Programming with Unicode Documentation

Release 2011

Victor Stinner

Aug 10, 2017

Contents

CHAPTER 1

About this book

The book is written in reStructuredText (reST) syntax and compiled by Sphinx.

I started to write in the 25th September 2010.

License

Thanks to

Reviewers: Alexander Belopolsky, Antoine Pitrou, Feth Arezki and Nelle Varoquaux, Natal Ngétal.

Notations

- `0bBBBBBBBB`: 8 bit unsigned number written in binary, first digit is the most significant. For example, `0b10000000` is 128.
- 0xHHHH: number written in hexadecimal, e.g. 0xFFFF is 65535.
- `0xHH 0xHH ...`: byte sequence with bytes written in hexadecimal, e.g. `0xC3 0xA9` (2 bytes) is the character é (U+00E9) *encoded* to UTF-8.
- U+HHHH: Unicode character with its code point written in hexadecimal. For example, U+20AC is the "euro sign" character, code point 8,364. Big code point are written with more than 4 hexadecimal digits, e.g. U+10FFFF is the biggest (unallocated) code point of *Unicode Character Set* 6.0: 1,114,111.
- A—B: range including start and end. Examples:
  - `0x00—0x7F` is the range 0 through 127 (128 bytes)
  - U+0000—U+00FF is the range 0 through 255 (256 characters)

- {U+HHHH, U+HHHH, ...}: a *character string*. For example, {U+0041, U+0042, U+0043} is the string “abc” (3 characters).

CHAPTER 2

Unicode nightmare

Unicode is the nightmare of many developers (and users) for different, and sometimes good reasons.

In the 1980s, only few people read documents in languages other their mother tongue and English. A computer supported only a small number of languages, the user configured his region to support languages of close countries. Memories and disks were expensive, all applications were written to use *byte strings* using 8 bits encodings: one byte per character was a good compromise.

Today with the Internet and the globalization, we all read and exchange documents from everywhere around the world (even if we don't understand everything). The problem is that documents rarely indicate their language (encoding), and displaying a document with the wrong encoding leads to a well known problem: *mojibake*.

It is difficult to get, or worse, *guess the encoding* of a document. Except for encodings of the UTF family (coming from the Unicode standard), there is no reliable algorithm for that. We have to rely on statistics to guess the most probable encoding, which is done by most Internet browsers.

Unicode support by *operating systems*, *programming languages* and *libraries* varies a lot. In general, the support is basic or non-existent. Each operating system manages Unicode differently. For example, *Windows* stores *filenames* as Unicode, whereas UNIX and BSD operating systems use bytes.

Mixing documents stored as bytes is possible, even if they use different encodings, but leads to *mojibake*. Because libraries and programs do also ignore encode and decode *warnings or errors*, writing a single character with a diacritic (any non-*ASCII* character) is sometimes enough to get an error.

Full Unicode support is complex because the Unicode charset is bigger than any other charset. For example, *ISO 8859-1* contains 256 code points including 191 characters, whereas Unicode version 6.0 contains *248,966 assigned code points*. The Unicode standard is larger than just a charset: it also explains how to display characters (e.g. left-to-right for English and right-to-left for persian), how to *normalize* a *character string* (e.g. precomposed characters versus the decomposed form), etc.

This book explains how to sympathize with Unicode, and how you should modify your program to avoid most, or all, issues related to encodings and Unicode.

CHAPTER 3

Definitions

Character

Generic term for a semantic symbol. Many possible interpretations exist in the context of encoding.

In computing, the most important aspect is that characters can be letters, spaces or control characters which represent the end of a file or can be used to trigger a sound.

Glyph

One or more shapes that may be combined into a grapheme.

In Latin, a glyph often has 2 variants like 'A' and 'a' and Arabic often has four. This term is context dependent and different styles or formats can be considered different glyphs.

Most relevant in programming is that diacritic marks (e.g. accents like ‘ and ^) are also glyphs, which are sometimes represented with another at one point, like the à in ISO 8859-1 or as two separate glyphs, so an a and the combining ‘ (U+0300 and U+0061 combined as U+00E0).

Code point

A **code point** is an unsigned integer. The smallest code point is zero. Code points are usually written as hexadecimal, e.g. "0x20AC" (8,364 in decimal).

Character set (charset)

A **character set**, abbreviated **charset**, is a mapping between *code points* and *characters*. The mapping has a fixed size. For example, most 7 bits encodings have 128 entries, and most 8 bits encodings have 256 entries. The biggest charset is the *Unicode Character Set* 6.0 with 1,114,112 entries.

In some charsets, code points are not all contiguous. For example, the *cp1252* charset maps code points from 0 though 255, but it has only 251 entries: 0x81, 0x8D, 0x8F, 0x90 and 0x9D code points are not assigned.

Examples of the *ASCII* charset: the digit five ("5", U+0035) is assigned to the code point 0x35 (53 in decimal), and the uppercase letter "A" (U+0041) to the code point 0x41 (65).

The biggest code point depends on the size of the charset. For example, the biggest code point of the ASCII charset is 127 ($2^7 - 1$)

Charset examples:

Charset	Code point	Character
ASCII	0x35	5 (U+0035)
ASCII	0x41	A (U+0041)
ISO-8859-15	0xA4	€ (U+20AC)
Unicode Character Set	0x20AC	€ (U+20AC)

Character string

A **character string**, or "Unicode string", is a string where each unit is a *character*. Depending on the implementation, each character can be any Unicode character, or only characters in the range U+0000—U+FFFF, range called the *Basic Multilingual Plane (BMP)*. There are 3 different implementations of character strings:

- array of 32 bits unsigned integers (the *UCS-4* encoding): full Unicode range
- array of 16 bits unsigned integers (*UCS-2*): BMP only
- array of 16 bits unsigned integers with *surrogate pairs* (*UTF-16*): full Unicode range

UCS-4 uses twice as much memory than UCS-2, but it supports all Unicode characters. UTF-16 is a compromise between UCS-2 and UCS-4: characters in the BMP range use one UTF-16 unit (16 bits), characters outside this range use two UTF-16 units (a *surrogate pair*, 32 bits). This advantage is also the main disadvantage of this kind of character string.

The length of a character string implemented using UTF-16 is the number of UTF-16 units, and not the number of characters, which is confusing. For example, the U+10FFFF character is *encoded* as two UTF-16 units: {U+DBFF, U+DFFF}. If the character string only contains characters of the BMP range, the length is the number of characters. Getting the n^{th} character or the length in characters using UTF-16 has a complexity of $O(n)$, whereas it has a complexity of $O(1)$ for UCS-2 and UCS-4 strings.

The *Java* language, the *Qt* library and *Windows 2000* implement character strings with UTF-16. The *C* and *Python* languages use UTF-16 or UCS-4 depending on: the size of the `wchar_t` type (16 or 32 bits) for C, and the compilation mode (narrow or wide) for Python. Windows 95 uses UCS-2 strings.

See also:

UCS-2, *UCS-4* and *UTF-16* encodings, and *surrogate pairs*.

Byte string

A **byte string** is a *character string encoded* to an *encoding*. It is implemented as an array of 8 bits unsigned integers. It can be called by its encoding. For example, a byte string encoded to *ASCII* is called an "ASCII encoded string", or simply an "ASCII string".

The *character range* supported by a byte string depends on its encoding, because an encoding is associated with a *charset*. For example, an ASCII string can only store characters in the range U+0000—U+007F.

The encoding is not stored explicitly in a byte string. If the encoding is not documented or attached to the byte string, *the encoding has to be guessed*, which is a difficult task. If a byte string is *decoded* from the wrong encoding, it will not be displayed correctly, leading to a well known issue: *mojibake*.

The same problem occurs if two byte strings encoded to different encodings are concatenated. **Never concatenate byte strings encoded to different encodings!** Use character strings, instead of byte strings, to avoid mojibake issues.

PHP5 only supports byte strings. In the *C language*, "strings" are usually byte strings which are implemented as the `char*` type (or `const char*`).

See also:

The `char*` type of the C language and the *mojibake* issue.

UTF-8 encoded strings and UTF-16 character strings

A *UTF-8* string is a particular case, because UTF-8 is able to encode all Unicode characters[1] . But a UTF-8 string is not a Unicode string because the string unit is byte and not character: you can get an individual byte of a multibyte character.

Another difference between UTF-8 strings and Unicode strings is the complexity of getting the nth character: $O(n)$ for the byte string and $O(1)$ for the Unicode string. There is one exception: if the Unicode string is implemented using UTF-16: it has also a complexity of $O(n)$.

Encoding

An **encoding** describes how to *encode code points* to bytes and how to *decode bytes* to code points.

An encoding is always associated with a *charset*. For example, the UTF-8 encoding is associated with the Unicode charset. So we can say that an encoding *encodes* characters to bytes and decode bytes to characters, or more generally, it encodes a *character string* to a *byte string* and decodes a byte string to a character string.

The 7 and 8 bits charsets have the simplest encoding: store a code point as a single byte. Since these charsets are also called encodings, it is easy to confuse them. The best example is the *ISO-8859-1 encoding*: all of the 256 possible bytes are considered as 8 bit code points (0 through 255) and are mapped to characters. For example, the character A (U+0041) has the code point 65 (0x41 in hexadecimal) and is stored as the byte `0x41`.

Charsets with more than 256 entries cannot encode all code points into a single byte. The encoding encodes all code points into byte sequences of the same length or of variable length. For example, *UTF-8* is a variable length encoding: code points lower than 128 use a single byte, whereas higher code points take 2, 3 or 4 bytes. The *UCS-2* encoding encodes all code points into sequences of two bytes (16 bits).

Encode a character string

Encode a *character string* to a *byte string*, to an encoding. For example, encode "Hé" to *UTF-8* gives `0x48 0xC3 0xA9`.

By default, most libraries are *strict*: raise an error at the first *unencodable character*. Some libraries allow to choose *how to handle them*.

Most encodings are stateless, but some encoding requires a stateful encoder. For example, the *UTF-16* encoding starts by generating a *BOM*, `0xFF 0xFE` or `0xFE 0xFF` depending on the endian.

[1] A UTF-8 encoder *should not encode surrogate characters* (U+D800—U+DFFF).

Decode a byte string

Decode a *byte string* from an encoding to a *character string*. For example, decode `0x48 0xC3 0xA9` from *UTF-8* gives "Hé".

By default, most libraries raise an error if *a byte sequence cannot be decoded*. Some libraries allow to choose *how to handle them*.

Most encodings are stateless, but some encoding requires a stateful decoder. For example, the *UTF-16* encoding decodes the two first bytes as a *BOM* to read the endian (use UTF-16-LE or UTF-16-BE).

Mojibake

When a *byte strings* is *decoded* from the wrong encoding, or when two byte strings encoded to different encodings are concatenated, a program will display **mojibake**.

The classical example is a latin string (with diacritics) encoded to UTF-8 but decoded from ISO-8859-1. It displays Ã© {U+00C3, U+00A9} for the é (U+00E9) letter, because é is encoded to `0xC3 0xA9` in UTF-8.

Other examples:

Text	Encoded to	Decoded from	Result
Noël	UTF-8	ISO-8859-1	NoÃ«l
	KOI-8	ISO-8859-1	ðÕÓÓËÉÊ

Note: "Mojibake" is japanese word meaning literally "unintelligible sequence of characters". This issue is called "" (krakozyabry) in Russian.

See also:

How to guess the encoding of a document?

Unicode: an Universal Character Set (UCS)

See also:

UCS-2, *UCS-4*, *UTF-8*, *UTF-16*, and *UTF-32* encodings.

CHAPTER 4

Unicode

Unicode is a character set. It is a superset of all the other character sets. In the version 6.0, Unicode has 1,114,112 code points (the last code point is U+10FFFF). Unicode 1.0 was limited to 65,536 code points (the last code point was U+FFFF), the range U+0000—U+FFFF called **BMP** (*Basic Multilingual Plane*). I call the range U+10000—U+10FFFF as **non-BMP** characters.

Unicode Character Set

The Unicode Character Set (UCS) contains 1,114,112 code points: U+0000—U+10FFFF. Characters and code point ranges are grouped by *categories*. Only encodings of the UTF family are able to encode the UCS.

Categories

Unicode 6.0 has 7 character categories, and each category has subcategories:

- Letter (L): lowercase (Ll), modifier (Lm), titlecase (Lt), uppercase (Lu), other (Lo)
- Mark (M): spacing combining (Mc), enclosing (Me), non-spacing (Mn)
- Number (N): decimal digit (Nd), letter (Nl), other (No)
- Punctuation (P): connector (Pc), dash (Pd), initial quote (Pi), final quote (Pf), open (Ps), close (Pe), other (Po)
- Symbol (S): currency (Sc), modifier (Sk), math (Sm), other (So)
- Separator (Z): line (Zl), paragraph (Zp), space (Zs)
- Other (C): control (Cc), format (Cf), not assigned (Cn), private use (Co), *surrogate* (Cs)

There are 3 ranges reserved for private use (Co subcategory): U+E000—U+F8FF (6,400 code points), U+F0000—U+FFFFD (65,534) and U+100000—U+10FFFD (65,534). Surrogates (Cs subcategory) use the range U+D800—U+DFFF (2,048 code points).

Statistics

On a total of 1,114,112 possible code points, only 248,966 code points are assigned: 77.6% are not assigned. Statistics excluding not assigned (Cn), private use (Co) and *surrogate* (Cs) subcategories:

- Letter: 100,520 (91.8%)
- Symbol: 5,508 (5.0%)
- Mark: 1,498 (1.4%)
- Number: 1,100 (1.0%)
- Punctuation: 598 (0.5%)
- Other: 205 (0.2%)
- Separator: 20 (0.0%)

On a total of 106,028 letters and symbols, 101,482 are in "other" subcategories (Lo and So): only 4.3% have well defined subcategories:

- Letter, lowercase (Ll): 1,759
- Letter, uppercase (Lu): 1,436
- Symbol, math (Sm): 948
- Letter, modifier (Lm): 210
- Symbol, modifier (Sk): 115
- Letter, titlecase (Lt): 31
- Symbol, currency (Sc): 47

Normalization

Unicode standard explains how to decompose a character. For example, the precomposed character ç (U+00C7, Latin capital letter C with cedilla) can be written as the sequence of two characters: {¸ (U+0327, Combining cedilla), c (U+0043, Latin capital letter C)}. This decomposition can be useful when searching for a substring in a text, e.g. removing the diacritic is pratical for the user. The decomposed form is called Normal Form D (**NFD**) and the precomposed form is called Normal Form C (**NFC**).

Form	String	Unicode
NFC	ç	U+00C7
NFD	¸c	{U+0327, U+0043}

Unicode database also contains a compatibility layer: if a character cannot be rendered (no font contain the requested character) or encoded to a specific encoding, Unicode proposes a *replacment character sequence which looks like the character*, but may have a different meaning.

For example, ij(U+0133, Latin small ligature ij) is replaced by the two characters {i (U+0069, Latin small letter I), j (U+006A, Latin small letter J)}. ijcharacter *cannot be encoded* to *ISO 8859-1*, whereas ij characters can.

Two extra normal forms use this compatibility layer: **NFKD** (decomposed) and **NFKC** (precomposed).

Note: The precomposed forms (NFC and NFKC) begin by a canonical decomposition before recomposing pre-combined characters again.

CHAPTER 5

Charsets and encodings

Encodings

There are many encodings around the world. Before Unicode, each manufacturer invented its own encoding to fit its client market and its usage. Most encodings are incompatible on at least one code, with some exceptions. A document stored in *ASCII* can be read using *ISO 8859-1* or UTF-8, because ISO-8859-1 and UTF-8 are supersets of ASCII. Each encoding can have multiple aliases, examples:

- ASCII: US-ASCII, ISO 646, ANSI_X3.4-1968, ...
- ISO-8859-1: Latin-1, iso88591, ...
- UTF-8: utf8, UTF_8, ...

Unicode is a charset and it requires a encoding. Only encodings of the UTF family are able to encode and decode all Unicode code points. Other encodings only support a subset of Unicode codespace. For example, ISO-8859-1 are the first 256 Unicode code points (U+0000—U+00FF).

This book presents the following encodings: *ASCII*, *cp1252*, *GBK*, *ISO 8859-1*, *ISO 8859-15*, *JIS*, *UCS-2*, *UCS-4*, *UTF-8*, *UTF-16* and *UTF-32*.

Popularity

The three most common encodings are, in chronological order of their creation: *ASCII* (1968), *ISO 8859-1* (1987) and *UTF-8* (1996).

Google posted an interesting graph of the usage of different encodings on the web: Unicode nearing 50% of the web (Mark Davis, january 2010). Because Google crawls a huge part of the web, these numbers should be reliable. In 2001, the most used encodings were:

- 1st (56%): *ASCII*
- 2nd (23%): Western Europe encodings (*ISO 8859-1*, *ISO 8859-15* and *cp1252*)
- 3rd (8%): Chinese encodings (*GB2312*, ...)

- and then come Korean (EUC-KR), Cyrillic (cp1251, KOI8-R, ...), East Europe (cp1250, ISO-8859-2), Arabic (cp1256, ISO-8859-6), etc.
- (UTF-8 was not used on the web in 2001)

In december 2007, for the first time: *UTF-8* becomes the most used encoding (near 25%). In january 2010, UTF-8 was close to 50%, and ASCII and Western Europe encodings were near 20%. The usage of other encodings doesn't change.

Encodings performances

Complexity of getting the n th character in a string, and of getting the length in character of a string:

- $O(1)$ for 7 and 8 bit encodings (*ASCII*, *ISO 8859 family*, ...), UCS-2 and UCS-4
- $O(n)$ for variable length encodings (e.g. the UTF family)

Examples

Encoding	A (U+0041)	é (U+00E9)	€ (U+20AC)	U+10FFFF
ASCII	`0x41`	—	—	—
ISO-8859-1	`0x41`	`0xE9`	—	—
UTF-8	`0x41`	`0xC3 0xA9`	`0xE2 0x82 0xAC`	`0xF4 0x8F 0xBF 0xBF`
UTF-16-LE	`0x41 0x00`	`0xE9 0x00`	`0xAC 0x20`	`0xFF 0xDB 0xFF 0xDF`
UTF-32-BE	`0x00 0x00 0x00 0x41`	`0x00 0x00 0x00 0xE9`	`0x00 0x00 0x20 0xAC`	`0x00 0x10 0xFF 0xFF`

— indicates that the character cannot be encoded.

Handle undecodable bytes and unencodable characters

Undecodable byte sequences

When a *byte string* is *decoded*, the decoder may fail to decode a specific byte sequence. For example, `0x61 0x62 0x63 0xE9` is not decodable from *ASCII* nor *UTF-8*, but it is decodable from *ISO 8859-1*.

Some encodings are able to decode any byte sequences. All encodings of the *ISO-8859 family* have this property, because all of the 256 code points of these 8 bits encodings are assigned.

Unencodable characters

When a *character string* is *encoded* to a *character set* smaller than the *Unicode character set (UCS)*, a character may not be encodable. For example, € (U+20AC) is not encodable to *ISO 8859-1*, but it is encodable to *ISO 8859-15* and *UTF-8*.

Error handlers

There are different choices to handle *undecodable byte sequences* and *unencodable characters*:

- strict: raise an error
- ignore
- replace by ? (U+003F) or (U+FFFD)
- replace by a similar glyph
- escape: format its code point
- etc.

Example of the "abcdé" string encoded to ASCII, é (U+00E9) is not encodable to ASCII:

Error handler	Output
strict	*raise an error*
ignore	`"abcd"`
replace by ?	`"abcd?"`
replace by a similar glyph	`"abcde"`
escape as hexadecimal	`"abcd\xe9"`
escape as XML entities	`"abcd&#233;"`

Replace unencodable characters by a similar glyph

By default, `WideCharToMultiByte()` replaces unencodable characters by similarly looking characters. The *normalization* to NFKC and NFKD does also such operation. Examples:

Character		Replaced by	
U+0141, latin capital letter l with stroke	Ł	L	U+004C, latin capital letter l
U+00B5, micro sign	µ	μ	U+03BC, greek small letter mu
U+221E, infinity	∞	8	U+0038, digit eight
U+0133, latin small ligature ij	ij	ij	{U+0069, U+006A}
U+20AC, euro sign	€	EUR	{U+0045, U+0055, U+0052}

∞ (U+221E) replaced by 8 (U+0038) is the worst example of the method: these two characters have completely different meanings.

Escape the character

Python "backslashreplace" error handler uses `\xHH`, `\uHHHH` or `\UHHHHHHHH` where HHH...H is the code point formatted in hexadecimal. PHP "long" error handler uses `U+HH`, `U+HHHH` or `encoding+HHHH` (e.g. `JIS+7E7E`).

PHP "entity" and Python "xmlcharrefreplace" error handlers escape the code point as an HTML/XML entity. For example, when U+00E9 is encoded to ASCII: it is replaced by `&#xE9;` in PHP and `&#233;` in Python.

Other charsets and encodings

There are much more charsets and encodings, but it is not useful to know them. The knowledge of a good conversion library, like *iconv*, is enough.

CHAPTER 6

Historical charsets and encodings

Between 1950 and 2000, each manufacturer and each operating system created its own 8 bits encoding. The problem was that 8 bits (256 code points) are not enough to store any character, and so the encoding tries to fit the user's language. Most 8 bits encodings are able to encode multiple languages, usually geographically close (e.g. ISO-8859-1 is intented for Western Europe).

It was difficult to exchange documents with different languages, because using an invalid encoding while loading the document leads to *mojibake*.

ASCII

ASCII encoding is supported by all applications. A document encoded in ASCII can be read decoded by any other encoding. This is explained by the fact that all 7 and 8 bits encodings are superset of ASCII, to be compatible with ASCII. Except *JIS X 0201* encoding: `0x5C` is decoded to the yen sign (U+00A5, ¥) instead of a backslash (U+005C, \).

ASCII is the smallest encoding, it only contains 128 codes including 95 printable characters (letters, digits, punctuation signs and some other various characters) and 33 control codes. Control codes are used to control the terminal. For example, the "line feed" (code point 10, usually written `"\n"`) marks the end of a line. There are some special control code. For example, the "bell" (code point 7, written `"\b"`) sent to ring a bell.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-a	-b	-c	-d	-e	-f
0-	NUL							BEL		TAB	LF			CR		
1-												ESC				
2-		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7-	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

0x00—0x1F and 0x7F are control codes:

- NUL (0x00): nul character (U+0000, `"\0"`)

- BEL (0x07): sent to ring a bell (U+0007, `"\b"`)
- TAB (0x09): horizontal tabulation (U+0009, `"\t"`)
- LF (0x0A): line feed (U+000A, `"\n"`)
- CR (0x0D): carriage return (U+000D, `"\r"`)
- ESC (0x1B): escape (U+001B)
- DEL (0x7F): delete (U+007F)
- other control codes are displayed as in this table

0x20 is a space.

Note: The first 128 code points of the Unicode charset (U+0000—U+007F) are the ASCII charset: Unicode is a superset of ASCII.

ISO 8859 family

Year	Norm	Description	Variant
1987	ISO 8859-1	Western European: German, French, Italian, ...	cp1252
1987	ISO 8859-2	Central European: Croatian, Polish, Czech, ...	cp1250
1988	ISO 8859-3	South European: Turkish and Esperanto	•
1988	ISO 8859-4	North European -	
1988	ISO 8859-5	Latin/Cyrillic: Macedonian, Russian, ...	KOI family
1987	ISO 8859-6	Latin/Arabic: Arabic language characters	cp1256
1987	ISO 8859-7	Latin/Greek: modern Greek language	cp1253
1988	ISO 8859-8	Latin/Hebrew: modern Hebrew alphabet	cp1255
1989	ISO 8859-9	Turkish: Largely the same as ISO 8859-1	cp1254
1992	ISO 8859-10	Nordic: a rearrangement of Latin-4	•
2001	ISO 8859-11	Latin/Thai: Thai language	TIS 620, cp874
1998	ISO 8859-13	Baltic Rim: Baltic languages	cp1257
1998	ISO 8859-14	Celtic: Gaelic, Breton	•
1999	ISO 8859-15	Revision of 8859-1: euro sign	cp1252
2001	ISO 8859-16	South-Eastern European	•

Note: ISO 8859-12 doesn't exist.

ISO 8859-1

ISO/CEI 8859-1, also known as "Latin-1" or "ISO-8859-1", is a superset of *ASCII*: it adds 128 code points, mostly latin letters with diacritics and 32 control codes. It is used in the USA and in Western Europe.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-a	-b	-c	-d	-e	-f
0-	NUL							BEL		TAB	LF			CR		
1-												ESC				
2-		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7-	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8-																
9-																
a-	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
b-	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
c-	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
d-	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
e-	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
f-	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

U+0000—U+001F, U+007F and U+0080—U+009F are control codes (displayed as in this table). See the ASCII table for U+0000—U+001F and U+007F control codes.

"NBSP" (U+00A0) is a non breaking space and "SHY" (U+00AD) is a soft hyphen.

Note: The 256 first code points of the Unicode charset (U+0000—U+00FF) are the ISO-8859-1 charset: Unicode is a superset of ISO-8859-1.

cp1252

Windows *code page* 1252, best known as cp1252, is a variant of *ISO 8859-1*. It is the default encoding of all English and Western Europe Windows setups. It is used as a fallback by web browsers if the webpage doesn't provide any encoding information (not in HTML, nor in HTTP).

cp1252 shares 224 code points with ISO-8859-1, the range 0x80—0x9F (32 characters, including 5 not assigned codes) are different. In ISO-8859-1, this range are 32 control codes (not printable).

Code point	ISO-8859-1	cp1252	Code point	ISO-8859-1	cp1252
0x80	U+0080	€ (U+20AC)	0x90	U+0090	*not assigned*
0x81	U+0081	*not assigned*	0x91	U+0091	‘ (U+2018)
0x82	U+0082	‚ (U+201A)	0x92	U+0092	’ (U+2019)
0x83	U+0083	*ƒ* (U+0192)	0x93	U+0093	“ (U+201C)
0x84	U+0084	„ (U+201E)	0x94	U+0094	” (U+201D)
0x85	U+0085	… (U+2026)	0x95	U+0095	• (U+2022)
0x86	U+0086	† (U+2020)	0x96	U+0096	– (U+2013)
0x87	U+0087	‡ (U+2021)	0x97	U+0097	— (U+2014)
0x88	U+0088	^ (U+02C6)	0x98	U+0098	~ (U+02DC)
0x89	U+0089	‰ (U+2030)	0x99	U+0099	™ (U+2122)
0x8A	U+008A	Š (U+0160)	0x9A	U+009A	š (U+0161)
0x8B	U+008B	‹ (U+2039)	0x9B	U+009B	› (U+203A)
0x8C	U+008C	Œ (U+0152)	0x9C	U+009C	œ (U+0153)
0x8D	U+008D	*not assigned*	0x9D	U+009D	*not assigned*
0x8E	U+008E	Ž (U+017D)	0x9E	U+009E	ž (U+017U)
0x8F	U+008F	*not assigned*	0x9F	U+009F	Ÿ (U+0178)

ISO 8859-15

ISO/CEI 8859-15, also known as Latin-9 or ISO-8859-15, is a variant of *ISO 8859-1*. 248 code points are identicals, 8 are different:

Code point	ISO-8859-1	ISO-8859-15	Code point	ISO-8859-1	ISO-8859-15
0xA4	¤ (U+00A4)	€ (U+20AC)	0xB8	¸ (U+00B8)	ž (U+017E)
0xA6	¦ (U+00A6)	Š (U+0160)	0xBC	¼ (U+00BC)	Œ (U+0152)
0xA8	¨ (U+00A8)	š (U+0161)	0xBD	½ (U+00BD)	œ (U+0152)
0xB4	´ (U+00B4)	Ž (U+017D)	0xBE	¾ (U+00BE)	Ÿ (U+0178)

CJK: asian encodings

Chinese encodings

GBK is a family of Chinese charsets using multibyte encodings:

- GB 2312 (1980): includes 6,763 Chinese characters
- GBK (1993) (*code page* 936)
- GB 18030 (2005, last revision in 2006)
- HZ (1989) (HG-GZ-2312)

Other encodings: Big5 (, Big Five Encoding, 1984), cp950.

Japanese encodings

JIS is a family of Japanese encodings:

- JIS X 0201 (1969): all code points are encoded to 1 byte
- 16 bits:

  - JIS X 0208 (first version in 1978: "JIS C 6226", last revision in 1997): code points are encoded to 1 or 2 bytes
  - JIS X 0212 (1990), extends JIS X 0208 charset: it is only a charset. Use EUC-JP or ISO 2022 to encode it.
  - JIS X 0213 (first version in 2000, last revision in 2004: EUC JIS X 2004), EUC JIS X 0213: it is only a charset, use EUC-JP, ISO 2022 or ShiftJIS 2004 to encode it.
- JIS X 0211 (1994), based on ISO/IEC 6429

Microsoft encodings:

- Shift JIS
- Windows *code page* 932 (cp932): extension of Shift JIS

In strict mode (flags=MB_ERR_INVALID_CHARS), cp932 cannot decode bytes in `0x81`—`0xA0` and `0xE0`—`0xFF` ranges. By default (flags=0), `0x81`—`0x9F` and `0xE0`—`0xFC` are decoded as U+30FB (Katakana middle dot), `0xA0` as U+F8F0, `0xFD` as U+F8F1, `0xFE` as U+F8F2 and `0xFF` as U+F8F3 (U+E000—U+F8FF is for private usage).

The JIS family causes *mojibake* on MS-DOS and Microsoft Windows because the yen sign (U+00A5, ¥) is encoded to `0x5C` which is a backslash (U+005C, \) in ASCII. For example, "C:\Windows\win.ini" is displayed "C:¥Windows¥win.ini". The backslash is encoded to `0x81 0x5F`.

To encode Japanese, there is also the ISO/IEC 2022 encoding family.

ISO 2022

ISO/IEC 2022 is an encoding family:

- ISO-2022-JP: JIS X 0201-1976, JIS X 0208-1978, JIS X 0208-1983
- ISO-2022-JP-1: JIS X 0212-1990
- ISO-2022-JP-2: GB 2312-1980, KS X 1001-1992, *ISO/IEC 8859-1*, ISO/IEC 8859-7
- ISO-2022-JP-3: JIS X 0201-1976, JIS X 0213-2000, JIS X 0213-2000
- ISO-2022-JP-2004: JIS X 0213-2004
- ISO-2022-KR: KS X 1001-1992
- ISO-2022-CN: GB 2312-1980, CNS 11643-1992 (planes 1 and 2)
- ISO-2022-CN-EXT: ISO-IR-165, CNS 11643-1992 (planes 3 though 7)

Extended Unix Code (EUC)

- EUC-CN: GB2312
- EUC-JP: JIS X 0208, JIS X 0212, JIS X 0201
- EUC-KR: KS X 1001, KS X 1003
- EUC-TW: CNS 11643 (16 planes)

Cyrillic

KOI family, " ":

- KOI-7: oldest KOI encoding (ASCII + some characters)

- KOI8-R: Russian
- KOI8-U: Ukrainian

Variants: ECMA-Cyrillic, KOI8-Unified, cp1251, MacUkrainian, Bulgarian MIK, ...

CHAPTER 7

Unicode encodings

UTF-8

UTF-8 is a multibyte encoding able to encode the whole Unicode charset. An encoded character takes between 1 and 4 bytes. UTF-8 encoding supports longer byte sequences, up to 6 bytes, but the biggest code point of Unicode 6.0 (U+10FFFF) only takes 4 bytes.

It is possible to be sure that a *byte string* is encoded to UTF-8, because UTF-8 adds markers to each byte. For the first byte of a multibyte character, bit 7 and bit 6 are set (`0b11xxxxxx`); the next bytes have bit 7 set and bit 6 unset (`0b10xxxxxx`).

Another cool feature of UTF-8 is that it has no endianness (it can be read in big or little endian order, it does not matter). Another advantage of UTF-8 is that most *C* bytes functions are compatible with UTF-8 encoded strings (e.g. `strcat()` or *`printf()`*), whereas they fail with UTF-16 and UTF-32 encoded strings because these encodings encode small codes with nul bytes.

The problem with UTF-8, if you compare it to *ASCII* or *ISO 8859-1*, is that it is a multibyte encoding: you cannot access a character by its character index directly, you have to iterate on each character because each character may have a different length in bytes. If getting a character by its index is a common operation in your program, use a *character string* instead of a *UTF-8 encoded string*.

See also:

Non-strict UTF-8 decoder and *Is UTF-8?*.

UCS-2, UCS-4, UTF-16 and UTF-32

UCS-2 and **UCS-4** encodings *encode* each code point to exactly one unit of, respectivelly, 16 and 32 bits. UCS-4 is able to encode all Unicode 6.0 code points, whereas UCS-2 is limited to *BMP* characters. These encodings are practical because the length in units is the number of characters.

UTF-16 and **UTF-32** encodings use, respectively, 16 and 32 bits units. UTF-16 encodes code points bigger than U+FFFF using two units: a *surrogate pair*. UCS-2 can be *decoded* from UTF-16. UTF-32 is also supposed to use

more than one unit for big code points, but in practice, it only requires one unit to store all code points of Unicode 6.0. That's why UTF-32 and UCS-4 are the same encoding.

Encoding	Word size	Unicode support
UCS-2	16 bits	BMP only
UTF-16	16 bits	Full
UCS-4	32 bits	Full
UTF-32	32 bits	Full

Windows 95 uses UCS-2, whereas Windows 2000 uses UTF-16.

Note: UCS stands for *Universal Character Set*, and UTF stands for *UCS Transformation format*.

UTF-7

The UTF-7 encoding is similar to the *UTF-8 encoding*, except that it uses 7 bits units instead of 8 bits units. It is used for example in emails with server which are not "8 bits clean".

Byte order marks (BOM)

UTF-16 and *UTF-32* use units bigger than 8 bits, and so are sensitive to endianness. A single unit can be stored as big endian (most significant bits first) or little endian (less significant bits first). BOM is a short byte sequence to indicate the encoding and the endian. It's the U+FEFF code point encoded with the given UTF encoding.

Unicode defines 6 different BOM:

BOM	Encoding	Endian
`0x2B 0x2F 0x76 0x38 0x2D` (5 bytes)	*UTF-7*	*endianless*
`0xEF 0xBB 0xBF` (3)	*UTF-8*	*endianless*
`0xFF 0xFE` (2)	*UTF-16-LE*	little endian
`0xFE 0xFF` (2)	*UTF-16-BE*	big endian
`0xFF 0xFE 0x00 0x00` (4)	*UTF-32-LE*	little endian
`0x00 0x00 0xFE 0xFF` (4)	*UTF-32-BE*	big endian

UTF-32-LE BOMs starts with UTF-16-LE BOM.

"UTF-16" and "UTF-32" encoding names are imprecise: depending of the context, format or protocol, it means UTF-16 and UTF-32 with BOM markers, or UTF-16 and UTF-32 in the host endian without BOM. On Windows, "UTF-16" usually means UTF-16-LE.

Some Windows applications, like notepad.exe, use UTF-8 BOM, whereas many applications are unable to detect the BOM, and so the BOM causes trouble. UTF-8 BOM should not be used for better interoperability.

UTF-16 surrogate pairs

Surrogates are characters in the Unicode range U+D800—U+DFFF (2,048 code points): it is also the *Unicode category* "surrogate" (Cs). The range is composed of two parts:

- U+D800—U+DBFF (1,024 code points): high surrogates
- U+DC00—U+DFFF (1,024 code points): low surrogates

In *UTF-16*, characters in ranges U+0000—U+D7FF and U+E000—U+FFFD are stored as a single 16 bits unit. *Non-BMP* characters (range U+10000—U+10FFFF) are stored as "surrogate pairs", two 16 bits units: an high surrogate (in range U+D800—U+DBFF) followed by a low surrogate (in range U+DC00—U+DFFF). A lone surrogate character is invalid in UTF-16, surrogate characters are always written as pairs (high followed by low).

Examples of surrogate pairs:

Character	Surrogate pair
U+10000	{U+D800, U+DC00}
U+10E6D	{U+D803, U+DE6D}
U+1D11E	{U+D834, U+DD1E}
U+10FFFF	{U+DBFF, U+DFFF}

Note: U+10FFFF is the highest code point encodable to UTF-16 and the highest code point of the *Unicode Character Set* 6.0. The {U+DBFF, U+DFFF} surrogate pair is the last available pair.

An *UTF-8* or *UTF-32* encoder should not encode surrogate characters (U+D800—U+DFFF), see *Non-strict UTF-8 decoder*.

C functions to create a surrogate pair (*encode* to UTF-16) and to join a surrogate pair (*decode* from UTF-16):

```
#include <stdint.h>

void
encode_utf16_pair(uint32_t character, uint16_t *units)
{
    unsigned int code;
    assert(0x10000 <= character && character <= 0x10FFFF);
    code = (character - 0x10000);
    units[0] = 0xD800 | (code >> 10);
    units[1] = 0xDC00 | (code & 0x3FF);
}

uint32_t
decode_utf16_pair(uint16_t *units)
{
    uint32_t code;
    assert(0xD800 <= units[0] && units[0] <= 0xDBFF);
    assert(0xDC00 <= units[1] && units[1] <= 0xDFFF);
    code = 0x10000;
    code += (units[0] & 0x03FF) << 10;
    code += (units[1] & 0x03FF);
    return code;
}
```

CHAPTER 8

How to guess the encoding of a document?

Only *ASCII*, *UTF-8* and encodings using a *BOM* (*UTF-7* with BOM, UTF-8 with BOM, *UTF-16*, and *UTF-32*) have reliable algorithms to get the encoding of a document. For all other encodings, you have to trust heuristics based on statistics.

Is ASCII?

Check if a document is encoded to *ASCII* is simple: test if the bit 7 of all bytes is unset (`0b0xxxxxxx`).

Example in *C*:

```c
int isASCII(const char *data, size_t size)
{
    const unsigned char *str = (const unsigned char*)data;
    const unsigned char *end = str + size;
    for (; str != end; str++) {
        if (*str & 0x80)
            return 0;
    }
    return 1;
}
```

In *Python*, the ASCII decoder can be used:

```python
def isASCII(data):
    try:
        data.decode('ASCII')
    except UnicodeDecodeError:
        return False
    else:
        return True
```

Note: Only use the Python function on short strings because it decodes the whole string into memory. For long strings, it is better to use the algorithm of the C function because it doesn't allocate any memory.

Check for BOM markers

If the string begins with a *BOM*, the encoding can be extracted from the BOM. But there is a problem with *UTF-16-BE* and *UTF-32-LE*: UTF-32-LE BOM starts with the UTF-16-LE BOM.

Example of a function written in *C* to check if a BOM is present:

```
#include <string.h>   /* memcmp() */

const char *UTF_16_BE_BOM = "\xFE\xFF";
const char *UTF_16_LE_BOM = "\xFF\xFE";
const char *UTF_8_BOM = "\xEF\xBB\xBF";
const char *UTF_32_BE_BOM = "\x00\x00\xFE\xFF";
const char *UTF_32_LE_BOM = "\xFF\xFE\x00\x00";

char* check_bom(const char *data, size_t size)
{
    if (size >= 3) {
        if (memcmp(data, UTF_8_BOM, 3) == 0)
            return "UTF-8";
    }
    if (size >= 4) {
        if (memcmp(data, UTF_32_LE_BOM, 4) == 0)
            return "UTF-32-LE";
        if (memcmp(data, UTF_32_BE_BOM, 4) == 0)
            return "UTF-32-BE";
    }
    if (size >= 2) {
        if (memcmp(data, UTF_16_LE_BOM, 2) == 0)
            return "UTF-16-LE";
        if (memcmp(data, UTF_16_BE_BOM, 2) == 0)
            return "UTF-16-BE";
    }
    return NULL;
}
```

For the UTF-16-LE/UTF-32-LE BOM conflict: this function returns `"UTF-32-LE"` if the string begins with `"\xFF\xFE\x00\x00"`, even if this string can be *decoded* from UTF-16-LE.

Example in *Python* getting the BOMs from the codecs library:

```
from codecs import BOM_UTF8, BOM_UTF16_BE, BOM_UTF16_LE, BOM_UTF32_BE, BOM_UTF32_LE

BOMS = (
    (BOM_UTF8, "UTF-8"),
    (BOM_UTF32_BE, "UTF-32-BE"),
    (BOM_UTF32_LE, "UTF-32-LE"),
    (BOM_UTF16_BE, "UTF-16-BE"),
    (BOM_UTF16_LE, "UTF-16-LE"),
)
```

```
def check_bom(data):
    return [encoding for bom, encoding in BOMS if data.startswith(bom)]
```

This function is different from the C function: it returns a list. It returns `['UTF-32-LE', 'UTF-16-LE']` if the string begins with `b"\xFF\xFE\x00\x00"`.

Is UTF-8?

UTF-8 encoding adds markers to each bytes and so it's possible to write a reliable algorithm to check if a *byte string* is encoded to UTF-8.

Example of a strict *C* function to check if a string is encoded with UTF-8. It rejects *overlong sequences* (e.g. `0xC0 0x80`) and *surrogate characters* (e.g. `0xED 0xB2 0x80`, U+DC80).

```
#include <stdint.h>

int isUTF8(const char *data, size_t size)
{
    const unsigned char *str = (unsigned char*)data;
    const unsigned char *end = str + size;
    unsigned char byte;
    unsigned int code_length, i;
    uint32_t ch;
    while (str != end) {
        byte = *str;
        if (byte <= 0x7F) {
            /* 1 byte sequence: U+0000..U+007F */
            str += 1;
            continue;
        }

        if (0xC2 <= byte && byte <= 0xDF)
            /* 0b110xxxxx: 2 bytes sequence */
            code_length = 2;
        else if (0xE0 <= byte && byte <= 0xEF)
            /* 0b1110xxxx: 3 bytes sequence */
            code_length = 3;
        else if (0xF0 <= byte && byte <= 0xF4)
            /* 0b11110xxx: 4 bytes sequence */
            code_length = 4;
        else {
            /* invalid first byte of a multibyte character */
            return 0;
        }

        if (str + (code_length - 1) >= end) {
            /* truncated string or invalid byte sequence */
            return 0;
        }

        /* Check continuation bytes: bit 7 should be set, bit 6 should be
         * unset (b10xxxxxx). */
        for (i=1; i < code_length; i++) {
            if ((str[i] & 0xC0) != 0x80)
                return 0;
        }
```

```
        if (code_length == 2) {
            /* 2 bytes sequence: U+0080..U+07FF */
            ch = ((str[0] & 0x1f) << 6) + (str[1] & 0x3f);
            /* str[0] >= 0xC2, so ch >= 0x0080.
               str[0] <= 0xDF, (str[1] & 0x3f) <= 0x3f, so ch <= 0x07ff */
        } else if (code_length == 3) {
            /* 3 bytes sequence: U+0800..U+FFFF */
            ch = ((str[0] & 0x0f) << 12) + ((str[1] & 0x3f) << 6) +
                  (str[2] & 0x3f);
            /* (0xff & 0x0f) << 12 | (0xff & 0x3f) << 6 | (0xff & 0x3f) = 0xffff,
               so ch <= 0xffff */
            if (ch < 0x0800)
                return 0;

            /* surrogates (U+D800-U+DFFF) are invalid in UTF-8:
               test if (0xD800 <= ch && ch <= 0xDFFF) */
            if ((ch >> 11) == 0x1b)
                return 0;
        } else if (code_length == 4) {
            /* 4 bytes sequence: U+10000..U+10FFFF */
            ch = ((str[0] & 0x07) << 18) + ((str[1] & 0x3f) << 12) +
                 ((str[2] & 0x3f) << 6) + (str[3] & 0x3f);
            if ((ch < 0x10000) || (0x10FFFF < ch))
                return 0;
        }
        str += code_length;
    }
    return 1;
}
```

In *Python*, the UTF-8 decoder can be used:

```
def isUTF8(data):
    try:
        data.decode('UTF-8')
    except UnicodeDecodeError:
        return False
    else:
        return True
```

In *Python 2*, this function is more tolerant than the C function, because the UTF-8 decoder of Python 2 accepts surrogate characters (U+D800—U+DFFF). For example, `isUTF8(b'\xED\xB2\x80')` returns `True`. With *Python 3*, the Python function is equivalent to the C function. If you would like to reject surrogate characters in Python 2, use the following strict function:

```
def isUTF8Strict(data):
    try:
        decoded = data.decode('UTF-8')
    except UnicodeDecodeError:
        return False
    else:
        for ch in decoded:
            if 0xD800 <= ord(ch) <= 0xDFFF:
                return False
        return True
```

Libraries

PHP has a builtin function to detect the encoding of a *byte string*: `mb_detect_encoding()`.

- chardet: *Python* version of the "chardet" algorithm implemented in Mozilla
- UTRAC: command line program (written in *C*) to recognize the encoding of an input file and its end-of-line type
- charguess: Ruby library to guess the charset of a document

CHAPTER 9

Good practices

Rules

To limit or avoid issues with Unicode, try to follow these rules:

- *decode* all bytes data as early as possible: keyboard strokes, files, data received from the network, ...
- *encode* back Unicode to bytes as late as possible: write text to a file, log a message, send data to the network, ...
- always store and manipulate text as *character strings*
- if you have to encode text and you can choose the encoding: prefer the *UTF-8* encoding. It is able to encode all Unicode 6.0 characters (including *non-BMP characters*), does not depend on endianness, is well supported by most programs, and its size is a good compromise.

Unicode support levels

There are different levels of Unicode support:

- **don't** support Unicode: only work correctly if all inputs and outputs are encoded to the same encoding, usually the *locale encoding*, use *byte strings*.
- **basic** Unicode support: decode inputs and encode outputs using the correct encodings, usually only support *BMP* characters. Use *Unicode strings*, or *byte strings* with the locale encoding or, better, an encoding of the UTF family (e.g. *UTF-8*).
- **full** Unicode support: have access to the Unicode database, *normalize text*, render correctly bidirectional texts and characters with diacritics.

These levels should help you to estimate the status of the Unicode support of your project. Basic support is enough if all of your users speak the same language or live in close countries. Basic Unicode support usually means excellent support of Western Europe languages. Full Unicode support is required to support Asian languages.

By default, the *C*, *C++* and *PHP5* languages have basic Unicode support. For the C and C++ languages, you can have basic or full Unicode support using a third-party library like *glib*, *Qt* or *ICU*. With PHP5, you can have basic Unicode support using "mb_" functions.

By default, the *Python 2* language doesn't support Unicode. You can have basic Unicode support if you store text into the `unicode` type and take care of input and output encodings. For *Python 3*, the situation is different: it has direct basic Unicode support by using the wide character API on Windows and by taking care of input and output encodings for you (e.g. decode command line arguments and environment variables). The `unicodedata` module is a first step for a full Unicode support.

Most UNIX and Windows programs don't support Unicode. Firefox web browser and OpenOffice.org office suite have full Unicode support. Slowly, more and more programs have basic Unicode support.

Don't expect to have full Unicode support directly: it requires a lot of work. Your project may be fully Unicode compliant for a specific task (e.g. *filenames*), but only have basic Unicode support for the other parts of the project.

Test the Unicode support of a program

Tests to evaluate the Unicode support of a program:

- Write non-ASCII characters (e.g. é, U+00E9) in all input fields: if the program fails with an error, it has no Unicode support.
- Write characters not encodable to the *locale encoding* (e.g. Ł, U+0141) in all input fields: if the program fails with an error, it probably has basic Unicode support.
- To test if a program is fully Unicode compliant, write text mixing different languages in different directions and characters with diacritics, especially in Persian characters. Try also *decomposed characters*, for example: {e, U+0301} (decomposed form of é, U+00E9).

See also:

Wikipedia article to test the Unicode support of your web browser. UTF-8 encoded sample plain-text file (Markus Kuhn, 2002).

Get the encoding of your inputs

Console:

- Windows: `GetConsoleCP()` for stdin and `GetConsoleOutputCP()` for stdout and stderr
- Other OSes: use the *locale encoding*

File formats:

- XML: the encoding can be specified in the `<?xml ...?>` header, use *UTF-8* if the encoding is not specified. For example, `<?xml version="1.0" encoding="iso-8859-1"?>`.
- HTML: the encoding can be specified in a "Content type" HTTP header, e.g. `<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1">`. If it is not, you have to guess the encoding.

Filesystem (filenames):

- *Windows* stores filenames as Unicode. It provides a bytes compatibily layer using the *ANSI code page* for applications using *byte strings*.
- *Mac OS X* encodes filenames to *UTF-8* and *normalize* see to a variant of the Normal Form D.

- Other OSes: use the *locale encoding*

See also:

How to guess the encoding of a document?

Switch from byte strings to character strings

Use character strings, instead of byte strings, to avoid *mojibake issues*.

CHAPTER 10

Operating systems

Windows

Since Windows 2000, Windows offers a nice Unicode API and supports *non-BMP characters*. It uses *Unicode strings* implemented as `wchar_t*` strings (LPWSTR). `wchar_t` is 16 bits long on Windows and so it uses *UTF-16*: *non-BMP* characters are stored as two `wchar_t` (a *surrogate pair*), and the length of a string is the number of UTF-16 units and not the number of characters.

Windows 95, 98 an Me had also Unicode strings, but were limited to *BMP characters*: they used *UCS-2* instead of UTF-16.

Code pages

A Windows application has two encodings, called code pages (abbreviated "cp"): ANSI and OEM code pages. The ANSI code page, `CP_ACP`, is used for the ANSI version of the *Windows API* to decode *byte strings* to *character strings* and has a number between 874 and 1258. The OEM code page or "IBM PC" code page, `CP_OEMCP`, comes from MS-DOS, is used for the *Windows console*, contains glyphs to create text interfaces (draw boxes) and has a number between 437 and 874. Example of a French setup: ANSI is *cp1252* and OEM is cp850.

There are code page constants:

- `CP_ACP`: Windows ANSI code page
- `CP_MACCP`: Macintosh code page
- `CP_OEMCP`: ANSI code page of the current process
- `CP_SYMBOL` (42): Symbol code page
- `CP_THREAD_ACP`: ANSI code page of the current thread
- `CP_UTF7` (65000): *UTF-7*
- `CP_UTF8` (65001): *UTF-8*

Functions.

UINT **GetACP**()
Get the ANSI code page number.

UINT **GetOEMCP**()
Get the OEM code page number.

BOOL **SetThreadLocale**(LCID *locale*)
Set the locale. It can be used to change the ANSI code page of current thread (`CP_THREAD_ACP`).

See also:

Wikipedia article: Windows code page.

Encode and decode functions

Encode and decode functions of `<windows.h>`.

MultiByteToWideChar()
Decode a *byte string* from a code page to a *character string*. Use `MB_ERR_INVALID_CHARS` flag to *return an error* on an *undecodable byte sequence*.

The default behaviour (flags=0) depends on the Windows version:

- Windows Vista and later: *replace undecodable bytes*
- Windows 2000, XP and 2003: *ignore undecodable bytes*

In strict mode (`MB_ERR_INVALID_CHARS`), the *UTF-8* decoder (`CP_UTF8`) returns an error on *surrogate characters* on Windows Vista and later. On Windows XP, the *UTF-8 decoder is not strict*: surrogates can be decoded in any mode.

The *UTF-7* decoder (`CP_UTF7`) only supports flags=0.

Examples on any Windows version:

Flags	default (0)	MB_ERR_INVALID_CHARS
`0xE9 0x80`, cp1252	é€ {U+00E9, U+20AC}	é€ {U+00E9, U+20AC}
`0xC3 0xA9`, CP_UTF8	é {U+00E9}	é {U+00E9}
`0xFF`, cp932	{U+F8F3}	*decoding error*
`0xFF`, CP_UTF7	{U+FF}	*invalid flags*

Examples on Windows Vista and later:

Flags	default (0)	MB_ERR_INVALID_CHARS
`0x81 0x00`, cp932	{U+30FB, U+0000}	*decoding error*
`0xFF`, CP_UTF8	{U+FFFD}	*decoding error*
`0xED 0xB2 0x80`, CP_UTF8	{U+FFFD, U+FFFD, U+FFFD}	*decoding error*

Examples on Windows 2000, XP, 2003:

Flags	default (0)	MB_ERR_INVALID_CHARS
`0x81 0x00`, cp932	{U+0000}	*decoding error*
`0xFF`, CP_UTF8	*decoding error*	*decoding error*
`0xED 0xB2 0x80`, CP_UTF8	{U+DC80}	{U+DC80}

Note: The U+30FB character is the Katakana middle dot (·). U+F8F3 code point is part of a Unicode range reserved for private use (U+E000—U+F8FF).

WideCharToMultiByte()

Encode a *character string* to a *byte string*. The behaviour on *unencodable characters* depends on the code page, the Windows version and the flags.

Code page	Windows version	Flags	Behaviour
CP_UTF8	2000, XP, 2003	0	Encode surrogates
	Vista or later	0	Replace surrogates by U+FFFD
		WC_ERR_INVALID_CHARS	Strict
CP_UTF7	*all versions*	0	Encode surrogates
Others	*all versions*	0	Replace by similar glyph
		WC_NO_BEST_FIT_CHARS	Replace by ? (1)

1.: Strict if you check for pusedDefaultChar pointer.

pusedDefaultChar is not supported by CP_UTF7 or CP_UTF8.

Use `WC_NO_BEST_FIT_CHARS` flag (or `WC_ERR_INVALID_CHARS` flag for `CP_UTF8`) to have a strict encoder: *return an error* on *unencodable character*. By default, if *a character cannot be encoded*, it is *replaced by a character with a similar glyph* or by "?" (U+003F). For example, with *cp1252*, Ł (U+0141) is replaced by L (U+004C).

On Windows Vista or later with `WC_ERR_INVALID_CHARS` flag, the *UTF-8* encoder (`CP_UTF8`) returns an error on *surrogate characters*. The default behaviour (flags=0) depends on the Windows version: surrogates are replaced by U+FFFD on Windows Vista and later, and are encoded to UTF-8 on older Windows versions. The `WC_NO_BEST_FIT_CHARS` flag is not supported by the UTF-8 encoder.

The `WC_ERR_INVALID_CHARS` flag is only supported by `CP_UTF8` and only on Windows Vista or later.

The *UTF-7* encoder (`CP_UTF7`) only supports flags=0. It is not strict: it encodes *surrogate characters*.

Examples (on any Windows version):

Flags	default (0)	WC_NO_BEST_FIT_CHARS
ÿ (U+00FF), cp932	`0x79` (y)	`0x3F` (?)
Ł (U+0141), cp1252	`0x4C` (L)	`0x3F` (?)
€ (U+20AC), cp1252	`0x80`	`0x80`
U+DC80, CP_UTF7	`0x2b 0x33 0x49 0x41 0x2d` (+3IA-)	*invalid flags*

Examples on Windows Vista an later:

Flags	default (0)	WC_ERR_INVALID_CHARS	WC_NO_BEST_FIT_CHARS
U+DC80, CP_UTF8	`0xEF 0xBF 0xBD`	*encoding error*	*invalid flags*

Examples on Windows 2000, XP, 2003:

Flags	default (0)	WC_ERR_INVALID_CHARS	WC_NO_BEST_FIT_CHARS
U+DC80, CP_UTF8	`0xED 0xB2 0x80`	*invalid flags*	*invalid flags*

Note: *`MultiByteToWideChar()`* and *`WideCharToMultiByte()`* functions are similar to *`mbstowcs()`* and *`wcstombs()`* functions.

Windows API: ANSI and wide versions

Windows has two versions of each function of its API: the ANSI version using *byte strings* (`A` suffix) and the *ANSI code page*, and the wide version (`W` suffix) using *character strings*. There are also functions without suffix using

TCHAR* strings: if the *C* define _UNICODE is defined, TCHAR is replaced by wchar_t and the Unicode functions are used; otherwise TCHAR is replaced by char and the ANSI functions are used. Example:

- CreateFileA(): bytes version, use *byte strings* encoded to the ANSI code page
- CreateFileW(): Unicode version, use *wide character strings*
- CreateFile(): TCHAR version depending on the _UNICODE define

Always prefer the Unicode version to avoid encoding/decoding errors, and use directly the W suffix to avoid compiling issues.

Note: There is a third version of the API: the MBCS API (multibyte character string). Use the TCHAR functions and define _MBCS to use the MBCS functions. For example, _tcsrev() is replaced by _mbsrev() if _MBCS is defined, by _wcsrev() if _UNICODE is defined, or by _strrev() otherwise.

Windows string types

- LPSTR (LPCSTR): *byte string*, char* (const char*)
- LPWSTR (LPCWSTR): *wide character string*, wchar_t* (const wchar_t*)
- LPTSTR (LPCTSTR): byte or wide character string depending of _UNICODE define, TCHAR* (const TCHAR*)

Filenames

Windows stores filenames as Unicode in the filesystem. Filesystem wide character POSIX-like API:

int **_wfstat**(const wchar_t* *filename*, struct _stat **statbuf*)
: Unicode version of stat().

FILE *__wfopen__(const wchar_t* *filename*, const wchar_t **mode*)
: Unicode version of fopen().

int **_wopen**(const wchar_t **filename*, int *oflag[*, int *pmode]*)
: Unicode version of open().

POSIX functions, like fopen(), use the *ANSI code page* to encode/decode strings.

Windows console

Console functions.

GetConsoleCP()
: Get the code page of the standard input (stdin) of the console.

GetConsoleOutputCP()
: Get the code page of the standard output (stdout and stderr) of the console.

WriteConsoleW()
: Write a *character string* into the console.

To improve the *Unicode support* of the console, set the console font to a TrueType font (e.g. "Lucida Console") and use the wide character API

If the console is unable to render a character, it tries to use a *character with a similar glyph*. For example, with OEM *code page* 850, Ł (U+0141) is replaced by L (U+0041). If no replacment character can be found, "?" (U+003F) is displayed instead.

In a console (`cmd.exe`), `chcp` command can be used to display or to change the *OEM code page* (and console code page). Changing the console code page is not a good idea because the ANSI API of the console still expects characters encoded to the previous console code page.

See also:

Conventional wisdom is retarded, aka What the @#%&* is _O_U16TEXT? (Michael S. Kaplan, 2008) and the Python bug report #1602: windows console doesn't print or input Unicode.

Note: Set the console *code page* to cp65001 (*UTF-8*) doesn't improve Unicode support, it is the opposite: non-ASCII are not rendered correctly and type non-ASCII characters (e.g. using the keyboard) doesn't work correctly, especially using raster fonts.

File mode

`_setmode()` and `_wsopen()` are special functions to set the encoding of a file:

- `_O_U8TEXT`: *UTF-8* without *BOM*
- `_O_U16TEXT`: *UTF-16* without BOM
- `_O_WTEXT`: UTF-16 with BOM

`fopen()` can use these modes using `ccs=` in the file mode:

- `ccs=UNICODE`: `_O_WTEXT`
- `ccs=UTF-8`: `_O_UTF8`
- `ccs=UTF-16LE`: `_O_UTF16`

Mac OS X

Mac OS X uses *UTF-8* for the filenames. If a filename is an invalid UTF-8 byte string, Mac OS X *returns an error*. The filenames are *decomposed* to an incompatible variant of the Normal Form D (NFD). Extract of the Technical Q&A QA1173: "For example, HFS Plus uses a variant of Normal Form D in which U+2000 through U+2FFF, U+F900 through U+FAFF, and U+2F800 through U+2FAFF are not decomposed."

Locales

To support different languages and encodings, UNIX and BSD operating systems have "locales". Locales are process-wide: if a thread or a library change the locale, the whole process is impacted.

Locale categories

Locale categories:

- `LC_COLLATE`: compare and sort strings

- LC_CTYPE: decode *byte strings* and encode *character strings*
- LC_MESSAGES: language of messages
- LC_MONETARY: monetary formatting
- LC_NUMERIC: number formatting (e.g. thousands separator)
- LC_TIME: time and date formatting

LC_ALL is a special category: if you set a locale using this category, it sets the locale for all categories.

Each category has its own environment variable with the same name. For example, LC_MESSAGES=C displays error messages in English. To get the value of a locale category, LC_ALL, LC_xxx (e.g. LC_CTYPE) or LANG environment variables are checked: use the first non empty variable. If all variables are unset, fallback to the C locale.

Note: The gettext library reads LANGUAGE, LC_ALL and LANG environment variables (and some others) to get the user language. The LANGUAGE variable is specific to gettext and is not related to locales.

The C locale

When a program starts, it does not get directly the user locale: it uses the default locale which is called the "C" locale or the "POSIX" locale. It is also used if no locale environment variable is set. For LC_CTYPE, the C locale usually means *ASCII*, but not always (see the locale encoding section). For LC_MESSAGES, the C locale means to speak the original language of the program, which is usually English.

Locale encoding

For Unicode, the most important locale category is LC_CTYPE: it is used to set the "locale encoding".

To get the locale encoding:

- Copy the current locale: `setlocale(LC_CTYPE, NULL)`
- Set the current locale encoding to the user preference: `setlocale(LC_CTYPE, "")`
- Use `nl_langinfo(CODESET)` if available
- or `setlocale(LC_CTYPE, NULL)`

For the C locale, `nl_langinfo(CODESET)` returns *ASCII*, or an alias to this encoding (e.g. "US-ASCII" or "646"). But on FreeBSD, Solaris and *Mac OS X*, codec functions (e.g. *`mbstowcs()`*) use *ISO 8859-1* even if `nl_langinfo(CODESET)` announces ASCII encoding. AIX uses *ISO 8859-1* for the C locale (and `nl_langinfo(CODESET)` returns `"ISO8859-1"`).

Locale functions

`<locale.h>` functions.

char* **`setlocale`**(category, NULL)
: Get the value of the specified locale category.

char* **`setlocale`**(category, name)
: Set the value of the specified locale category.

`<langinfo.h>` functions.

char* **nl_langinfo**(CODESET)
: Get the name of the locale encoding.

`<stdlib.h>` functions.

size_t **mbstowcs**(wchar_t **dest*, const char **src*, size_t *n*)
: *Decode* a *byte string* from the *locale encoding* to a *character string*. The decoder is *strict*: it returns an error on *undecodable byte sequence*. If available, prefer the reentrant version: `mbsrtowcs()`.

size_t **wcstombs**(char **dest*, const wchar_t **src*, size_t *n*)
: *Encode* a *character string* to a *byte string* in the *locale encoding*. The encoder is *strict* : it returns an error if *a character cannot by encoded*. If available, prefer the reentrant version: `wcsrtombs()`.

mbstowcs() and wcstombs() are *strict* and don't support *error handlers*.

Note: "mbs" stands for "multibyte string" (byte string) and "wcs" stands for "wide character string".

On Windows, the "locale encoding" are the *ANSI and OEM code pages*. A Windows program uses the user preferred code pages at startup, whereas a program starts with the C locale on UNIX.

Filesystems (filenames)

CD-ROM and DVD

CD-ROM uses the ISO 9660 filesystem which stores filenames as *byte strings*. This filesystem is very restrictive: only A-Z, 0-9, _ and "." are allowed. Microsoft has developed the Joliet extension: store filenames as *UCS-2*, up to 64 characters (*BMP* only). It was first supported by Windows 95. Today, all operating systems are able to read it.

UDF (Universal Disk Format) is the filesystem of DVD: it stores filenames as character strings.

Microsoft: FAT and NTFS filesystems

MS-DOS uses the FAT filesystems (FAT 12, FAT 16, FAT 32): filenames are stored as *byte strings*. Filenames are limited to 8+3 characters (8 for the name, 3 for the extension) and displayed differently depending on the *code page* (*mojibake issue*).

Microsoft extended its FAT filesystem in Windows 95: the Virtual FAT (VFAT) supports "long filenames", filenames are stored as *UCS-2*, up to 255 characters (BMP only). Starting at Windows 2000, *non-BMP characters* can be used: *UTF-16* replaces UCS-2 and the limit is now 255 UTF-16 units.

The NTFS filesystem stores filenames using UTF-16 encoding.

Apple: HFS and HFS+ filesystems

HFS stores filenames as byte strings.

HFS+ stores filenames as *UTF-16*: the maximum length is 255 UTF-16 units.

Others

JFS and ZFS also use Unicode.

The ext family (ext2, ext3, ext4) store filenames as byte strings.

CHAPTER 11

Programming languages

C language

The C language is a low level language, close to the hardware. It has a builtin *character string* type (`wchar_t*`), but only few libraries support this type. It is usually used as the first "layer" between the kernel (system calls, e.g. open a file) and applications, higher level libraries and other programming languages. This first layer uses the same type as the kernel: except *Windows*, all kernels use *byte strings*.

There are higher level libraries, like *glib* or *Qt*, offering a Unicode API, even if the underlying kernel uses byte strings. Such libraries use a codec to *encode* data to the kernel and to *decode* data from the kernel. The codec is usually the current *locale encoding*.

Because there is no Unicode standard library, most third-party libraries chose the simple solution: use *byte strings*. For example, the OpenSSL library, an open source cryptography toolkit, expects *filenames* as byte strings. On Windows, you have to encode Unicode filenames to the current *ANSI code page*, which is a small subset of the Unicode charset.

Byte API (char)

`char`

For historical reasons, `char` is the C type for a character ("char" as "character"). In pratical, it's only true for 7 and 8 bits encodings like *ASCII* or *ISO 8859-1*. With multibyte encodings, a `char` is only one byte. For example, the character "é" (U+00E9) is encoded as two bytes (`0xC3 0xA9`) in *UTF-8*.

`char` is a 8 bits integer, it is signed or not depending on the operating system and the compiler. On Linux, the GNU compiler (gcc) uses a signed type for Intel CPU. It defines `__CHAR_UNSIGNED__` if `char` type is unsigned. Check if the `CHAR_MAX` constant from `<limits.h>` is equal to 255 to check if `char` is unsigned.

A literal byte is written between apostrophes, e.g. `'a'`. Some control characters can be written with an backslash plus a letter (e.g. `'\n'` = 10). It's also possible to write the value in octal (e.g. `'\033'` = 27) or hexadecimal (e.g. `'\x20'` = 32). An apostrophe can be written `'\''` or `'\x27'`. A backslash is written `'\\'`.

`<ctype.h>` contains functions to manipulate bytes, like `toupper()` or `isprint()`.

Byte string API (char*)

char*

`char*` is a a *byte string*. This type is used in many places in the C standard library. For example, `fopen()` uses `char*` for the filename.

`<string.h>` is the byte string library. Most functions starts with "str" (string) prefix: `strlen()`, `strcat()`, etc. `<stdio.h>` contains useful string functions like `snprintf()` to format a message.

The length of a string is stored directly in the string as a nul byte at the end. This is a problem with encodings using nul bytes (e.g. *UTF-16* and *UTF-32*): `strlen()` cannot be used to get the length of the string, whereas most C functions suppose that `strlen()` gives the length of the string. To support such encodings, the length should be stored differently (e.g. in another variable or function argument) and `str*()` functions should be replaced by `mem*` functions (e.g. replace `strcmp(a, b) == 0` by `memcmp(a, b) == 0`).

A literal byte strings is written between quotes, e.g. `"Hello World!"`. As byte literal, it's possible to add control characters and characters in octal or hexadecimal, e.g. `"Hello World!\n"`.

Character API (wchar_t)

wchar_t

With ISO C99 comes `wchar_t`: the *character* type. It can be used to store Unicode characters. As `char`, it has a library: `<wctype.h>` contains functions like `towupper()` or `iswprint()` to manipulate characters.

`wchar_t` is a 16 or 32 bits integer, signed or not. Linux uses 32 bits signed integer. Mac OS X uses 32 bits integer. Windows and AIX use 16 bits integer (*BMP* only). Check if the `WCHAR_MAX` constant from `<wchar.h>` is equal to 0xFFFF to check if `wchar_t` is a 16 bits unsigned integer.

A literal character is written between apostrophes with the `L` prefix, e.g. `L'a'`. As byte literal, it's possible to write control character with an backslash and a character with its value in octal or hexadecimal. For codes bigger than 255, `'\uHHHH'` syntax can be used. For codes bigger than 65535, `'\UHHHHHHHH'` syntax can be used with 32 bits `wchar_t`.

Character string API (wchar_t*)

wchar_t*

With ISO C99 comes `wchar_t*`: the *character string* type. The standard library `<wchar.h>` contains character string functions like `wcslen()` or *`wprintf()`*, and constants like `WCHAR_MAX`. If `wchar_t` is 16 bits long, *non-BMP* characters are encoded to *UTF-16* as *surrogate pairs*.

A literal character strings is written between quotes with the `L` prefix, e.g. `L"Hello World!\n"`. As character literals, it supports also control character, codes written in octal, hexadecimal, `L"\uHHHH"` and `L"\UHHHHHHHH"`.

POSIX.1-2001 has no function ignoring case to compare character strings. POSIX.1-2008, a recent standard, adds `wcscasecmp()`: the GNU libc has it as an extension (if `_GNU_SOURCE` is defined). Windows has the `_wcsnicmp()` function.

Windows uses (*UTF-16*) wchar_t* strings for its Unicode API.

printf functions family

int **printf** (const char* *format*, ...)

int **wprintf** (const wchar_t* *format*, ...)

Formats of string arguments for the printf functions:

- `"%s"`: literal byte string (`char*`)
- `"%ls"`: literal character string (`wchar_t*`)

`printf("%ls")` is *strict*: it stops immediatly if a *character string* argument *cannot be encoded* to the *locale encoding*. For example, the following code prints the truncated string "Latin capital letter L with stroke: [" if Ł (U+0141) cannot be encoded to the locale encoding.

```
printf("Latin capital letter L with stroke: [%ls]\n", L"\u0141");
```

`wprintf("%s")` and `wprintf("%.<length>s")` are *strict*: they stop immediatly if *a byte string* argument *cannot be decoded* from the *locale encoding*. For example, the following code prints the truncated string "Latin capital letter L with stroke: [" if `0xC5 0x81` (U+0141 encoded to *UTF-8*) cannot be decoded from the *locale encoding*.

```
wprintf(L"Latin capital letter L with stroke): [%s]\n", "\xC5\x81");
wprintf(L"Latin capital letter L with stroke): [%.10s]\n", "\xC5\x81");
```

`wprintf("%ls")` *replaces unencodable character string* arguments by ? (U+003F). For example, the following example print "Latin capital letter L with stroke: [?]" if Ł (U+0141) cannot be encoded to the *locale encoding*:

```
wprintf(L"Latin capital letter L with stroke: [%s]\n", L"\u0141");
```

So to avoid truncated strings, try to use only *`wprintf()`* with character string arguments.

Note: There is also `"%S"` format which is a deprecated alias to the `"%ls"` format, don't use it.

C++

- `std::wstring`: *character string* using the `wchar_t` type, Unicode version of `std::string` (*byte string*)
- `std::wcin`, `std::wcout` and `std::wcerr`: standard input, output and error output; Unicode version of `std::cin`, `std::cout` and `std::cerr`
- `std::wostringstream`: character stream buffer; Unicode version of `std::ostringstream`.

To initialize the *locales*, equivalent to `setlocale(LC_ALL, "")`, use:

```
#include <locale>
std::locale::global(std::locale(""));
```

If you use also C and C++ functions (e.g. *`printf()`* and `std::cout`) to access the standard streams, you may have issues with *non-ASCII* characters. To avoid these issues, you can disable the automatic synchronization between C (`std*`) and C++ (`std::c*`) streams using:

```
#include <iostream>
std::ios_base::sync_with_stdio(false);
```

Note: Use `typedef basic_ostringstream<wchar_t> wostringstream;` if wostringstream is not available.

Python

Python supports Unicode since its version 2.0 released in October 2000. *Byte* and *Unicode* strings store their length, so it's possible to embed nul byte/character.

Python can be compiled in two modes: narrow (*UTF-16*) and wide (*UCS-4*). `sys.maxunicode` constant is 0xFFFF in narrow build, and 0x10FFFF in wide build. Python is compiled in narrow mode on Windows, because `wchar_t` is also 16 bits on Windows and so it is possible to use Python Unicode strings as `wchar_t*` strings without any (expensive) conversion.

See also:

Python Unicode HOWTO.

Python 2

`str` is the *byte string* type and `unicode` is the *character string* type. Literal byte strings are written `b'abc'` (syntax compatible with Python 3) or `'abc'` (legacy syntax), `\xHH` can be used to write a byte by its hexadecimal value (e.g. `b'\x80'` for 128). Literal Unicode strings are written with the prefix `u`: `u'abc'`. Code points can be written as hexadecimal: `\xHH` (U+0000—U+00FF), `\uHHHH` (U+0000—U+FFFF) or `\UHHHHHHHH` (U+0000—U+10FFFF), e.g. `'euro sign:\u20AC'`.

In Python 2, `str + unicode` gives `unicode`: the byte string is *decoded* from the default encoding (*ASCII*). This coercion was a bad design idea because it was the source of a lot of confusion. At the same time, it was not possible to switch completely to Unicode in 2000: computers were slower and there were fewer Python core developers. It took 8 years to switch completely to Unicode: Python 3 was relased in December 2008.

Narrow build of Python 2 has a partial support of *non-BMP* characters. The unichr() function raises an error for code bigger than U+FFFF, whereas literal strings support non-BMP characters (e.g. `'\U0010FFFF'`). Non-BMP characters are encoded as *surrogate pairs*. The disavantage is that `len(u'\U00010000')` is 2, and `u'\U0010FFFF'[0]` is `u'\uDBFF'` (lone surrogate character).

Note: **DO NOT CHANGE THE DEFAULT ENCODING!** Calling sys.setdefaultencoding() is a very bad idea because it impacts all libraries which suppose that the default encoding is ASCII.

Python 3

`bytes` is the *byte string* type and `str` is the *character string* type. Literal byte strings are written with the `b` prefix: `b'abc'`. `\xHH` can be used to write a byte by its hexadecimal value, e.g. `b'\x80'` for 128. Literal Unicode strings are written `'abc'`. Code points can be used directly in hexadecimal: `\xHH` (U+0000—U+00FF), `\uHHHH` (U+0000—U+FFFF) or `\UHHHHHHHH` (U+0000—U+10FFFF), e.g. `'euro sign:\u20AC'`. Each item of a byte string is an integer in range 0—255: `b'abc'[0]` gives 97, whereas `'abc'[0]` gives `'a'`.

Python 3 has a full support of *non-BMP* characters, in narrow and wide builds. But as Python 2, chr(0x10FFFF) creates a string of 2 characters (a *UTF-16 surrogate pair*) in a narrow build. `chr()` and `ord()` supports non-BMP characters in both modes.

Python 3 uses U+DC80—U+DCFF character range to store *undecodable bytes* with the `surrogateescape` error handler, described in the PEP 383 (*Non-decodable Bytes in System Character Interfaces*). It is used for filenames and environment variables on UNIX and BSD systems. Example: `b'abc\xff'.decode('ASCII', 'surrogateescape')` gives `'abc\uDCFF'`.

Differences between Python 2 and Python 3

`str + unicode` gives `unicode` in Python 2 (the byte string is decoded from the default encoding, *ASCII*) and it raises a `TypeError` in Python 3. In Python 3, comparing `bytes` and `str` gives `False`, emits a `BytesWarning` warning or raises a `BytesWarning` exception depending of the bytes warning flag (`-b` or `-bb` option passed to the Python program). In Python 2, the byte string is *decoded* from the default encoding (ASCII) to Unicode before being compared.

UTF-8 decoder of Python 2 accept *surrogate characters*, even if there are invalid, to keep backward compatibility with Python 2.0. In Python 3, the *UTF-8 decoder is strict*: it rejects surrogate characters.

It is possible to make Python 2 behave more like Python 3 with *from __future__ import unicode_literals*.

Codecs

The `codecs` and `encodings` modules provide text encodings. They support a lot of encodings. Some examples: ASCII, ISO-8859-1, UTF-8, UTF-16-LE, ShiftJIS, Big5, cp037, cp950, EUC_JP, etc.

`UTF-8`, `UTF-16-LE`, `UTF-16-BE`, `UTF-32-LE` and `UTF-32-BE` don't use *BOM*, whereas `UTF-8-SIG`, `UTF-16` and `UTF-32` use BOM. `mbcs` is only available on Windows: it is the *ANSI code page*.

Python provides also many *error handlers* used to specify how to handle *undecodable byte sequences* and *unencodable characters*:

- `strict` (default): raise a `UnicodeDecodeError` or a `UnicodeEncodeError`
- `replace`: replace undecodable bytes by (U+FFFD) and unencodable characters by `?` (U+003F)
- `ignore`: ignore undecodable bytes and unencodable characters
- `backslashreplace` (only encode): replace unencodable bytes by `\xHH`

Python 3 has three more error handlers:

- `surrogateescape`: replace undecodable bytes (non-ASCII: `0x80`—`0xFF`) by *surrogate characters* (in U+DC80—U+DCFF) on decoding, replace characters in range U+DC80—U+DCFF by bytes in `0x80`—`0xFF` on encoding. Read the PEP 383 (*Non-decodable Bytes in System Character Interfaces*) for the details.
- `surrogatepass`, specific to `UTF-8` codec: allow encoding/decoding surrogate characters in *UTF-8*. It is required because UTF-8 decoder of Python 3 rejects surrogate characters by default.
- `backslashreplace` (for decode): replace undecodable bytes by `\xHH`

Decoding examples in Python 3:

- `b'abc\xff'.decode('ASCII')` uses the `strict` error handler and raises an `UnicodeDecodeError`
- `b'abc\xff'.decode('ASCII', 'ignore')` gives `'abc'`
- `b'abc\xff'.decode('ASCII', 'replace')` gives `'abc\uFFFD'`
- `b'abc\xff'.decode('ASCII', 'surrogateescape')` gives `'abc\uDCFF'`

Encoding examples in Python 3:

- `'\u20ac'.encode('UTF-8')` gives `b'\xe2\x82\xac'`
- `'abc\xff'.encode('ASCII')` uses the `strict` error handler and raises an `UnicodeEncodeError`
- `'abc\xff'.encode('ASCII', 'backslashreplace')` gives `b'abc\\xff'`

String methods

Byte string (`str` in Python 2, `bytes` in Python 3) methods:

- `.decode(encoding, errors='strict')`: *decode* from the specified encoding and (optional) *error handler*.

Character string (`unicode` in Python 2, `str` in Python 3) methods:

- `.encode(encoding, errors='strict')`: *encode* to the specified encoding with an (optional) *error handler*
- `.isprintable()`: `False` if the *character category* is other (Cc, Cf, Cn, Co, Cs) or separator (Zl, Zp, Zs), `True` otherwise. There is an exception: even if U+0020 is a separator, `' '.isprintable()` gives `True`.
- `.toupper()`: convert to uppercase

Filesystem

Python decodes bytes filenames and encodes Unicode filenames using the filesystem encoding, `sys.getfilesystemencoding()`:

- `mbcs` (*ANSI code page*) on *Windows*
- `UTF-8` on *Mac OS X*
- *locale encoding* otherwise

Python uses the `strict` *error handler* in Python 2, and `surrogateescape` (PEP 383) in Python 3. In Python 2, if `os.listdir(u'.')` cannot decode a filename, it keeps the bytes filename unchanged. Thanks to `surrogateescape`, decoding a filename never fails in Python 3. But encoding a filename can fail in Python 2 and 3 depending on the filesystem encoding. For example, on Linux with the C locale, the Unicode filename `"h\xe9.py"` cannot be encoded because the filesystem encoding is ASCII.

In Python 2, use `os.getcwdu()` to get the current directory as Unicode.

Windows

Encodings used on Windows:

- locale.getpreferredencoding(): *ANSI code page*
- `'mbcs'` codec: *ANSI code page*
- sys.stdout.encoding, sys.stderr.encoding: encoding of the *Windows console*.
- sys.argv, os.environ, subprocess.Popen(args): native Unicode support (no encoding)

Modules

`codecs` module:

- `BOM_UTF8`, `BOM_UTF16_BE`, `BOM_UTF32_LE`, ...: *Byte order marks (BOM)* constants
- `lookup(name)`: get a Python codec. `lookup(name).name` gets the Python normalized name of a codec, e.g. `codecs.lookup('ANSI_X3.4-1968').name` gives `'ascii'`.
- `open(filename, mode='rb', encoding=None, errors='strict', ...)`: legacy API to open a binary or text file. To open a file in Unicode mode, use `io.open()` instead

`io` module:

- `open(name, mode='r', buffering=-1, encoding=None, errors=None, ...)`: open a binary or text file in read and/or write mode. For text file, `encoding` and `errors` can be used to specify the encoding and the *error handler*. By default, it opens text files with the *locale encoding* in *strict* mode.
- `TextIOWrapper()`: wrapper to read and/or write text files, encode from/decode to the specified encoding (and *error handler*) and normalize newlines (`\r\n` and `\r` are replaced by `\n`). It requires a buffered file. Don't use it directly to open a text file: use `open()` instead.

`locale` module (*locales*):

- `LC_ALL`, `LC_CTYPE`, ...: *locale categories*
- `getlocale(category)`: get the value of a *locale category* as the tuple (language code, encoding name)
- `getpreferredencoding()`: get the *locale encoding*
- `setlocale(category, value)`: set the value of a locale category

`sys` module:

- `getdefaultencoding()`: get the default encoding, e.g. used by `'abc'.encode()`. In Python 3, the default encoding is fixed to `'utf-8'`, in Python 2, it is `'ascii'` by default.
- `getfilesystemencoding()`: get the filesystem encoding used to decode and encode filenames
- `maxunicode`: biggest Unicode code point storable in a single Python Unicode character, 0xFFFF in narrow build or 0x10FFFF in wide build.

`unicodedata` module:

- `category(char)`: get the *category* of a character
- `name(char)`: get the name of a character
- `normalize(string)`: *normalize* a string to the NFC, NFD, NFKC or NFKD form

PHP

In PHP 5, a literal string (e.g. `"abc"`) is a *byte string*. PHP has no *character string* type, only a "string" type which is a *byte string*.

PHP has "multibyte" functions to manipulate byte strings using their encoding. These functions have an optional encoding argument. If the encoding is not specified, PHP uses the default encoding (called "internal encoding"). Some multibyte functions:

- `mb_internal_encoding()`: get or set the internal encoding
- `mb_substitute_character()`: change how to *handle unencodable characters*:
  - `"none"`: *ignore* unencodable characters
  - `"long"`: *escape as hexadecimal* value, e.g. `"U+E9"` or `"JIS+7E7E"`
  - `"entity"`: *escape as HTML entities*, e.g. `"&#xE9;"`
- `mb_convert_encoding()`: *decode* from an encoding and *encode* to another encoding
- `mb_ereg()`: search a pattern using a regular expression
- `mb_strlen()`: get the length in characters
- `mb_detect_encoding()`: *guess the encoding* of a *byte string*

Perl compatible regular expressions (PCRE) have an `u` flag ("PCRE8") to process byte strings as UTF-8 encoded strings.

PHP also includes a binding for the *iconv* library.

- `iconv()`: *decode* a *byte string* from an encoding and *encode* to another encoding, you can use `//IGNORE` or `//TRANSLIT` suffix to choose the *error handler*
- `iconv_mime_decode()`: decode a MIME header field

PHP 6 was a project to improve Unicode support of Unicode. This project died at the beginning of 2010. Read The Death of PHP 6/The Future of PHP 6 (May 25, 2010 by Larry Ullman) and Future of PHP6 (March 2010 by Johannes Schlüter) for more information.

Perl

Write a character using its code point written in hexadecimal:

- `chr(0x1F4A9)`
- `"\x{2639}"`
- `"\N{U+A0}"`

Using `use charnames qw( :full );`, you can use a Unicode character in a string using `"\N{name}"` syntax. Example:

```
say "\N{long s} \N{ae} \N{Omega} \N{omega} \N{UPWARDS ARROW}"
```

Declare that filehandles opened within this lexical scope but not elsewhere are in UTF-8, until and unless you say otherwise. The `:std` adds in `STDIN`, `STDOUT`, and `STDERR`. This critical step implicitly decodes incoming data and encodes outgoing data as UTF-8:

```
use open qw( :encoding(UTF-8) :std );
```

If `PERL_UNICODE` environment variable is set to `AS`, the following data will use UTF-8:

- `@ARGV`
- `STDIN`, `STDOUT`, `STDERR`

If you have a `DATA` handle, you must explicitly set its encoding. If you want this to be UTF-8, then say:

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

Misc:

```
use feature qw< unicode_strings >;
use Unicode::Normalize qw< NFD NFC >;
use Encode qw< encode decode >;
@ARGV = map { decode("UTF-8", $_) } @ARGV;
open(OUTPUT, "> :raw :encoding(UTF-16LE) :crlf", $filename);
```

Misc:

- Encode
- Unicode::Normalize
- Unicode::Collate

- Unicode::Collate::Locale
- Unicode::UCD
- DBM_Filter::utf8

History:

- Perl 5.6 (2000): initial Unicode support, support *character strings*
- Perl 5.8 (2002): regex supports Unicode
- use "`use utf8;`" pragma to specify that your Perl script is encoded to *UTF-8*

Read `perluniintro`, `perlunicode` and `perlunifaq` manuals.

See Tom Christiansen's Materials for OSCON 2011 for more information.

Java

`char` is a character able to store Unicode *BMP* only characters (U+0000—U+FFFF), whereas `Character` is a wrapper of the `char` with static helper functions. `Character` methods:

- `.getType(ch)`: get the *category* of a character
- `.isWhitespace(ch)`: test if a character is a whitespace according to Java
- `.toUpperCase(ch)`: convert to uppercase
- `.codePointAt(CharSequence, int)`: return the code point at the given index of the CharSequence

`String` is a *character string* implemented using a `char` array and *UTF-16*. `String` methods:

- `String(bytes, encoding)`: *decode* a *byte string* from the specified encoding. The decoder is *strict*: throw a `CharsetDecoder` exception if a *byte sequence cannot be decoded*.
- `.getBytes(encoding)`: *encode* to the specified encoding, throw a `CharsetEncoder` exception if a character *cannot be encoded*.
- `.length()`: get the length in UTF-16 units.

As *Python* compiled in narrow mode, *non-BMP* characters are stored as *UTF-16 surrogate pairs* and the length of a string is the number of UTF-16 units, not the number of Unicode characters.

Java, as the Tcl language, uses a variant of *UTF-8* which encodes the nul character (U+0000) as the *overlong byte sequence* `0xC0 0x80`, instead of `0x00`. So it is possible to use *C* functions like `strlen()` on *byte string* with embeded nul characters.

Go and D

The Go and D languages use *UTF-8* as internal encoding to store *Unicode strings*.

CHAPTER 12

Database systems

MySQL

MySQL supports two UTF-8 variants: * utf8mb4: This is the full UTF-8 character set supported since MySQL 5.5 * utf8: Also known as utf8mb3. This only supports the Basic Multilingual Plane of Unicode 3.0 and doesn't support 4-byte characters.

https://dev.mysql.com/doc/refman/5.7/en/charset-unicode.html

In MySQL a character set is used on a per-column basis. A default characater set for new columns is set on a table level. And the default for tables is set on a database level.

PostgreSQL

Unicode support is set on database level. There is a cluster level default.

To create a database with UTF-8 support: *createdb -E utf8*

To convert a non-unicode database to UTF-8 the recommended method is a dump/restore.

http://www.postgresql.org/docs/9.5/static/charset.html

SQLite

CHAPTER 13

Libraries

Programming languages have no or basic support of Unicode. Libraries are required to get a full support of Unicode on all platforms.

Qt library

Qt is a big *C++* library covering different topics, but it is typically used to create graphical interfaces. It is distributed under the GNU LGPL license (version 2.1), and is also available under a commercial license.

Character and string classes

`QChar` is a Unicode character, only able to store *BMP characters*. It is implemented using a 16 bits unsigned number. Interesting `QChar` methods:

- `isSpace()`: True if the *character category* is separator (Zl, Zp or Zs)
- `toUpper()`: convert to upper case

`QString` is a *character string* implemented as an array of `QChar` using *UTF-16*. A *Non-BMP character* is stored as two `QChar` (a *surrogate pair*). Interesting `QString` methods:

- `toAscii()`, `fromAscii()`: encode to/decode from *ASCII*
- `toLatin1()`, `fromLatin1()`: encode to/decode from *ISO 8859-1*
- `utf16()`, `fromUtf16()`: encode to/decode to *UTF-16* (in the host endian)
- `normalized()`: *normalize* to NFC, NFD, NFKC or NFKD

Qt *decodes* literal byte strings from *ISO 8859-1* using the `QLatin1String` class, a thin wrapper to `char*`. `QLatin1String` is a character string storing each character as a single byte. It is possible because it only supports characters in U+0000—U+00FF range. `QLatin1String` cannot be used to manipulate text, it has a smaller API than `QString`. For example, it is not possible to concatenate two `QLatin1String` strings.

Codec

`QTextCodec.codecForLocale()` gets the locale encoding codec:

- Windows: *ANSI code page*
- Otherwise: the *locale encoding*. Try `nl_langinfo(CODESET)`, or `LC_ALL`, `LC_CTYPE`, `LANG` environment variables. If no one gives any useful information, fallback to *ISO 8859-1*.

Filesystem

`QFile.encodeName()`:

- *Windows*: encode to *UTF-16*
- *Mac OS X*: *normalize* to the D form and then encode to *UTF-8*
- Other (UNIX/BSD): encode to the *local encoding* (`QTextCodec.codecForLocale()`)

`QFile.decodeName()` is the reverse operation.

Qt has two implementations of its `QFSFileEngine`:

- Windows: use Windows native API
- UNIX: use POSIX API. Examples: `fopen()`, `getcwd()` or `get_current_dir_name()`, `mkdir()`, etc.

Related classes: `QFile`, `QFileInfo`, `QAbstractFileEngineHandler`, `QFSFileEngine`.

The glib library

The glib library is a great *C* library distributed under the GNU LGPL license (version 2.1).

Character strings

The `gunichar` type is a character. It is able to store any Unicode 6.0 character (U+0000—U+10FFFF).

The glib library has no *character string* type. It uses *byte strings* using the `gchar*` type, but most functions use *UTF-8* encoded strings.

Codec functions

- `g_convert()`: *decode* from an encoding and *encode* to another encoding with the *iconv library*. Use `g_convert_with_fallback()` to choose *how to handle undecodable bytes* and *unencodable characters*.
- `g_locale_from_utf8()` / `g_locale_to_utf8()`: encode to/decode from the current *locale encoding*.
- `g_get_charset()`: get the locale encoding
  - Windows: current *ANSI code page*
  - OS/2: current code page (call `DosQueryCp()`)
  - other: try `nl_langinfo(CODESET)`, or `LC_ALL`, `LC_CTYPE` or `LANG` environment variables
- `g_utf8_get_char()`: get the first character of an UTF-8 string as `gunichar`

Filename functions

- `g_filename_from_utf8()` / `g_filename_to_utf8()`: *encode*/*decode* a filename to/from UTF-8
- `g_filename_display_name()`: human readable version of a filename. Try to decode the filename from each encoding of `g_get_filename_charsets()` encoding list. If all decoding failed, decode the filename from *UTF-8* and *replace undecodable bytes* by (U+FFFD).
- `g_get_filename_charsets()`: get the list of charsets used to decode and encode filenames. `g_filename_display_name()` tries each encoding of this list, other functions just use the first encoding. Use *UTF-8* on *Windows*. On other operating systems, use:
  - `G_FILENAME_ENCODING` environment variable (if set): comma-separated list of character set names, the special token `"@locale"` is taken to mean the *locale encoding*
  - or UTF-8 if `G_BROKEN_FILENAMES` environment variable is set
  - or call `g_get_charset()` (the *locale encoding*)

iconv library

libiconv is a library to encode and decode text in different encodings. It is distributed under the GNU LGPL license. It supports a lot of encodings including rare and old encodings.

By default, libiconv is *strict*: an *unencodable character* raise an error. You can *ignore* these characters by adding the `//IGNORE` suffix to the encoding name. There is also the `//TRANSLIT` suffix to *replace unencodable characters* by similarly looking characters.

PHP has a builtin binding of iconv.

ICU libraries

International Components for Unicode (ICU) is a mature, widely used set of *C*, *C++* and *Java* libraries providing Unicode and Globalization support for software applications. ICU is an open source project distributed under the MIT license.

libunistring

libunistring provides functions for manipulating Unicode strings and for manipulating C strings according to the Unicode standard. It is distributed under the GNU LGPL license version 3.

CHAPTER 14

Unicode issues

Security vulnerabilities

Special characters

Fullwidth (U+FF01—U+FF60) and halfwidth (U+FF61—U+FFEE) characters have been used in 2007 to bypass security checks. Examples with the *Unicode normalization*:

- U+FF0E is normalized to . (U+002E) in NFKC
- U+FF0F is normalized to / (U+002F) in NFKC

Some important characters have also "alternatives" in Unicode:

- Windows directory separator, \ (U+005C): U+20E5, U+FF3C
- UNIX directory separator, / (U+002F): U+2215, U+FF0F
- Parent directory, .. (U+002E, U+002E): U+FF0E

For more information, read GS07-01 Full-Width and Half-Width Unicode Encoding IDS/IPS/WAF Bypass Vulnerability (GamaTEAM, april 2007).

Non-strict UTF-8 decoder: overlong byte sequences and surrogates

An *UTF-8 decoder* has to reject overlong byte sequences, or an attacker can use them to bypass security checks (e.g. check rejecting string containing nul bytes, `0x00`). For example, `0xC0 0x80` byte sequence must raise an error and not be decoded as U+0000, and "." (U+002E) can be encoded to `0xC0 0xAE` (two bytes instead of one) to bypass directory traversal checks.

Surrogates characters are also invalid in UTF-8: characters in U+D800—U+DFFF have to be rejected. See the table 3-7 in the Conformance chapter of the Unicode standard (december 2009); and the section 3 (UTF-8 definition) of UTF-8, a transformation format of ISO 10646 (RFC 3629, november 2003).

The libxml2 library had such vulnerability until january 2008: CVE-2007-6284.

Some PHP functions use a strict UTF-8 decoder (e.g. `mb_convert_encoding()`), some other don't. For example, `utf8_decode()` and `mb_strlen()` accept `0xC0 0x80` in PHP 5.3.2. The UTF-8 decoder of Python 3 is strict, whereas the UTF-8 decoder of Python 2 accepts surrogates (to keep the backward compatibility). In Python 3, the error handler `surrogatepass` can be used to encode and decode surrogates.

Note: The *Java* and Tcl languages use a variant of *UTF-8* which encodes the nul character (U+0000) as the overlong byte sequence `0xC0 0x80`, instead of `0x00`, for practical reasons.

Check byte strings before decoding them to character strings

Some applications check user inputs as *byte strings*, but then process them as *character strings*. This vulnerability can be used to bypass security checks.

The WordPress blog tool had such issue with *PHP5* and MySQL: WordPress Charset SQL Injection Vulnerability (Abel Cheung, december 2007). WordPress used the PHP function `addslashes()` on the input byte strings. This function adds `0x5C` prefix to `0x00`, `0x22`, `0x27` and `0x5C` bytes. If a input string is encoded to *ISO 8859-1*, this operation escapes a quote: `'` (U+0027) becomes `\'` ({U+005C, U+0027}).

The problem is that `addslashes()` process byte strings, whereas the result is used by MySQL which process character strings. Example with *Big5* encoding: `0xB5 0x27` *cannot be decoded* from Big5, but escaped it becomes `0xB5 0x5C 0x27` which is decoded to {U+8A31, U+0027}. The `0x5C` byte is no more a backslash: it is part of the multibyte character U+8A31 encoded to `0xB5 0x5C`. The solution is to use `mysql_real_escape_string()` function, instead of `addslashes()`, which process inputs as character strings using the MySQL connection encoding.

See also:

CVE-2006-2314 (PostgreSQL, may 2006), CVE-2006-2753 (MySQL, may 2006) and CVE-2008-2384 (libapache2-mod-auth-mysql, january 2009).

CHAPTER 15

See also

- UTF-8 and Unicode FAQ for Unix/Linux by Markus Kuhn, first version in june 1999, last edit in may 2009

Index