaning of the code point has been *determined*. This is because 66 code points will always be unassigned, and so the Age for them is the Unicode version in which the decision to make them so was made. For example, U+FDD0 is to be permanently unassigned to a character, and the decision to do that was made in version 3.1, so \p{Age=3.1} matches this character, as also does \p{Present_In: 3.1} and up.

\p{Print}

This matches any character that is graphical or blank, except controls.

\p{SpacePerl}

This is the same as \s, including beyond ASCII.

Mnemonic: Space, as modified by Perl. (It doesn't include the vertical tab until v5.18, which both the Posix standard and Unicode consider white space.)

\p{Title} and **\p{Titlecase}**

Under case-sensitive matching, these both match the same code points as \p{General Category=Titlecase_Letter} (\p{gc=lt}). The difference is that under /i caseless matching, these match the same as \p{Cased}, whereas \p{gc=lt} matches \p{Cased_Letter}).

\p{Unicode}

This matches any of the 1_114_112 Unicode code points. \p{Any}.

\p{VertSpace}

This is the same as \v: A character that changes the spacing vertically.

\p{Word}

This is the same as \w, including over 100_000 characters beyond ASCII.

`\p{XPosix...}`

There are several of these, which are the standard Posix classes extended to the full Unicode range. They are described in *"POSIX Character Classes" in perlrecharclass*.

User-Defined Character Properties

You can define your own binary character properties by defining subroutines whose names begin with `"In"` or `"Is"`. (The experimental feature *"(?[ ])" in perlre* provides an alternative which allows more complex definitions.) The subroutines can be defined in any package. The user-defined properties can be used in the regular expression `\p{}` and `\P{}` constructs; if you are using a user-defined property from a package other than the one you are in, you must specify its package in the `\p{}` or `\P{}` construct.

```
# assuming property Is_Foreign defined in Lang::
package main;  # property package name required
if ($txt =~ /\p{Lang::IsForeign}+/) { ... }

package Lang;  # property package name not required
if ($txt =~ /\p{IsForeign}+/) { ... }
```

Note that the effect is compile-time and immutable once defined. However, the subroutines are passed a single parameter, which is 0 if case-sensitive matching is in effect and non-zero if caseless matching is in effect. The subroutine may return different values depending on the value of the flag, and one set of values will immutably be in effect for all case-sensitive matches, and the other set for all case-insensitive matches.

Note that if the regular expression is tainted, then Perl will die rather than calling the subroutine when the name of the subroutine is determined by the tainted data.

The subroutines must return a specially-formatted string, with one or more newline-separated lines. Each line must be one of the following:

- A single hexadecimal number denoting a code point to include.
- Two hexadecimal numbers separated by horizontal whitespace (space or tabular characters) denoting a range of code points to include.
- Something to include, prefixed by `"+"`: a built-in character property (prefixed by `"utf8::"`) or a fully qualified (including package name) user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to exclude, prefixed by `"-"`: an existing character property (prefixed by `"utf8::"`) or a fully qualified (including package name) user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to negate, prefixed `"!"`: an existing character property (prefixed by `"utf8::"`) or a fully qualified (including package name) user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to intersect with, prefixed by `"&"`: an existing character property (prefixed by `"utf8::"`) or a fully qualified (including package name) user-defined character property, for all the characters except the characters in the property; two hexadecimal code points for a range; or a single hexadecimal code point.

For example, to define a property that covers both the Japanese syllabaries (hiragana and katakana), you can define

```
sub InKana {
    return <<END;
3040\t309F
30A0\t30FF
END
}
```

Imagine that the here-doc end marker is at the beginning of the line. Now you can use `\p{InKana}` and `\P{InKana}`.

You could also have used the existing block property names:

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}
```

Suppose you wanted to match only the allocated characters, not the raw block ranges: in other words, you want to remove the unassigned characters:

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}
```

The negation is useful for defining (surprise!) negated classes.

```
sub InNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}
```

This will match all non-Unicode code points, since every one of them is not in Kana. You can use intersection to exclude these, if desired, as this modified example shows:

```
sub InNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
&utf8::Any
END
}
```

`&utf8::Any` must be the last line in the definition.

Intersection is used generally for getting the common characters matched by two (or more) classes. It's important to remember not to use `"&"` for the first set; that would be intersecting with nothing,

resulting in an empty set.

Unlike non-user-defined `\p{}` property matches, no warning is ever generated if these properties are matched against a non-Unicode code point (see *Beyond Unicode code points* below).

User-Defined Case Mappings (for serious hackers only)

This feature has been removed as of Perl 5.16. The CPAN module *`Unicode::Casing`* provides better functionality without the drawbacks that this feature had. If you are using a Perl earlier than 5.16, this feature was most fully documented in the 5.14 version of this pod: *http://perldoc.perl.org/5.14.0/perlunicode.html#User-Defined-Case-Mappings-%28for-serious-hackers-only%29*

Character Encodings for Input and Output

See *Encode*.

Unicode Regular Expression Support Level

The following list of Unicode supported features for regular expressions describes all features currently directly supported by core Perl. The references to "Level N" and the section numbers refer to the Unicode Technical Standard #18, "Unicode Regular Expressions", version 13, from August 2008.

- Level 1 - Basic Unicode Support

```
RL1.1   Hex Notation                    - done          [1]
RL1.2   Properties                      - done          [2][3]
RL1.2a  Compatibility Properties        - done          [4]
RL1.3   Subtraction and Intersection    - experimental  [5]
RL1.4   Simple Word Boundaries          - done          [6]
RL1.5   Simple Loose Matches            - done          [7]
RL1.6   Line Boundaries                 - MISSING       [8][9]
RL1.7   Supplementary Code Points       - done          [10]
```

[1] `\N{U+...}` and `\x{...}`

[2] `\p{...}` `\P{...}`

[3] supports not only minimal list, but all Unicode character properties (see Unicode Character Properties above)

[4] `\d` `\D` `\s` `\S` `\w` `\W` `\X` `[:`*`prop`*`:]` `[:^`*`prop`*`:]`

[5] The experimental feature starting in v5.18 `"(?[...])"` accomplishes this.

See *"(?[ ])" in perlre*. If you don't want to use an experimental feature, you can use one of the following:

- Regular expression lookahead

  You can mimic class subtraction using lookahead. For example, what UTS#18 might write as

  ```
  [{Block=Greek}-[{UNASSIGNED}]]
  ```

  in Perl can be written as:

  ```
  (?!\p{Unassigned})\p{Block=Greek}
  (?=\p{Assigned})\p{Block=Greek}
  ```

  But in this particular example, you probably really want

  ```
  \p{Greek}
  ```

  which will match assigned characters known to be part of the Greek script.

- CPAN module *`Unicode::Regex::Set`*

It does implement the full UTS#18 grouping, intersection, union, and removal (subtraction) syntax.

- *User-Defined Character Properties*

  "+" for union, "-" for removal (set-difference), "&" for intersection

[6] \b \B

[7] Note that Perl does Full case-folding in matching, not Simple:

For example U+1F88 is equivalent to U+1F00 U+03B9, instead of just U+1F80. This difference matters mainly for certain Greek capital letters with certain modifiers: the Full case-folding decomposes the letter, while the Simple case-folding would map it to a single character.

[8] Perl treats \n as the start- and end-line delimiter. Unicode specifies more characters that should be so-interpreted.

These are:

```
VT   U+000B  (\v in C)
FF   U+000C  (\f)
CR   U+000D  (\r)
NEL  U+0085
LS   U+2028
PS   U+2029
```

^ and $ in regular expression patterns are supposed to match all these, but don't. These characters also don't, but should, affect <> $., and script line numbers.

Also, lines should not be split within CRLF (i.e. there is no empty line between \r and \n). For CRLF, try the :crlf layer (see *PerlIO*).

[9] But qr/\b{lb}/ and *Unicode::LineBreak* are available.

qr/\b{lb}/ supplies default line breaking conformant with *UAX#14 "Unicode Line Breaking Algorithm"*.

And, the module *Unicode::LineBreak* also conformant with UAX#14, provides customizable line breaking.

[10] UTF-8/UTF-EBDDIC used in Perl allows not only U+10000 to U+10FFFF but also beyond U+10FFFF

- Level 2 - Extended Unicode Support

```
RL2.1   Canonical Equivalents           - MISSING       [10][11]
RL2.2   Default Grapheme Clusters       - MISSING       [12]
RL2.3   Default Word Boundaries         - DONE          [14]
RL2.4   Default Loose Matches           - MISSING       [15]
RL2.5   Name Properties                 - DONE
RL2.6   Wildcard Properties             - MISSING

[10] see UAX#15 "Unicode Normalization Forms"
[11] have Unicode::Normalize but not integrated to regexes
[12] have \X and \b{gcb} but we don't have a "Grapheme Cluster
     Mode"
[14] see UAX#29, Word Boundaries
[15] This is covered in Chapter 3.13 (in Unicode 6.0)
```

- Level 3 - Tailored Support

```
RL3.1   Tailored Punctuation            - MISSING
```

```
RL3.2   Tailored Grapheme Clusters      - MISSING       [17][18]
RL3.3   Tailored Word Boundaries        - MISSING
RL3.4   Tailored Loose Matches          - MISSING
RL3.5   Tailored Ranges                 - MISSING
RL3.6   Context Matching                - MISSING       [19]
RL3.7   Incremental Matches             - MISSING
     ( RL3.8   Unicode Set Sharing )
RL3.9   Possible Match Sets             - MISSING
RL3.10  Folded Matching                 - MISSING       [20]
RL3.11  Submatchers                     - MISSING

[17] see UAX#10 "Unicode Collation Algorithms"
[18] have Unicode::Collate but not integrated to regexes
[19] have (?<=x) and (?=x), but lookaheads or lookbehinds
     should see outside of the target substring
[20] need insensitive matching for linguistic features other
     than case; for example, hiragana to katakana, wide and
     narrow, simplified Han to traditional Han (see UTR#30
     "Character Foldings")
```

Unicode Encodings

Unicode characters are assigned to *code points*, which are abstract numbers. To use these numbers, various encodings are needed.

- UTF-8

  UTF-8 is a variable-length (1 to 4 bytes), byte-order independent encoding. In most of Perl's documentation, including elsewhere in this document, the term "UTF-8" means also "UTF-EBCDIC". But in this section, "UTF-8" refers only to the encoding used on ASCII platforms. It is a superset of 7-bit US-ASCII, so anything encoded in ASCII has the identical representation when encoded in UTF-8.

  The following table is from Unicode 3.2.

```
 Code Points            1st Byte  2nd Byte  3rd Byte 4th Byte

   U+0000..U+007F       00..7F
   U+0080..U+07FF     * C2..DF    80..BF
   U+0800..U+0FFF       E0      * A0..BF    80..BF
   U+1000..U+CFFF       E1..EC    80..BF    80..BF
   U+D000..U+D7FF       ED        80..9F    80..BF
   U+D800..U+DFFF       +++++ utf16 surrogates, not legal utf8 +++++
   U+E000..U+FFFF       EE..EF    80..BF    80..BF
  U+10000..U+3FFFF      F0      * 90..BF    80..BF    80..BF
  U+40000..U+FFFFF      F1..F3    80..BF    80..BF    80..BF
 U+100000..U+10FFFF     F4        80..8F    80..BF    80..BF
```

  Note the gaps marked by "*" before several of the byte entries above. These are caused by legal UTF-8 avoiding non-shortest encodings: it is technically possible to UTF-8-encode a single code point in different ways, but that is explicitly forbidden, and the shortest possible encoding should always be used (and that is what Perl does).

  Another way to look at it is via bits:

```
               Code Points  1st Byte  2nd Byte  3rd Byte  4th Byte

                  0aaaaaaa  0aaaaaaa
         00000bbbbbaaaaaa  110bbbbb  10aaaaaa
         ccccbbbbbbaaaaaa  1110cccc  10bbbbbb  10aaaaaa
```

```
00000dddccccccbbbbbbaaaaaa  11110ddd  10cccccc  10bbbbbb  10aaaaaa
```

As you can see, the continuation bytes all begin with "10", and the leading bits of the start byte tell how many bytes there are in the encoded character.

The original UTF-8 specification allowed up to 6 bytes, to allow encoding of numbers up to 0x7FFF_FFFF. Perl continues to allow those, and has extended that up to 13 bytes to encode code points up to what can fit in a 64-bit word. However, Perl will warn if you output any of these as being non-portable; and under strict UTF-8 input protocols, they are forbidden. In addition, it is deprecated to use a code point larger than what a signed integer variable on your system can hold. On 32-bit ASCII systems, this means 0x7FFF_FFFF is the legal maximum going forward (much higher on 64-bit systems).

- UTF-EBCDIC

  Like UTF-8, but EBCDIC-safe, in the way that UTF-8 is ASCII-safe. This means that all the basic characters (which includes all those that have ASCII equivalents (like "A", "0", "%", *etc.*) are the same in both EBCDIC and UTF-EBCDIC.)

  UTF-EBCDIC is used on EBCDIC platforms. It generally requires more bytes to represent a given code point than UTF-8 does; the largest Unicode code points take 5 bytes to represent (instead of 4 in UTF-8), and, extended for 64-bit words, it uses 14 bytes instead of 13 bytes in UTF-8.

- UTF-16, UTF-16BE, UTF-16LE, Surrogates, and BOM's (Byte Order Marks)

  The followings items are mostly for reference and general Unicode knowledge, Perl doesn't use these constructs internally.

  Like UTF-8, UTF-16 is a variable-width encoding, but where UTF-8 uses 8-bit code units, UTF-16 uses 16-bit code units. All code points occupy either 2 or 4 bytes in UTF-16: code points U+0000..U+FFFF are stored in a single 16-bit unit, and code points U+10000..U+10FFFF in two 16-bit units. The latter case is using *surrogates*, the first 16-bit unit being the *high surrogate*, and the second being the *low surrogate*.

  Surrogates are code points set aside to encode the U+10000..U+10FFFF range of Unicode code points in pairs of 16-bit units. The *high surrogates* are the range U+D800..U+DBFF and the *low surrogates* are the range U+DC00..U+DFFF. The surrogate encoding is

  ```
  $hi = ($uni - 0x10000) / 0x400 + 0xD800;
  $lo = ($uni - 0x10000) % 0x400 + 0xDC00;
  ```

  and the decoding is

  ```
  $uni = 0x10000 + ($hi - 0xD800) * 0x400 + ($lo - 0xDC00);
  ```

  Because of the 16-bitness, UTF-16 is byte-order dependent. UTF-16 itself can be used for in-memory computations, but if storage or transfer is required either UTF-16BE (big-endian) or UTF-16LE (little-endian) encodings must be chosen.

  This introduces another problem: what if you just know that your data is UTF-16, but you don't know which endianness? Byte Order Marks, or BOM's, are a solution to this. A special character has been reserved in Unicode to function as a byte order marker: the character with the code point U+FEFF is the BOM.

  The trick is that if you read a BOM, you will know the byte order, since if it was written on a big-endian platform, you will read the bytes 0xFE 0xFF, but if it was written on a little-endian platform, you will read the bytes 0xFF 0xFE. (And if the originating platform was writing in ASCII platform UTF-8, you will read the bytes 0xEF 0xBB 0xBF.)

  The way this trick works is that the character with the code point U+FFFE is not supposed to be in input streams, so the sequence of bytes 0xFF 0xFE is unambiguously "BOM, represented in little-endian format" and cannot be U+FFFE, represented in big-endian format".

  Surrogates have no meaning in Unicode outside their use in pairs to represent other code

points. However, Perl allows them to be represented individually internally, for example by saying `chr(0xD801)`, so that all code points, not just those valid for open interchange, are representable. Unicode does define semantics for them, such as their *General_Category* is `"Cs"`. But because their use is somewhat dangerous, Perl will warn (using the warning category `"surrogate"`, which is a sub-category of `"utf8"`) if an attempt is made to do things like take the lower case of one, or match case-insensitively, or to output them. (But don't try this on Perls before 5.14.)

- UTF-32, UTF-32BE, UTF-32LE

  The UTF-32 family is pretty much like the UTF-16 family, except that the units are 32-bit, and therefore the surrogate scheme is not needed. UTF-32 is a fixed-width encoding. The `BOM` signatures are `0x00 0x00 0xFE 0xFF` for BE and `0xFF 0xFE 0x00 0x00` for LE.

- UCS-2, UCS-4

  Legacy, fixed-width encodings defined by the ISO 10646 standard. UCS-2 is a 16-bit encoding. Unlike UTF-16, UCS-2 is not extensible beyond `U+FFFF`, because it does not use surrogates. UCS-4 is a 32-bit encoding, functionally identical to UTF-32 (the difference being that UCS-4 forbids neither surrogates nor code points larger than `0x10_FFFF`).

- UTF-7

  A seven-bit safe (non-eight-bit) encoding, which is useful if the transport or storage is not eight-bit safe. Defined by RFC 2152.

Noncharacter code points

66 code points are set aside in Unicode as "noncharacter code points". These all have the `Unassigned` (`Cn`) *General_Category*, and no character will ever be assigned to any of them. They are the 32 code points between `U+FDD0` and `U+FDEF` inclusive, and the 34 code points:

```
U+FFFE   U+FFFF
U+1FFFE  U+1FFFF
U+2FFFE  U+2FFFF
...
U+EFFFE  U+EFFFF
U+FFFFE  U+FFFFF
U+10FFFE U+10FFFF
```

Until Unicode 7.0, the noncharacters were "**forbidden** for use in open interchange of Unicode text data", so that code that processed those streams could use these code points as sentinels that could be mixed in with character data, and would always be distinguishable from that data. (Emphasis above and in the next paragraph are added in this document.)

Unicode 7.0 changed the wording so that they are "**not recommended** for use in open interchange of Unicode text data". The 7.0 Standard goes on to say:

> "If a noncharacter is received in open interchange, an application is not required to interpret it in any way. It is good practice, however, to recognize it as a noncharacter and to take appropriate action, such as replacing it with `U+FFFD` replacement character, to indicate the problem in the text. It is not recommended to simply delete noncharacter code points from such text, because of the potential security issues caused by deleting uninterpreted characters. (See conformance clause C7 in Section 3.2, Conformance Requirements, and *Unicode Technical Report #36, "Unicode Security Considerations"*)."

This change was made because it was found that various commercial tools like editors, or for things like source code control, had been written so that they would not handle program files that used these code points, effectively precluding their use almost entirely! And that was never the intent. They've always been meant to be usable within an application, or cooperating set of applications, at will.

If you're writing code, such as an editor, that is supposed to be able to handle any Unicode text data, then you shouldn't be using these code points yourself, and instead allow them in the input. If you need sentinels, they should instead be something that isn't legal Unicode. For UTF-8 data, you can use the bytes 0xC1 and 0xC2 as sentinels, as they never appear in well-formed UTF-8. (There are equivalents for UTF-EBCDIC). You can also store your Unicode code points in integer variables and use negative values as sentinels.

If you're not writing such a tool, then whether you accept noncharacters as input is up to you (though the Standard recommends that you not). If you do strict input stream checking with Perl, these code points continue to be forbidden. This is to maintain backward compatibility (otherwise potential security holes could open up, as an unsuspecting application that was written assuming the noncharacters would be filtered out before getting to it, could now, without warning, start getting them). To do strict checking, you can use the layer `:encoding('UTF-8')`.

Perl continues to warn (using the warning category `"nonchar"`, which is a sub-category of `"utf8"`) if an attempt is made to output noncharacters.

Beyond Unicode code points

The maximum Unicode code point is `U+10FFFF`, and Unicode only defines operations on code points up through that. But Perl works on code points up to the maximum permissible unsigned number available on the platform. However, Perl will not accept these from input streams unless lax rules are being used, and will warn (using the warning category `"non_unicode"`, which is a sub-category of `"utf8"`) if any are output.

Since Unicode rules are not defined on these code points, if a Unicode-defined operation is done on them, Perl uses what we believe are sensible rules, while generally warning, using the `"non_unicode"` category. For example, `uc("\x{11_0000}")` will generate such a warning, returning the input parameter as its result, since Perl defines the uppercase of every non-Unicode code point to be the code point itself. (All the case changing operations, not just uppercasing, work this way.)

The situation with matching Unicode properties in regular expressions, the `\p{}` and `\P{}` constructs, against these code points is not as clear cut, and how these are handled has changed as we've gained experience.

One possibility is to treat any match against these code points as undefined. But since Perl doesn't have the concept of a match being undefined, it converts this to failing or FALSE. This is almost, but not quite, what Perl did from v5.14 (when use of these code points became generally reliable) through v5.18. The difference is that Perl treated all `\p{}` matches as failing, but all `\P{}` matches as succeeding.

One problem with this is that it leads to unexpected, and confusting results in some cases:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}      # Failed on <= v5.18
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}     # Failed! on <= v5.18
```

That is, it treated both matches as undefined, and converted that to false (raising a warning on each). The first case is the expected result, but the second is likely counterintuitive: "How could both be false when they are complements?" Another problem was that the implementation optimized many Unicode property matches down to already existing simpler, faster operations, which don't raise the warning. We chose to not forgo those optimizations, which help the vast majority of matches, just to generate a warning for the unlikely event that an above-Unicode code point is being matched against.

As a result of these problems, starting in v5.20, what Perl does is to treat non-Unicode code points as just typical unassigned Unicode characters, and matches accordingly. (Note: Unicode has atypical unassigned code points. For example, it has noncharacter code points, and ones that, when they do get assigned, are destined to be written Right-to-left, as Arabic and Hebrew are. Perl assumes that no non-Unicode code point has any atypical properties.)

Perl, in most cases, will raise a warning when matching an above-Unicode code point against a Unicode property when the result is TRUE for `\p{}`, and FALSE for `\P{}`. For example:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}      # Fails, no warning
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}     # Succeeds, with warning
```

In both these examples, the character being matched is non-Unicode, so Unicode doesn't define how it should match. It clearly isn't an ASCII hex digit, so the first example clearly should fail, and so it does, with no warning. But it is arguable that the second example should have an undefined, hence FALSE, result. So a warning is raised for it.

Thus the warning is raised for many fewer cases than in earlier Perls, and only when what the result is could be arguable. It turns out that none of the optimizations made by Perl (or are ever likely to be made) cause the warning to be skipped, so it solves both problems of Perl's earlier approach. The most commonly used property that is affected by this change is `\p{Unassigned}` which is a short form for `\p{General_Category=Unassigned}`. Starting in v5.20, all non-Unicode code points are considered `Unassigned`. In earlier releases the matches failed because the result was considered undefined.

The only place where the warning is not raised when it might ought to have been is if optimizations cause the whole pattern match to not even be attempted. For example, Perl may figure out that for a string to match a certain regular expression pattern, the string has to contain the substring `"foobar"`. Before attempting the match, Perl may look for that substring, and if not found, immediately fail the match without actually trying it; so no warning gets generated even if the string contains an above-Unicode code point.

This behavior is more "Do what I mean" than in earlier Perls for most applications. But it catches fewer issues for code that needs to be strictly Unicode compliant. Therefore there is an additional mode of operation available to accommodate such code. This mode is enabled if a regular expression pattern is compiled within the lexical scope where the `"non_unicode"` warning class has been made fatal, say by:

```
use warnings FATAL => "non_unicode"
```

(see *warnings*). In this mode of operation, Perl will raise the warning for all matches against a non-Unicode code point (not just the arguable ones), and it skips the optimizations that might cause the warning to not be output. (It currently still won't warn if the match isn't even attempted, like in the `"foobar"` example above.)

In summary, Perl now normally treats non-Unicode code points as typical Unicode unassigned code points for regular expression matches, raising a warning only when it is arguable what the result should be. However, if this warning has been made fatal, it isn't skipped.

There is one exception to all this. `\p{All}` looks like a Unicode property, but it is a Perl extension that is defined to be true for all possible code points, Unicode or not, so no warning is ever generated when matching this against a non-Unicode code point. (Prior to v5.20, it was an exact synonym for `\p{Any}`, matching code points `0` through `0x10FFFF`.)

Security Implications of Unicode

First, read *Unicode Security Considerations*.

Also, note the following:

- Malformed UTF-8

  Unfortunately, the original specification of UTF-8 leaves some room for interpretation of how many bytes of encoded output one should generate from one input Unicode character. Strictly speaking, the shortest possible sequence of UTF-8 bytes should be generated, because otherwise there is potential for an input buffer overflow at the receiving end of a UTF-8

connection. Perl always generates the shortest length UTF-8, and with warnings on, Perl will warn about non-shortest length UTF-8 along with other malformations, such as the surrogates, which are not Unicode code points valid for interchange.

- Regular expression pattern matching may surprise you if you're not accustomed to Unicode. Starting in Perl 5.14, several pattern modifiers are available to control this, called the character set modifiers. Details are given in *"Character set modifiers" in perlre*.

As discussed elsewhere, Perl has one foot (two hooves?) planted in each of two worlds: the old world of ASCII and single-byte locales, and the new world of Unicode, upgrading when necessary. If your legacy code does not explicitly use Unicode, no automatic switch-over to Unicode should happen.

Unicode in Perl on EBCDIC

Unicode is supported on EBCDIC platforms. See *perlebcdic*.

Unless ASCII vs. EBCDIC issues are specifically being discussed, references to UTF-8 encoding in this document and elsewhere should be read as meaning UTF-EBCDIC on EBCDIC platforms. See *"Unicode and UTF" in perlebcdic*.

Because UTF-EBCDIC is so similar to UTF-8, the differences are mostly hidden from you; `use utf8` (and NOT something like `use utfebcdic`) declares the the script is in the platform's "native" 8-bit encoding of Unicode. (Similarly for the `":utf8"` layer.)

Locales

See *"Unicode and UTF-8" in perllocale*

When Unicode Does Not Happen

There are still many places where Unicode (in some encoding or another) could be given as arguments or received as results, or both in Perl, but it is not, in spite of Perl having extensive ways to input and output in Unicode, and a few other "entry points" like the `@ARGV` array (which can sometimes be interpreted as UTF-8).

The following are such interfaces. Also, see *The "Unicode Bug"*. For all of these interfaces Perl currently (as of v5.16.0) simply assumes byte strings both as arguments and results, or UTF-8 strings if the (deprecated) `encoding` pragma has been used.

One reason that Perl does not attempt to resolve the role of Unicode in these situations is that the answers are highly dependent on the operating system and the file system(s). For example, whether filenames can be in Unicode and in exactly what kind of encoding, is not exactly a portable concept. Similarly for `qx` and `system`: how well will the "command-line interface" (and which of them?) handle Unicode?

- `chdir`, `chmod`, `chown`, `chroot`, `exec`, `link`, `lstat`, `mkdir`, `rename`, `rmdir`, `stat`, `symlink`, `truncate`, `unlink`, `utime`, `-X`
- `%ENV`
- `glob` (aka the `<*>`)
- `open`, `opendir`, `sysopen`
- `qx` (aka the backtick operator), `system`
- `readdir`, `readlink`

The "Unicode Bug"

The term, "Unicode bug" has been applied to an inconsistency with the code points in the `Latin-1 Supplement` block, that is, between 128 and 255. Without a locale specified, unlike all other characters or code points, these characters can have very different semantics depending on the rules in effect. (Characters whose code points are above 255 force Unicode rules; whereas the rules for

ASCII characters are the same under both ASCII and Unicode rules.)

Under Unicode rules, these upper-Latin1 characters are interpreted as Unicode code points, which means they have the same semantics as Latin-1 (ISO-8859-1) and C1 controls.

As explained in *ASCII Rules versus Unicode Rules*, under ASCII rules, they are considered to be unassigned characters.

This can lead to unexpected results. For example, a string's semantics can suddenly change if a code point above 255 is appended to it, which changes the rules from ASCII to Unicode. As an example, consider the following program and its output:

```
$ perl -le'
    no feature 'unicode_strings';
    $s1 = "\xC2";
    $s2 = "\x{2660}";
    for ($s1, $s2, $s1.$s2) {
        print /\w/ || 0;
    }
'
0
0
1
```

If there's no `\w` in `s1` nor in `s2`, why does their concatenation have one?

This anomaly stems from Perl's attempt to not disturb older programs that didn't use Unicode, along with Perl's desire to add Unicode support seamlessly. But the result turned out to not be seamless. (By the way, you can choose to be warned when things like this happen. See *`encoding::warnings`*.)

`use feature 'unicode_strings'` was added, starting in Perl v5.12, to address this problem. It affects these things:

- Changing the case of a scalar, that is, using `uc()`, `ucfirst()`, `lc()`, and `lcfirst()`, or `\L`, `\U`, `\u` and `\l` in double-quotish contexts, such as regular expression substitutions.

  Under `unicode_strings` starting in Perl 5.12.0, Unicode rules are generally used. See *"lc" in perlfunc* for details on how this works in combination with various other pragmas.

- Using caseless (`/i`) regular expression matching.

  Starting in Perl 5.14.0, regular expressions compiled within the scope of `unicode_strings` use Unicode rules even when executed or compiled into larger regular expressions outside the scope.

- Matching any of several properties in regular expressions.

  These properties are `\b` (without braces), `\B` (without braces), `\s`, `\S`, `\w`, `\W`, and all the Posix character classes *except* `[[:ascii:]]`.

  Starting in Perl 5.14.0, regular expressions compiled within the scope of `unicode_strings` use Unicode rules even when executed or compiled into larger regular expressions outside the scope.

- In `quotemeta` or its inline equivalent `\Q`.

  Starting in Perl 5.16.0, consistent quoting rules are used within the scope of `unicode_strings`, as described in *"quotemeta" in perlfunc*. Prior to that, or outside its scope, no code points above 127 are quoted in UTF-8 encoded strings, but in byte encoded strings, code points between 128-255 are always quoted.

You can see from the above that the effect of `unicode_strings` increased over several Perl

releases. (And Perl's support for Unicode continues to improve; it's best to use the latest available release in order to get the most complete and accurate results possible.) Note that `unicode_strings` is automatically chosen if you `use 5.012` or higher.

For Perls earlier than those described above, or when a string is passed to a function outside the scope of `unicode_strings`, see the next section.

Forcing Unicode in Perl (Or Unforcing Unicode in Perl)

Sometimes (see *When Unicode Does Not Happen* or *The "Unicode Bug"*) there are situations where you simply need to force a byte string into UTF-8, or vice versa. The standard module *Encode* can be used for this, or the low-level calls *`utf8::upgrade($bytestring)`* and *`utf8::downgrade($utf8string[, FAIL_OK])`*.

Note that `utf8::downgrade()` can fail if the string contains characters that don't fit into a byte.

Calling either function on a string that already is in the desired state is a no-op.

ASCII Rules versus Unicode Rules gives all the ways that a string is made to use Unicode rules.

Using Unicode in XS

See *"Unicode Support" in perlguts* for an introduction to Unicode at the XS level, and *"Unicode Support" in perlapi* for the API details.

Hacking Perl to work on earlier Unicode versions (for very serious hackers only)

Perl by default comes with the latest supported Unicode version built-in, but the goal is to allow you to change to use any earlier one. In Perls v5.20 and v5.22, however, the earliest usable version is Unicode 5.1. Perl v5.18 is able to handle all earlier versions.

Download the files in the desired version of Unicode from the Unicode web site *http://www.unicode.org*). These should replace the existing files in *lib/unicore* in the Perl source tree. Follow the instructions in *README.perl* in that directory to change some of their names, and then build perl (see *INSTALL*).

Porting code from perl-5.6.X

Perls starting in 5.8 have a different Unicode model from 5.6. In 5.6 the programmer was required to use the `utf8` pragma to declare that a given scope expected to deal with Unicode data and had to make sure that only Unicode data were reaching that scope. If you have code that is working with 5.6, you will need some of the following adjustments to your code. The examples are written such that the code will continue to work under 5.6, so you should be safe to try them out.

- A filehandle that should read or write UTF-8

```
if ($] > 5.008) {
  binmode $fh, ":encoding(utf8)";
}
```

- A scalar that is going to be passed to some extension

  Be it `Compress::Zlib`, `Apache::Request` or any extension that has no mention of Unicode in the manpage, you need to make sure that the UTF8 flag is stripped off. Note that at the time of this writing (January 2012) the mentioned modules are not UTF-8-aware. Please check the documentation to verify if this is still true.

```
if ($] > 5.008) {
  require Encode;
  $val = Encode::encode_utf8($val); # make octets
}
```

- A scalar we got back from an extension

If you believe the scalar comes back as UTF-8, you will most likely want the UTF8 flag restored:

```
if ($] > 5.008) {
  require Encode;
  $val = Encode::decode_utf8($val);
}
```

- Same thing, if you are really sure it is UTF-8

```
if ($] > 5.008) {
  require Encode;
  Encode::_utf8_on($val);
}
```

- A wrapper for *DBI* `fetchrow_array` and `fetchrow_hashref`

  When the database contains only UTF-8, a wrapper function or method is a convenient way to replace all your `fetchrow_array` and `fetchrow_hashref` calls. A wrapper function will also make it easier to adapt to future enhancements in your database driver. Note that at the time of this writing (January 2012), the DBI has no standardized way to deal with UTF-8 data. Please check the *DBI documentation* to verify if that is still true.

```
sub fetchrow {
  # $what is one of fetchrow_{array,hashref}
  my($self, $sth, $what) = @_;
  if ($] < 5.008) {
    return $sth->$what;
  } else {
    require Encode;
    if (wantarray) {
      my @arr = $sth->$what;
      for (@arr) {
        defined && /[^\000-\177]/ && Encode::_utf8_on($_);
      }
      return @arr;
    } else {
      my $ret = $sth->$what;
      if (ref $ret) {
        for my $k (keys %$ret) {
          defined
          && /[^\000-\177]/
          && Encode::_utf8_on($_) for $ret->{$k};
        }
        return $ret;
      } else {
        defined && /[^\000-\177]/ && Encode::_utf8_on($_) for $ret;
        return $ret;
      }
    }
  }
}
```

- A large scalar that you know can only contain ASCII

  Scalars that contain only ASCII and are marked as UTF-8 are sometimes a drag to your program. If you recognize such a situation, just remove the UTF8 flag:

```
utf8::downgrade($val) if $] > 5.008;
```

BUGS

See also *The "Unicode Bug"* above.

Interaction with Extensions

When Perl exchanges data with an extension, the extension should be able to understand the UTF8 flag and act accordingly. If the extension doesn't recognize that flag, it's likely that the extension will return incorrectly-flagged data.

So if you're working with Unicode data, consult the documentation of every module you're using if there are any issues with Unicode data exchange. If the documentation does not talk about Unicode at all, suspect the worst and probably look at the source to learn how the module is implemented. Modules written completely in Perl shouldn't cause problems. Modules that directly or indirectly access code written in other programming languages are at risk.

For affected functions, the simple strategy to avoid data corruption is to always make the encoding of the exchanged data explicit. Choose an encoding that you know the extension can handle. Convert arguments passed to the extensions to that encoding and convert results back from that encoding. Write wrapper functions that do the conversions for you, so you can later change the functions when the extension catches up.

To provide an example, let's say the popular `Foo::Bar::escape_html` function doesn't deal with Unicode data yet. The wrapper function would convert the argument to raw UTF-8 and convert the result back to Perl's internal representation like so:

```
sub my_escape_html ($) {
    my($what) = shift;
    return unless defined $what;
    Encode::decode_utf8(Foo::Bar::escape_html(
                                    Encode::encode_utf8($what)));
}
```

Sometimes, when the extension does not convert data but just stores and retrieves it, you will be able to use the otherwise dangerous *`Encode::_utf8_on()`* function. Let's say the popular `Foo::Bar` extension, written in C, provides a `param` method that lets you store and retrieve data according to these prototypes:

```
$self->param($name, $value);            # set a scalar
$value = $self->param($name);           # retrieve a scalar
```

If it does not yet provide support for any encoding, one could write a derived class with such a `param` method:

```
sub param {
  my($self,$name,$value) = @_;
  utf8::upgrade($name);     # make sure it is UTF-8 encoded
  if (defined $value) {
    utf8::upgrade($value);  # make sure it is UTF-8 encoded
    return $self->SUPER::param($name,$value);
  } else {
    my $ret = $self->SUPER::param($name);
    Encode::_utf8_on($ret); # we know, it is UTF-8 encoded
    return $ret;
  }
}
```

Some extensions provide filters on data entry/exit points, such as `DB_File::filter_store_key` and family. Look out for such filters in the documentation of your extensions; they can make the

transition to Unicode data much easier.

Speed

Some functions are slower when working on UTF-8 encoded strings than on byte encoded strings. All functions that need to hop over characters such as `length()`, `substr()` or `index()`, or matching regular expressions can work **much** faster when the underlying data are byte-encoded.

In Perl 5.8.0 the slowness was often quite spectacular; in Perl 5.8.1 a caching scheme was introduced which improved the situation. In general, operations with UTF-8 encoded strings are still slower. As an example, the Unicode properties (character classes) like `\p{Nd}` are known to be quite a bit slower (5-20 times) than their simpler counterparts like `[0-9]` (then again, there are hundreds of Unicode characters matching `Nd` compared with the 10 ASCII characters matching `[0-9]`).

SEE ALSO

perlunitut, *perluniintro*, *perluniprops*, *Encode*, *open*, *utf8*, *bytes*, *perlretut*, *"${^UNICODE}" in perlvar*, *http://www.unicode.org/reports/tr44*).