Lecture 3

408/508 *Computational Techniques for Linguists*

From Last Time … everything is **binary**

Introduction

- Storage:
  - based on digital logic
  - binary (base 2) – everything is a power of 2
  - Byte: 8 bits
    - 01011011
    - = $2^6+2^4+2^3+2^1+2^0$
    - = 64 + 16 + 8 + 2 + 1
    - = 91 (in decimal)
  - Hexadecimal (base 16)
    - 0-9,A,B,C,D,E,F (need 4 bits)
    - 5B (= 1 byte)
    - = $5*16^1 + 11$
    - = 80 + 11
    - = 91

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	1	1	0	1	1

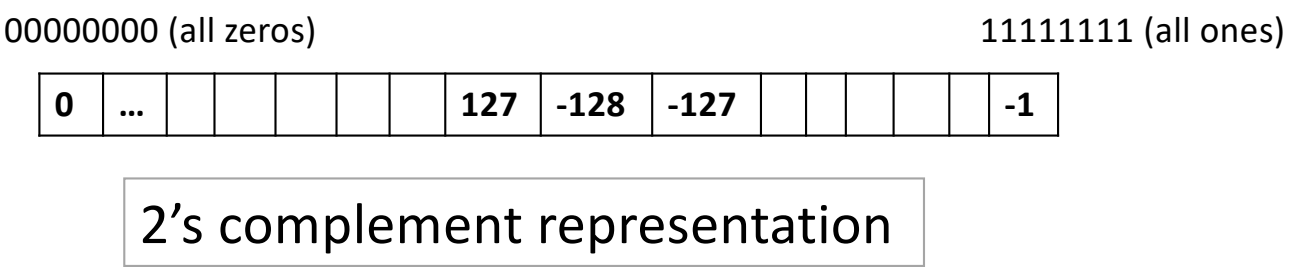
2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
0	1	0	1	1	0	1	1

5 B

16^1	16^0
5	B

Introduction: data types

- Integers
  - In one byte (= 8 bits), what's the largest and smallest number, we can represent?
  - 00000000 = 0
  - 01111111 = 127
  - 10000000 = -128
  - 11111111 = -1
  - Number line:



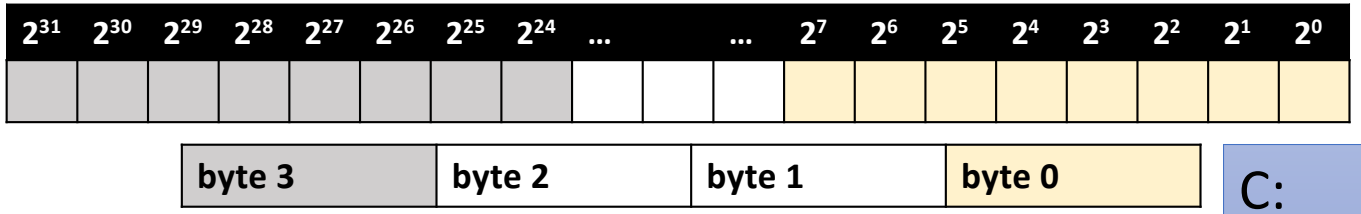
Introduction: data types

- Integers
  - In one byte, what's the largest and smallest number, we can represent?
  - **Answer**: -128 .. 0 .. 127 using the **2's complement representation**
  - Why? super-convenient for arithmetic operations
  - "to convert a positive integer X to its negative counterpart, flip all the bits, and add 1"
  - **Example**:
  - 00001010 = $2^3 + 2^1 = 10$ (decimal)
  - 11110101 + 1 = 11110110 = -10 (decimal)
  - 11110110 flip + 1 = 00001001 + 1 = 00001010

Addition:
-10 + 10
= 11110110
+ 00001010 = 0 (ignore overflow)

Introduction: data types

- Typically 32 bits (4 bytes) are used to store an integer
  - range: -2,147,483,648 ($2^{(31-1)}$ -1) to 2,147,483,647 ($2^{(32-1)}$ -1)

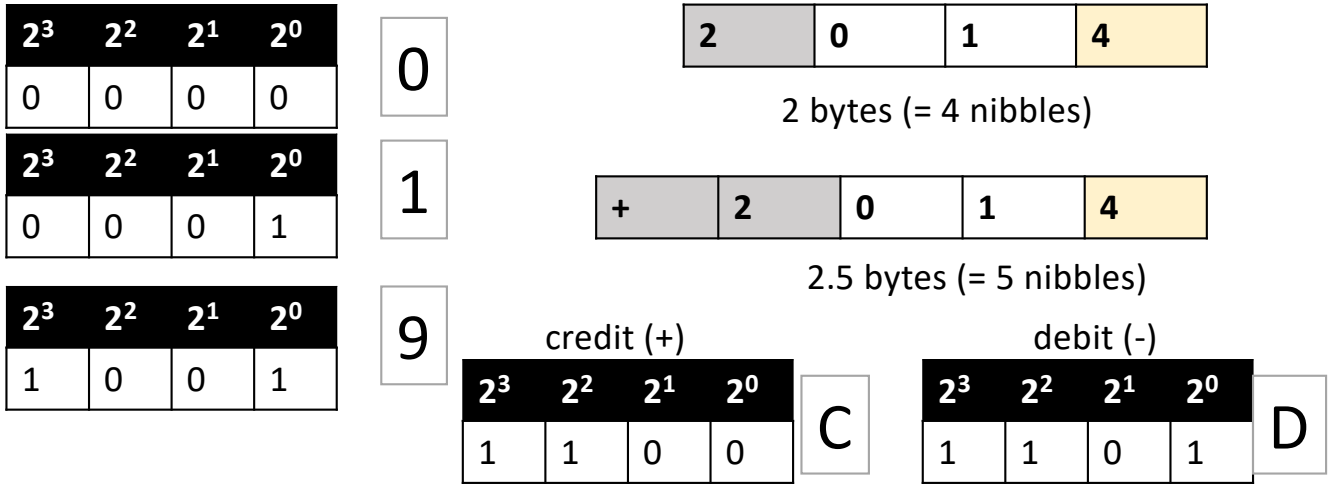


C:
`int i;`
Python
(no declaration)

- what if you want to store even larger numbers?
  - Binary Coded Decimal (BCD)
  - code each decimal digit separately, use a string (sequence) of decimal digits …

Introduction: data types

- what if you want to store even larger numbers?
  - Binary Coded Decimal (BCD)
  - 1 byte can code two digits (0-9 requires 4 bits)
  - 1 nibble (4 bits) codes the sign (+/-), e.g. hex C/D



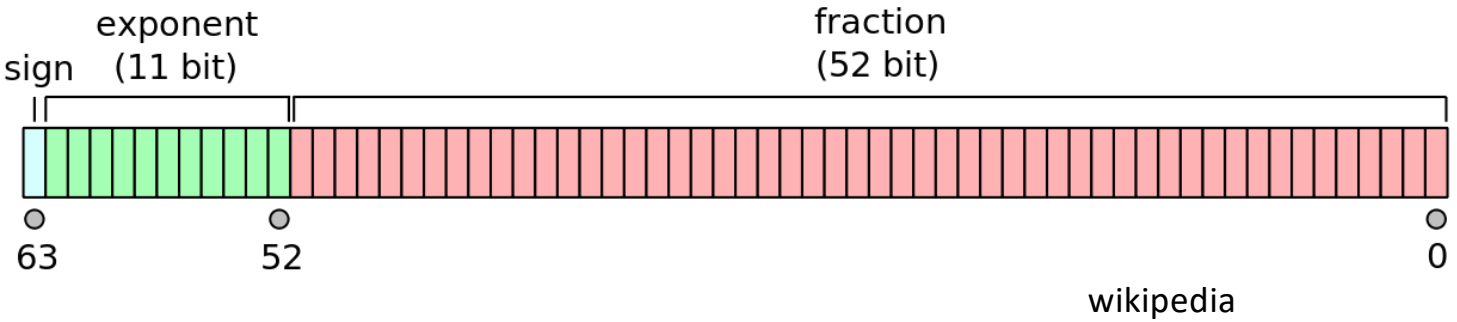
Introduction: data types

e.g. probabilities

- Typically, 64 bits (8 bytes) are used to represent floating point numbers (double precision)
  - c = 2.99792458 x 10^8 (m/s)
  - coefficient: 52 bits (**implied 1**, therefore treat as 53)
  - exponent: 11 bits (usually not 2's complement, unsigned with bias $2^{(10-1)}-1$ = 511)
  - sign: 1 bit (+/-)

C:
float
double

x86 CPUs have a built-in floating point coprocessor (x87)
80 bit long registers



Example 1

- The speed of light:
  - $c = 2.99792458 \times 10^8$ (m/s)

1. Can a 4 byte integer be used to represent *c* exactly?
   - 4 bytes = 32 bits
   - 32 bits in 2's complement format
   - Largest positive number is
   - $2^{31}-1 = 2,147,483,647$
   - *c* = 299,792,458

Example 2

- Recall the speed of light:
  - $c = 2.99792458 \times 10^8$ (m/s)

2. How much memory would you need to encode *c* using BCD notation as an integer?
   - 9 digits
   - each digit requires 4 bits (a nibble)
   - BCD notation includes a sign nibble
   - total is 5 bytes

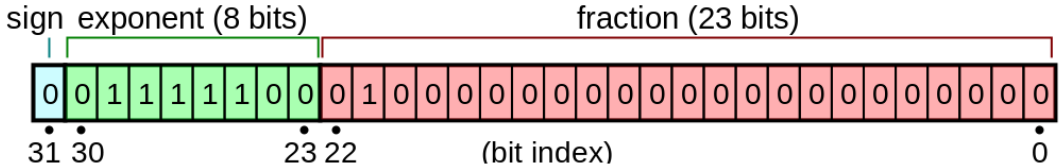
Example 3

- Recall the speed of light:
  - $c = 2.99792458 \times 10^8$ (m/s)

3. Can the 64 bit floating point representation (`double`) encode *c* without loss of precision?
   - Recall significand precision: 53 bits (52 explicitly stored)
   - $2^{53}-1 = 9{,}007{,}199{,}254{,}740{,}991$
   - almost 16 digits
   - (we only need 9 digits of precision)

Example 4

- Recall the speed of light:
  - c = 2.99792458 x 10^8 (m/s)
- The 32 bit floating point representation (`float`) – sometimes called single precision - is composed of 1 bit sign, 8 bits exponent (unsigned with bias $2^{(8-1)}$-1), and 23 bits coefficient (24 bits effective).



- Can it represent c without loss of precision?
  - 2^{24}-1 = 16,777,215
  - Nope (7 digits of precision)

Example 5

Wikipedia: when the atom is irradiated with electromagnetic radiation having an energy corresponding to the energetic difference between the two sub-levels the radiation is absorbed and the atom is excited, going from the *F* = 3 sub-level to the *F* = 4 one. After some time the atom will re-emit the radiation and return to its *F* = 3 ground state.

- Atomic clocks use the caesium standard. Frequency is exactly **9 192 631 770** Hz.
- Does the 32 bit binary integer representation have enough bits to represent this frequency?
- Relevant integer is:
  - 9 192 631 770 (Hz)
  - Largest possible integer using 2's complement is:
  - $2^{31} - 1 = 2\ 147\ 483\ 648$ (too small)
  - Largest possible integer even if we use *unsigned* representation is:
  - $2^{32} - 1 = 4\ 294\ 967\ 296$ (still too small)

Example 5

- *Kinda tedious to calculate the binary bit pattern by hand*
- At the **Terminal** (on Mac or Linux):

```
$ which bc
/usr/bin/bc
$ bc
>>> obase=2
>>> 9192631770
1000100011111011000110110111011010
10 0010 0011 1110 1100 0110 1101 1101 1010
```

(bc - *arbitrary-precision calculator*)

(*group by nibbles*)

Example 5

- In **Python**:

```
$ python
Python 3.9.16 | packaged by conda-forge | (main, Feb  1 2023, 21:38:11)
[Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> bin(9192631770)
'0b1000100011111011000110110111011010'
```

Example 5

- In Python, function `bin()` expects an integer:

```
>>> bin(9.19263177e9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
>>> 9.19263177e9
9192631770.0
```

- function `int()` converts number into an integer:

```
>>> bin(int(9.19263177e9))
'0b1000100011111011000110110111011010'
```

ChatGPT: Example 5

What is the caesium atom frequency in binary?

Great question! The **cesium (caesium) standard** that defines the **second** uses the transition frequency of the cesium-133 atom:

$$\nu_{\mathrm{Cs}} = 9,192,631,770 \text{ Hz}.$$

Now let's express that in **binary**:

$$9{,}192{,}631{,}770_{10} = 1000\ 1000\ 1110\ 1111\ 0100\ 1010\ 0000\ 1010_2.$$

✅ So, in binary it is:

csharp Copy Edit

```
10001000111011110100101000001010 (base 2)
```

Would you like me to also give you the **hexade al** form (which is much more compact) alongside the binary?

Example 5

- Since binary and hexadecimal are so common, **Python** has built-in functions `bin()` and `hex()`
  - `docs.python.org/3/library/functions.html#hex`

hex(*integer*, /)

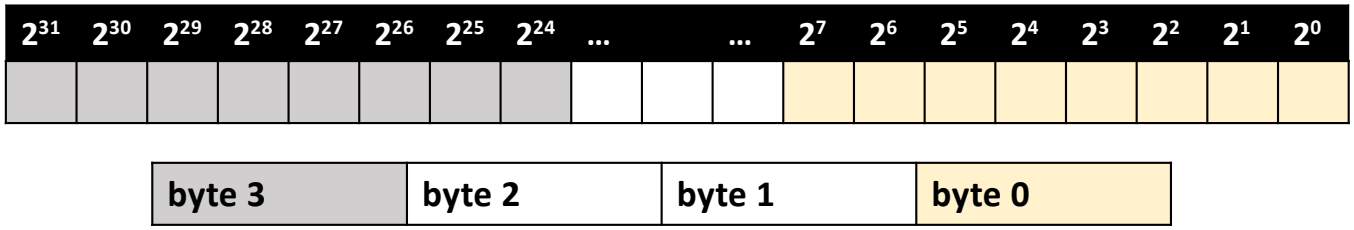
Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *integer* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

192 631 770 Hz in hexadecimal?

Quick Homework 1: Binary Exercise

- 2,147,483,659 is a prime number.
- Does the 32 bit binary integer representation have enough bits to represent this prime? Explain.



Instructions

- Email to sandiway@arizona.edu
- By tomorrow midnight
- SUBJECT: 408/508 Homework 1: **YOUR NAME**
- Send me the binary bit pattern!
- Either Plain Text or PDF accepted
  - no MS Word .doc/.docx files please

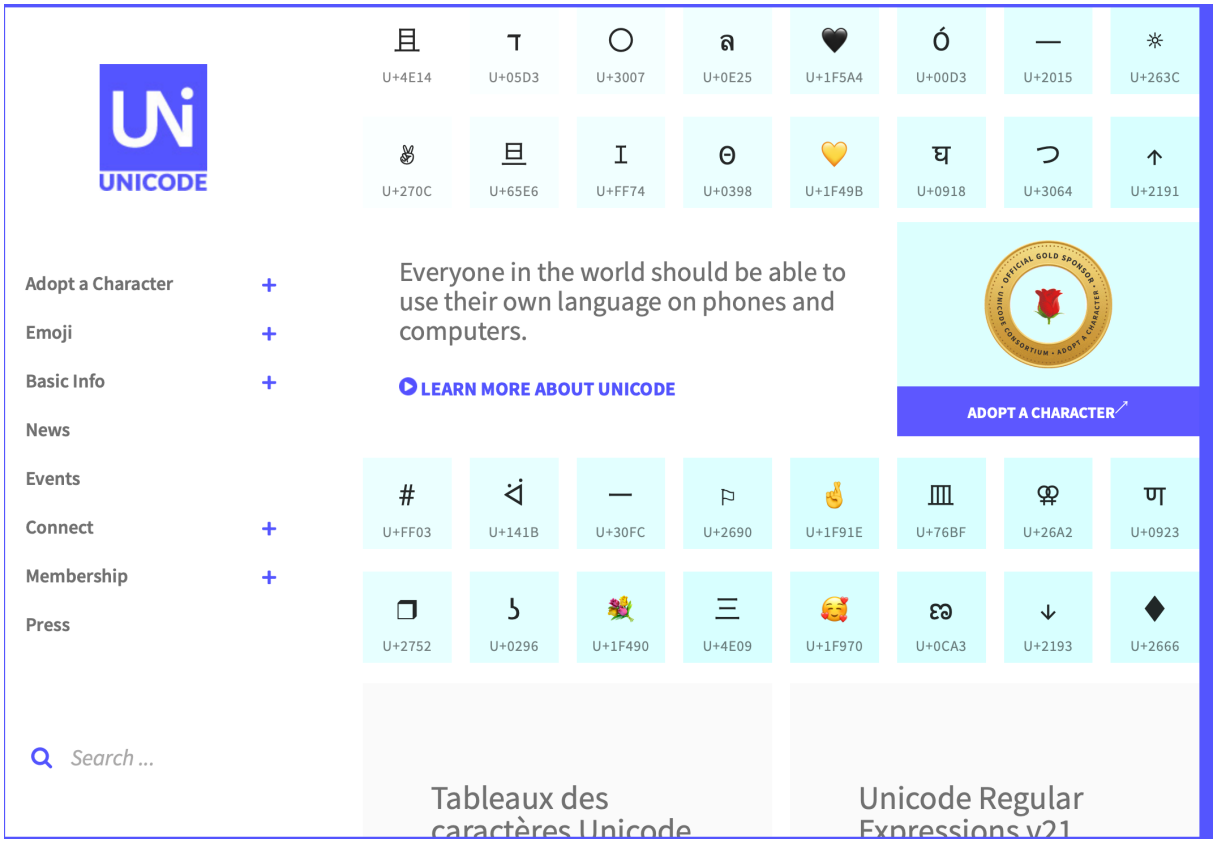
New Topic

- Enough about numbers for now.
- How about language data, i.e. characters we type?

https://home.unicode.org

Unicode

number of encoded characters in Unicode 17.0: 159,845

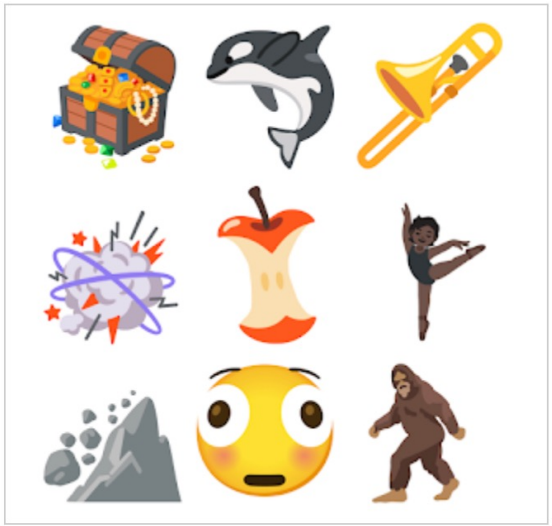


Unicode adoption:

MacOS Tahoe 26.2.1 and i/Pad/OS 26.2.1 include Unicode standard 16.0. Apple's iOS 18.4 or later contains Unicode standard 16.0. Earlier versions of iOS 18 contain Unicode standard 15.1.

WEDNESDAY, JULY 16, 2025

🥳 Say Hello to the New Emoji Coming in Unicode 17.0 This Fall! ✨



From 🥹 to 🦕 to 🎸, emoji have become the world's favorite way to say anything—without saying a word. Whether you're texting your best friend, posting on social media, or cheering someone up with a perfectly-timed 😭, emoji help us connect across languages, cultures, and continents.

Unicode

- https://www.unicode.org/versions/Unicode13.0.0/UnicodeStandard-13.0.pdf

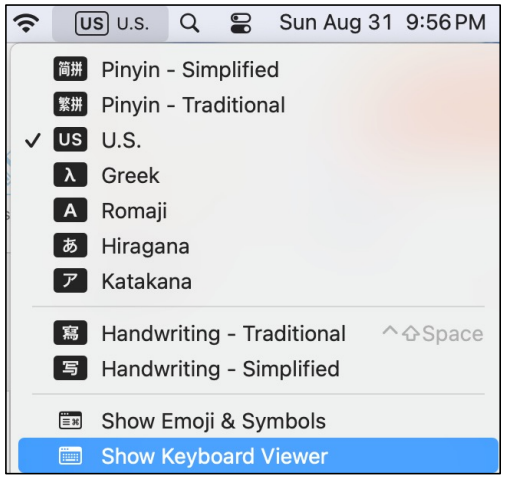
UTF-8

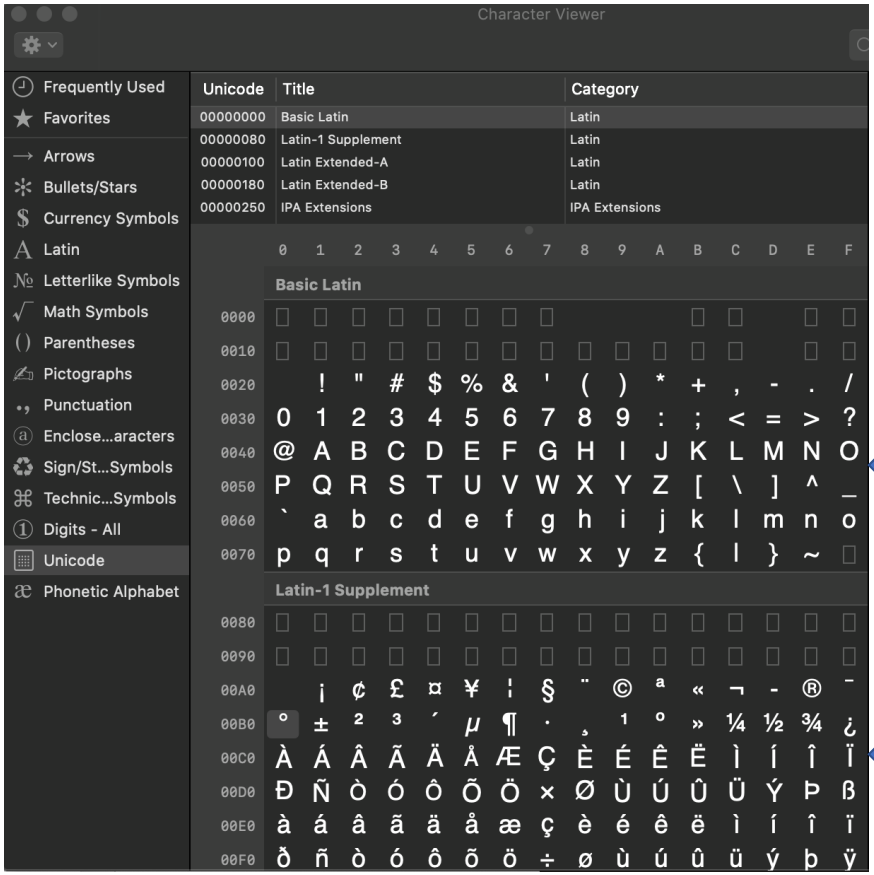
To meet the requirements of byte-oriented, ASCII-based systems, a third encoding form is specified by the Unicode Standard: UTF-8. This variable-width encoding form preserves ASCII transparency by making use of 8-bit code units.

Preferred Usage. UTF-8 is typically the preferred encoding form for HTML and similar protocols, particularly for the Internet. The ASCII transparency helps migration. UTF-8 also has the advantage that it is already inherently byte-serialized, as for most existing 8-bit character sets; strings of UTF-8 work easily with the C standard library, and many existing APIs that work for typical East Asian multibyte character sets adapt to UTF-8 as well with little or no change required.

On the Mac

Show Emojis and Symbols > Unicode





ASCII

Latin-1 supplement

Introduction: data types

- How about letters, punctuation, etc.?
- ASCII
  - American Standard Code for Information Interchange
  - Based on English alphabet (upper and lower case) + space + digits + punctuation + control (Teletype Model 33)
  - **Question**: how many bits do we need?
  - 7 bits + 1 bit parity
  - Remember everything is in binary …

C:
char

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Teletype Model 33 ASR Teleprinter (Wikipedia)

Introduction: data types

- UTF-8
  - standard in the post-ASCII world
  - backwards compatible with ASCII
  - (*previously, different languages had multi-byte character sets that clashed*)
  - Universal Character Set (UCS) Transformation Format 8-bits

Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4
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`

(Wikipedia)

Introduction: data types

order is important in sorting!

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

0-9: there's a connection with BCD. **Notice:** code 30 (hex) through 39 (hex)

Introduction: data types

- Parity bit:
  - transmission can be noisy over telephone lines
  - parity bit can be added to ASCII code (total: 8 bits, byte)
  - can spot single bit transmission errors
  - even/odd parity:
    - receiver understands each byte should be even/odd
  - Example:
    - 0 (zero) is ASCII 30 (hex) = 011000 (binary)
    - even parity: 0011000, odd parity: 1011000
  - Checking parity:
    - Exclusive or (XOR): basic machine instruction
      - A xor B true if either A or B true but not both
  - Example:
    - (even parity 0) 0011000 xor bit by bit
    - 0 xor 0 = 0 xor 1 = 1 xor 1 = 0 xor 0 = 0 xor 0 = 0 xor 0 = 0 xor 0 = 0

x86 assemby language:

1. PF: even parity flag set by arithmetic ops.
2. TEST: AND (don't store result), sets PF
3. JP: jump if PF set

Example:

MOV al,<char>
TEST al, al
JP <location if even>
<go here if odd>

Introduction: data types

- UTF-8
  - standard in the post-ASCII world
  - backwards compatible with ASCII
  - (*previously, different languages had multi-byte character sets that clashed*)
  - Universal Character Set (UCS) Transformation Format 8-bits

	Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4
ASCII	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

(Wikipedia)

Introduction: data types

In your browser, which runs HTML5, UTF-8, is the default character encoding

ç	231	00E7	ç	LATIN SMALL LETTER C WITH CEDILLA
è	232	00E8	è	LATIN SMALL LETTER E WITH GRAVE
é	233	00E9	é	LATIN SMALL LETTER E WITH ACUTE
ê	234	00EA	ê	LATIN SMALL LETTER E WITH CIRCUMFLEX
ë	235	00EB	ë	LATIN SMALL LETTER E WITH DIAERESIS

Introduction: data types

é	233	00E9	é	LATIN SMALL LETTER E WITH ACUTE

File: `pokemon.html`

```
1<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">¶
2<html> <head>¶
3<title></title>¶
4</head>¶
5<body>¶
6<h1>Pok&eacute;mon</h1>¶
7</body>¶
8</html>¶
```

Introduction: data types

line 7: é typed on my mac keyboard using OPT-e e

File: `pokemon2.html`

```
1 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
2 <html> <head>
3 <title></title>
4 </head>
5 <body>
6 <h1>Pok&eacute;mon</h1>
7 Pokémon
8 </body>
9 </html>
```

Introduction: data types

File: `pokemon3.html`

```
1 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
2 <html> <head>
3 <meta charset="UTF-8">
4 <title></title>
5 </head>
6 <body>
7 <h1>Pok&eacute;mon</h1>
8 Pokémon
9 </body>
10 </html>
```

Introduction: data types

Bits of code point	First code point	Last code point	Bytes in sequence	Byte 1	Byte 2	Byte 3	Byte 4
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

- Example:
  - あ Hiragana letter A: UTF-8: E38182
  - Byte 1: E = 1110, 3 = 0011
  - Byte 2: 8 = 1000, 1 = 0001
  - Byte 3: 8 = 1000, 2 = 0010
  - い Hiragana letter I: UTF-8: E38184

Shift-JIS (Hex):
あ: 82A0
い: 82A2

Introduction: data types

- あ Hiragana letter A: UTF-8: E3 81 82
- い Hiragana letter I: UTF-8: E3 81 84

227	E3	ã

Latin-1 character set (8 bits) https://kb.iu.edu/d/aepu

Introduction: data types

Many Windows programs (including Windows Notepad) add the bytes 0xEF, 0xBB, 0xBF at the start of any document saved as UTF-8. This is the UTF-8 encoding of the Unicode byte order mark (BOM), and is commonly referred to as a UTF-8 BOM, even though it is not relevant to byte order. A BOM can also appear if another encoding with a BOM is translated to UTF-8 without stripping it. Software that is not aware of multibyte encodings will display the BOM as three strange characters (e.g. "ï»¿" in software interpreting the document as ISO 8859-1 or Windows-1252) at the start of the document.

Introduction: data types

- How can you tell what encoding your file is using?
- Detecting UTF-8
  - Microsoft:
    - 1st three bytes in the file is EF BB BF
    - (*not all software understands this; not everybody uses it*)
  - HTML:
    - `<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">`
    - (*not always present*)
  - Analyze the file:
    - Find non-valid UTF-8 sequences: if found, not UTF-8…
    - Interesting paper:
      - http://www-archive.mozilla.org/projects/intl/UniversalCharsetDetection.html