NAME

perlunitut - Perl Unicode Tutorial

DESCRIPTION

The days of just flinging strings around are over. It's well established that modern programs need to be capable of communicating funny accented letters, and things like euro symbols. This means that programmers need new habits. It's easy to program Unicode capable software, but it does require discipline to do it right.

There's a lot to know about character sets, and text encodings. It's probably best to spend a full day learning all this, but the basics can be learned in minutes.

These are not the very basics, though. It is assumed that you already know the difference between bytes and characters, and realise (and accept!) that there are many different character sets and encodings, and that your program has to be explicit about them. Recommended reading is "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" by Joel Spolsky, at *http://joelonsoftware.com/articles/Unicode.html*.

This tutorial speaks in rather absolute terms, and provides only a limited view of the wealth of character string related features that Perl has to offer. For most projects, this information will probably suffice.

Definitions

It's important to set a few things straight first. This is the most important part of this tutorial. This view may conflict with other information that you may have found on the web, but that's mostly because many sources are wrong.

You may have to re-read this entire section a few times...

Unicode

Unicode is a character set with room for lots of characters. The ordinal value of a character is called a **code point**.

There are many, many code points, but computers work with bytes, and a byte can have only 256 values. Unicode has many more characters, so you need a method to make these accessible.

Unicode is encoded using several competing encodings, of which UTF-8 is the most used. In a Unicode encoding, multiple subsequent bytes can be used to store a single code point, or simply: character.

UTF-8

UTF-8 is a Unicode encoding. Many people think that Unicode and UTF-8 are the same thing, but they're not. There are more Unicode encodings, but much of the world has standardized on UTF-8.

UTF-8 treats the first 128 codepoints, 0..127, the same as ASCII. They take only one byte per character. All other characters are encoded as two or more (up to six) bytes using a complex scheme. Fortunately, Perl handles this for us, so we don't have to worry about this.

Text strings (character strings)

Text strings, or **character strings** are made of characters. Bytes are irrelevant here, and so are encodings. Each character is just that: the character.

Text strings are also called **Unicode strings**, because in Perl, every text string is a Unicode string.

On a text string, you would do things like:

```
$text =~ s/foo/bar/;
if ($string =~ /^\d+$/) { ... }
$text = ucfirst $text;
```

```
my $character_count = length $text;
```

The value of a character (`ord`, `chr`) is the corresponding Unicode code point.

Binary strings (byte strings)

Binary strings, or **byte strings** are made of bytes. Here, you don't have characters, just bytes. All communication with the outside world (anything outside of your current Perl process) is done in binary.

On a binary string, you would do things like:

```
my (@length_content) = unpack "(V/a)*", $binary;
$binary =~ s/\x00\x0F/\xFF\xF0/;  # for the brave :)
print {$fh} $binary;
my $byte_count = length $binary;
```

Encoding

Encoding (as a verb) is the conversion from *text* to *binary*. To encode, you have to supply the target encoding, for example `iso-8859-1` or `UTF-8`. Some encodings, like the `iso-8859` ("latin") range, do not support the full Unicode standard; characters that can't be represented are lost in the conversion.

Decoding

Decoding is the conversion from *binary* to *text*. To decode, you have to know what encoding was used during the encoding phase. And most of all, it must be something decodable. It doesn't make much sense to decode a PNG image into a text string.

Internal format

Perl has an **internal format**, an encoding that it uses to encode text strings so it can store them in memory. All text strings are in this internal format. In fact, text strings are never in any other format!

You shouldn't worry about what this format is, because conversion is automatically done when you decode or encode.

Your new toolkit

Add to your standard heading the following line:

```
use Encode qw(encode decode);
```

Or, if you're lazy, just:

```
use Encode;
```

I/O flow (the actual 5 minute tutorial)

The typical input/output flow of a program is:

```
1. Receive and decode
2. Process
3. Encode and output
```

If your input is binary, and is supposed to remain binary, you shouldn't decode it to a text string, of course. But in all other cases, you should decode it.

Decoding can't happen reliably if you don't know how the data was encoded. If you get to choose, it's a good idea to standardize on UTF-8.

```
my $foo   = decode('UTF-8', get 'http://example.com/');
my $bar   = decode('ISO-8859-1', readline STDIN);
my $xyzzy = decode('Windows-1251', $cgi->param('foo'));
```

Processing happens as you knew before. The only difference is that you're now using characters instead of bytes. That's very useful if you use things like `substr`, or `length`.

It's important to realize that there are no bytes in a text string. Of course, Perl has its internal encoding to store the string in memory, but ignore that. If you have to do anything with the number of bytes, it's probably best to move that part to step 3, just after you've encoded the string. Then you know exactly how many bytes it will be in the destination string.

The syntax for encoding text strings to binary strings is as simple as decoding:

```
$body = encode('UTF-8', $body);
```

If you needed to know the length of the string in bytes, now's the perfect time for that. Because `$body` is now a byte string, `length` will report the number of bytes, instead of the number of characters. The number of characters is no longer known, because characters only exist in text strings.

```
my $byte_count = length $body;
```

And if the protocol you're using supports a way of letting the recipient know which character encoding you used, please help the receiving end by using that feature! For example, E-mail and HTTP support MIME headers, so you can use the `Content-Type` header. They can also have `Content-Length` to indicate the number of *bytes*, which is always a good idea to supply if the number is known.

```
"Content-Type: text/plain; charset=UTF-8",
"Content-Length: $byte_count"
```

SUMMARY

Decode everything you receive, encode everything you send out. (If it's text data.)

Q and A (or FAQ)

After reading this document, you ought to read *perlunifaq* too.

ACKNOWLEDGEMENTS

Thanks to Johan Vromans from Squirrel Consultancy. His UTF-8 rants during the Amsterdam Perl Mongers meetings got me interested and determined to find out how to use character encodings in Perl in ways that don't break easily.

Thanks to Gerard Goossen from TTY. His presentation "UTF-8 in the wild" (Dutch Perl Workshop 2006) inspired me to publish my thoughts and write this tutorial.

Thanks to the people who asked about this kind of stuff in several Perl IRC channels, and have constantly reminded me that a simpler explanation was needed.

Thanks to the people who reviewed this document for me, before it went public. They are: Benjamin Smith, Jan-Pieter Cornet, Johan Vromans, Lukas Mai, Nathan Gray.

AUTHOR

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

SEE ALSO

perlunifaq, *perlunicode*, *perluniintro*, *Encode*