NAME

perlunifaq - Perl Unicode FAQ

Q and A

This is a list of questions and answers about Unicode in Perl, intended to be read after *perlunitut*.

perlunitut isn't really a Unicode tutorial, is it?

No, and this isn't really a Unicode FAQ.

Perl has an abstracted interface for all supported character encodings, so this is actually a generic `Encode` tutorial and `Encode` FAQ. But many people think that Unicode is special and magical, and I didn't want to disappoint them, so I decided to call the document a Unicode tutorial.

What character encodings does Perl support?

To find out which character encodings your Perl supports, run:

```
perl -MEncode -le "print for Encode->encodings(':all')"
```

Which version of perl should I use?

Well, if you can, upgrade to the most recent, but certainly `5.8.1` or newer. The tutorial and FAQ assume the latest release.

You should also check your modules, and upgrade them if necessary. For example, HTML::Entities requires version >= 1.32 to function correctly, even though the changelog is silent about this.

What about binary data, like images?

Well, apart from a bare `binmode $fh`, you shouldn't treat them specially. (The binmode is needed because otherwise Perl may convert line endings on Win32 systems.)

Be careful, though, to never combine text strings with binary strings. If you need text in a binary stream, encode your text strings first using the appropriate encoding, then join them with binary strings. See also: "What if I don't encode?".

When should I decode or encode?

Whenever you're communicating text with anything that is external to your perl process, like a database, a text file, a socket, or another program. Even if the thing you're communicating with is also written in Perl.

What if I don't decode?

Whenever your encoded, binary string is used together with a text string, Perl will assume that your binary string was encoded with ISO-8859-1, also known as latin-1. If it wasn't latin-1, then your data is unpleasantly converted. For example, if it was UTF-8, the individual bytes of multibyte characters are seen as separate characters, and then again converted to UTF-8. Such double encoding can be compared to double HTML encoding (`&gt;`), or double URI encoding (`%253E`).

This silent implicit decoding is known as "upgrading". That may sound positive, but it's best to avoid it.

What if I don't encode?

Your text string will be sent using the bytes in Perl's internal format. In some cases, Perl will warn you that you're doing something wrong, with a friendly warning:

```
Wide character in print at example.pl line 2.
```

Because the internal format is often UTF-8, these bugs are hard to spot, because UTF-8 is usually the encoding you wanted! But don't be lazy, and don't use the fact that Perl's internal format is UTF-8 to your advantage. Encode explicitly to avoid weird bugs, and to show to maintenance programmers that you thought this through.

Is there a way to automatically decode or encode?

If all data that comes from a certain handle is encoded in exactly the same way, you can tell the PerlIO system to automatically decode everything, with the `encoding` layer. If you do this, you can't accidentally forget to decode or encode anymore, on things that use the layered handle.

You can provide this layer when opening the file:

```
open my $fh, '>:encoding(UTF-8)', $filename;  # auto encoding on write
open my $fh, '<:encoding(UTF-8)', $filename;  # auto decoding on read
```

Or if you already have an open filehandle:

```
binmode $fh, ':encoding(UTF-8)';
```

Some database drivers for DBI can also automatically encode and decode, but that is sometimes limited to the UTF-8 encoding.

What if I don't know which encoding was used?

Do whatever you can to find out, and if you have to: guess. (Don't forget to document your guess with a comment.)

You could open the document in a web browser, and change the character set or character encoding until you can visually confirm that all characters look the way they should.

There is no way to reliably detect the encoding automatically, so if people keep sending you data without charset indication, you may have to educate them.

Can I use Unicode in my Perl sources?

Yes, you can! If your sources are UTF-8 encoded, you can indicate that with the `use utf8` pragma.

```
use utf8;
```

This doesn't do anything to your input, or to your output. It only influences the way your sources are read. You can use Unicode in string literals, in identifiers (but they still have to be "word characters" according to `\w`), and even in custom delimiters.

Data::Dumper doesn't restore the UTF8 flag; is it broken?

No, Data::Dumper's Unicode abilities are as they should be. There have been some complaints that it should restore the UTF8 flag when the data is read again with `eval`. However, you should really not look at the flag, and nothing indicates that Data::Dumper should break this rule.

Here's what happens: when Perl reads in a string literal, it sticks to 8 bit encoding as long as it can. (But perhaps originally it was internally encoded as UTF-8, when you dumped it.) When it has to give that up because other characters are added to the text string, it silently upgrades the string to UTF-8.

If you properly encode your strings for output, none of this is of your concern, and you can just `eval` dumped data as always.

Why do regex character classes sometimes match only in the ASCII range?

Why do some characters not uppercase or lowercase correctly?

It seemed like a good idea at the time, to keep the semantics the same for standard strings, when Perl got Unicode support. The plan is to fix this in the future, and the casing component has in fact mostly been fixed, but we have to deal with the fact that Perl treats equal strings differently, depending on the internal state.

First the casing. Just put a `use feature 'unicode_strings'` near the beginning of your program. Within its lexical scope, `uc`, `lc`, `ucfirst`, `lcfirst`, and the regular expression escapes

`\U`, `\L`, `\u`, `\l` use Unicode semantics for changing case regardless of whether the UTF8 flag is on or not. However, if you pass strings to subroutines in modules outside the pragma's scope, they currently likely won't behave this way, and you have to try one of the solutions below. There is another exception as well: if you have furnished your own casing functions to override the default, these will not be called unless the UTF8 flag is on)

This remains a problem for the regular expression constructs `\d`, `\s`, `\w`, `\D`, `\S`, `\W`, `/.../i`, `(?i:...)`, and `/[[:posix:]]/`.

To force Unicode semantics, you can upgrade the internal representation to by doing `utf8::upgrade($string)`. This can be used safely on any string, as it checks and does not change strings that have already been upgraded.

For a more detailed discussion, see *Unicode::Semantics* on CPAN.

How can I determine if a string is a text string or a binary string?

You can't. Some use the UTF8 flag for this, but that's misuse, and makes well behaved modules like Data::Dumper look bad. The flag is useless for this purpose, because it's off when an 8 bit encoding (by default ISO-8859-1) is used to store the string.

This is something you, the programmer, has to keep track of; sorry. You could consider adopting a kind of "Hungarian notation" to help with this.

How do I convert from encoding FOO to encoding BAR?

By first converting the FOO-encoded byte string to a text string, and then the text string to a BAR-encoded byte string:

```
my $text_string = decode('FOO', $foo_string);
my $bar_string  = encode('BAR', $text_string);
```

or by skipping the text string part, and going directly from one binary encoding to the other:

```
use Encode qw(from_to);
from_to($string, 'FOO', 'BAR');  # changes contents of $string
```

or by letting automatic decoding and encoding do all the work:

```
open my $foofh, '<:encoding(FOO)', 'example.foo.txt';
open my $barfh, '>:encoding(BAR)', 'example.bar.txt';
print { $barfh } $_ while <$foofh>;
```

What are decode_utf8 and encode_utf8?

These are alternate syntaxes for `decode('utf8', ...)` and `encode('utf8', ...)`.

What is a "wide character"?

This is a term used both for characters with an ordinal value greater than 127, characters with an ordinal value greater than 255, or any character occupying more than one byte, depending on the context.

The Perl warning "Wide character in ..." is caused by a character with an ordinal value greater than 255. With no specified encoding layer, Perl tries to fit things in ISO-8859-1 for backward compatibility reasons. When it can't, it emits this warning (if warnings are enabled), and outputs UTF-8 encoded data instead.

To avoid this warning and to avoid having different output encodings in a single stream, always specify an encoding explicitly, for example with a PerlIO layer:

```
binmode STDOUT, ":encoding(UTF-8)";
```

INTERNALS

What is "the UTF8 flag"?

Please, unless you're hacking the internals, or debugging weirdness, don't think about the UTF8 flag at all. That means that you very probably shouldn't use `is_utf8`, `_utf8_on` or `_utf8_off` at all.

The UTF8 flag, also called SvUTF8, is an internal flag that indicates that the current internal representation is UTF-8. Without the flag, it is assumed to be ISO-8859-1. Perl converts between these automatically. (Actually Perl usually assumes the representation is ASCII; see *Why do regex character classes sometimes match only in the ASCII range?* above.)

One of Perl's internal formats happens to be UTF-8. Unfortunately, Perl can't keep a secret, so everyone knows about this. That is the source of much confusion. It's better to pretend that the internal format is some unknown encoding, and that you always have to encode and decode explicitly.

What about the use bytes pragma?

Don't use it. It makes no sense to deal with bytes in a text string, and it makes no sense to deal with characters in a byte string. Do the proper conversions (by decoding/encoding), and things will work out well: you get character counts for decoded data, and byte counts for encoded data.

`use bytes` is usually a failed attempt to do something useful. Just forget about it.

What about the use encoding pragma?

Don't use it. Unfortunately, it assumes that the programmer's environment and that of the user will use the same encoding. It will use the same encoding for the source code and for STDIN and STDOUT. When a program is copied to another machine, the source code does not change, but the STDIO environment might.

If you need non-ASCII characters in your source code, make it a UTF-8 encoded file and `use utf8`.

If you need to set the encoding for STDIN, STDOUT, and STDERR, for example based on the user's locale, `use open`.

What is the difference between :encoding and :utf8?

Because UTF-8 is one of Perl's internal formats, you can often just skip the encoding or decoding step, and manipulate the UTF8 flag directly.

Instead of `:encoding(UTF-8)`, you can simply use `:utf8`, which skips the encoding step if the data was already represented as UTF8 internally. This is widely accepted as good behavior when you're writing, but it can be dangerous when reading, because it causes internal inconsistency when you have invalid byte sequences. Using `:utf8` for input can sometimes result in security breaches, so please use `:encoding(UTF-8)` instead.

Instead of `decode` and `encode`, you could use `_utf8_on` and `_utf8_off`, but this is considered bad style. Especially `_utf8_on` can be dangerous, for the same reason that `:utf8` can.

There are some shortcuts for oneliners; see `-C` in *perlrun*.

What's the difference between UTF-8 and utf8?

`UTF-8` is the official standard. `utf8` is Perl's way of being liberal in what it accepts. If you have to communicate with things that aren't so liberal, you may want to consider using `UTF-8`. If you have to communicate with things that are too liberal, you may have to use `utf8`. The full explanation is in *Encode*.

`UTF-8` is internally known as `utf-8-strict`. The tutorial uses UTF-8 consistently, even where utf8 is actually used internally, because the distinction can be hard to make, and is mostly irrelevant.

For example, utf8 can be used for code points that don't exist in Unicode, like 9999999, but if you encode that to UTF-8, you get a substitution character (by default; see *"Handling Malformed Data" in Encode* for more ways of dealing with this.)

Okay, if you insist: the "internal format" is utf8, not UTF-8. (When it's not some other encoding.)

I lost track; what encoding is the internal format really?

It's good that you lost track, because you shouldn't depend on the internal format being any specific encoding. But since you asked: by default, the internal format is either ISO-8859-1 (latin-1), or utf8, depending on the history of the string. On EBCDIC platforms, this may be different even.

Perl knows how it stored the string internally, and will use that knowledge when you `encode`. In other words: don't try to find out what the internal encoding for a certain string is, but instead just encode it into the encoding that you want.

AUTHOR

Juerd Waalboer <#####@juerd.nl>

SEE ALSO

perlunicode, *perluniintro*, *Encode*