Lecture 3: Input/output Encoding

PCL II, CL, UZH

March 09, 2016

Universität Zürich UZH

Outline

- Part 1: General Part
  - files and folders
  - how does a file look on the inside?
  - hexadecimal and binary numeral systems
  - bits, bytes
  - file formats
  - text encodings
  - unicode
- Part 2: Python Part
  - i/o functionality: opening, closing, reading, writing
  - encoding
  - unicode support

Part 1: General Part

Files and Folders

are what?

Files and Folders

- File: a block of arbitrary **information**, usually based on some kind of **durable storage**
- Folder: a group of files and/or other folders

Storing Information

- Information
  - text
  - **pretty** *text*
  -
  - sound
  - video
  - recipe of a perfect Bolognese sauce
  - secret plans to blow up the Kremlin
- Durable storage: HDD/DVD/SSD/...

Computer Storage

Universität Zürich^UZH

- Bit (**bi**nary digi**t**), value of either 0 or 1
- Easy to implement electronically:
  - punchcard sector **punched** (1) / **not** (0)
  - tape/disk/... sector **magnetized** (1) / **not** (0)
  - CD / DVD track section is a **bump** (1) / **land** (0)
  - flash memory gate electrode **open** (1) / **closed** (0)

- **All information on the computer is binary !**

Binary representation

- 1 bit can have one of **two** values
  - `0, 1`
- 2 bits -- **four** values:
  - `00, 01, 10, 11`
- 3 bits -- **eight** values:
  - `000, 001, 010, 011, 100, 101, 110, 111`
- ...
- N bits -- **2^N** values

Binary representation

- How many bits for a single (latin) letter?

Binary representation

- How many bits for a single (latin) letter?
  - how many letters are there?

Binary representation

- How many bits for a single (latin) letter?
  - 26 letters fit into 5 bits, which have 32 possible values:
    - `00000`: a
    - `00001`: b
    - `00010`: c
    - ...
    - `10111`: x
    - `11000`: y
    - `11001`: z
    - `11010`, `11011`, `11100`, `11101`, `11110`, `11111` are left over

Binary representation

- How many bits for a single (latin) letter?
  - 26 letters fit into 5 bits, which have 32 possible values:
    - ...
  - upper+lower case: ? bits

Binary representation

- How many bits for a single (latin) letter?
  - 26 letters fit into 5 bits, which have 32 possible values:
    - ...
  - upper+lower case: 6 bits
  - 26 + 26 = 52 different letters < 64 = 2^6: 6 bits

Binary representation

- How many bits for a single (latin) letter?
    - 26 letters fit into 5 bits, which have 32 possible values:
        - ...
    - upper+lower case: 6 bits
    - 26 + 26 = 52 different letters < 64 = 2^6: 6 bits
    - (any) 5-letter word: 30 bits

Units of digital information

- **Byte**:
    - Nowadays: **8 bits**
    - Historically: number of bits to used to encode one character
    - Symbol: B
    - fits in upper/lower-case letters, numbers, punctuation, etc.
    - can represent 256 distinct values (=2^8)

- 4 bytes of information:
    - `01101000 01111110 10101100 01101110`

Binary representation

- How many bits for a natural number?
    - between 1 and 256?

Binary representation

- How many bits for a natural number?
  - between 1 and 256?
    - encoded as text:
    - encoded as bits:

Binary representation

- How many bits for a natural number?
  - between 1 and 256?
    - encoded as text: 3 bytes (max "2", "5", "6"), 24 bits
    - encoded as bits: 8 bits / 1 byte

Binary representation

- How many bits for a natural number?
  - between 1 and 256?
    - encoded as text: 3 bytes (max "2", "5", "6"), 24 bits
    - encoded as bits: 8 bits / 1 byte
  - between -2 billion and +2 billion?
    - as text:

Binary representation

- How many bits for a natural number?
  - between 1 and 256?
    - encoded as text: 3 bytes (max "2", "5", "6"), 24 bits
    - encoded as bits: 8 bits ($\log_2 256$) / 1 byte
  - between -2 billion and +2 billion?
    - as text: up to 11 bytes / 88 bits
    - encoded as bits: 32 bits / 4 bytes (4 billion values)
    - more intuitive representation:
      - 1 bit for the sign (- / +)
      - 31 bits for the absolute value, 0 to 2 billion

Decimal numeral system

- Also "base-10"
- 10 digits: 0, 1, ..., 9
- positional notation
- 5203 =

  3 2 1 0

  $= 5000 + 200 + 3 =$

  $= 5 * 1000 + 2 * 100 + 0 * 10 + 3 =$

  $= 5 * 10^3 + 2 * 10^2 + 0 * 10^1 + 3 * 10^0$

Binary numeral system

- Also "base-2"
- 2 digits: 0, 1
- positional notation
- $\underset{3}{1}\underset{2}{0}\underset{1}{1}\underset{0}{1}$ (binary) =

$$= 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 =$$

11

Hexadecimal numeral system

- Also "base-16"
- 1 digits: 0, 1, ..., 9, A, B, C, D, E, F

  10 11 12 13 14 15

- positional notation
- 3D58 (hex) =

  3 2 1 0

$$= 3 * 16^3 + 13 * 16^2 + 5 * 16^1 + 8 * 16^0 =$$

15704

Hexadecimal numeral system

- Also "base-16"
- 1 digits: 0, 1, ..., 9, A, B, C, D, E, F
  10 11 12 13 14 15

Why Hex?

- Better readable/understandable for humans than binary
- Better convertible from/into binary than decimal

Binary to hex conversion

- Conversion to decimal involves a lot of summing over the whole number
    - 1 + 0 * 2 + 1 * 4 + 1 * 8 + 0 * 16 + 1 * 32 + 0 * 64 +...
- Conversion to hex?
    - bin to dec, dec to hex -- tiresome

Binary to hex conversion

- Conversion to decimal involves a lot of summing over the whole number
  - 1 + 0 * 2 + 1 * 4 + 1 * 8 + 0 * 16 + 1 * 32 + 0 * 64 +...
- Conversion to hex?
  - bin to dec, dec to hex -- tiresome
  - $16 = 2^4$

Binary to hex conversion

- Conversion to decimal involves a lot of summing over the whole number
  - 1 + 0 * 2 + 1 * 4 + 1 * 8 + 0 * 16 + 1 * 32 + 0 * 64 +...
- Conversion to hex?
  - bin to dec, dec to hex -- tiresome
  - $16 = 2^4$
  - **4 bits** = 4-digit binary number =
    = 16 values = 1-digit hexadecimal number;
    1 hex digit

Units of digital information

- Byte = 8 bits = 2-digit hex number
  - `00000000`, `00000001`, ..., `11111111` binary
  - `00`, `01`, ..., `09`, `0A`, ..., `0F`, `10`, `11`, ..., `FF` hex
  - `00`, `01`, ..., `09`, `10`, ..., `254`, `255` decimal
  - different values: $256 = 2^8 = 16^2$
- 4 bytes of information
  - `01101000 01111110 10101100 01101110` binary
  - hex ?

Units of digital information

- Byte = 8 bits = 2-digit hex number
  - `00000000, 00000001, ..., 11111111` binary
  - `00, 01, ..., 09, 0A, ..., 0F, 10, 11, ..., FF` hex
  - `00, 01, ..., 09, 10, ..., 254, 255` decimal
  - different values: $256 = 2^8 = 16^2$
- 4 bytes of information
  - `01101000 01111110 10101100 01101110` binary
  - `68 7E AC 6E` hex

Files

- A file is a block of arbitrary information
  - Represented with a list of bits
  - Easier to see as a list of bytes (same thing)

File formats

- A file is a block of arbitrary information
- In most cases it is **organized** information
- File format = an agreement on how the information of the file is organized/encoded
- e.g.:
  - text file: every byte represents a text character
  - the whole file represents the whole text

File formats

- A file is a block of arbitrary information
- In most cases it is **organized** information
- File format = an agreement on how the information of the file is organized/encoded
- e.g.:
  - text file: every byte represents a text character
  - the whole file represents the whole text

  - **Q:** which byte represents which character?
  - **Q:** how about line breaks, tabs?

File formats

- A file is a block of arbitrary information
- In most cases it is **organized** information
- File format = an agreement on how the information of the file is organized/encoded
- e.g.:
    - text file: every byte represents a text character
    - the whole file represents the whole text

    - **Q:** which byte represents which character?
    - **Q:** how about line breaks, tabs?
    - **A: character-encoding scheme**

Encoding

- Character-encoding scheme
  - short: encoding
- Sort of a file format for text files
- Includes an agreement on which byte (in binary, hex, or whatever) represents which text character
  - codepage / encoding table

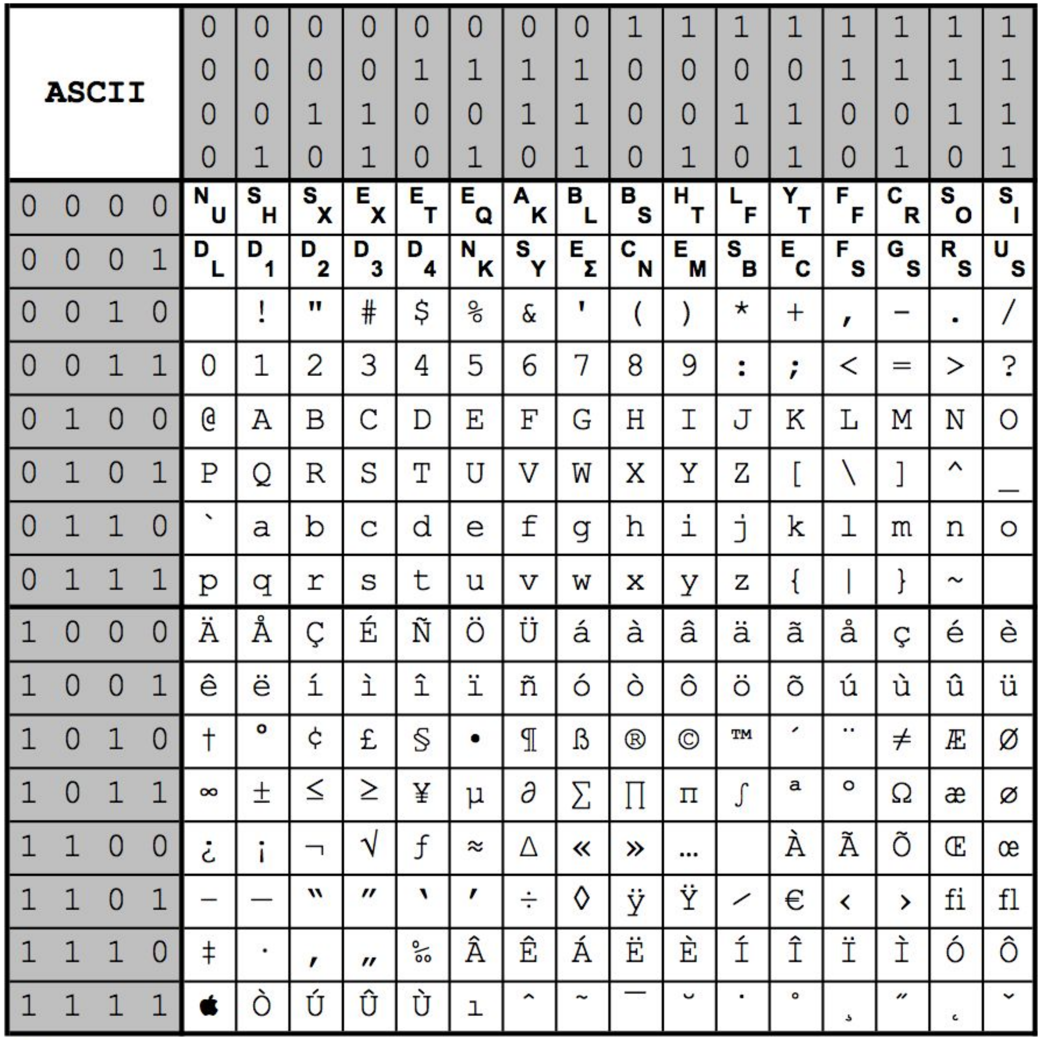
ASCII

- ASCII = American Standard Code for Information Interchange
- Pronounced “ass-kee”
- First published: 1963
- Developed from telegraph codes
- Used to be most common character scheme on the WWW (UTF-8 since 2007)

ASCII

- Encoding for English and several special characters
- 1 byte (= 8 bits) = 1 text character
- characters encoded: 128
  - 95 printable characters (including space)
  - 33 non-printable (control) characters
    - originated with teletype machines, many now obsolete
- Encoding table/ASCII chart
  - ASCII only uses 7 bits = 128 values, the other 128 can be used for extra characters (=extended ASCII)

Binary ASCII Chart



ASCII	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0 0 0 0	NU	SH	SX	EX	ET	EQ	AK	BL	BS	HT	LF	YT	FF	CR	SO	SI
0 0 0 1	DL	D1	D2	D3	D4	NK	SY	EΣ	CN	EM	SB	EC	FS	GS	RS	US
0 0 1 0		!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
0 0 1 1	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0 1 0 0	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0 1 0 1	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0 1 1 0	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0 1 1 1	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
1 0 0 0	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
1 0 0 1	ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
1 0 1 0	†	°	¢	£	§	•	¶	ß	®	©	™	´	¨	≠	Æ	Ø
1 0 1 1	∞	±	≤	≥	¥	µ	∂	∑	∏	π	∫	ª	º	Ω	æ	ø
1 1 0 0	¿	¡	¬	√	ƒ	≈	∆	«	»	…		À	Ã	Õ	Œ	œ
1 1 0 1	–	—	“	”	‘	’	÷	◊	ÿ	Ÿ	⁄	€	‹	›	ﬁ	ﬂ
1 1 1 0	‡	·	‚	„	‰	Â	Ê	Á	Ë	È	Í	Î	Ï	Ì	Ó	Ô
1 1 1 1		Ò	Ú	Û	Ù	ı	ˆ	˜	¯	˘	˙	˚	¸	˝	˛	ˇ

Standard ASCII Character set (7 bits)

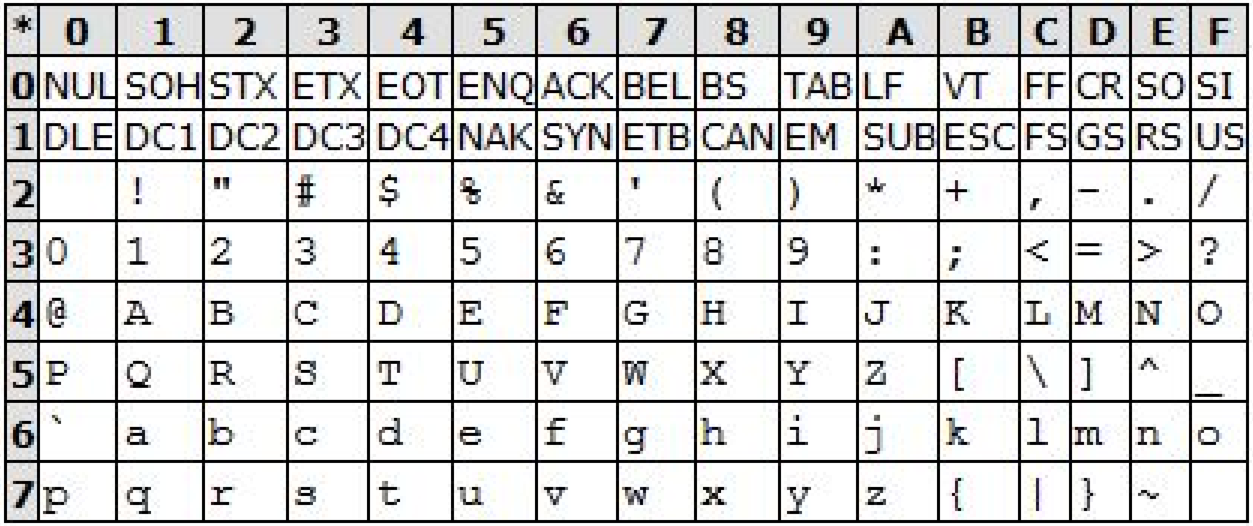
Extended ASCII Character set (8 bits)

↑

This bit specifies whether the character is in the top half of the table (Standard ASCII) or the bottom half (Extended ASCII).

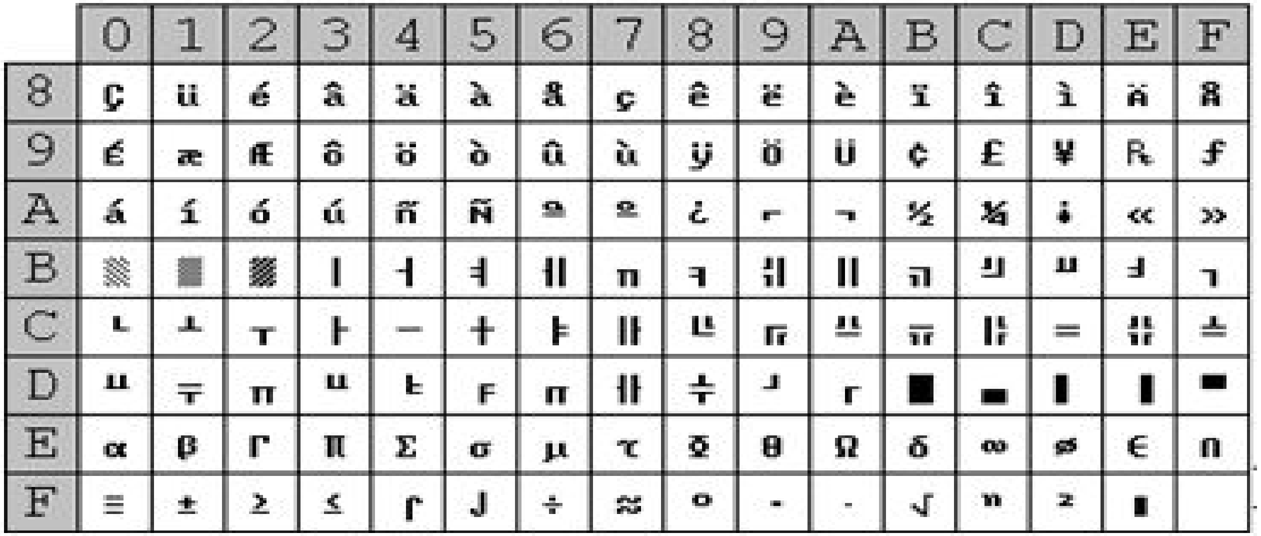
Source: https://canvas.instructure.com/courses/884561/pages/rep-introduction-to-alphanumeric-representation

Hexadecimal ASCII Chart



*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
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	{	|	}	~	

- standard ASCII

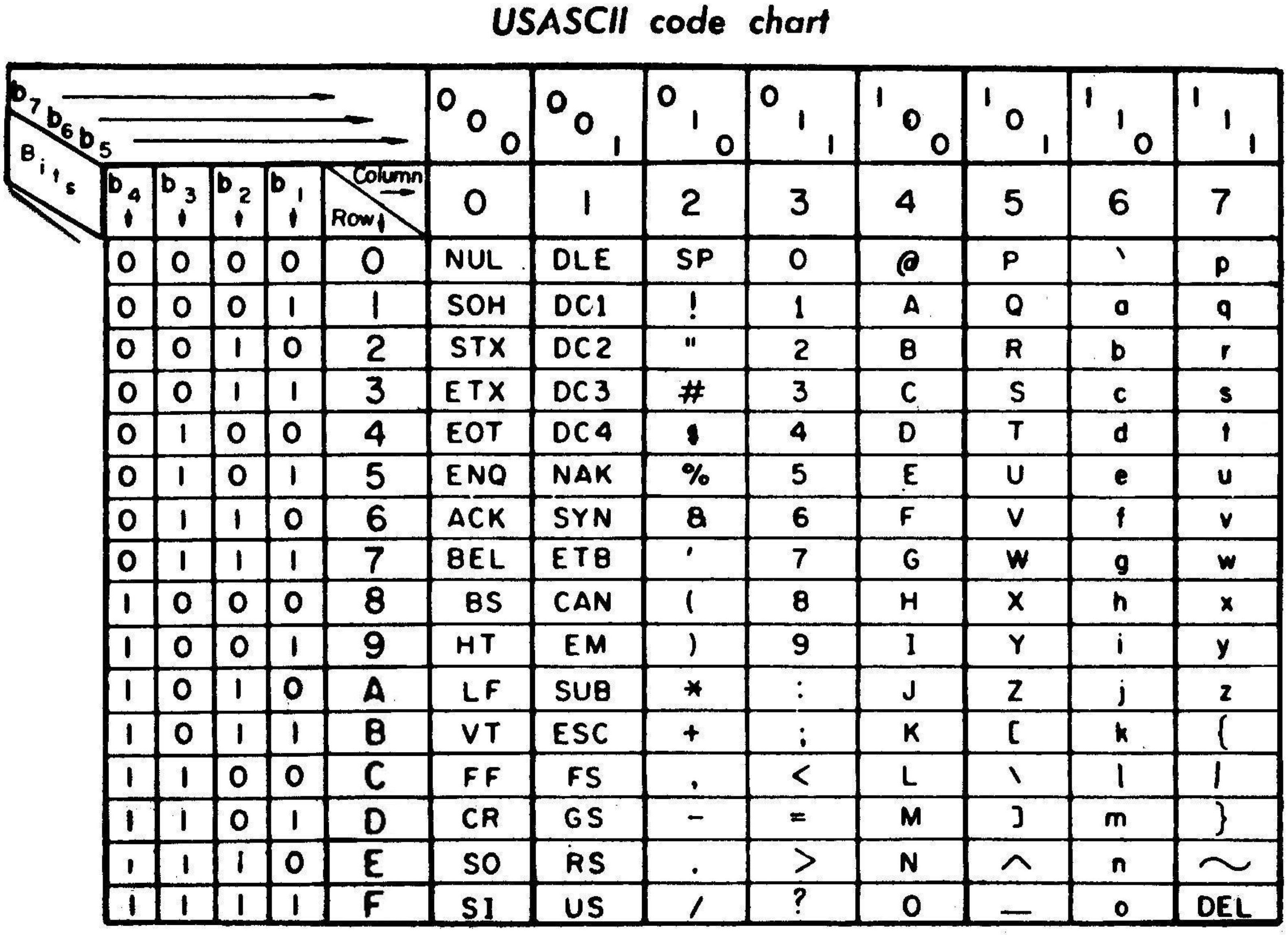


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

- extended ASCII

here:
OEM extended code ASCII

Source: http://www.maxi-pedia.com/code+ASCII



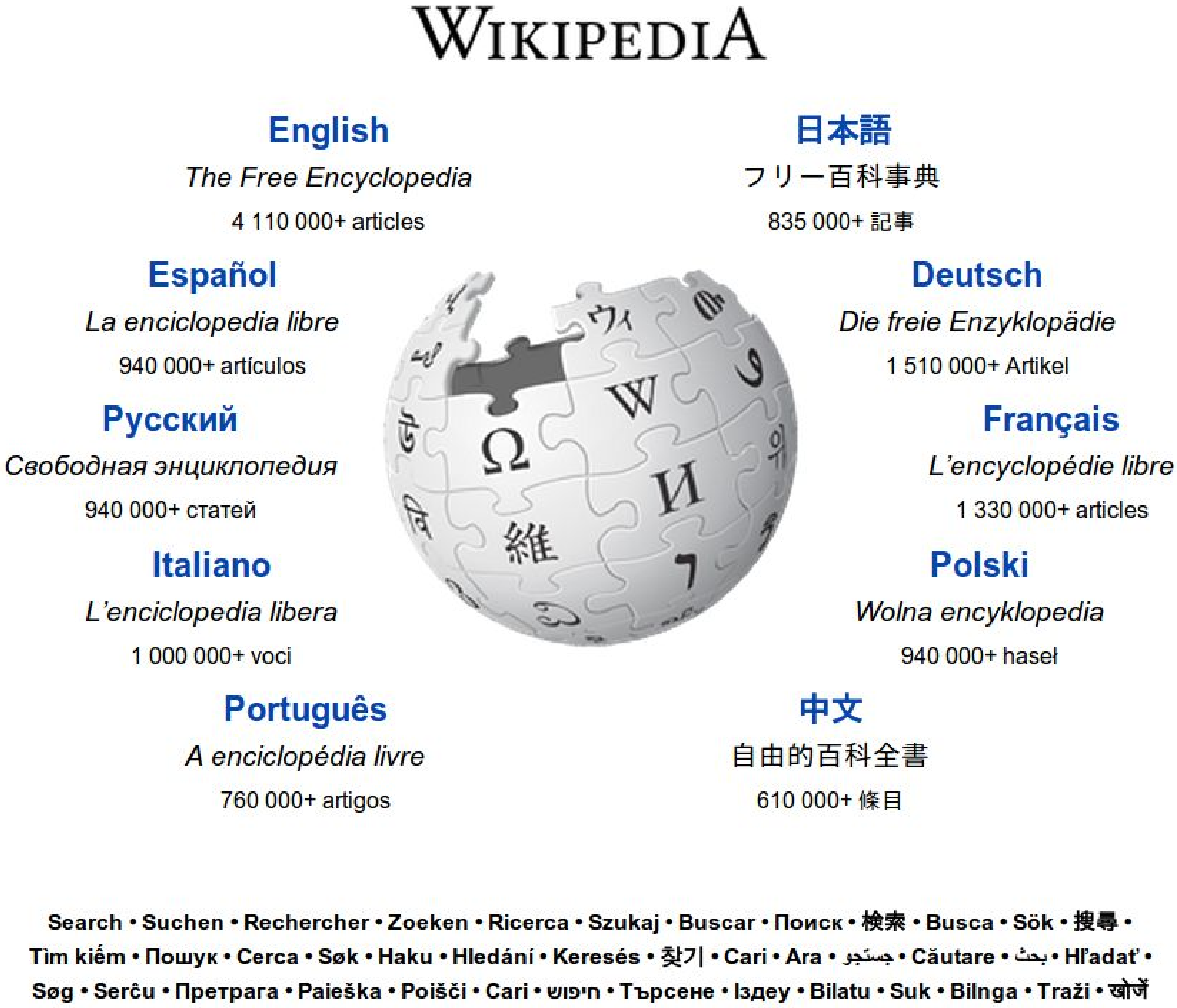
USASCII code chart

b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b4 ↓	b3 ↓	b2 ↓	b1 ↓	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
1	0	1	1	B	VT	ESC	+	;	K	[	k	{
1	1	0	0	C	FF	FS	,	<	L	\	l	|
1	1	0	1	D	CR	GS	−	=	M	]	m	}
1	1	1	0	E	SO	RS	.	>	N	^	n	~
1	1	1	1	F	SI	US	/	?	O	_	o	DEL

Source: http://en.wikipedia.org/wiki/ASCII

Demo

```
> cat song.txt
Song
    Well here we are again
    It's always such a pleasure
    Remember how you tried to kill me
    twice?
    ...
> hexdump -C song.txt
53 6f 6e 67  0a 09 57 65  6c 6c 20 68  65 72 65 20  |Song..Well here |
77 65 20 61  72 65 20 61  67 61 69 6e  0a 09 49 74  |we are again..It|
27 73 20 61  6c 77 61 79  73 20 73 75  63 68 20 61  |'s always such a|
20 70 6c 65  61 73 75 72  65 0a 09 52  65 6d 65 6d  | pleasure..Remem|
62 65 72 20  68 6f 77 20  79 6f 75 20  74 72 69 65  |ber how you trie|
64 20 74 6f  20 6b 69 6c  6c 20 6d 65  0a 09 74 77  |d to kill me..tw|
69 63 65 3f  0a 09 2e 2e  2e 0a                     |ice?......|
```



WIKIPEDIA

English
The Free Encyclopedia
4 110 000+ articles

日本語
フリー百科事典
835 000+ 記事

Español
La enciclopedia libre
940 000+ artículos

Deutsch
Die freie Enzyklopädie
1 510 000+ Artikel

Русский
Свободная энциклопедия
940 000+ статей

Français
L’encyclopédie libre
1 330 000+ articles

Italiano
L’enciclopedia libera
1 000 000+ voci

Polski
Wolna encyklopedia
940 000+ haseł

Português
A enciclopédia livre
760 000+ artigos

中文
自由的百科全書
610 000+ 條目

**Search • Suchen • Rechercher • Zoeken • Ricerca • Szukaj • Buscar • Поиск • 検索 • Busca • Sök • 搜尋 •
Tìm kiếm • Пошук • Cerca • Søk • Haku • Hledání • Keresés • 찾기 • Cari • Ara • جستجو • Căutare • بحث • Hľadať •
Søg • Serĉu • Претрага • Paieška • Poišči • Cari • חיפוש • Търсене • Іздеу • Bilatu • Suk • Bilnga • Traži • खोजें**

Extended ASCII

- Mostly 8 bit character encodings
  - “eight-bit extended ASCII codes”
- includes 7-bit standard ASCII characters
  - the first part (bytes `00`, ..., `7F`) same as ASCII
  - the second part (bytes `80`, ..., `FF`) used for extra characters
    - latin letters with diacritics, cyrillic, arabic, ...

Extended ASCII

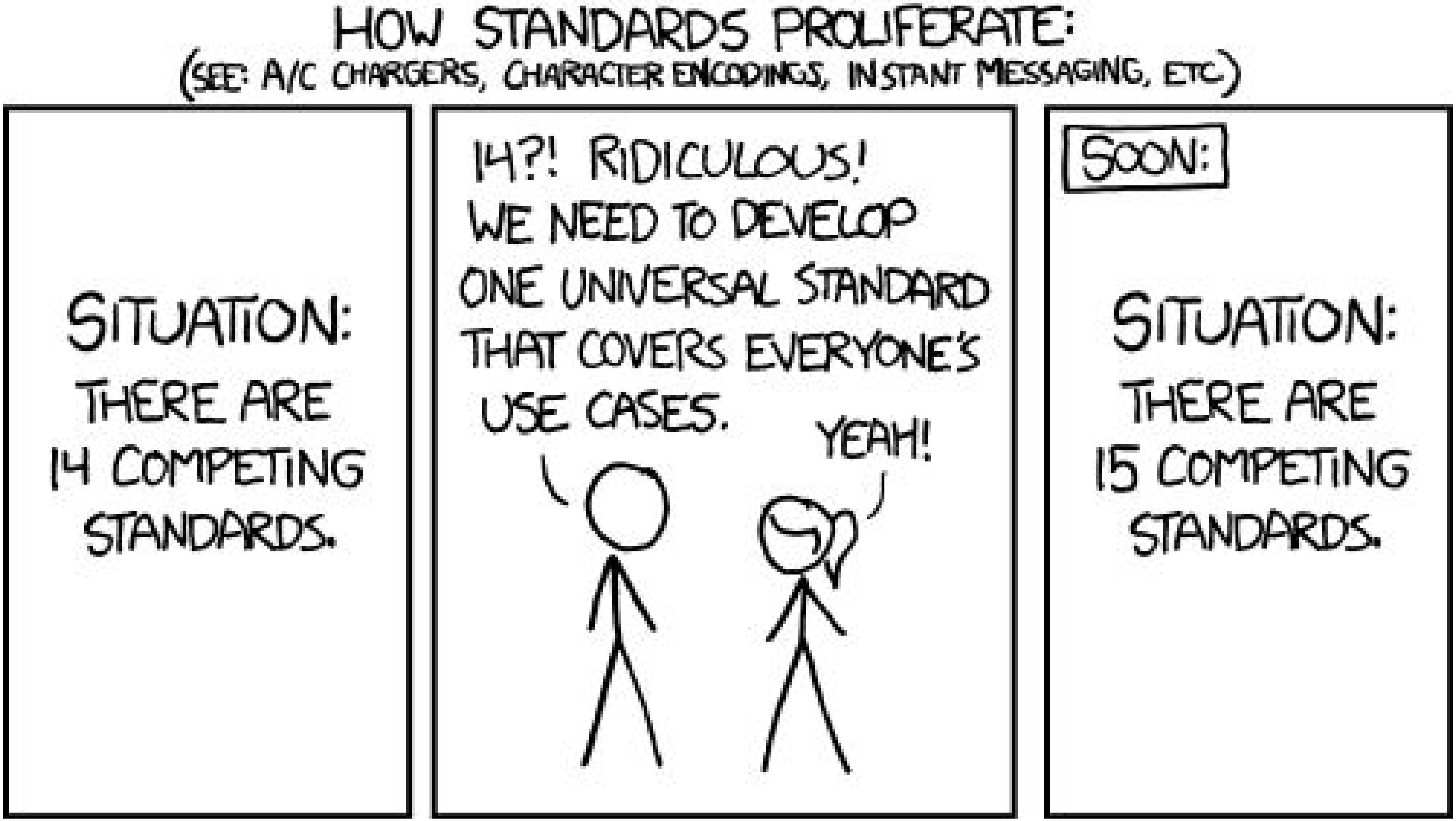
- **Common extended ASCII encodings:**
  - KOI8-R
  - BIG-5
  - Mac-Roman
  - ISO/IEC 8859 series encodings
    - 16 series of 8-bit character encodings:
      - ISO-8859-1: Latin, Western European
      - ISO-8859-2: Latin, Central European
      - ...
- There are many incompatible extensions to ASCII
  - many local variants
  - encoding of specific text needs to be known
  - possible problems exchanging files
  - what about “mixed texts”?

	ASCII	ISO-8859-15 (latin-9)	CP-1252 (Windows 1252)	UTF-8
a	01100001	01100001	01100001	01100001
€	NA	10100100	10000000	11100010 10000010 10101100
¤	NA	NA	10100100	11000010 10100100

Source: Travis Fischer, Ester Nam: How to (╯°□°)╯︵ ┻━┻ with dignity (Pycon 2014)

Encoding Standards

- Q: one encoding to rule them all?


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Source: http://xkcd.com/927/

Unicode

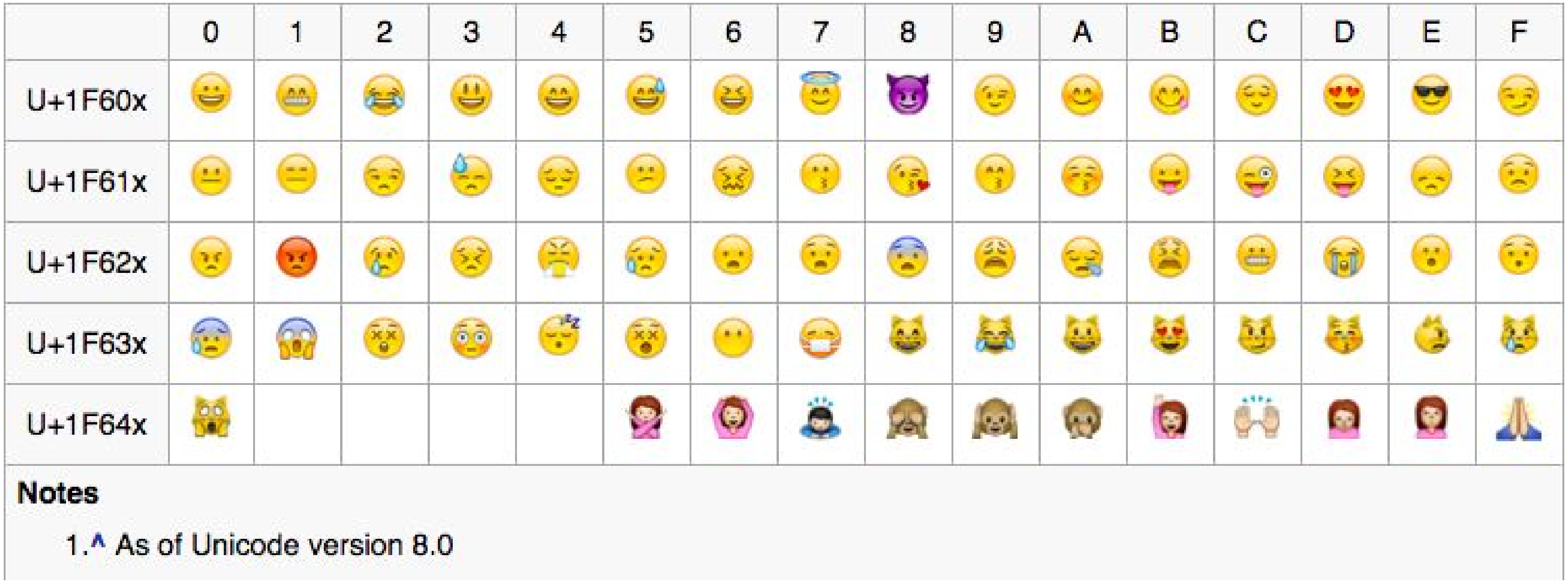
- "The Unicode Standard"/ "Meta-encoding"
  - a number of requirements to encodings
- supports all the characters one can think of in one encoding
    - more than 120,000 characters
    - supporting 129 modern and historic scripts + multiple symbol sets
  - developed by the Unicode Consortium (Unicode Inc.), California
- Most recent: Unicode 8.0 (2015)
- implemented by various encodings:
    - *UTF-1*, *UTF-7*, UTF-8, *UTF-EBCDIC*, UTF-16, UTF-32, ...

Unicode

- "Meta-encoding" or an encoding standard
  (not *Zeichenkodierung*, but *Zeichenkodierungsstandard*)
  - describes how characters are presented by code points
    - code point: hexadecimal integer value
      - one code point per character
      - represents characters in an abstract way
        - e.g. U+2746 (U + hexadecimal number)
      - range U+0000..U+10FFFF

- Unicode code points can be encoded according to defined format
  - UTF = “Unicode Transformation Format”
  - mapping from unicode code points to unique byte sequence

UTF-8



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+1F60x	😀	😁	😂	😃	😄	😅	😆	😇	😈	😉	😊	😋	😌	😍	😎	😏
U+1F61x	😐	😑	😒	😓	😔	😕	😖	😗	😘	😙	😚	😛	😜	😝	😞	😟
U+1F62x	😠	😡	😢	😣	😤	😥	😦	😧	😨	😩	😪	😫	😬	😭	😮	😯
U+1F63x	😰	😱	😲	😳	😴	😵	😶	😷	😸	😹	😺	😻	😼	😽	😾	😿
U+1F64x	🙀					🙅	🙆	🙇	🙈	🙉	🙊	🙋	🙌	🙍	🙎	🙏

Notes

1.^ As of Unicode version 8.0

Official Unicode Webpage:
http://www.unicode.org/

(All encoding charts can be found here)

Source: https://en.wikipedia.org/wiki/Emoticons_(Unicode_block)

UTF-32

- Unicode Transformation Format, 32 bits
- Fixed-length encoding
- 4 bytes for every character
    - how many values?

UTF-32

- Unicode Transformation Format, 32 bits
- Fixed-length encoding
  - 4 bytes (=32 bits) for every character
  - 2^{32} = 4 billion
- Direct presentation of the numerical value of a code point

UTF-32

- Unicode Transformation Format, 32 bits
- Fixed-length encoding
- 4 bytes for every character
  - 2^{32} = 4 billion
- Direct presentation of the numerical value of a code point
- Problem: space-inefficient
  - compared to ASCII, required memory/space is 4 times bigger
  - so are the files and strings encoded in UTF-32
  - many (or most) characters beyond ASCII are rarely used

UTF-32

- Unicode Transformation Format, 32 bits
- Fixed-length encoding
- 4 bytes for every character
  - 2^{32} = 4 billion
- Direct presentation of the numerical value of a code point
- Problem: space-inefficient
  - compared to ASCII, required memory/space is 4 times bigger
  - so are the files and strings encoded in UTF-32
  - many (or most) characters beyond ASCII are rarely used
- Answer: variable-length encoding

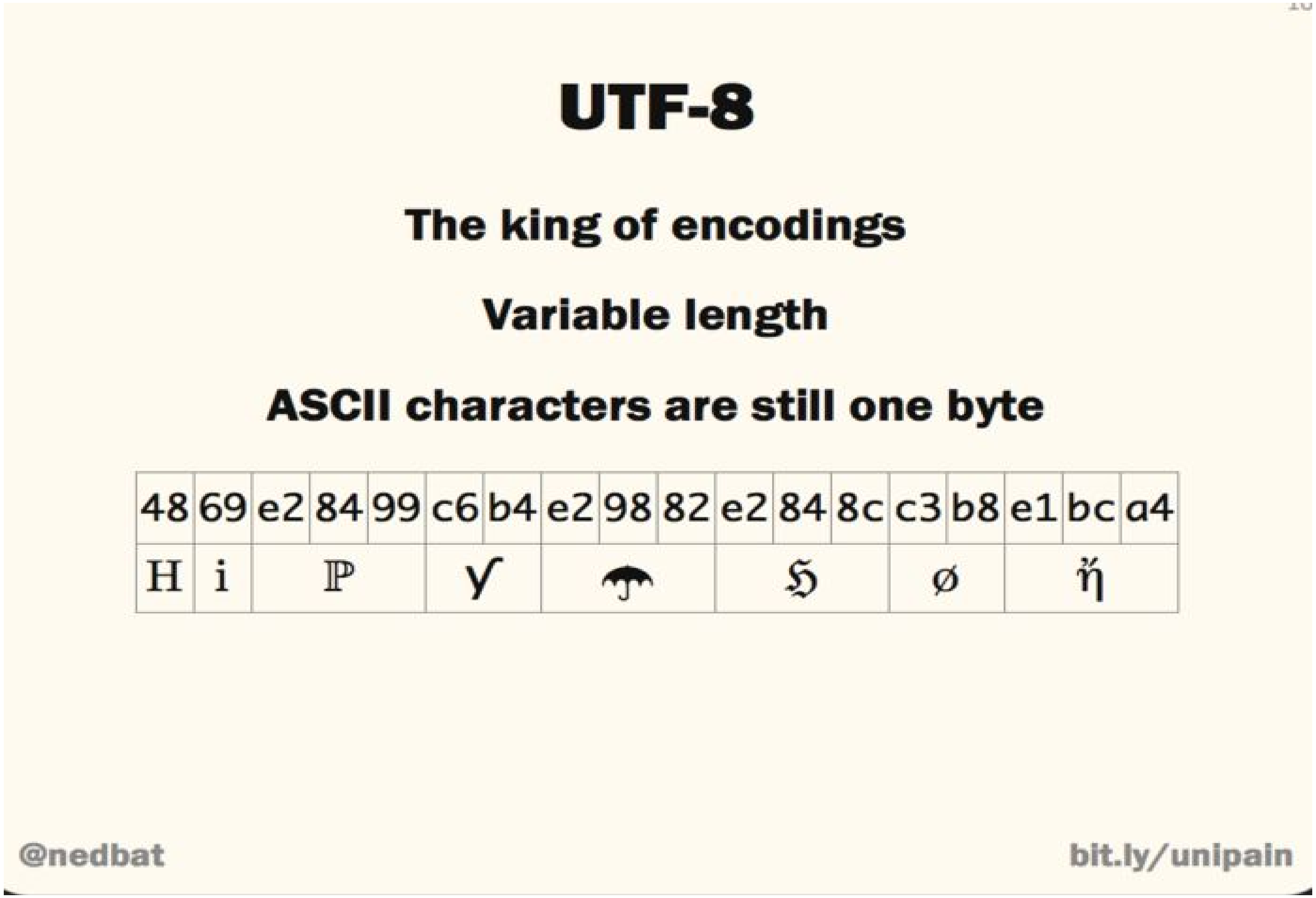
UTF-16

- Each *code point* represented with one or two 16-bit *code units*
- Variable-length encoding
- 2 bytes for more common characters
  - "BMP" = basic multilingual plane (up to 65k characters)
- 4 bytes for the rest
  - Supplementary planes (millions more characters)
- Still at least 2 times bigger memory/space requirements compared to ASCII
- Backwards compatible to UCS-2

UTF-8

- Dominant Character Encoding for the WWW since 2007
- Variable-length encoding
- Each character encoded with 1 to 6 bytes
- **Backward-compatible with ASCII**
  - i.e. characters that can be represented with ASCII are represented in UTF-8 in the same way as in ASCII
- How?
  - some bits used for encoding characters
  - some bits used to indicate whether this byte is the only one or last one or etc.

UTF-8



UTF-8

The king of encodings

Variable length

ASCII characters are still one byte

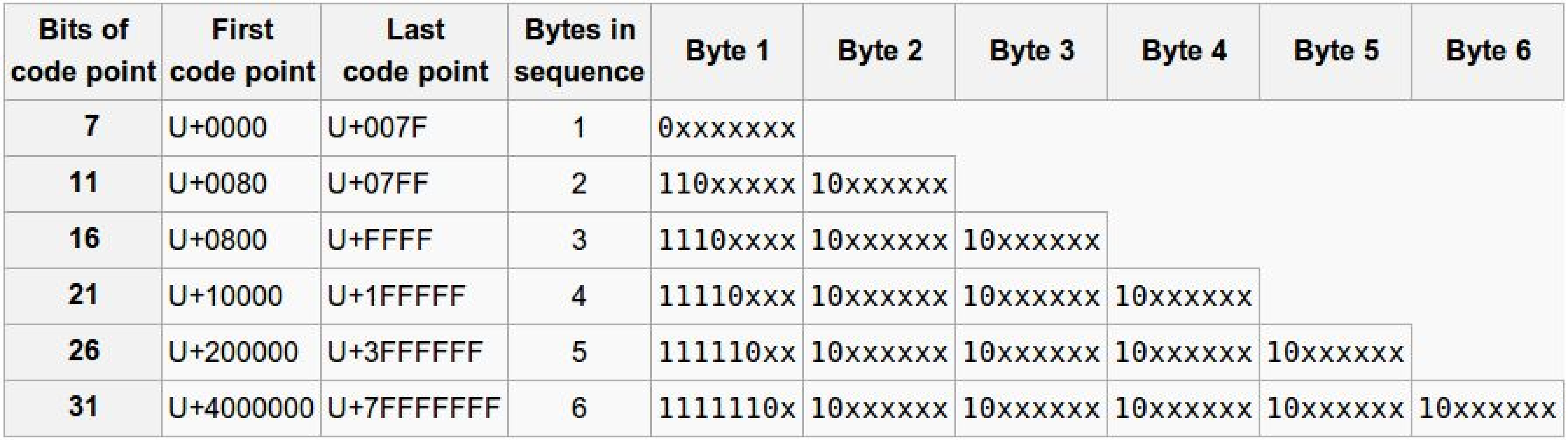
48	69	e2	84	99	c6	b4	e2	98	82	e2	84	8c	c3	b8	e1	bc	a4
H	i	ℙ			ƴ		☂			ℌ			ø		ἤ		

@nedbat

bit.ly/unipain

Source: http://nedbatchelder.com/text/unipain.html

UTF-8



Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	U+0000	U+007F	1	0xxxxxxx					
11	U+0080	U+07FF	2	110xxxxx	10xxxxxx				
16	U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx			
21	U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx		
26	U+200000	U+3FFFFFF	5	111110xx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	
31	U+4000000	U+7FFFFFFF	6	1111110x	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx	10xxxxxx

- 1 byte (128 values) covers ASCII characters
- 2 bytes (1920 values) cover all latin-based alphabets + Greek, Cyrillic, Arabic, etc.
- 3 bytes (63k values) cover the BMP of UTF-16
- the rest covers the supplementary planes

String encoding

Grüezi zäme:

- ASCII (no "ä"):

```
47 72 -- 65  7a 69 20 7a  -- 6d 65                    Gr.ezi z.me
```

- ISO-8859-1:

```
47 72 fc 65  7a 69 20 7a  e4 6d 65                    Gr.ezi z.me
```

- UTF-32:

```
ff fe 00 00  47 00 00 00  72 00 00 00  fc 00 00 00   ....G...r.......
65 00 00 00  7a 00 00 00  69 00 00 00  20 00 00 00   e...z...i... ...
7a 00 00 00  e4 00 00 00  6d 00 00 00  65 00 00 00   z.......m...e...
```

- UTF-16:

```
ff fe 47 00  72 00 fc 00  65 00 7a 00  69 00 20 00   ..G.r...e.z.i. .
7a 00 e4 00  6d 00 65 00                             z...m.e.
```

- UTF-8:

```
47 72 c3 bc  65 7a 69 20  7a c3 a4 6d  65            Gr..ezi z..me
```

String encoding

```
1 茶:
```

- ASCII (no "茶"):

```
31 20 --                                             1 .
```

- **Shift-JIS**:

```
31 20 92 83                                          1 ..
```

- UTF-32:

```
ff fe 00 00  31 00 00 00  20 00 00 00  36 83 00 00   ....1... ...6...
```

- UTF-16:

```
ff fe 31 00  20 00 36 83                             ..1. .6.
```

- UTF-8:

31 20 **e8 8c b6**   1 ...

00110001 **0**0100000 **1110**1000 **10**001100 **10**110110

Encoding

- **3 Levels:**
  - **letters = *"glyphs"*** (Graphic Representation)
    - `a`
  - **Unicode code points**
    - `U+0061`
  - **numeric representation**
    - decimal representation
      - `97`
    - hexadecimal representation
      - more readable than binary
      - `61`
    - binary representation (stored in file)
      - Depends on encoding standard
        - e.g. `01100001` (ASCII, UTF-8)

Part 2: Python Part

- **File i/o =** file input/output
- Unicode and conversion

RAM vs Durable storage

Your computer has

- a "hard disk" -- HDD/SSD
  - keeps the information even without electric power
  - data transfer rate:
    - HDD: 50-120 MB/s, SSD: 200-500 MB/s
- "memory" -- RAM (random-access memory)
  - power off: RAM empty
  - data transfer rate:
    - DDR3: 6400 MB/s

Work with files

- File kept on disc
- To access/modify:
  - copy file from disk into memory = **open** file
  - read and process text/play music/display picture or text/show video from memory
  - either copy back from memory to the disk (if changes made) or discard the copy in memory = **close** file

- Advice: Back up your important files before working on them...

Python file i/o

- Handled by the `file` object
- Opening (loading from disc into memory):

```
f = open('/home/lmascarell/file.txt', 'r')
```

  - *open(filePath, accessMode)*
  - *accessMode*:
    - `'r'`: reading
    - `'w'`: writing (deletes the current content)
    - `'a'`: appending (adds new content to old)
    - `'r+'`/`'w+'`/`'a+'`: reading and writing

- Closing (clearing memory/saving to the disk):

```
f.close()
```

Python file input

- `f.read()`: read the entire file into a string
- `f.read(N)`: read at most N bytes
- `f.readline()`: read one line
  - can be repeated to read file one line at a time
- `f.readlines()`: read the entire file into a list of lines as strings
- `for line in f`: will iterate over the file, line by line

Python file input

Example:

```
path = '/home/lmascarell/file.txt'

f = open(path, 'r')
for l in f:
   print l,
f.close()

f = open(path, 'r')
print "".join(f.readlines()),
f.close()
```

Python file input

Example:

```
path = '/home/lmascarell/file.txt'

f = open(path, 'r')
for l in f:
    print l,
f.close()

f = open(path, 'r')
print "".join(f.readlines()),
f.close()
```

the 2 code snippets will print the same output

Python file output

- `f.write(str)`: write string to file
- `f.writelines(strList)`: write a list of strings to file

```
f = open('/home/lmascarell/file.txt', 'w')
f.write("hello\n")
f.writelines(["one\n", "two\n", "three\n"])
f.close()
```

- only strings:

```
val = ['g', 2, "hoho", 5]
f.write(val)       # TypeError
f.write(str(val)) # ok!
```

Python file reading position

Reading/writing updates the current position in file

- `f.tell()`: position from file beginning in bytes
- `f.seek(N)`: change position to `N` bytes from file beginning
- `f.seek(N, 1)`: change position to `N` bytes from current position
- `f.seek(N, 2)`: change position to `N` bytes from file end

```
f.seek(-3, 2)  # go to 3 bytes before the end
```

Python folder handling

- Folder = list of files/folders that it includes
- module for handling: `os`
- `os.listdir(path)`: return a list of files/folders in given path
- `os.path.isdir(path)`: checks if path is a directory/folder or not
- `os.path.isfile`, `os.path.isabs`, ...

```
import os


for subPath in os.listdir('.'):
    print subPath,
    if os.path.isdir(subPath):
        print '/'
```

Python argument handling

Define input and output through arguments:

```
$ python test.py arg1 arg2 arg3
```

argument list: `['test.py', 'arg1', 'arg2', 'arg3']`

```
import sys

var1 = sys.argv[1]
var2 = sys.argv[2]
var3 = sys.argv[3]
```

Python argument handling

- `argparse`
  - Module for argument and option handling
  - https://docs.python.org/2/howto/argparse.html

- Help message, arguments information
- Better user experience

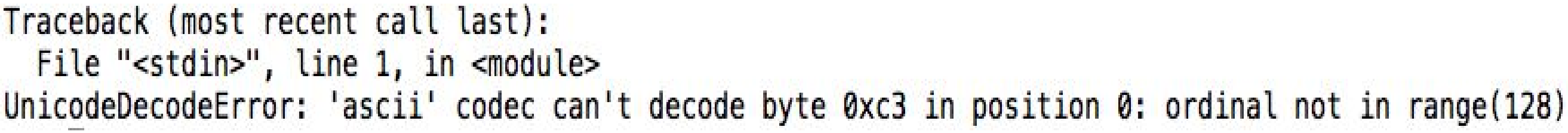
- Class: `ArgumentParser`
- Class Methods:
  - `add_argument()`
  - `parse_args()`

Python argument handling

Example:

```
import argparse
import sys

def parse_command_line():
    parser = argparse.ArgumentParser(description="Prints the first 'num_lines' lines of the file 'src' ")

    parser.add_argument('src', type=argparse.FileType('r'), metavar='FILE', help='source file');
    parser.add_argument('num_lines', help="number of lines to print", type=int)
    parser.add_argument('-o', '--out', type=argparse.FileType('w'), default=sys.stdout, metavar='FILE', help='out file');

    return parser.parse_args()

def main(args):
    for i in range(args.num_lines):
        args.out.write(args.src.readline())

if __name__ == '__main__':
    args = parse_command_line()
    main(args)
```

Python and unicode



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 0: ordinal not in range(128)
```

Possible Solutions:

- Add random decode and encode commands in your code
- Know how to deal with encoding in Python

Unicode in Python 2.x

Universität Zürich^UZH

- **str** type vs **unicode** type

```
a_umlaut = 'ä'
type(a_umlaut)
<type 'str'>
```

```
a_umlaut_unicode = u'\xe4'
type(a_umlaut_unicode)
<type 'unicode'>
```

Unicode in Python 2.x

- **bytes** vs **unicode**

```
a_umlaut = 'ä'
print a_umlaut
'\xc3\xa4'
```

UTF-8 Byte representation (hexadecimal)

```
a_umlaut_unicode = u'\xe4'
print a_umlaut_unicode
ä
```

Unicode Code point

Unicode in Python 2.x

Conversion

- from **bytes** to **unicode**
  - map bytes to unicode code points

```
'\xc3\xa4'.decode('utf-8') → u'\xe4'
unicode('ä', 'utf-8')  → u'\xe4'
unicode('\xc3\xa4', 'utf-8')
```

- from **unicode** to **bytes**
  - map unicode code points to bytes

```
u'\xe4'.encode('utf-8') → '\xc3\xa4'
```

Unicode in Python 2.x

Conversion

- decode()
  - map bytes to unicode code points
- encode()
  - map unicode code points to bytes

Note : Never use encode() on bytes or decode() on Unicode objects.

Encoding Errors in Python

Possible Reasons?

- **UnicodeEncodeError:**
    - Writing out Unicode using the wrong encoding
        - e.g. ASCII for UTF-9 characters
            - (ASCII is default encoding scheme in Python 2.x)
- **UnicodeDecodeError:**
    - Decoding using the wrong encoding (= file encoding)
        - Know the file encoding!
    - Make sure all byte sequences are correct!

Encoding Error Handling in Python

- **Replace:**

```
unicode_str = u'Zürich'
unicode_str.encode('ascii', 'replace')
'Z?rich'
```

- **Ignore:**

```
unicode_str = u'Zürich'
i = unicode_str.encode('ascii', 'ignore')
'Zrich'
```

Encoding of Source Code

Define Python source code encoding through “magic encoding comment”:

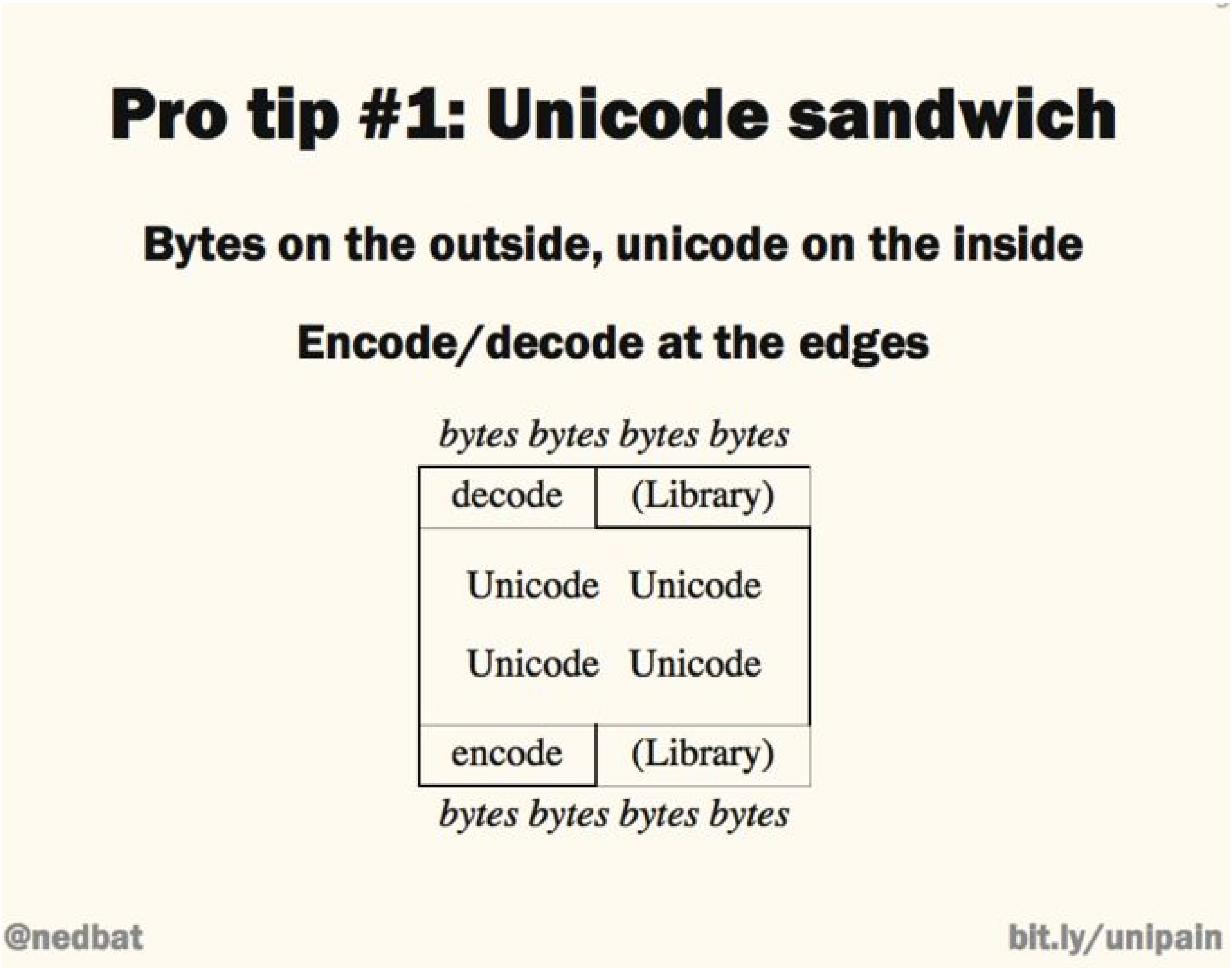
```
#!/usr/bin/python #

-*- coding: <encoding name> -*-
```

Example:

```
#!/usr/bin/python #

-*- coding: utf-8 -*-
```

- Allows for unicode literals in the source code
- Default: ASCII
- Error is raised if unknown encoding is given

The Unicode Sandwich



Pro tip #1: Unicode sandwich

Bytes on the outside, unicode on the inside

Encode/decode at the edges

bytes bytes bytes bytes

decode	(Library)
Unicode	Unicode
Unicode	Unicode
encode	(Library)

bytes bytes bytes bytes

@nedbat

bit.ly/unipain

Source: http://nedbatchelder.com/text/unipain.html

Overview: Avoiding Encoding Errors in Python

- Never use non-ASCII filenames
- Define the encoding of the source code
- Know the encoding of the input files
    - there is no way to guess from a binary file
        - check for encoding definition in the file
            - file headers
            - BOM = “Byte Order Mark” (UTF-16, UTF-32, sometimes UTF-8)
        - try popular encodings
        - guess with chardet module
- Make use of “The Unicode Sandwich”
- Be prepared for all eventualities
    - test your code with different input

Python 2.x vs Python 3.x

- Python 2.x
  - 8-bit strings (bytes loaded from file, any other string)

    ```
    "blaah"
    ```

  - unicode object (converted explicitly/loaded with a unicode codec)

    ```
    u"blaah" = unicode("blaah", "utf-8")
    ```

- Python 3.0
  - text (list of unicode characters)

    ```
    "blaah"
    ```

  - data (list of bytes)

Python and unicode

- Python 2.x:

```
f = open('data.txt')

x = f.read()
# '\xd0\x9f\xd0\xb5\xd0\xbb\xd1\x8c\xd0\xbc\xd0\xb5\xd0\xbd\xd0\xb8'

y = x.decode('utf-8')
# u'\u041f\u0435\u043b\u044c\u043c\u0435\u043d\u0438'
```

- Python 3.0:

```
>>> "Пельмени"
'Пельмени'

>>> "Пельмени".encode()
b'\xd0\x9f\xd0\xb5\xd0\xbb\xd1\x8c\xd0\xbc\xd0\xb5\xd0\xbd\xd0\xb8'
```

Python and unicode

- Python 2.x:

```
>>> "ö"
'\xc3\xb6'

>>> len("ö")
2

>>> u"ö"
u'\xf6'

>>> len(u"ö")
1

>>> re.findall("ö", u"Höhentraining")
[]
```

File i/o with unicode: codecs

```
import codecs


f = open('song.txt', 'r')
lines = f.readlines()
print lines
# ['Song\n', '\tWell here we are again\n', ...]

g = codecs.open('song.txt', 'r', 'UTF-8')
print g.readlines()
# [u'Song\n', u'\tWell here we are again\n', ...]
```

https://docs.python.org/2/library/codecs.html

File i/o with unicode: codecs

```
import codecs

g_in = codecs.open('hallo.txt', 'r', 'UTF-8')
g_lines = g_in.readlines()
print g_lines
# [u'H\xe4llo there...']

g_o = codecs.open('hallo_again.txt', 'w', 'UTF-8')
g_o.write(' '.join(g_lines))
g_o.close()
# Note: Writing an encoded string to a file would result in a
UnicodeDecodeError (Confusing!)
```

https://docs.python.org/2/library/codecs.html

Useful Help on Unicode

Net Batchelder: “How Do I Stop the Pain?”

http://nedbatchelder.com/text/unipain.html

Official Python Documentation: Unicode HOWTO

https://docs.python.org/2/howto/unicode.html (Python 2.x)

https://docs.python.org/3/howto/unicode.html (Python 3.x)

Unicode in Python, Completely Demystified

https://docs.python.org/2/howto/unicode.html

What we learned

- durable storage devices hold bits
- bits are grouped into bytes
- byte representation is used to encode information, grouped into files
- an encoding is an agreement of how to represent text with bytes
- UTF-8 supports practically all scripts in a single encoding
- python has
  - the `file` object for file opening/closing/handling
  - methods and module `codecs` for encoding conversion
  - modules `os` and `os.path` for folder handling

Qµê§†ïðñ§?