NAME

perlunicook - cookbookish examples of handling Unicode in Perl

DESCRIPTION

This manpage contains short recipes demonstrating how to handle common Unicode operations in Perl, plus one complete program at the end. Any undeclared variables in individual recipes are assumed to have a previous appropriate value in them.

EXAMPLES

X 0: Standard preamble

Unless otherwise notes, all examples below require this standard preamble to work correctly, with the "#!" adjusted to work on your system:

```
#!/usr/bin/env perl

use utf8;      # so literals and identifiers can be in UTF-8
use v5.12;     # or later to get "unicode_strings" feature
use strict;    # quote strings, declare variables
use warnings;  # on by default
use warnings  qw(FATAL utf8);    # fatalize encoding glitches
use open      qw(:std :encoding(UTF-8)); # undeclared streams in UTF-8
use charnames qw(:full :short);  # unneeded in v5.16
```

This *does* make even Unix programmers "binmode" your binary streams, or open them with ":raw", but that’s the only way to get at them portably anyway.

WARNING: "use autodie" (pre 2.26) and "use open" do not get along with each other.

X 1: Generic Unicode-savvy filter

Always decompose on the way in, then recompose on the way out.

```
use Unicode::Normalize;

while (<>) {
   $_ = NFD($_);   # decompose + reorder canonically
   ...
} continue {
   print NFC($_);  # recompose (where possible) + reorder canonically
}
```

X 2: Fine-tuning Unicode warnings

As of v5.14, Perl distinguishes three subclasses of UTFX8 warnings.

```
use v5.14;              # subwarnings unavailable any earlier
no warnings "nonchar";  # the 66 forbidden non-characters
no warnings "surrogate";  # UTF-16/CESU-8 nonsense
no warnings "non_unicode";  # for codepoints over 0x10_FFFF
```

X 3: Declare source in utf8 for identifiers and literals

Without the all-critical "use utf8" declaration, putting UTFX8 in your literals and identifiers wonXt work right. If you used the standard preamble just given above, this already happened. If you did, you can do things like this:

```
use utf8;

my $measure   = "Aangstroem";
my @Xsoft     = qw( cp852 cp1251 cp1252 );
my @XXXXXXXXX = qw( XXXX  XXXXX );
my @X         = qw( koi8-f koi8-u koi8-r );
my $motto     = "X X X"; # FAMILY, GROWING HEART, DROMEDARY CAMEL
```

If you forget "use utf8", high bytes will be misunderstood as separate characters, and nothing will work right.

X 4: Characters and their numbers

The "ord" and "chr" functions work transparently on all codepoints, not just on ASCII alone X nor in fact, not even just on Unicode alone.

```
# ASCII characters
ord("A")
chr(65)

# characters from the Basic Multilingual Plane
ord("X")
chr(0x3A3)

# beyond the BMP
ord("X")            # MATHEMATICAL ITALIC SMALL N
chr(0x1D45B)
```

```
# beyond Unicode! (up to MAXINT)
ord("\x{20_0000}")
chr(0x20_0000)
```

X 5: Unicode literals by character number

In an interpolated literal, whether a double-quoted string or a regex, you may specify a character by its number using the "\x{*HHHHHH*}" escape.

```
String: "\x{3a3}"
Regex:  /\x{3a3}/

String: "\x{1d45b}"
Regex:  /\x{1d45b}/

# even non-BMP ranges in regex work fine
/[\x{1D434}-\x{1D467}]/
```

X 6: Get character name by number

```
use charnames ();
my $name = charnames::viacode(0x03A3);
```

X 7: Get character number by name

```
use charnames ();
my $number = charnames::vianame("GREEK CAPITAL LETTER SIGMA");
```

X 8: Unicode named characters

Use the "\N{*charname*}" notation to get the character by that name for use in interpolated literals (double-quoted strings and regexes). In v5.16, there is an implicit

```
use charnames qw(:full :short);
```

But prior to v5.16, you must be explicit about which set of charnames you want. The ":full" names are the official Unicode character name, alias, or sequence, which all share a namespace.

```
use charnames qw(:full :short latin greek);

"\N{MATHEMATICAL ITALIC SMALL N}"     # :full
"\N{GREEK CAPITAL LETTER SIGMA}"      # :full
```

Anything else is a Perl-specific convenience abbreviation. Specify one or more scripts by names if you

want short names that are script-specific.

```
"\N{Greek:Sigma}"                # :short
"\N{ae}"                       #  latin
"\N{epsilon}"                    #  greek
```

The v5.16 release also supports a ":loose" import for loose matching of character names, which works just like loose matching of property names: that is, it disregards case, whitespace, and underscores:

```
"\N{euro sign}"                  # :loose (from v5.16)
```

Starting in v5.32, you can also use

```
qr/\p{name=euro sign}/
```

to get official Unicode named characters in regular expressions. Loose matching is always done for these.

X 9: Unicode named sequences

These look just like character names but return multiple codepoints. Notice the %vx vector-print functionality in "printf".

```
use charnames qw(:full);
my $seq = "\N{LATIN CAPITAL LETTER A WITH MACRON AND GRAVE}";
printf "U+%v04X\n", $seq;
U+0100.0300
```

X 10: Custom named characters

Use ":alias" to give your own lexically scoped nicknames to existing characters, or even to give unnamed private-use characters useful names.

```
use charnames ":full", ":alias" => {
   ecute => "LATIN SMALL LETTER E WITH ACUTE",
   "APPLE LOGO" => 0xF8FF, # private use character
};

"\N{ecute}"
"\N{APPLE LOGO}"
```

X 11: Names of CJK codepoints

Sinograms like XXXX come back with character names of "CJK UNIFIED IDEOGRAPH-6771" and "CJK UNIFIED IDEOGRAPH-4EAC", because their XnamesX vary. The CPAN "Unicode::Unihan" module has a large database for decoding these (and a whole lot more), provided you know how to understand its output.

```
# cpan -i Unicode::Unihan
use Unicode::Unihan;
my $str = "XX";
my $unhan = Unicode::Unihan->new;
for my $lang (qw(Mandarin Cantonese Korean JapaneseOn JapaneseKun)) {
   printf "CJK $str in %-12s is ", $lang;
   say $unhan->$lang($str);
}
```

prints:

```
CJK XX in Mandarin     is DONG1JING1
CJK XX in Cantonese    is dung1ging1
CJK XX in Korean       is TONGKYENG
CJK XX in JapaneseOn   is TOUKYOU KEI KIN
CJK XX in JapaneseKun  is HIGASHI AZUMAMIYAKO
```

If you have a specific romanization scheme in mind, use the specific module:

```
# cpan -i Lingua::JA::Romanize::Japanese
use Lingua::JA::Romanize::Japanese;
my $k2r = Lingua::JA::Romanize::Japanese->new;
my $str = "XX";
say "Japanese for $str is ", $k2r->chars($str);
```

prints

```
Japanese for XX is toukyou
```

X 12: Explicit encode/decode

On rare occasion, such as a database read, you may be given encoded text you need to decode.

```
use Encode qw(encode decode);

my $chars = decode("shiftjis", $bytes, 1);
```

```
# OR
 my $bytes = encode("MIME-Header-ISO_2022_JP", $chars, 1);
```

For streams all in the same encoding, don't use encode/decode; instead set the file encoding when you open the file or immediately after with "binmode" as described later below.

X 13: Decode program arguments as utf8

```
    $ perl -CA ...
or
    $ export PERL_UNICODE=A
or
  use Encode qw(decode);
  @ARGV = map { decode('UTF-8', $_, 1) } @ARGV;
```

X 14: Decode program arguments as locale encoding

```
  # cpan -i Encode::Locale
  use Encode qw(locale);
  use Encode::Locale;

  # use "locale" as an arg to encode/decode
  @ARGV = map { decode(locale => $_, 1) } @ARGV;
```

X 15: Declare STD{IN,OUT,ERR} to be utf8

Use a command-line option, an environment variable, or else call "binmode" explicitly:

```
    $ perl -CS ...
or
    $ export PERL_UNICODE=S
or
    use open qw(:std :encoding(UTF-8));
or
    binmode(STDIN,  ":encoding(UTF-8)");
    binmode(STDOUT, ":utf8");
    binmode(STDERR, ":utf8");
```

X 16: Declare STD{IN,OUT,ERR} to be in locale encoding

```
  # cpan -i Encode::Locale
  use Encode;
  use Encode::Locale;
```

```
    # or as a stream for binmode or open
    binmode STDIN,  ":encoding(console_in)"  if -t STDIN;
    binmode STDOUT, ":encoding(console_out)" if -t STDOUT;
    binmode STDERR, ":encoding(console_out)" if -t STDERR;
```

X 17: Make file I/O default to utf8

Files opened without an encoding argument will be in UTF-8:

```
    $ perl -CD ...
or
    $ export PERL_UNICODE=D
or
    use open qw(:encoding(UTF-8));
```

X 18: Make all I/O and args default to utf8

```
    $ perl -CSDA ...
or
    $ export PERL_UNICODE=SDA
or
    use open qw(:std :encoding(UTF-8));
    use Encode qw(decode);
    @ARGV = map { decode('UTF-8', $_, 1) } @ARGV;
```

X 19: Open file with specific encoding

Specify stream encoding. This is the normal way to deal with encoded text, not by calling low-level functions.

```
# input file
    open(my $in_file, "< :encoding(UTF-16)", "wintext");
OR
    open(my $in_file, "<", "wintext");
    binmode($in_file, ":encoding(UTF-16)");
THEN
    my $line = <$in_file>;

# output file
    open($out_file, "> :encoding(cp1252)", "wintext");
OR
    open(my $out_file, ">", "wintext");
    binmode($out_file, ":encoding(cp1252)");
```

```
THEN
   print $out_file "some text\n";
```

More layers than just the encoding can be specified here. For example, the incantation ":raw :encoding(UTF-16LE) :crlf" includes implicit CRLF handling.

X 20: Unicode casing

Unicode casing is very different from ASCII casing.

```
uc("henry X")  # "HENRY X"
uc("tschuess")   # "TSCHUeSS"  notice ss => SS

# both are true:
"tschuess"  =~ /TSCHUeSS/i   # notice ss => SS
"XXXXXXX" =~ /XXXXXXX/i   # notice X,X,X sameness
```

X 21: Unicode case-insensitive comparisons

Also available in the CPAN Unicode::CaseFold module, the new "fc" XfoldcaseX function from v5.16 grants access to the same Unicode casefolding as the "/i" pattern modifier has always used:

```
use feature "fc"; # fc() function is from v5.16

# sort case-insensitively
my @sorted = sort { fc($a) cmp fc($b) } @list;

# both are true:
fc("tschuess")  eq fc("TSCHUeSS")
fc("XXXXXXX") eq fc("XXXXXXX")
```

X 22: Match Unicode linebreak sequence in regex

A Unicode linebreak matches the two-character CRLF grapheme or any of seven vertical whitespace characters. Good for dealing with textfiles coming from different operating systems.

```
\R

s/\R/\n/g;  # normalize all linebreaks to \n
```

X 23: Get character category

Find the general category of a numeric codepoint.

```
use Unicode::UCD qw(charinfo);
my $cat = charinfo(0x3A3)->{category};  # "Lu"
```

X 24: Disabling Unicode-awareness in builtin charclasses

Disable "\w", "\b", "\s", "\d", and the POSIX classes from working correctly on Unicode either in this scope, or in just one regex.

```
use v5.14;
use re "/a";

# OR

my($num) = $str =~ /(\d+)/a;
```

Or use specific un-Unicode properties, like "\p{ahex}" and "\p{POSIX_Digit"}. Properties still work normally no matter what charset modifiers ("/d /u /l /a /aa") should be effect.

X 25: Match Unicode properties in regex with \p, \P

These all match a single codepoint with the given property. Use "\P" in place of "\p" to match one codepoint lacking that property.

```
\pL, \pN, \pS, \pP, \pM, \pZ, \pC
\p{Sk}, \p{Ps}, \p{Lt}
\p{alpha}, \p{upper}, \p{lower}
\p{Latin}, \p{Greek}
\p{script_extensions=Latin}, \p{scx=Greek}
\p{East_Asian_Width=Wide}, \p{EA=W}
\p{Line_Break=Hyphen}, \p{LB=HY}
\p{Numeric_Value=4}, \p{NV=4}
```

X 26: Custom character properties

Define at compile-time your own custom character properties for use in regexes.

```
# using private-use characters
sub In_Tengwar { "E000\tE07F\n" }

if (/\p{In_Tengwar}/) { ... }

# blending existing properties
sub Is_GraecoRoman_Title {<<’END_OF_SET’}
```

```
+utf8::IsLatin
+utf8::IsGreek
&utf8::IsTitle
END_OF_SET

if (/\p{Is_GraecoRoman_Title}/ { ... }
```

X 27: Unicode normalization

Typically render into NFD on input and NFC on output. Using NFKC or NFKD functions improves recall on searches, assuming you've already done to the same text to be searched. Note that this is about much more than just pre- combined compatibility glyphs; it also reorders marks according to their canonical combining classes and weeds out singletons.

```
use Unicode::Normalize;
my $nfd  = NFD($orig);
my $nfc  = NFC($orig);
my $nfkd = NFKD($orig);
my $nfkc = NFKC($orig);
```

X 28: Convert non-ASCII Unicode numerics

Unless youXve used "/a" or "/aa", "\d" matches more than ASCII digits only, but PerlXs implicit string-to-number conversion does not current recognize these. HereXs how to convert such strings manually.

```
use v5.14;  # needed for num() function
use Unicode::UCD qw(num);
my $str = "got X and XXXX and X and here";
my @nums = ();
while ($str =~ /(\d+|\N)/g) {  # not just ASCII!
  push @nums, num($1);
}
say "@nums";   #     12      4567      0.875

use charnames qw(:full);
my $nv = num("\N{RUMI DIGIT ONE}\N{RUMI DIGIT TWO}");
```

X 29: Match Unicode grapheme cluster in regex

Programmer-visible XcharactersX are codepoints matched by "/./s", but user-visible XcharactersX are graphemes matched by "/\X/".

```
# Find vowel *plus* any combining diacritics,underlining,etc.
my $nfd = NFD($orig);
$nfd =~ / (?=[aeiou]) \X /xi
```

X 30: Extract by grapheme instead of by codepoint (regex)

```
# match and grab five first graphemes
my($first_five) = $str =~ /^ ( \X{5} ) /x;
```

X 31: Extract by grapheme instead of by codepoint (substr)

```
# cpan -i Unicode::GCString
use Unicode::GCString;
my $gcs = Unicode::GCString->new($str);
my $first_five = $gcs->substr(0, 5);
```

X 32: Reverse string by grapheme

Reversing by codepoint messes up diacritics, mistakenly converting "creme brulee" into "eelXurb emXerc" instead of into "eelurb emerc"; so reverse by grapheme instead. Both these approaches work right no matter what normalization the string is in:

```
$str = join("", reverse $str =~ /\X/g);

# OR: cpan -i Unicode::GCString
use Unicode::GCString;
$str = reverse Unicode::GCString->new($str);
```

X 33: String length in graphemes

The string "brulee" has six graphemes but up to eight codepoints. This counts by grapheme, not by codepoint:

```
my $str = "brulee";
my $count = 0;
while ($str =~ /\X/g) { $count++ }

 # OR: cpan -i Unicode::GCString
use Unicode::GCString;
my $gcs = Unicode::GCString->new($str);
my $count = $gcs->length;
```

X 34: Unicode column-width for printing

PerlXs "printf", "sprintf", and "format" think all codepoints take up 1 print column, but many take 0 or

2. Here to show that normalization makes no difference, we print out both forms:

```
use Unicode::GCString;
use Unicode::Normalize;

my @words = qw/creme brulee/;
@words = map { NFC($_), NFD($_) } @words;

for my $str (@words) {
   my $gcs = Unicode::GCString->new($str);
   my $cols = $gcs->columns;
   my $pad = " " x (10 - $cols);
   say str, $pad, " |";
}
```

generates this to show that it pads correctly no matter the normalization:

```
creme     |
creXme    |
brulee    |
bruXleXe  |
```

X 35: Unicode collation

Text sorted by numeric codepoint follows no reasonable alphabetic order; use the UCA for sorting text.

```
use Unicode::Collate;
my $col = Unicode::Collate->new();
my @list = $col->sort(@old_list);
```

See the *ucsort* program from the Unicode::Tussle CPAN module for a convenient command-line interface to this module.

X 36: Case- ***and*** **accent-insensitive Unicode sort**

Specify a collation strength of level 1 to ignore case and diacritics, only looking at the basic character.

```
use Unicode::Collate;
my $col = Unicode::Collate->new(level => 1);
my @list = $col->sort(@old_list);
```

X 37: Unicode locale collation

Some locales have special sorting rules.

```
# either use v5.12, OR: cpan -i Unicode::Collate::Locale
use Unicode::Collate::Locale;
my $col = Unicode::Collate::Locale->new(locale => "de__phonebook");
my @list = $col->sort(@old_list);
```

The *ucsort* program mentioned above accepts a "--locale" parameter.

X 38: Making "cmp" work on text instead of codepoints

Instead of this:

```
@srecs = sort {
   $b->{AGE}   <=>  $a->{AGE}
            ||
   $a->{NAME}  cmp  $b->{NAME}
} @recs;
```

Use this:

```
my $coll = Unicode::Collate->new();
for my $rec (@recs) {
   $rec->{NAME_key} = $coll->getSortKey( $rec->{NAME} );
}
@srecs = sort {
   $b->{AGE}      <=>  $a->{AGE}
               ||
   $a->{NAME_key}  cmp  $b->{NAME_key}
} @recs;
```

X 39: Case- *and* accent-insensitive comparisons

Use a collator object to compare Unicode text by character instead of by codepoint.

```
use Unicode::Collate;
my $es = Unicode::Collate->new(
   level => 1,
   normalization => undef
);

 # now both are true:
```

```
$es->eq("Garcia",  "GARCIA" );
$es->eq("Marquez", "MARQUEZ");
```

X 40: Case- *and* **accent-insensitive locale comparisons**

Same, but in a specific locale.

```
my $de = Unicode::Collate::Locale->new(
         locale => "de__phonebook",
        );

# now this is true:
$de->eq("tschuess", "TSCHUESS");  # notice ue => UE, ss => SS
```

X 41: Unicode linebreaking

Break up text into lines according to Unicode rules.

```
# cpan -i Unicode::LineBreak
use Unicode::LineBreak;
use charnames qw(:full);

my $para = "This is a super\N{HYPHEN}long string. " x 20;
my $fmt = Unicode::LineBreak->new;
print $fmt->break($para), "\n";
```

X 42: Unicode text in DBM hashes, the tedious way

Using a regular Perl string as a key or value for a DBM hash will trigger a wide character exception if any codepoints wonXt fit into a byte. HereXs how to manually manage the translation:

```
  use DB_File;
  use Encode qw(encode decode);
  tie %dbhash, "DB_File", "pathname";

# STORE

  # assume $uni_key and $uni_value are abstract Unicode strings
  my $enc_key   = encode("UTF-8", $uni_key, 1);
  my $enc_value = encode("UTF-8", $uni_value, 1);
  $dbhash{$enc_key} = $enc_value;

# FETCH
```

```
# assume $uni_key holds a normal Perl string (abstract Unicode)
my $enc_key   = encode("UTF-8", $uni_key, 1);
my $enc_value = $dbhash{$enc_key};
my $uni_value = decode("UTF-8", $enc_value, 1);
```

X 43: Unicode text in DBM hashes, the easy way

HereXs how to implicitly manage the translation; all encoding and decoding is done automatically, just as with streams that have a particular encoding attached to them:

```
use DB_File;
use DBM_Filter;

my $dbobj = tie %dbhash, "DB_File", "pathname";
$dbobj->Filter_Value("utf8");  # this is the magic bit

# STORE

# assume $uni_key and $uni_value are abstract Unicode strings
$dbhash{$uni_key} = $uni_value;

# FETCH

# $uni_key holds a normal Perl string (abstract Unicode)
my $uni_value = $dbhash{$uni_key};
```

X 44: PROGRAM: Demo of Unicode collation and printing

HereXs a full program showing how to make use of locale-sensitive sorting, Unicode casing, and managing print widths when some of the characters take up zero or two columns, not just one column each time. When run, the following program produces this nicely aligned output:

```
Creme Brulee..... X2.00
Eclair..... X1.60
Fideua..... X4.20
Hamburger..... X6.00
Jamon Serrano..... X4.45
Linguica..... X7.00
Pate..... X4.15
Pears..... X2.00
Peches..... X2.25
Smorbrod..... X5.75
```

```
    Spaetzle..... X5.50
    Xorico..... X3.00
    XXXXX..... X6.50
    XXX..... X4.00
    XXX..... X2.65
    XXXXX..... X8.00
    XXXXXXX..... X1.85
    XX..... X9.99
    XX..... X7.50
```

Here's that program; tested on v5.14.

```
#!/usr/bin/env perl
# umenu - demo sorting and printing of Unicode food
#
# (obligatory and increasingly long preamble)
#
use utf8;
use v5.14;                  # for locale sorting
use strict;
use warnings;
use warnings  qw(FATAL utf8);    # fatalize encoding faults
use open      qw(:std :encoding(UTF-8)); # undeclared streams in UTF-8
use charnames qw(:full :short);  # unneeded in v5.16

# std modules
use Unicode::Normalize;         # std perl distro as of v5.8
use List::Util qw(max);         # std perl distro as of v5.10
use Unicode::Collate::Locale;    # std perl distro as of v5.14

# cpan modules
use Unicode::GCString;          # from CPAN

# forward defs
sub pad($$$);
sub colwidth(_);
sub entitle(_);

my %price = (
    "XXXXX"           => 6.50, # gyros
```

```
    "pears"            => 2.00, # like um, pears
    "linguica"         => 7.00, # spicy sausage, Portuguese
    "xorico"           => 3.00, # chorizo sausage, Catalan
    "hamburger"        => 6.00, # burgermeister meisterburger
    "eclair"           => 1.60, # dessert, French
    "smorbrod"         => 5.75, # sandwiches, Norwegian
    "spaetzle"         => 5.50, # Bayerisch noodles, little sparrows
    "XX"               => 7.50, # bao1 zi5, steamed pork buns, Mandarin
    "jamon serrano"    => 4.45, # country ham, Spanish
    "peches"           => 2.25, # peaches, French
    "XXXXXXX"          => 1.85, # cream-filled pastry like eclair
    "XXX"              => 4.00, # makgeolli, Korean rice wine
    "XX"               => 9.99, # sushi, Japanese
    "XXX"              => 2.65, # omochi, rice cakes, Japanese
    "creme brulee"     => 2.00, # crema catalana
    "fideua"           => 4.20, # more noodles, Valencian
                           # (Catalan=fideuada)
    "pate"             => 4.15, # gooseliver paste, French
    "XXXXX"            => 8.00, # okonomiyaki, Japanese
);

my $width = 5 + max map { colwidth } keys %price;

# So the Asian stuff comes out in an order that someone
# who reads those scripts won't freak out over; the
# CJK stuff will be in JIS X 0208 order that way.
my $coll  = Unicode::Collate::Locale->new(locale => "ja");

for my $item ($coll->sort(keys %price)) {
    print pad(entitle($item), $width, ".");
    printf " X%.2f\n", $price{$item};
}

sub pad($$$) {
    my($str, $width, $padchar) = @_;
    return $str . ($padchar x ($width - colwidth($str)));
}

sub colwidth(_) {
    my($str) = @_;
```

```
    return Unicode::GCString->new($str)->columns;
}

sub entitle(_) {
    my($str) = @_;
    $str =~ s{ (?=\pL)(\S)     (\S*) }
            { ucfirst($1) . lc($2)  }xge;
    return $str;
}
```

SEE ALSO

See these manpages, some of which are CPAN modules: perlunicode, perluniprops, perlre, perlrecharclass, perluniintro, perlunitut, perlunifaq, PerlIO, DB_File, DBM_Filter, DBM_Filter::utf8, Encode, Encode::Locale, Unicode::UCD, Unicode::Normalize, Unicode::GCString, Unicode::LineBreak, Unicode::Collate, Unicode::Collate::Locale, Unicode::Unihan, Unicode::CaseFold, Unicode::Tussle, Lingua::JA::Romanize::Japanese, Lingua::ZH::Romanize::Pinyin, Lingua::KO::Romanize::Hangul.

The Unicode::Tussle CPAN module includes many programs to help with working with Unicode, including these programs to fully or partly replace standard utilities: *tcgrep* instead of *egrep*, *uniquote* instead of *cat -v* or *hexdump*, *uniwc* instead of *wc*, *unilook* instead of *look*, *unifmt* instead of *fmt*, and *ucsort* instead of *sort*. For exploring Unicode character names and character properties, see its *uniprops*, *unichars*, and *uninames* programs. It also supplies these programs, all of which are general filters that do Unicode-y things: *unititle* and *unicaps*; *uniwide* and *uninarrow*; *unisupers* and *unisubs*; *nfd*, *nfc*, *nfkd*, and *nfkc*; and *uc*, *lc*, and *tc*.

Finally, see the published Unicode Standard (page numbers are from version 6.0.0), including these specific annexes and technical reports:

X3.13 Default Case Algorithms, page 113; X4.2 Case, pages 120X122; Case Mappings, page 166X172, especially Caseless Matching starting on page 170.
UAX #44: Unicode Character Database
UTS #18: Unicode Regular Expressions
UAX #15: Unicode Normalization Forms
UTS #10: Unicode Collation Algorithm
UAX #29: Unicode Text Segmentation
UAX #14: Unicode Line Breaking Algorithm
UAX #11: East Asian Width

AUTHOR

Tom Christiansen <tchrist@perl.com> wrote this, with occasional kibbitzing from Larry Wall and Jeffrey Friedl in the background.

COPYRIGHT AND LICENCE

REVISION HISTORY

v1.0.0 X first public release, 2012-02-27