Topic 1: Introduction and Computational Foundations

LSE MY459: Computational Text Analysis and Large Language Models

https://lse-my459.github.io/

Ryan Hübert
Associate Professor of Methodology

Slides last updated: 18 January 2026

Why are you here?

Most social interactions involve communication

So, we are *awash* in natural language data:

- ➜ Social media posts
- ➜ Political statements
- ➜ Literature and artistic expression
- ➜ Laws, legal opinions, government records
- ➜ News articles
- ➜ Emails, text messages, WhatsApp messages
- ➜ LLM generated text(!!)

Analysing natural language data can reveal important things about the social world: politics, economics, culture, industry, finance, etc.

But how can we do it *at scale*? **That's the focus of this course**

Outline for today

- What is this course about?
- Course logistics
- Computational foundations

What is this course about?

What is this course about?

This course is about analysing natural language data (which we loosely call *texts*) with computational methods

Why do this? Justin Grimmer's "texts-as-haystack" metaphor:

- ➜ Analysing a straw of hay: understanding *meaning* of a text
  - ➜ Humans are great! But computer struggle
- ➜ Organising haystack: describing, classifying, extracting topics/meanings, etc.
  - ➜ Humans struggle. But computers are great!

Computational methods allow for analysis at huge scale—this is *awesome* for social science

Three "cultures"

Methods in this course are built on a foundation largely from computer science (hence "computational" in title)

- ➜ Sometimes called **natural language processing (NLP)**
- ➜ Focused on modelling language for its own sake, e.g. for industry applications like chatbots, translation, etc.
- ➜ Jurafsky and Martin (3rd ed) is the canonical textbook

Computational text methods have come to the "liberal arts" relatively recently

- ➜ Scholars in **digital humanities** use them to learn about texts
- ➜ Social scientists use them to learn about social phenomena, thus treating **text as data**

Three distinct "cultures" but they are intertwined

Three "cultures"

MY459 is primarily about doing social scientific analysis with methods adapted from computer science (text-as-data approach)

This requires us to straddle the line in terms of what we cover:

- ➜ Some statistics and quantitative methods theory
- ➜ Some coverage of computational issues
- ➜ Some discussion of social science research design

But our core goal: doing social science

Social science *first*

Social science analysis is typically interested in using data to learn about **positive theories**

From Gailmard (2014), ch. 1: "a *positive theory* in social science asserts that two or more concepts or events are related to each other and specifies the reason why."

A positive theory from Catalinac (2016): Japan's 1994 electoral reform led governing party politicians to put more political focus on national security issues

- ➜ Mechanism: decreased intra-party competition
- ➜ Two distinct concepts/events: *electoral reform* and *political focus*, related in a (specific) way

Social science *first*

The core question: does this theory teach us about the real world?

We can turn to data to try to answer this, but it's hard:

- ➜ How do we measure these concepts?
- ➜ How do we figure out whether there is any relationship?
- ➜ If we find evidence of a relationship, how do we know if it is good evidence? (And what does "good evidence" mean?)

Catalinac's key contribution: measure *political focus*

- ➜ Data source: candidate election manifestos
- ➜ Method: probabilistic topic modelling (LDA)

Then used standard quantitative techniques to show relationship

Social science *first*

Of interest for MY459: Catalinac's method for measuring focus

- ➜ We'll cover LDA in week 4!
- ➜ The "standard" analysis of the correlation is for other courses

More generally, we'll explore how CTA tools help with:

1. **Discovery**: "process of creating new conceptualizations or ways to organize the world" (GRS, p. 4)
2. **Measurement**: "process where concepts are connected to data, allowing us to describe the prevalence of those concepts in the real world" (GRS, p. 4)

First 6 topics: more "established" methods (more social science-y);
Last 4 topics: wild west of LLMs (more technical/NLP-y)

MY459 is a (quick) overview—lots more to learn if interested

A manifesto (of sorts)

Three core assumptions:

1. Texts are observable implications of underlying social phenomena
2. Texts can be represented by extracting their "features"
3. Text features can be quantified and analysed using quantitative methods to produce meaningful and valid inferences about social phenomena

Three core principles:

1. You must start with substantive question rooted in clear social science theory
2. Choice of method must be clearly matched to research question
3. A human must always be in the loop (lots of reading!)

Course logistics

Course meetings

Lectures: Mondays from 10:00 to 12:00 in CLM.3.02

- ➜ Lecture slides posted to GitHub no later than one hour before lecture (https://github.com/lse-my459/lectures)

Classes/seminars: only in weeks 2, 4, 7, 9, 11:

- ➜ *See LSE timetable for times and locations!*
- ➜ Two hours: mostly coding and doing exercises but some conceptual material too
- ➜ Materials posted no later than one hour before seminar (https://github.com/lse-my459/seminars)

No lectures or classes during Reading Week (week 6)

To audit: submit a request to the department here

Teaching team

Dr. Ryan Hübert
r.hubert@lse.ac.uk

- ➜ Course convenor
- ➜ Leads all lectures and classes/seminars during weeks 1-7
- ➜ Book office hours through StudentHub
- ➜ Please call me "Ryan"

Dr. Friedrich Geiecke
f.c.geiecke@lse.ac.uk

- ➜ Leads all lectures and classes/seminars during weeks 8-11
- ➜ Book office hours through StudentHub

Prerequisites

This is a technical graduate-level course

We will assume:

- You have exposure to calculus, elementary linear algebra and have taken quantitative methods courses up to and including the equivalent of MY452 (regression)
- You have access to a laptop that can run the required software for the course and that you will bring to class
- You are comfortable using Python, VS Code and Jupyter/iPython notebooks (see Moodle for more information)
- You are self-driven, and are eager to engage deeply with mathematical concepts and with Python code

Course resources

Course website: https://lse-my459.github.io/

- ➜ Course schedule and syllabus
- ➜ Links to lecture and seminar materials
- ➜ Reading lists: "primary" and "further"
- ➜ Weeks 1-7 will draw *heavily* from Grimmer, Roberts and Stewart (2022) (abbreviated GSR)

Course GitHub: https://github.com/lse-my459/

- ➜ Lecture slides and seminar materials

Moodle page: https://moodle.lse.ac.uk/course/view.php?id=13640

- ➜ Announcements and some administrative information
- ➜ Some (non-public) supporting materials

Assessments

Formative:

- ➜ Exercises during bi-weekly seminars
- ➜ Quiz at end of course—useful for exam prep

Summative: there is a two-hour pen-and-paper exam during Spring Term exam period, which is worth 100% of your mark

Please review department and school policies and information on assessments, including:

- ➜ Exam procedures (see here)
- ➜ Academic integrity (see here)
- ➜ Grading scales (see here)

Support

For administrative support:

- For auditing, registration, extensions, late assignments and department/school policy questions: methodology.admin@lse.ac.uk
- For questions about course platforms, including Moodle, website or other course-related platforms: r.hubert@lse.ac.uk
- For other administrative issues: r.hubert@lse.ac.uk
- For questions about course material: schedule office hours for the relevant instructors via StudentHub

Course outline

There are 10 topics, roughly corresponding to each week

- Lecture slides and seminar materials posted no later than one hour before beginning of lecture/seminar
- Some topics will span more than one lecture, some topics less than one lecture
- See the website for detailed outline

Some overlap with other courses, e.g. MY472, MY474, MY475

- This is a feature not a bug
- But here, our primary motivation is distinctive: do analysis with *text*
- We'll often discuss the peculiarities of applying "general" methods to texts
- Plus: many students don't take these courses!

Preparing for lectures and seminars

- Make sure that you have both Python and VS Code installed and working on your computer
  - Warning #1: the instructors are not tech support
  - Warning #2: this is a course about text methods, not a Python coding course
- We generally expect that you are constantly reviewing course materials, readings and other sources as we go
- This course is on a massive topic—we'll move fast!

Digital data

Digital data

Text analysis: learning things about the world from texts

Computational text analysis: using *computational* methods to learn things about the world from texts

➜ This is (obviously) a quantitative exercise

"Computational" does not necessarily imply using a computer for computations, but in practice it does (due to scale)

Modern CTA requires **digital data** that computers can read

Because CTA relies heavily on computers, we need to spend some time on "computer stuff"

Digital data

Digital text (and all digital data) is stored as **bits**:

- With n bits, can store 2^n patterns
- E.g., 2 bits of storage gives four possibilities: 00, 01, 10, 11
- A sequence of bits is referred to as **binary**

Bytes are an aggregation of bits: 8 bits $=$ 1 byte

- kilobytes (KB), megabytes (MB), gigabytes (GB), etc. are metric aggregations of bytes (roughly)
- $\neq$ kilobits (Kb), megabits (Mb), gigabits (Gb), etc.

Digital data

One byte can have 256 possible "values" (2^8) and each possible "value" can be written in several formats:

- **Binary**: a sequence of eight bits
- **Dec(imal)**: an integer from 0 to 255
- **Hex(adecimal)**: a two-character code from 00 to FF

Example: the byte `01110101` can be written in dec format as 117, or in hex format as 75

In programming: hex codes are often preceded with `0x`, e.g. `0x75`

You can have sequences of bytes, e.g. this two byte sequence:

- Binary: `11000011 10111100`
- Dec: 195 188
- Hex: C3 BC

Digital data

Bytes of digital data live on computers in two main places:

1. **Storage hardware** such as SSDs, hard drives, USB sticks, CDs, etc.

2. **Random access memory ("memory" or "RAM")**

Keep in mind: storage $\neq$ memory

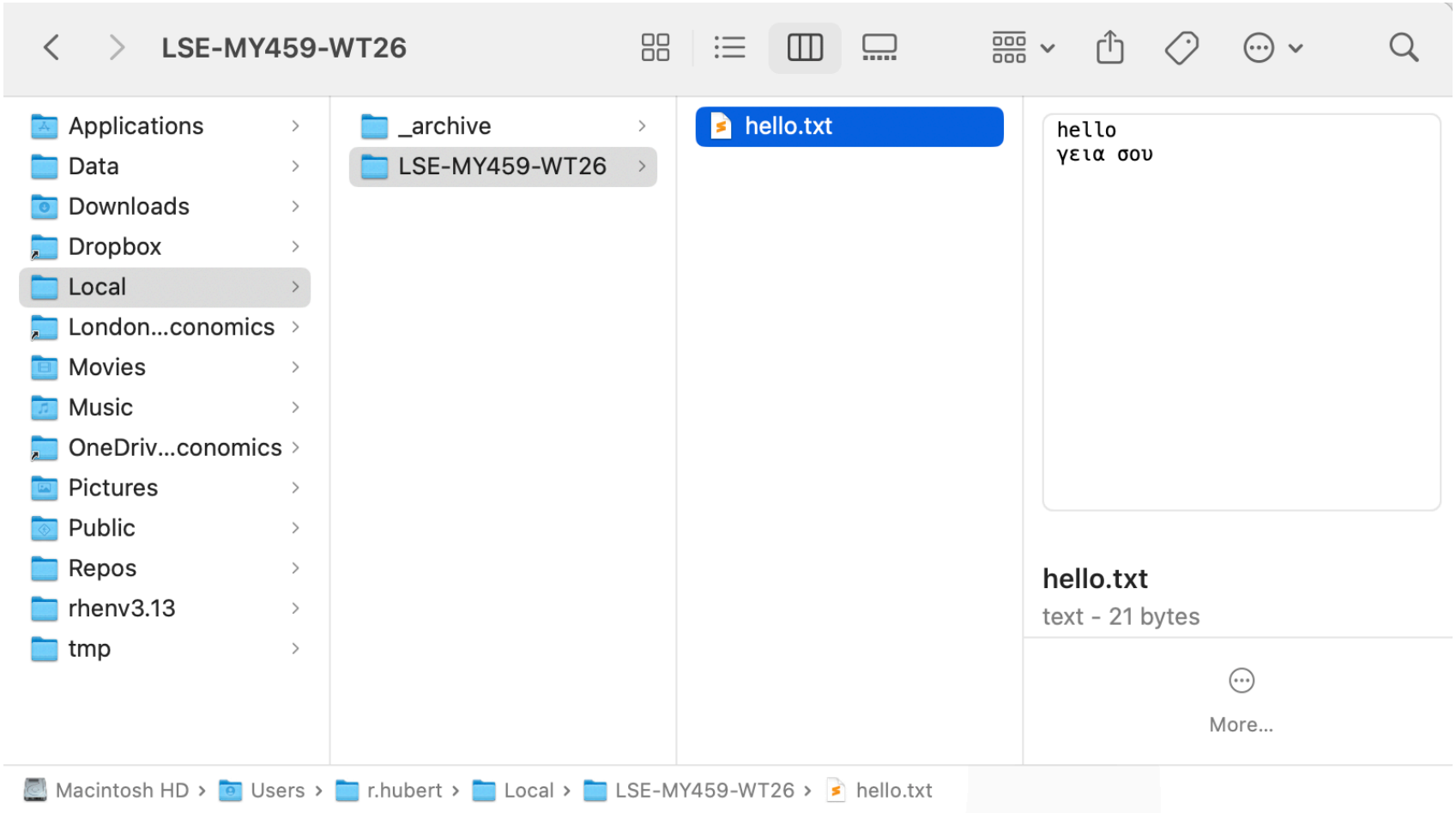
➜ Distinction: archiving versus active use

File Input/Output (I/O): refers to a set of technical issues around storing and accessing digital data on a device

➜ We will need to read *stored* data into Python

Reading digital data into Python

For example: `hello.txt` stored on my computer



Reading digital data into Python

(1) Specify where it is:

```
import os
home = os.path.expanduser("~")
wdir = os.path.join(home, "Local", "LSE-MY459-WT26")
wdir
```

```
'/Users/rhubert/Local/LSE-MY459-WT26'
```

```
my_file = wdir + "/hello.txt"
my_file
```

```
'/Users/rhubert/Local/LSE-MY459-WT26/hello.txt'
```

(2) Read the file's bytes directly into Python

- ➜ Recall that all digital data is stored as bytes
- ➜ Read raw bytes using `mode = "rb"` (for "read binary")

```
with open(my_file, mode = "rb") as f:
  my_data_raw = f.read()
```

Reading digital data into Python

`my_data_raw` is a `bytes` object (Python's type for raw bytes):

```
type(my_data_raw)
```

```
<class 'bytes'>
```

Python prints `bytes` objects in a weird way, mixing formats:

```
my_data_raw
```

```
b'hello\n\xce\xb3\xce\xb5\xce\xb9\xce\xb1 \xcf\x83\xce\xbf\xcf\x85'
```

- ➜ Some bytes are shown as Latin characters
- ➜ Other bytes are shown in hex (e.g., `\xb5` is hex code B5)

To see the hex codes for *all* the bytes:

```
# [hex(x) for x in my_data_raw] # also works
my_data_raw.hex(" ")
```

```
'68 65 6c 6c 6f 0a ce b3 ce b5 ce b9 ce b1 20 cf 83 ce bf cf 85'
```

Reading digital data into Python

Raw bytes can be expressed in multiple formats besides hex codes

To get the dec format of a specific byte, use `ord()`

```
ord(b"\xb5")
```

```
181
```

And to get binary, use `format()` with option `"08b"`

```
format(b"\xb5"[0], "08b")
```

```
'10110101'
```

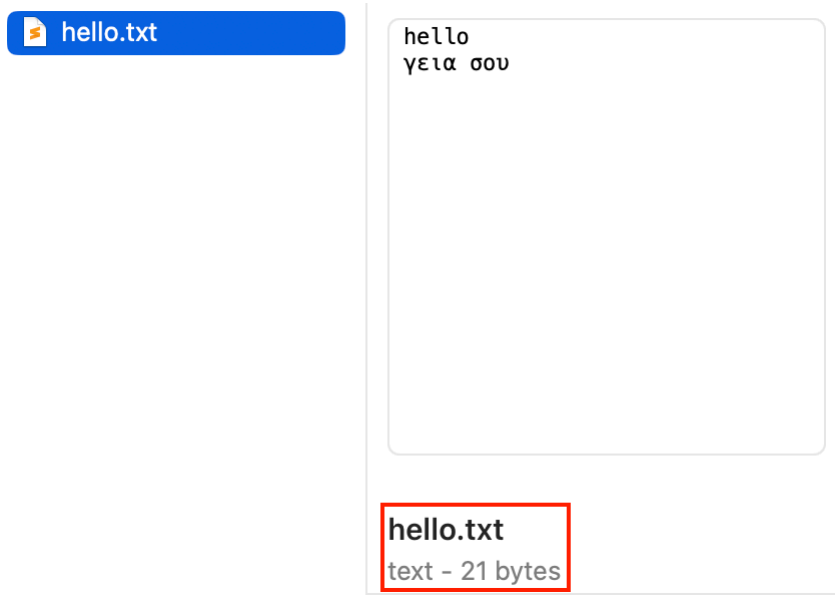
Reading digital data into Python

A file should be as big as the number of bytes in it

```
len(my_data_raw) # how many bytes did we read in?
```

```
21
```

In the computer file browser, verify the file size:



Reading digital data into Python

Recall: digital data is bits—this is how your computer stores it

Usually, when you read any digital data into Python, it "interprets" it for you under the hood

- ➜ But if you specify `mode = "rb"`, you're asking for minimal interpreting—just return the raw bytes as a `bytes` object

What if we don't ask for the raw bytes?

```
with open(my_file, mode = "r") as f:
  my_data = f.read()
my_data
```

```
'hello\nγεια σου'
```

Python knew how to convert raw bytes in the file to readable text—but how?

Character encoding

Character encoding

A **character encoding** defines how a machine "translates" bits/bytes of data into strings

A character encoding consists of:

- A **character set**: a list of characters with associated numerical representations
  - **Code points** are the unique "numbers" associated with characters in a character set
- An **encoding rule** telling a computer program how to render text when given raw bytes
  - An encoding rule specifies the number of bits used to represent characters, e.g. 7-bit, 8-bit, 16-bit, etc.

The origins of encoding: ASCII

ASCII: foundational character set/encoding, uses 7 bits

➜ Could only encode up to $2^7 = 128$ characters... not enough!

ASCII was later extended to 8 bits, e.g. ISO-8859-1

➜ Now could encode $2^8 = 256$ characters... still not enough!

Full tables here: original ASCII, ISO-8859-1

Example: hex code point for "A" is 41 with bits: `1000001` (original) `01000001` (extended)

As you can imagine: different languages, different characters → different character sets and encodings

A mess... see http://en.wikipedia.org/wiki/Character_encoding

Widely used character encoding today: Unicode

Unicode: standard for representing any character from any language

- ➜ Most recent version is Unicode 17.0 (September 2025)
- ➜ Core feature: universal character set
- ➜ Currently contains 159,801 characters from 172 "scripts"
- ➜ However, it can accommodate up to 1,114,111 characters

In Unicode, code points are unique integer IDs

- ➜ For example, the letter ü has the ID number 252

Unicode code points are conventionally written in a standardised hex format (more compact than integers)

- ➜ Code point for ü is usually written U+00FC ($\leftarrow$ not bytes!)

Widely used character encoding today: Unicode

In addition to the universal character set, Unicode contains encoding standards

These are called **Unicode Transformation Formats (UTFs)**

- ➜ Three most common: **UTF-8**, **UTF-16**, **UTF-32**

These UTFs differ by how many bytes they use:

- ➜ UTF-8 is **variable width encoding**: 1, 2, 3 or 4 bytes
- ➜ UTF-16 mostly uses 2 bytes, sometimes 4 bytes
- ➜ UTF-32 is a **fixed width encoding** using 4 bytes

UTF-8 is now the default character encoding for most digital data

- ➜ Accommdates vast number of characters very efficiently
- ➜ Backward compatible with ASCII

UTF-8 examples

	Code point	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	00100110			
u	U+0075	01110101			
ü	U+00FC	11000011	10111100		
Д	U+0414	11010000	10010100		
𝓐	U+120A	11100001	10001000	10001010	
🫠	U+1FAE0	11110000	10011111	10101011	10100000

UTF-8 examples

	Code point	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	26			
u	U+0075	75			
ü	U+00FC	C3	BC		
Д	U+0414	D0	94		
𝓐	U+120A	E1	88	8A	
	U+1FAE0	F0	9F	AB	A0

UTF-16 examples

	Code point	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	00100110	00000000		
u	U+0075	01110101	00000000		
ü	U+00FC	11111100	00000000		
Д	U+0414	00000100	00010100		
Ⴊ	U+120A	00010010	00001010		
🫠	U+1FAE0	11011000	00111110	11011110	11100000

Note: these are **little endian** where null bytes go last (instead of **big endian**)

UTF-32 examples

	Code point	Byte 1	Byte 2	Byte 3	Byte 4
&	U+0026	00100110	00000000	00000000	00000000
u	U+0075	01110101	00000000	00000000	00000000
ü	U+00FC	11111100	00000000	00000000	00000000
Д	U+0414	00010100	00000100	00000000	00000000
𝒜	U+120A	00001010	00010010	00000000	00000000
🫠	U+1FAE0	11100000	11111010	00000001	00000000

Note: these are **little endian** where null bytes go last (instead of **big endian**)

Character encoding in Python

In Python, text is typically stored as a `str` object, which are sequences of Unicode code points (one for each character)

```
ord("ü") # gives the Unicode code point for a character
```

```
252
```

```
chr(252) # gives the character for a Unicode code point
```

```
'ü'
```

```
[ord(x) for x in "Hübert"] # Unicode code points for a full string
```

```
[72, 252, 98, 101, 114, 116]
```

If you want to know the bytes that correspond to characters, you need to encode with a *specific encoding* (and format):

```
[hex(x) for x in "Hübert".encode("utf-8")]
```

```
['0x48', '0xc3', '0xbc', '0x62', '0x65', '0x72', '0x74']
```

Encoding and file types

All digital data is stored in files as bits/bytes, and they are therefore **machine-readable**

But *with an encoding*, a file can also be **human-readable**

- ➜ Many files are machine-readable but not human-readable
- ➜ All human-readable files are machine-readable
- ➜ Most of the time, when someone says a file is "machine-readable" they are implying *not* human-readable
- ➜ Files that are not human readable are often called **binary** files

There are "degrees" of human-readability, but at a minimum, a human-readable file must be *text-based*

But, encoding creates problems...

Potential encoding issues with human-readable files

1. Wrongly detected encoding

- ➜ When a plain text file is initially saved, it has an encoding
- ➜ But encoding is *not* stored as metadata in plain text files
- ➜ Software used to access plain text files guesses which encoding is used, sometimes incorrectly
- ➜ Assuming the wrong encoding when reading in/parsing a text file leads to import errors and corrupted characters
- ➜ This is known as mojibake: underlying bit sequences are translated into the wrong characters

Potential encoding issues with human-readable files

2. Space constraints

- ➜ Each bit used to represent a character uses storage
- ➜ 8 bit encoding uses less storage, but is not enough for a character set that has all known characters
- ➜ Encoding with 32 bits ($2^{32} \approx 4.3$ billion code points), however, ensures all known characters can be stored
- ➜ But, in most situations, it implies storing a lot of "unused" bits and unnecessarily large file sizes
  - ➜ For example, the letter "A" encoded with UTF-32 will be four times larger than when encoded with UTF-8

Things to watch out for

- Some text production applications (e.g. MS Office-based products) might still use proprietary character encoding formats, such as Windows-1252
- Some parts of Windows use UTF-16, while Unix-based platforms mostly use UTF-8
- Text editors can be misleading: they may display mojibake but the encoding might still be as intended
- Some computers use a different default encoding, e.g. if purchased in country not using Latin-based characters
- No easy method of detecting encoding in basic text files

File encodings in Python

Recall we read in `hello.txt` as a string earlier:

```
with open(my_file, mode = "r") as f:
  my_data = f.read()
my_data
```

```
'hello\nγεια σου'
```

When you use `mode = "r"`, Python tries to **decode** the raw bytes in the file into Unicode

- ➜ It *assumes* the computer's default encoding
- ➜ Most modern computers: this is UTF-8

But you can (and should) specify to be safe:

```
with open(my_file, mode = "r", encoding = "utf-8") as f:
  my_data = f.read()
my_data
```

```
'hello\nγεια σου'
```

File encodings in Python

What if the file is not encoded in UTF-8?

If you don't specify encoding, it assumes system default (probably UTF-8), and gives you an error when decoding fails:

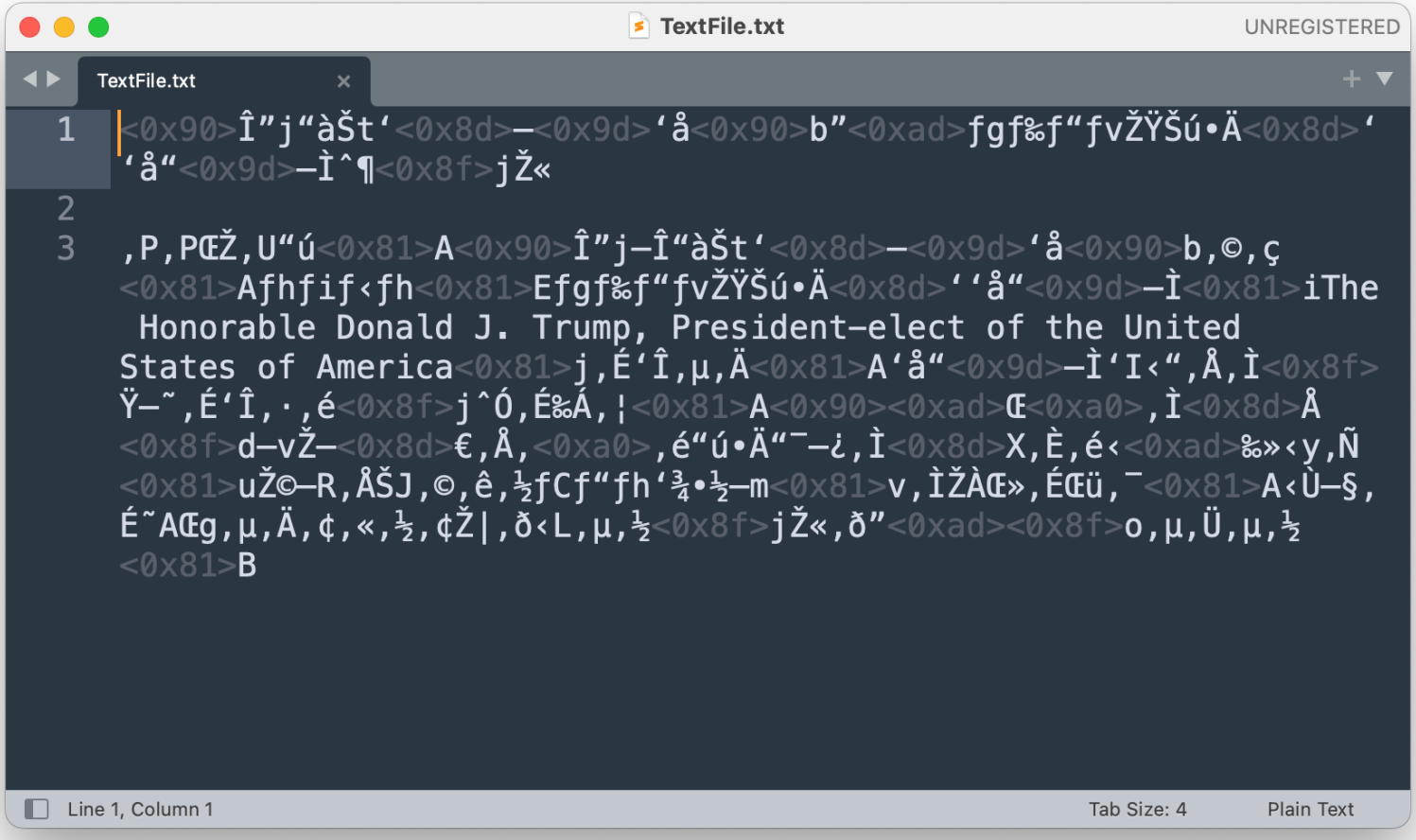
```
unknown_enc_file = wdir + "/TextFile.txt"
with open(unknown_enc_file, mode = "r") as f:
  unk_data = f.read()
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x90 in position 0: invalid
```

File encodings in Python

Not just a Python problem!

Text editor also assumes wrong encoding and shows mojibake:



File encodings in Python

What can you do?

First load the file as raw bytes:

```
with open(unknown_enc_file, mode = "rb") as f:
  unk_data = f.read()
unk_data[0:10] # print out the first 10 bytes
```

```
b'\x90\xce\x94j\x93\xe0\x8at\x91\x8d'
```

Use `charset_normalizer` module to guess encoding:

```
import charset_normalizer
print(charset_normalizer.detect(unk_data))
```

```
{'encoding': 'CP932', 'language': 'Japanese', 'confidence': 1.0}
```

Here: we see it's pretty confident — but it isn't always

File encodings in Python

Then, if you know encoding (or Python's guess seems promising) you can "decode" a character object:

```
my_string = unk_data.decode("CP932")
```

```
print(my_string)
```

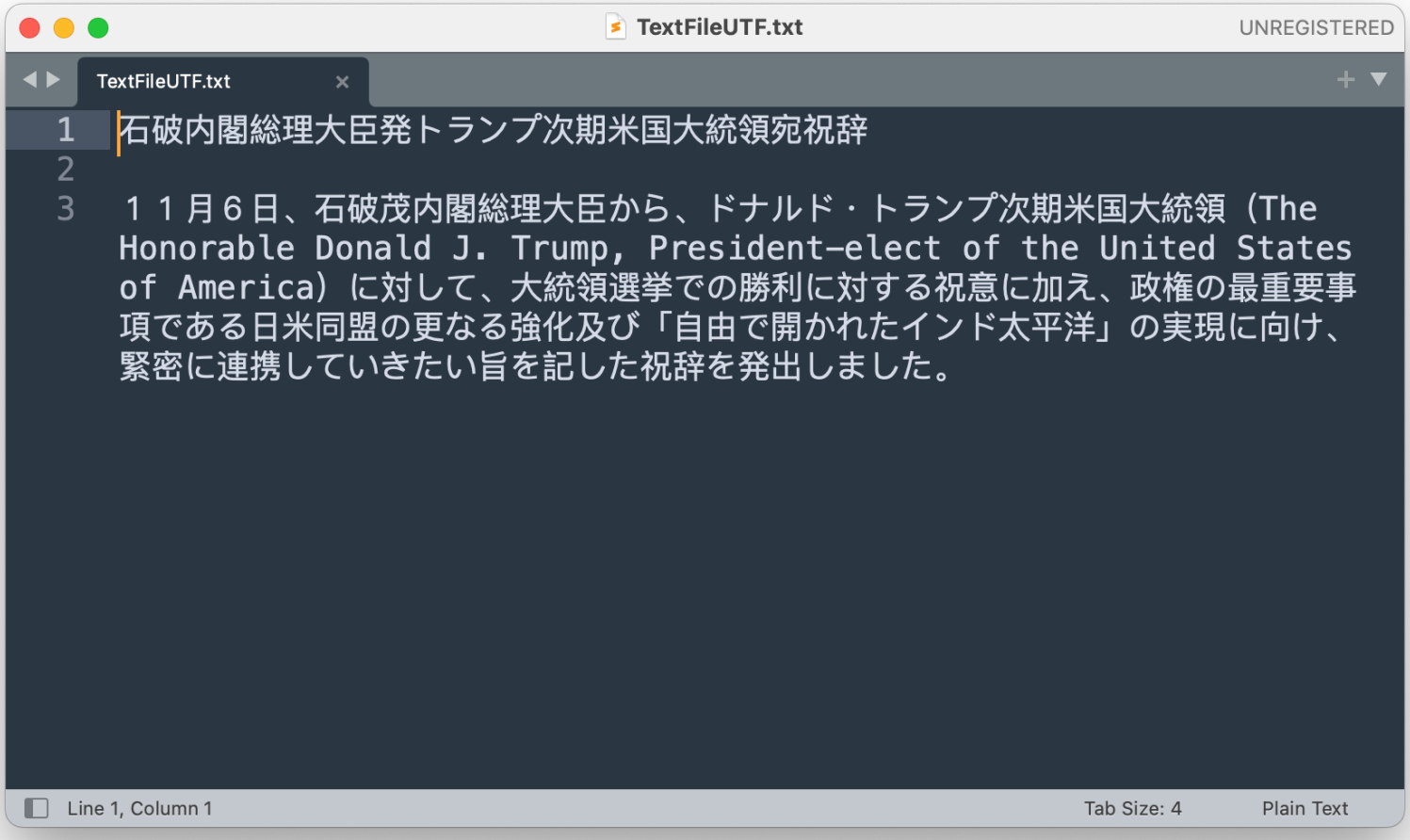
```
石破内閣総理大臣発トランプ次期米国大統領宛祝辞

１１月６日、石破茂内閣総理大臣から、ドナルド・トランプ次期米国大統領（The Honorable Donald
```

When satisfied, save the resulting text as UTF-8 for future

```
utf_enc_file = wdir + "/TextFileUTF8.txt"
with open(utf_enc_file, mode = "w", encoding = "utf-8") as f:
  f.write(my_string)
```

Note: `encoding = "utf-8"` is safe, but probably not required

File encodings in Python



File encodings in Python

A (much) more common problem these days: UTF-8 encoded file but with some *corrupted* text

```
corrupted_file = wdir + "/CorruptedFile.txt"
with open(corrupted_file, mode = "r") as f:
  c_data = f.read()
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xbb in position 14: invalid
```

If you *know* it's supposed to be encoded with UTF-8, you can opt to replace the corrupted characters:

```
with open(corrupted_file, mode = "r", errors='replace') as f:
  c_data = f.read()
c_data
```

```
'This file has corrupted text in it.'
```

String manipulation

Simple string manipulations with built-in methods

```
my_name = "Dr. Ryan" + " " + " H" # note two spaces
my_name
```

```
'Dr. Ryan  H'
```

```
my_name += "ubert "
my_name
```

```
'Dr. Ryan  Hubert '
```

```
my_name = my_name.replace("u", "ü")
my_name
```

```
'Dr. Ryan  Hübert '
```

```
my_name = my_name.strip()
my_name
```

```
'Dr. Ryan  Hübert'
```

Simple string manipulations with built-in methods

```
name_list = my_name.split(" ")
name_list
```

```
['Dr.', 'Ryan', '', 'Hübert']
```

```
name_list = [x for x in name_list if x != ""]
name_list
```

```
['Dr.', 'Ryan', 'Hübert']
```

```
name_list = [x.lower() for x in name_list]
name_list
```

```
['dr.', 'ryan', 'hübert']
```

```
name_list = [x for x in name_list if "dr" not in x]
name_list
```

```
['ryan', 'hübert']
```

Regular expressions

Searching for patterns in strings is very common in CTA

Regular expressions (usually called "regex") provide a powerful and flexible tool to search (and replace) text

➜ Similar to globs, but more powerful

Many editors that work with plain text (e.g. VS Code) can usually find and replace terms with regular expressions

Can also be used in many programming languages, e.g. when counting or collecting certain keywords in text analysis

Topic could fill lectures itself, we will cover some basics here

Regular expressions: syntax

Regular expressions can consist of literal characters and metacharacters

- **Literal characters**: usual text
- **Metacharacters**: `^ $ [] () {} * + . ?` etc.

When a meta character shall be treated as usual text in a search, **escape** it with a backslash `\` (unless it is in a set `[]`)

- For example:
  - searching for `.` will select *any* character
  - searching for `\.` will select the period

Regular expressions: syntax

Consider: "Dr Hübert teaches MY472 and MY459."

If you wanted to find all the mentions of MY courses in this string, then you could use any one of these regex patterns:

- `MY[0-9][0-9][0-9]`
- `MY\d\d\d`
- `MY[0-9]+`
- `MY.{3}`
- `MY[^A-z]{3}`

Some are better and some are worse, but regex gives *tonnes* of flexibility to find patterns

In Python, you indicate regex patterns using `r""`, such as:

```
import re
re.findall(r"MY\d+", "Dr Hübert teaches MY472 and MY459.")
```

```
['MY472', 'MY459']
```

Regular expressions: syntax

In addition to finding characters and "wildcards", regex allows for:

- **Booleans**, e.g., find one pattern *or* another
- **Capturing groups**, find a specific group of characters (good for extracting later)

For example:

- `r"MY(459|472)"` finds either MY459 or MY472 (or both)

In this pattern, `(459|472)` contains a boolean but it is also a capturing group

Regular expressions: syntax

```
rh = "Dr Hübert teaches MY472 and MY459."
re.search(r"(MY)(459|472)", rh).group(0)
```

```
'MY472'
```

```
re.search(r"(MY)(459|472)", rh).group(2)
```

```
'472'
```

```
re.findall(r"(MY)(459|472)", rh)
```

```
[('MY', '472'), ('MY', '459')]
```

```
re.findall(r"MY(?:459|472)", rh)
```

```
['MY472', 'MY459']
```

```
re.sub(r"(MY[^A-z ]{3})", "LSE \\1", rh)
```

```
'Dr Hübert teaches LSE MY472 and LSE MY459.'
```

Extra slides (important review)

Digital data in Python

We'll work with digital data within Python, where it is contained in **objects** (some mutable, some not, see extra slides)

There are several basic object types in Python:

Object type	Python	Example
Bytes literal	`bytes`	`x = b"hello"`
Float	`float`	`x = 3.14`
Integer	`int`	`x = 42`
String	`str`	`x = "hello"`
Logical	`bool`	`x = True`
List	`list`	`x = [1, 2, 3]`
Tuple	`tuple`	`x = ("1", "2", "3")`
Dictionary	`dict`	`x = {"a": 1, "b": 2}`

Other object types available in modules (e.g. data frames)

Digital data in Python

Let's create an object named `x` that contains the word "hello"

We use the equal sign to **assign** the name `x` to the string "hello":

```
x = "hello"
x
```

```
'hello'
```

Since it is text, we can see above it is stored as a `str` object

We can also confirm using `type()`:

```
type(x)
```

```
<class 'str'>
```

Global scope

When you assign objects, they are in Python's **global scope**:

```
"x" in globals() # check the name "x" is in global scope
```

```
True
```

```
globals()["x"]   # what object is assigned to the name?
```

```
'hello'
```

This means they are occupying your computer's memory, and actively available for use (while current Python session is running)

`sys` module allows you to see how much memory an object uses

```
import sys
sys.getsizeof(x) # in bytes
```

```
46
```

Global scope

You can also clear objects from memory (scope) in Python using:

```
del x
```

To verify this works, you can again check:

```
"x" in globals() # check the name "x" is still in global scope
```

```
False
```

And of course, if you try to use it, you'll get an error:

```
print(x)
```

```
NameError: name 'x' is not defined
```

Life lesson: **read error messages!**

Mutability

In Python, some objects are **immutable** while others are **mutable**

- ➜ Immutable: once created, they cannot be changed in place (mutable is opposite)
- ➜ Immutable objects work on a **copy-on-modify** logic: if you change a immutable object, a new object is created in memory

Other OOP languages are different in this respect, e.g. R's objects are all immutable

Mutability

Object type	Python	Example	Mutable?
Bytes literal	`bytes`	`x = b"hello"`	No
Float	`float`	`x = 3.14`	No
Integer	`int`	`x = 42`	No
String	`str`	`x = "hello"`	No
Logical	`bool`	`x = True`	No
List	`list`	`x = [1, 2, 3]`	Yes
Tuple	`tuple`	`x = ("1", "2", "3")`	No
Dictionary	`dict`	`x = {"a": 1, "b": 2}`	Yes

Mutability

Let's see how copy-on-modify works for immutable objects:

```
x = "hello"
id(x)  # memory "address" for `x`
```

```
4793138656
```

```
x = 1.2
id(x)  # Copy-on-modify
```

```
4792334800
```

```
z = x
id(z)  # `z` is an alias for `x`
```

```
4792334800
```

```
z = 7
id(z)  # `z` is now its own object
```

```
4785632200
```

Mutability

Compare to mutable objects, like `list` objects:

```
y = ["hello", "hola", "hei"]
id(y)  # memory "address" for `y`
```

```
4795222144
```

```
y.append(["olá"])
id(y)  # same address!
```

```
4795222144
```

```
y[0] = "hi"
id(y)  # same address!
```

```
4795222144
```

Type coercion in python

You can also **coerce** objects from one type to another:

```
y = 1.2
type(y)
```

```
<class 'float'>
```

```
y
```

```
1.2
```

```
y = str(y)
type(y)
```

```
<class 'str'>
```

```
y
```

```
'1.2'
```

Type coercion in python

In some contexts, Python does automatic (implicit) coercion:

```
3 + 2.5  # coerced int to float
```

```
5.5
```

```
3 + True # coerced bool to int
```

```
4
```

But it fails under ambiguous operations, like:

```
"3" + 2  # cannot coerce
```

```
TypeError: can only concatenate str (not "int") to str
```

Python doesn't know if you want to paste ("32") or add (5)