Lecture 8

Text encoding, text formats

Learning objectives

By the end of this lecture, you should:

- Know how text is represented in binary
- Understand what Unicode and text encodings are
- Understand why encÎ¿ding errors happen
- Know how to handle binary files and corrupted text files in Python
- Know three common text-based data formats (CSV, JSON, XML) and how to use them in Python

Code in notebook

Get the notebook!

`text_encoding.ipynb`

Binary and hexadecimal notation

Decimal notation:

10 000 000	1 000 000	100 000	10 000	1000	100	10	1
0	0	0	0	0	1	0	7

Binary notation:

128	64	32	16	8	4	2	1
0	1	1	0	1	0	1	1

1 Byte

Hexadecimal notation:

268 435 456	16 777 216	1 048 576	65 536	4096	256	16	1
0	0	0	0	0	0	6	b

1 Byte

What is the largest number that can be represented in 1 Byte?

128	64	32	16	8	4	2	1
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**

1 Byte

Text encodings and Unicode

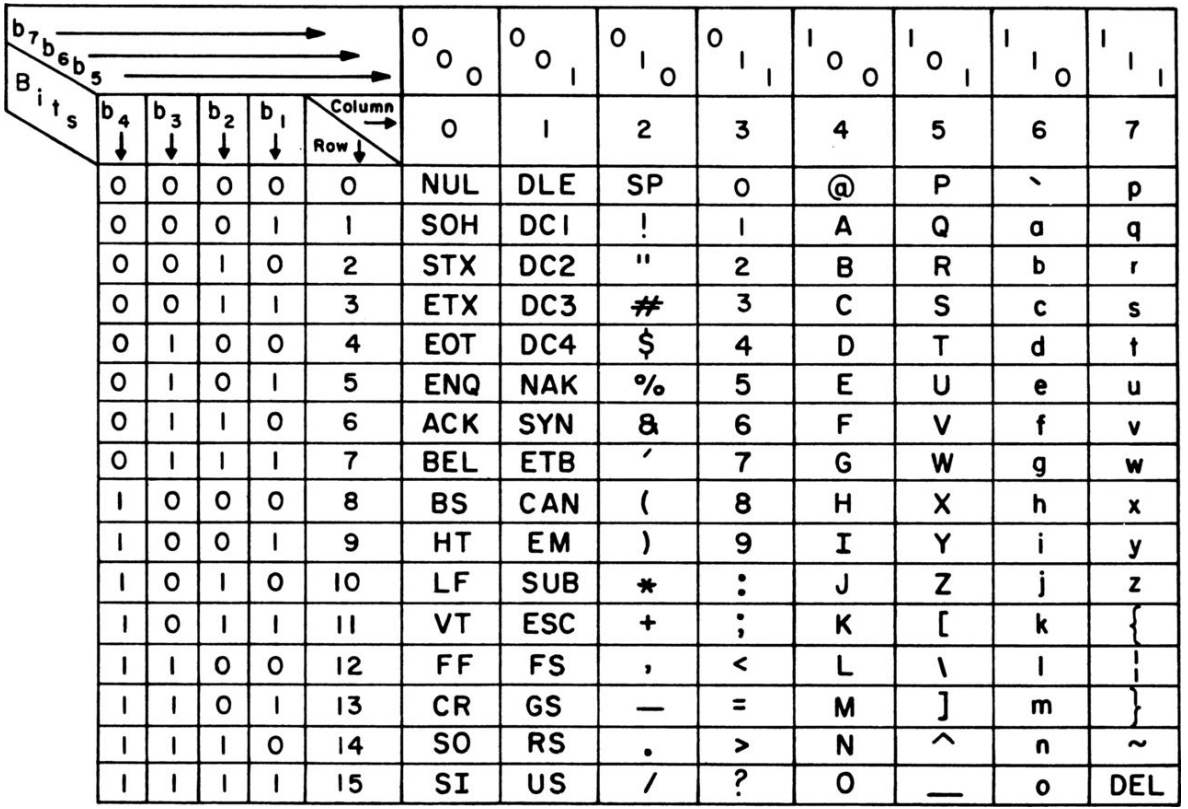
Code in notebook

What's in a string?

String	Hello					"Hello"
Characters	H	e	l	l	o	`list("Hello")`
Code points	72	101	108	108	111	`[ord(char) for char in "Hello"]`
Binary representation	0100 1000	0110 0101	0110 1100	0110 1100	0110 1111	`[bin(ord(char)) for char in "Hello"]`

- `ord(character)` converts a character into a code point
- `chr(codepoint)` converts a code point into a character
- **Remember:** In Python, characters are just strings of length 1

ASCII



$b_7 b_6 b_5$ →	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 b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	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	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

ASCII chart (1972)

ASCII

(ASCII, 1963)

00000000 →	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
00010000 →	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
00100000 →	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
00110000 →	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
01000000 →	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
01010000 →	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
01100000 →	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
01110000 →	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
10000000 →	**Unused code points**															
10010000 →																
10100000 →																
10110000 →																
11000000 →																
11010000 →																
11100000 →																
11110000 →																

ISO-8859-1

(Latin-1, 1987)

00000000 →	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
00010000 →	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
00100000 →	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
00110000 →	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
01000000 →	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
01010000 →	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
01100000 →	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
01110000 →	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
10000000 →	**Control characters** (not printable)															
10010000 →																
10100000 →		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
10110000 →	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
11000000 →	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
11010000 →	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
11100000 →	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
11110000 →	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

ISO-8859-15

(Latin-9, 1999)

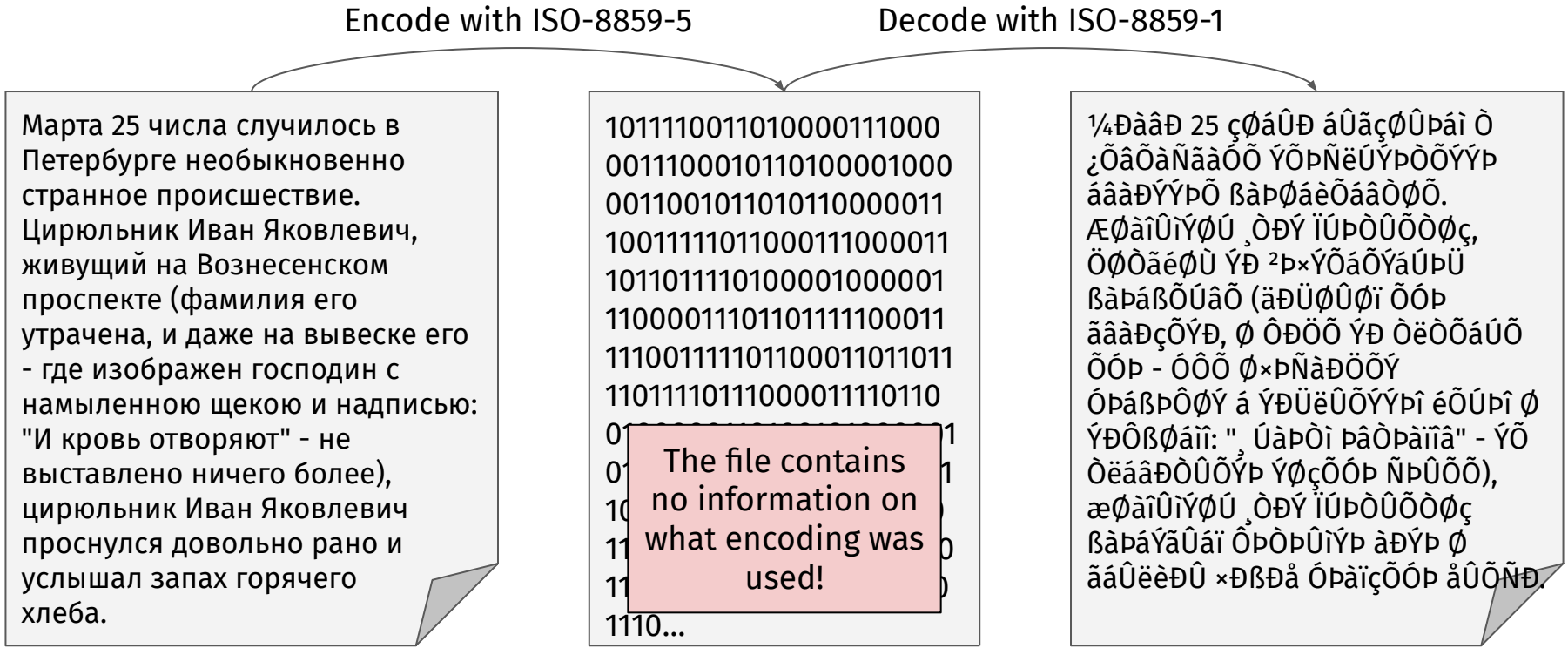
00000000 →	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
00010000 →	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
00100000 →	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
00110000 →	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
01000000 →	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
01010000 →	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
01100000 →	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
01110000 →	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
10000000 →	**Control characters** (not printable)															
10010000 →																
10100000 →		¡	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	¯
10110000 →	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
11000000 →	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
11010000 →	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
11100000 →	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
11110000 →	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Windows-1252

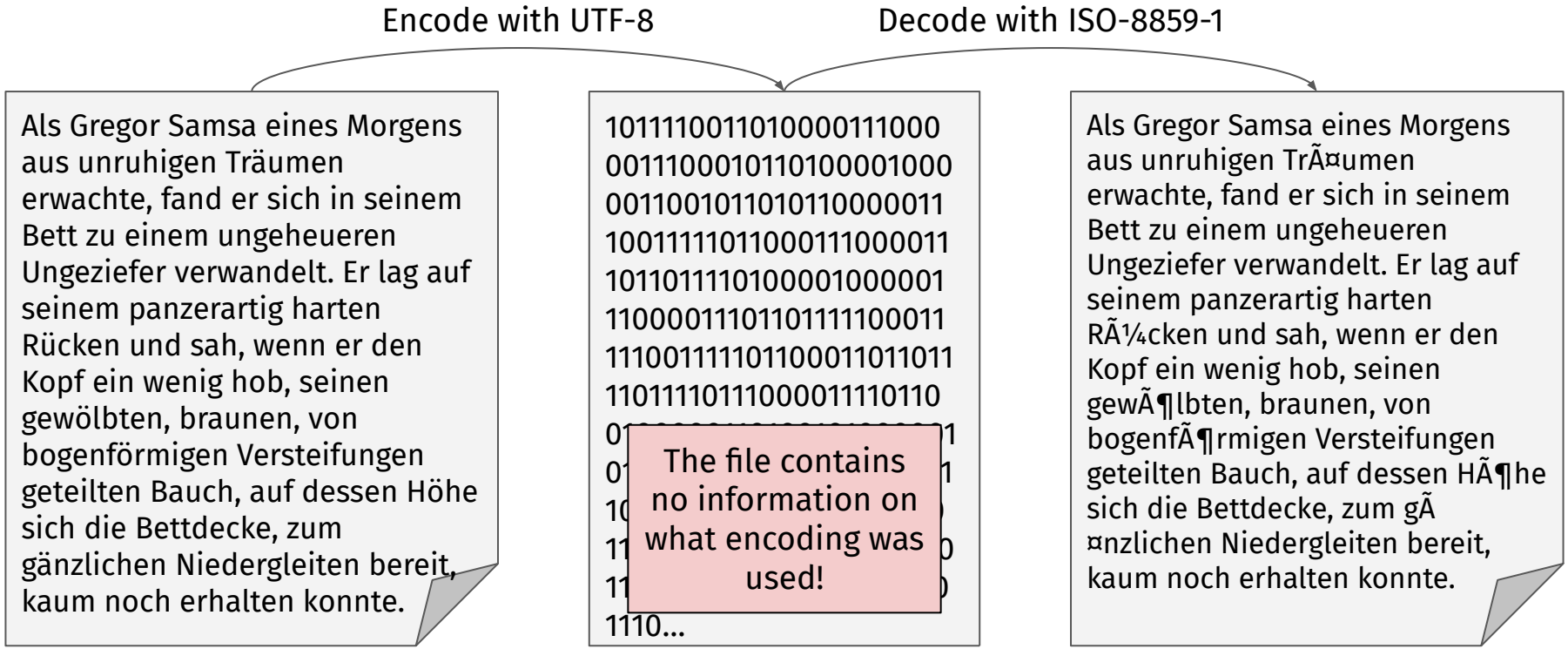
(CP-1252, 1985)

00000000 →	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
00010000 →	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
00100000 →	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
00110000 →	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
01000000 →	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
01010000 →	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
01100000 →	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
01110000 →	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
10000000 →	€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž	
10010000 →		‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ
10100000 →		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
10110000 →	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
11000000 →	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
11010000 →	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
11100000 →	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
11110000 →	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Encoding mismatch



Encoding mismatch



Pre-Unicode text encodings

- Use fixed number of bits per character
  Usually 1 byte (= 8 bits) per character → 2^8 = 256 code points
- Use different code point mappings (encodings) for different writing systems
  - **ASCII**: English alphabet
  - **ISO-8859-1** (**Latin-1**): Latin alphabet
  - **ISO-8859-5**: English + Cyrillic alphabet
  - **ISO 8859-6**: English + Arabic alphabet
  - **Windows-1252** (**CP-1252**): Latin alphabet

  } The “English” (ASCII) part is the same in all of these
- Problems
  - Very limited number of encodable characters
  - Using multiple alphabets/scripts within one file/string is impossible
  - Opening a file with the wrong encoding results in garbled text

The Unicode Standard

- Goal: consistent representation of text written in any writing system
- Maps characters to code points (numbers)
- Currently contains 149,186 characters (Unicode 15.0)

Name	Character	Code point
LATIN CAPITAL LETTER A	A	65
INTERROBANG	‽	8253
CJK UNIFIED IDEOGRAPH-5B57	字	23383
SIGNWRITING HAND-FIST INDEX MIDDLE THUMB CUPPED		120872
NERD FACE	🤓	129299

Unicode character properties (r12a.github.io/app-analysestring)

U+00E9 LATIN SMALL LETTER E WITH ACUTE

Unicode block: Latin-1 Supplement, Letters

General category:	Ll - Letter, lowercase
Canonical combining class:	0 - Spacing, split, enclosing, reordrant, & Tibetan subjoined
Bidirectional category:	L - Left-to-right
Character decomposition mapping:	0065 0301 é
Uppercase mapping:	00C9 É
Titlecase mapping:	00C9 É
Unicode 1.0 name:	LATIN SMALL LETTER E ACUTE
Unicode version:	a
As text:	é
Decimal:	233
Show more properties	

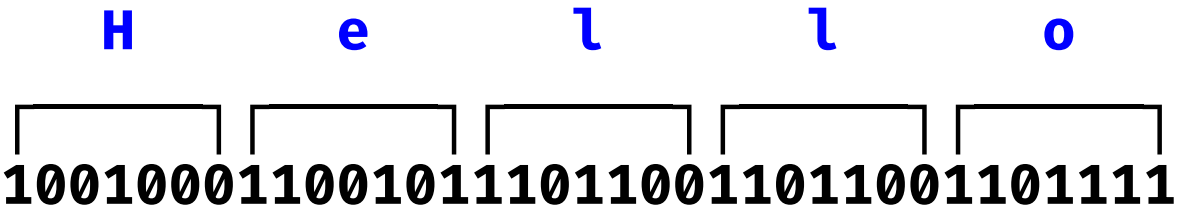
unicodedata

```
>>> import unicodedata
>>> unicodedata.digit("²")
2
>>> unicodedata.name("#")
'NUMBER SIGN'
>>> unicodedata.lookup("nerd face")
'🤓'
>>> unicodedata.category("🤓")
'So'
```

Representing strings in memory / on disk

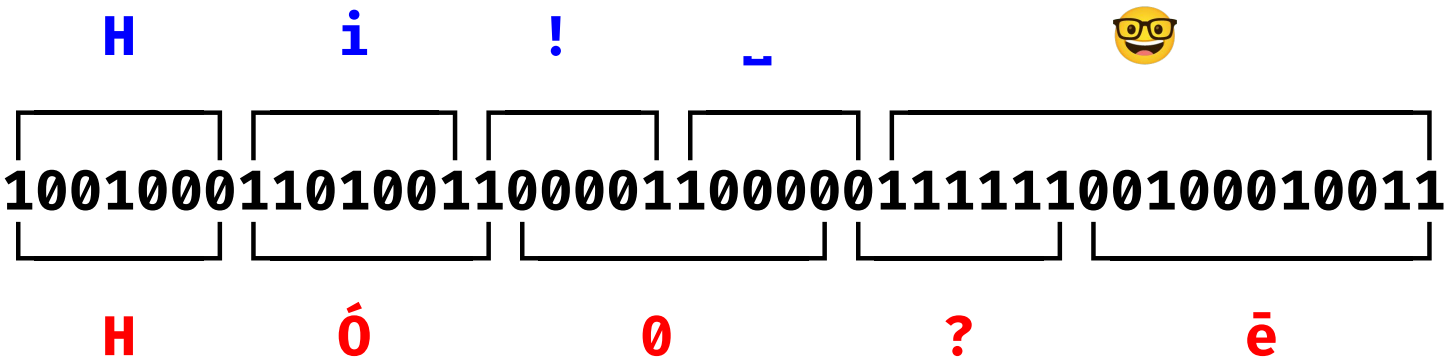
Memory is a one-dimensional sequence of bits (ones and zeros)

Simply concatenating binary representations of code points would look like this:



Representing strings in memory / on disk

But what if the code points use varying numbers of binary digits?



→ We need constant-length binary units for unambiguous boundaries

→ We need some way of indicating the start of a new character

Unicode Transformation Formats (UTF)

Variable-length encodings:

- **UTF-8**: 1-4 bytes per character, compatible with ASCII
- **UTF-16**: 2 or 4 bytes per character

Fixed-length encoding:

- **UTF-32**: 4 bytes per character (rarely used)

These encodings cover **all** characters defined by the Unicode Standard.

Today, **UTF-8** is the de-facto standard (at least for the web).

Code in notebook

Latin-1:

Characters:	h	ä
Unicode code points:	104	228
Encoded bytes (bin):	**01101000**	**11100100**
Encoded bytes (hex):	68	e4

UTF-8:

Characters:	h	ä	
Unicode code points:	104	228	
Encoded bytes (bin):	**01101000**	**11000011**	**10100100**
Encoded bytes (hex):	68	c3	a4

UTF-16:

Characters:	h		ä	
Unicode code points:	104		228	
Encoded bytes (bin):	**00000000**	**01101000**	**00000000**	**11100100**
Encoded bytes (hex):	00	68	00	e4

UTF-32:

Characters:	h				ä			
Unicode code points:	104				228			
Encoded bytes (bin):	**00000000**	**00000000**	**00000000**	**01101000**	**00000000**	**00000000**	**00000000**	**11100100**
Encoded bytes (hex):	00	00	00	68	00	00	00	e4

Variable length encoding (UTF-8)

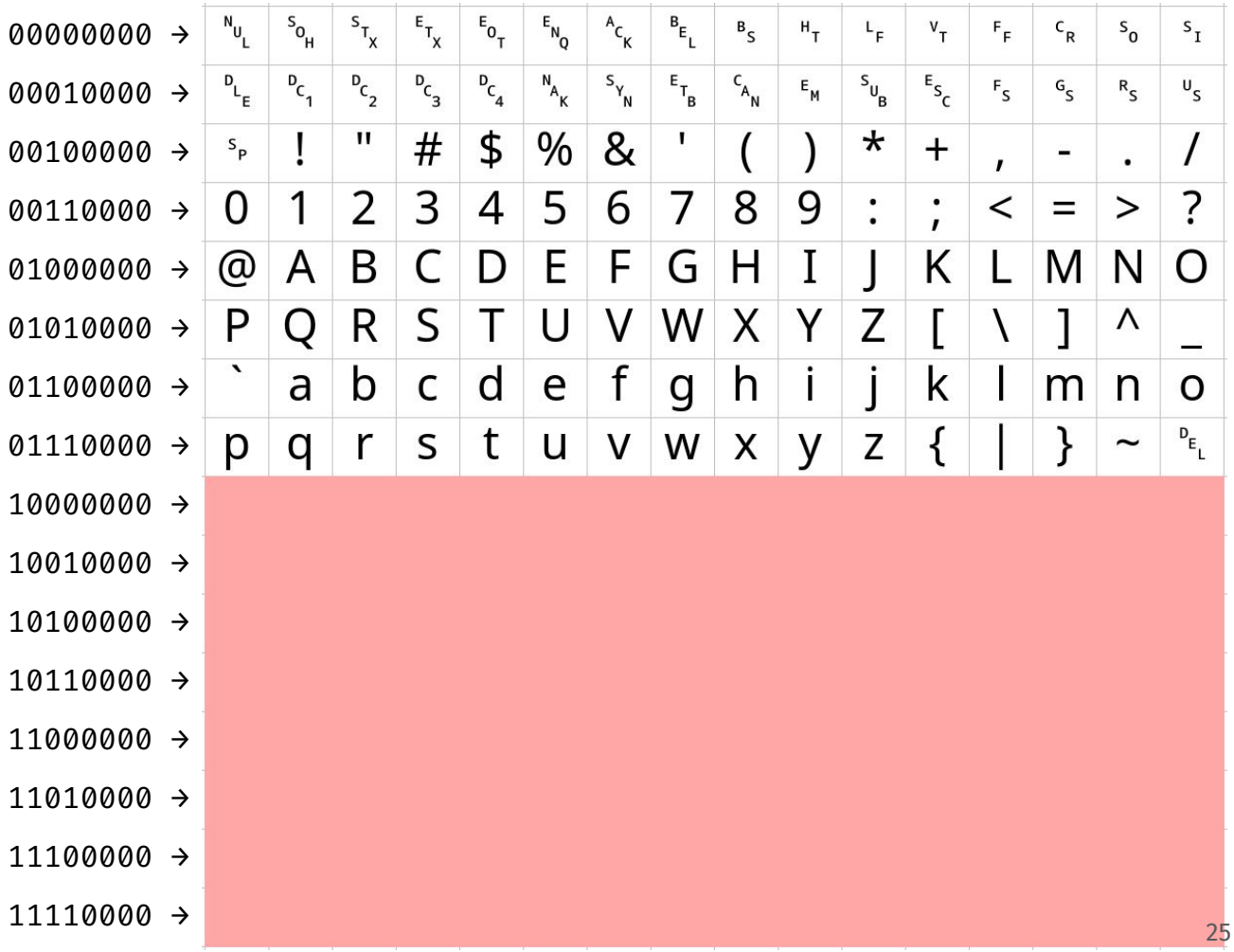
Examples of UTF-8 encoding

Character		Binary code point	Binary UTF-8	Hex UTF-8
$	U+0024	010 0100	00100100	24
£	U+00A3	000 1010 0011	11000010 10100011	C2 A3
ह	U+0939	0000 1001 0011 1001	11100000 10100100 10111001	E0 A4 B9
€	U+20AC	0010 0000 1010 1100	11100010 10000010 10101100	E2 82 AC
한	U+D55C	1101 0101 0101 1100	11101101 10010101 10011100	ED 95 9C
𐍈	U+10348	0 0001 0000 0011 0100 1000	11110000 10010000 10001101 10001000	F0 90 8D 88

Prefix:
Indicates how many bytes the character will use

Source: https://en.wikipedia.org/wiki/UTF-8

UTF-8


00000000 → NUL SOH STX ETX EOT ENQ ACK BEL BS HT LF VT FF CR SO SI
00010000 → DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC FS GS RS US
00100000 → SP ! " # $ % & ' ( ) * + , - . /
00110000 → 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
01000000 → @ A B C D E F G H I J K L M N O
01010000 → P Q R S T U V W X Y Z [\ ] ^ _
01100000 → ` a b c d e f g h i j k l m n o
01110000 → p q r s t u v w x y z { | } ~ DEL
10000000 →
10010000 →
10100000 →
10110000 →
11000000 →
11010000 →
11100000 →
11110000 →

Quiz

pwa.klicker.uzh.ch/join/asaeub

Strings vs. bytes

Code in notebook

Encoding/decoding strings in Python

```
>>> my_string = "Hi! 🤓"

>>> my_bytes = my_string.encode("utf-8")

>>> my_bytes
b'Hi! \xf0\x9f\xa4\x93'

>>> my_bytes[0]
72

>>> my_bytes.decode("utf-8")
'Hi! 🤓'
```

Bytes vs. strings

`bytes`:

- … is a sequence of **bytes**
  (= numbers between 0 and 255)
- … can store **any type of data** (text, image, audio, …)
- You have to know **how the content was encoded** (UTF-8, PNG, MP3, …) in order to interpret it
- … can be **decoded** to get a string

`str`:

- … is a sequence of **characters**
  (= Unicode code points)
- … can only store **text data**
- … is **independent of encoding**
- … can be **encoded** to get bytes

Code in notebook

Opening files in Python (text mode)

By default, Python assumes that a file contains text:

```
>>> my_file = open("my_file.txt")

>>> my_file
<_io.TextIOWrapper name='my_file.txt' mode='r' encoding='UTF-8'>

>>> type(my_file.read())
<class 'str'>
```

Code in notebook

Opening files in Python (text mode)

Reading from a file with the wrong encoding will raise an exception:

```
>>> my_file = open("my_file.txt", "ascii")

>>> my_file
<_io.TextIOWrapper name='my_file.txt' mode='r' encoding='ascii'>

>>> my_file.read()
UnicodeDecodeError: 'ascii' codec can't decode byte 0×f0 in
position 4: ordinal not in range(128)
```

Code in notebook

Opening files in Python (byte mode)

When specifying mode **"b"** (*byte*), Python will read raw bytes instead of strings:

```
>>> my_file = open("my_file.txt", "rb")
>>> my_file
<_io.BufferedReader name='my_file.txt'>
>>> type(my_file.read())
<class 'bytes'>
```

Default encodings in Python

- The default encoding for Python source files is UTF-8 on all platforms.
  → Always use UTF-8 to save your `*.py` files.
- ⚠️ The default encoding for opening files in text mode in Python depends on the platform:
  - Windows mostly uses CP-1252 (depending on locale settings)
  - Mac and Linux use UTF-8

  → Always specify: `open("file.txt", encoding="utf-8")`
- Starting with version 3.15 (to be released in 2026), Python will use UTF-8 as the default encoding for opening files everywhere

Inspecting binary files

- Try opening the file in a text editor (with different encodings)
- Open the file using byte mode in Python:
  - Print bytes to find ASCII characters
  - Use **`my_bytes.hex()`** to see bytes as hexadecimals
- Use hexdump on Linux/Mac:
  - Show bytes as hexadecimals:
    `$ hexdump my_file.bin`
  - Show bytes as printable ASCII characters along with hexadecimals:
    `$ hexdump -C my_file.bin`

Exercise: Inspecting a corrupt text file

1. Download the file **`corrupt.txt`** from OLAT.
2. **Inspect the file** and try to figure out what went wrong.
3. Try to **fix the file** and restore its original content using Python.

Unicode quirks

Code in notebook

Combining characters

a + ◌̩ = ạ

e + ◌́ = é

k + ◌͡◌ + p = k͡p

t + ◌̃ + ◌́ + ◌̩ = t̪̩̃́

Zalgo text:

```
>>> len("Unicode")
95
```

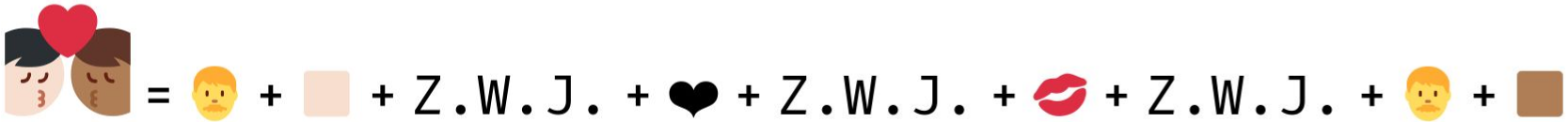
Normal forms

Some strings can look exactly the same, even though they are different:

```
>>> "é" == "é"
False
```

- **NFC:** Use single characters whenever possible: `len("é") == 1`
- **NFD:** Use combining characters whenever possible: `len("é") == 2`
- `unicodedata.normalize("NFD", "é")`

Emoji modifiers



Text-based data formats

Comma-separated values (**CSV**)

```
date,author,text
2018-05-01,Philomena Cunk,What a wonderful day
,Guido van Rossum,"Hello, world!"
2006-08-04,Borat Sagdiyev,"This suit is black ...

NOT!"
1637-08-04,Pierre de Fermat,aⁿ + bⁿ = cⁿ for n > 2
```

Comma-separated values (**CSV**)

```
date,author,text
2018-05-01,Philomena Cunk,What a wonderful day
,Guido van Rossum,"Hello, world!"
2006-08-04,Borat Sagdiyev,"This suit is black...

NOT!"
1637-08-04,Pierre de Fermat,aⁿ + bⁿ = cⁿ for n > 2
```

Header (optional)

Missing value

Comma-separated values (**CSV**)

```
date,author,text
2018-05-01,Philomena Cunk,What a wonderful day
,Guido van Rossum,"Hello, world!"
2006-08-04,Borat Sagdiyev,"This suit is black...

NOT!"
1637-08-04,Pierre de Fermat,aⁿ + bⁿ = cⁿ for n > 2
```

Code in notebook

Reading CSV files in Python

```
>>> import csv

>>> with open("tweets.csv", encoding="utf-8") as infile:
...     reader = csv.reader(infile)
...     for row in reader:
...         print(row)
```

```
['date', 'author', 'text']
['2018-05-01', 'Philomena Cunk', 'What a wonderful day']
['', 'Guido van Rossum', 'Hello, world!']
['2006-08-04', 'Borat Sagdiyev', 'This suit is black...\n\nNOT!']
['1637-08-04', 'Pierre de Fermat', 'aⁿ + bⁿ = cⁿ for n > 2']
```

How can we skip the header?

Code in notebook

Reading CSV files in Python

```
>>> import csv

>>> with open("tweets.csv", encoding="utf-8") as infile:
...     reader = csv.DictReader(infile)
...     for row in reader:
...         print(row)

{'date': '2018-05-01', 'author': 'Philomena Cunk', 'text': 'What a wor
{'date': '', 'author': 'Guido van Rossum', 'text': 'Hello, world!'}
{'date': '2006-08-04', 'author': 'Borat Sagdiyev', 'text': 'This suit
{'date': '1637-08-04', 'author': 'Pierre de Fermat', 'text': 'a^n + b^
```

Code in notebook

Writing CSV files in Python

```
>>> import csv

>>> with open("new.csv", "w", encoding="utf-8") as outfile:
...     writer = csv.writer(outfile)
...     writer.writerow(["name", "age"])
...     writer.writerow(["Martha", 36])
...     writer.writerow(["Carl", 19])
```

Code in notebook

Writing CSV files in Python

```
>>> import csv

>>> with open("new.csv", "w", encoding="utf-8") as outfile:
...     writer = csv.DictWriter(outfile, ["name", "age"])
...     writer.writeheader()
...     writer.writerow({"name": "Martha", "age": 36})
...     writer.writerow({"name": "Carl", "age": 19})
```

CSV has many dialects

- Column separators: commas, tabs, semicolons, …
- Row separators: `\n`, `\r\n`, `\r`
- Text encodings: any

CSV is **not standardized** and different programs interpret it differently.

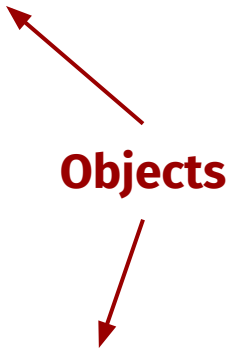
→ There are usually better options for sharing and archiving data

JavaScript Object Notation (JSON)

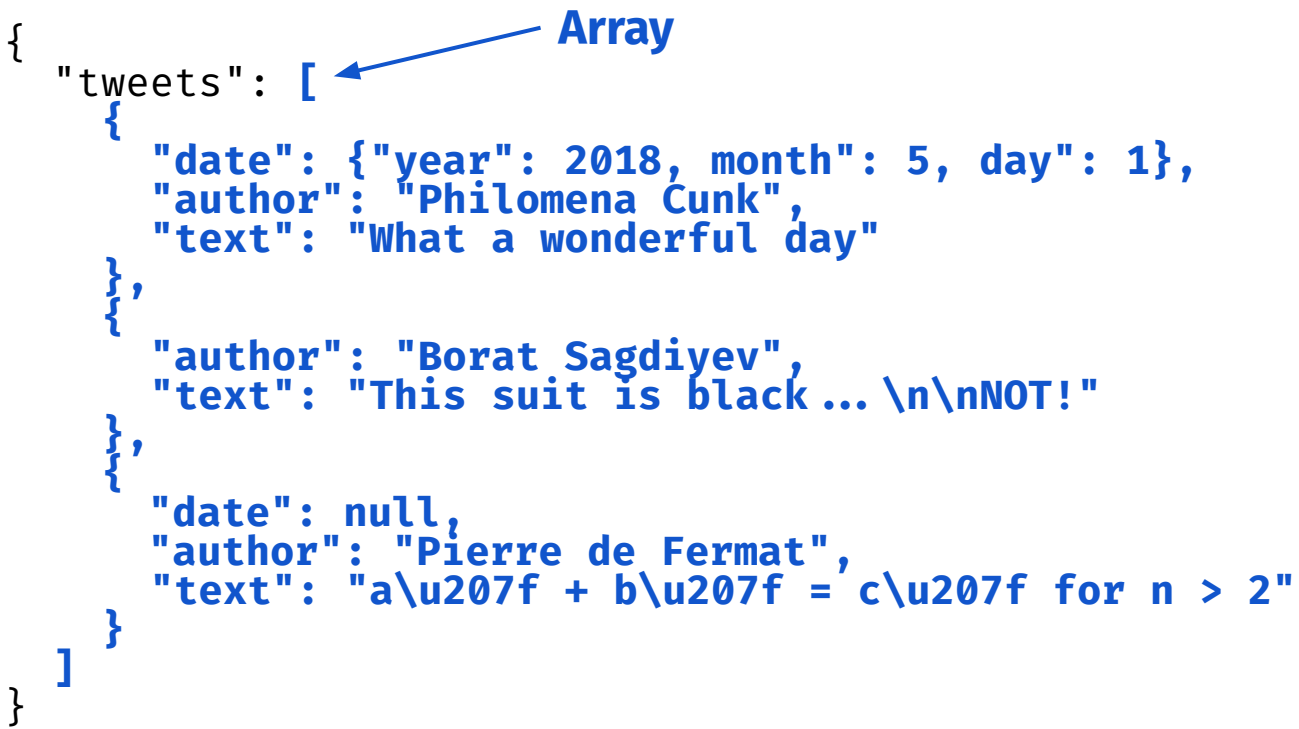
```
{
  "tweets": [
    {
      "date": {"year": 2018, month": 5, day": 1},
      "author": "Philomena Cunk",
      "text": "What a wonderful day"
    },
    {
      "author": "Borat Sagdiyev",
      "text": "This suit is black ... \n\nNOT!"
    },
    {
      "date": null,
      "author": "Pierre de Fermat",
      "text": "a\u207f + b\u207f = c\u207f for n > 2"
    }
  ]
}
```

JavaScript Object Notation (JSON)

```
{
  "tweets": [
    {
      "date": {"year": 2018, month": 5, day": 1},
      "author": "Philomena Cunk",
      "text": "What a wonderful day"
    },
    {
      "author": "Borat Sagdiyev",
      "text": "This suit is black ... \n\nNOT!"
    },
    {
      "date": null,
      "author": "Pierre de Fermat",
      "text": "a\u207f + b\u207f = c\u207f for n > 2"
    }
  ]
}
```



JavaScript Object Notation (JSON)



JavaScript Object Notation (JSON)

```
{
  "tweets": [
    {
      "date": {"year": 2018, month": 5, day": 1},
      "author": "Philomena Cunk",
      "text": "What a wonderful day"
    },
    {
      "author": "Borat Sagdiyev",
      "text": "This suit is black... \n\nNOT!"
    },
    {
      "date": null,
      "author": "Pierre de Fermat",
      "text": "a\u207f + b\u207f = c\u207f for n > 2"
    }
  ]
}
```

Character escaping

- Some characters in JSON string literals have special (non-literal) meanings:
  - **Double quotes:** `"Hello"`
  - **Backslashes:** `"Hello\nworld"`
- If we want to use these characters in their literal meaning, we have to **escape** them using a backslash (same as in Python strings):
  - `"He said \"hello!\""`
  - `"In JSON, \\n means newline"`
- There are no raw string literals in JSON!

Code in notebook

Reading JSON files in Python

```
>>> import json

>>> with open("tweets.json", encoding="utf-8") as infile:
...     data = json.load(infile)

>>> data
{'tweets': [{'date': {'year': 2018, 'month': 5, 'day': 1},
'author': 'Philomena Cunk', 'text': 'What a wonderful day'},
{'author': 'Borat Sagdiyev', 'text': 'This suit is
black ... \n\nNOT!'}, {'date': None, 'author': 'Pierre de
Fermat', 'text': 'aⁿ + bⁿ = cⁿ for n > 2'}]}
```

Code in notebook

Writing JSON files in Python

```
>>> import json

>>> with open("new.json", "w", encoding="utf-8") as outfile:
...     data = {"example": [1, 2, 3]}
...     json.dump(data, outfile)
```

Data types in JSON and Python

Type in JSON	**Type in Python**
Number	`int` or `float`
String	`str`
Boolean (`true`, `false`)	`bool` (`True`, `False`)
`null`	`None`
Object	`dict`
Array	`list`

JSON Lines: the best of JSON and CSV

- One JSON value (object/array/string/…) per line
- Line delimiter: `"\n"`
- Encoding: UTF-8 (just like JSON)
- File extension: `.jsonl`

Extensible Markup Language (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black …

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>aⁿ + bⁿ = cⁿ for n &gt; 2</text>
    </tweet>
</tweets>
```

Extensible Markup Language (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black ...

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>aⁿ + bⁿ = cⁿ for n &gt; 2</text>
    </tweet>
</tweets>
```

XML declaration

Extensible Markup Language (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black ...

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>aⁿ + bⁿ = cⁿ for n &gt; 2</text>
    </tweet>
</tweets>
```

Element containing text

Element containing other elements

Self-closing element (no content)

Extensible Markup Language (XML)


```
<?xml version="1.0" encoding="UTF-8"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black …

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>aⁿ + bⁿ = cⁿ for n &gt; 2</text>
    </tweet>
</tweets>
```

Extensible Markup Language (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black …

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>aⁿ + bⁿ = cⁿ for n &gt; 2</text>
    </tweet>
</tweets>
```

Extensible Markup Language (XML)

```
<?xml version="1.0" encoding="ASCII"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black ...

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>a&#8319; + b&#8319; = c&#8319; for n &gt; 2</text>
    </tweet>
</tweets>
```

Extensible Markup Language (XML)

```
<?xml version="1.0" encoding="UTF-8"?>
<tweets>
    <tweet id="1847263">
        <date year="2018" month="5" day="1" />
        <author>Philomena Cunk</author>
        <text>What a wonderful day</text>
    </tweet>
    <tweet id="4732614">
        <author>Borat Sagdiyev</author>
        <text>This suit is black ...

        <b>NOT!</b></text>
    </tweet>
    <tweet id="10423298">
        <date year="2018" month="5" day="1" />
        <author>Pierre de Fermat</author>
        <text>a^n + b^n = c^n for n &gt; 2</text>
    </tweet>
</tweets>
```

Every XML document has exactly one root element

Code in notebook

Reading XML files in Python

```
>>> import xml.etree.ElementTree as ET

>>> tree = ET.parse("tweets.xml")
>>> tree
<xml.etree.ElementTree.ElementTree object at 0×7f8ff4e574f0>

>>> texts = tree.findall("./tweet/text")
>>> texts
[<Element 'text' at 0×7f8ff4b2b790>, <Element 'text' at
0×7f8ff4b2b970>, <Element 'text' at 0×7f8ff4b2ba60>, <Element
'text' at 0×7f8ff4b2bc90>]

>>> texts[-1].text
'aⁿ + bⁿ = cⁿ for n > 2'
```

XPath

XPath

`./tweet/author`

... matches any **`author`** element which is a **child of a `tweet`** element which is a **child of the selected element**

`.//author`

... matches any **`author`** element which is a **(direct or indirect) descendant** of the selected element

`.//date[@year="2018"]`

... matches any **`date`** element which has an **attribute `year` with value `2018`** and is a (direct or indirect) descendant of the selected element

ElementTree API

- `element.find(tagname_or_xpath: str)`
- `element.findall(tagname_or_xpath: str)`
- `element.iterfind(tagname_or_xpath: str)`
- `element.text`
- `element.attrib`

Code in notebook

Exercise: Extracting information from XML

1. In the Jupyter notebook **`text_encoding.ipynb`**, scroll to the section **“Exercise: Extracting information from XML”**.
2. Run the first two cells to download and parse the XML file.
3. Look at the downloaded XML file (`archimob_1044.xml`).
4. Add code to **find the longest noun** (tag **NN**) in the document.

XML namespaces

```
<data>
    <table>
        <tr>
            <td>Apples</td>
            <td>Bananas</td>
        </tr>
    </table>

    <table>
        <name>African Coffee Table</name>
        <width>80</width>
        <length>120</length>
    </table>
</data>
```

XML namespaces

```
<data xmlns:h="http://www.w3.org/TR/html4/"
      xmlns:f="https://www.w3schools.com/furniture">
    <h:table>
        <h:tr>
            <h:td>Apples</h:td>
            <h:td>Bananas</h:td>
        </h:tr>
    </h:table>

    <f:table>
        <f:name>African Coffee Table</f:name>
        <f:width>80</f:width>
        <f:length>120</f:length>
    </f:table>
</data>
```

XML schemas

A schema defines how the elements in a namespace have to look like:

- What attributes are allowed/required?
- What child elements are allowed/required?
- What data types do the values have (numbers, strings, etc.)?
- …

Schemas are defined in **XML Schema Definition (XSD)** files. XSD files are XML documents themselves.

XML documents can be automatically **validated** against a schema.

Digression

XML schema example (https://www.w3schools.com/xml/schema_example.asp)

my_shiporder.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

shiporder.xsd:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

Writing XML files in Python

```
xml = f"""
<?xml version="1.0" encoding="UTF-9"?>
<data>
"""

for line in infile:
    xml += f"<sample>{line}</sample>\n"
xml += f"</data>"

outfile.write(xml)
```

🤦 do NOT do this! 🤦

Writing XML files in Python

✅ DO do this! ✅

```
>>> root = ET.Element("examples")

>>> ET.SubElement(root, "example")

>>> subelement = ET.SubElement(root, "example", id="123")
>>> subelement.text = "Content! <:-)"

>>> tree = ET.ElementTree(root)
>>> with open("new.xml", "wb") as outfile:
...     tree.write(outfile, xml_declaration=True, encoding="utf-8")
```

Overview

CSV:

- Tabular data
- Not really standardized, many dialects

JSON:

- Arbitrary nested data structures
- Commonly used in web APIs

XML:

- Arbitrary nested data structures
- Large ecosystem for computational linguistics (e.g. TEI)

Other tools for searching and editing CSV/JSON/XML

CSV:

- Spreadsheet programs (Excel, LibreOffice Calc, …)
- Command line: Miller, pwk

JSON:

- Command line: jq, Miller

XML:

- XML extension for VS Code
- Command line: xmllint

Quiz

pwa.klicker.uzh.ch/join/asaeub

Conclusion

Take-home messages

- **Unicode** is a standard that defines a unique integer for each character.
  - Use **`ord()`** and **`chr()`** to convert between Unicode code points and characters.
- **Text encodings** define how to convert text into “ones and zeros” and back.
  - **ASCII**, **Latin-1**, and **CP-1252** can only encode a small set of characters.
  - **UTF-8**, **UTF-16**, and **UTF-32** can encode all Unicode characters (with varying numbers of bytes).
- CSV, JSON, and XML are examples of **text-based data formats**.
  - **CSV** for tabular data, not standardized
  - **JSON** for nested data structures, very simple
  - **XML** for nested data structures, more complex, large ecosystem
    → namespaces, schemas, XPath

Midterm exam next week!

- Read the information page on OLAT (*Exercise & Exam Info*)
- Please be on time (start at **10:15**)
- Bring a **non-erasable pen** and your **student ID card**

You can do it!

Image generated by DALL·E 3