For some combinations, there are *precomposed* characters. `LATIN CAPITAL LETTER A WITH ACUTE`, for example, is defined as a single code point. These precomposed characters are, however, only available for some combinations, and are mainly meant to support round-trip conversions between Unicode and legacy standards (like the ISO 8859). In the general case, the composing method is more extensible. To support conversion between different compositions of the characters, various *normalization forms* to standardize representations are also defined.

Because of backward compatibility with legacy encodings, the "a unique number for every character" idea breaks down a bit: instead, there is "at least one number for every character". The same character could be represented differently in several legacy encodings. The converse is also not true: some code points do not have an assigned character. Firstly, there are unallocated code points within otherwise used blocks. Secondly, there are special Unicode control characters that do not represent true characters.

A common myth about Unicode is that it would be "16-bit", that is, Unicode is only represented as `0x10000` (or 65536) characters from `0x0000` to `0xFFFF`. **This is untrue.** Since Unicode 2.0 (July 1996), Unicode has been defined all the way up to 21 bits (`0x10FFFF`), and since Unicode 3.1 (March 2001), characters have been defined beyond `0xFFFF`. The first `0x10000` characters are called the *Plane 0*, or the *Basic Multilingual Plane* (BMP). With Unicode 3.1, 17 (yes, seventeen) planes in all were defined–but they are nowhere near full of defined characters, yet.

Another myth is that the 256-character blocks have something to do with languages–that each block would define the characters used by a language or a set of languages. **This is also untrue.** The division into blocks exists, but it is almost completely accidental–an artifact of how the characters have been and still are allocated. Instead, there is a concept called *scripts*, which is more useful: there is `Latin` script, `Greek` script, and so on. Scripts usually span varied parts of several blocks. For further information see *Unicode::UCD*.

The Unicode code points are just abstract numbers. To input and output these abstract numbers, the numbers must be *encoded* or *serialised* somehow. Unicode defines several *character encoding forms*, of which *UTF-8* is perhaps the most popular. UTF-8 is a variable length encoding that encodes Unicode characters as 1 to 6 bytes (only 4 with the currently defined characters). Other encodings include UTF-16 and UTF-32 and their big- and little-endian variants (UTF-8 is byte-order independent) The ISO/IEC 10646 defines the UCS-2 and UCS-4 encoding forms.

For more information about encodings–for instance, to learn what *surrogates* and *byte order marks* (BOMs) are–see *perlunicode*.

54.1.2 Perl's Unicode Support

Starting from Perl 5.6.0, Perl has had the capacity to handle Unicode natively. Perl 5.8.0, however, is the first recommended release for serious Unicode work. The maintenance release 5.6.1 fixed many of the problems of the initial Unicode implementation, but for example regular expressions still do not work with Unicode in 5.6.1.

Starting from Perl 5.8.0, the use of `use utf8` is no longer necessary. In earlier releases the `utf8` pragma was used to declare that operations in the current block or file would be Unicode-aware. This model was found to be wrong, or at least clumsy: the "Unicodeness" is now carried with the data, instead of being attached to the operations. Only one case remains where an explicit `use utf8` is needed: if your Perl script itself is encoded in UTF-8, you can use UTF-8 in your identifier names, and in string and regular expression literals, by saying `use utf8`. This is not the default because scripts with legacy 8-bit data in them would break. See *utf8*.

54.1.3 Perl's Unicode Model

Perl supports both pre-5.6 strings of eight-bit native bytes, and strings of Unicode characters. The principle is that Perl tries to keep its data as eight-bit bytes for as long as possible, but as soon as Unicodeness cannot be avoided, the data is transparently upgraded to Unicode.

Internally, Perl currently uses either whatever the native eight-bit character set of the platform (for example Latin-1) is, defaulting to UTF-8, to encode Unicode strings. Specifically, if all code points in the string are `0xFF` or less, Perl uses the native eight-bit character set. Otherwise, it uses UTF-8.

A user of Perl does not normally need to know nor care how Perl happens to encode its internal strings, but it becomes relevant when outputting Unicode strings to a stream without a PerlIO layer – one with the "default" encoding. In such a case, the raw bytes used internally (the native character set or UTF-8, as appropriate for each string) will be used, and a "Wide character" warning will be issued if those strings contain a character beyond 0x00FF.

For example,