Week 2: Fundamentals of Digital Data

MY472: Data for Data Scientists
https://lse-my472.github.io/

Autumn Term 2025

Ryan Hübert
Associate Professor of Methodology

Slides last updated: 7 October 2025

Outline

What is data?

Data: representations of facts or observations

- ▷ Symbols, numbers, words, measurements
- ▷ E.g. “All”, “the”, “world’s”, “a”, “stage”

Information: meaning drawn from data

- ▷ Context + organization
- ▷ E.g. “All the world’s a stage”

A key idea: **data doesn’t speak for itself**

- ▷ Raw data alone doesn’t guarantee useful information
- ▷ Human needs to decide what information they want and then organise and interpret data to reveal that information

Making sense of data

Data science is all about making sense of data

- ▷ That is: turning data into information

A lot (if not all) data used by data scientists is **digital data**

- ▷ Digital data represents information as **bits**: 0s and 1s

Humans need a “computer” to turn digital data into information

- ▷ This is why we will spend a lot of time on “computer stuff”
- ▷ But, a computer cannot—on its own—turn data into information a data scientist finds useful (not even modern AI… week 11)

Storing digital data

A computer **stores** digital data as bits

- ▷ With n bits, can store 2^n patterns
- ▷ E.g., 2 bits of storage gives four possibilities: 00, 01, 10, 11

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 is stored in files that are written to (and read from) some kind of storage hardware

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

Storing digital data

Storage hardware varies by *mode* of file I/O:

- ▷ **flash storage**: no moving parts
  - Very fast I/O
  - Examples: SSDs, USB sticks, SD cards, modern laptops
- ▷ **magnetic storage**: magnets write to spinning “platters”
  - Slower I/O
  - Examples: internal/external hard drives, floppy disks
- ▷ **optical storage**: lasers write to disks
  - Slowest I/O
  - Examples: CDs, DVDs, Blue-ray

Reading stored digital data

Reading data: process of retrieving stored digital data

Simplifying quite a bit: when you "open" a file, you are "reading it into memory" so that you can actively work with it

- ▷ **Random access memory ("memory" or "RAM")**: where the data you retrieved resides until you clear it from memory (e.g. by closing an application)
- ▷ Most devices these days have at least 8 GB of RAM
- ▷ More RAM $\rightarrow$ your computer can work with more data simultanously

Keep in mind: storage $\neq$ memory

- ▷ Distinction: archiving versus active use

Reading stored digital data

Operating systems host software designed to help you retrieve and work with stored digital data

Some examples:

- Microsoft Word is an application designed to read data stored in certain file types, most notably `.docx` and `.doc`
- Positron is an application designed to read data from plain text scripting files (mostly R and Python) and execute code
- Web browsers are designed to read data from web files (`.html`, `.css`, etc.) and render visual webpages

Outline

Creating digital data

How do you *create* new digital data?

1. Make it yourself on the fly
2. Source it from somewhere else—i.e., **copy/paste** either locally or via a network from remote source

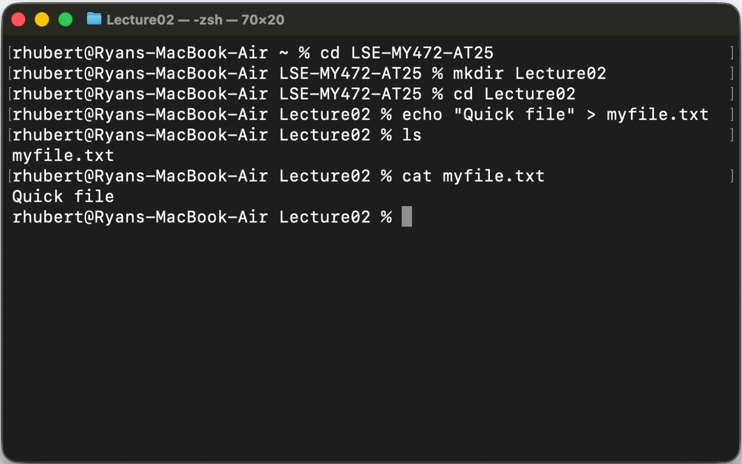
Many, many, many ways to create digital data

▷ You do this all the time: writing notes, taking pictures, talking on the phone, posting on social media, etc.

▷ I will show you a simple example with command line

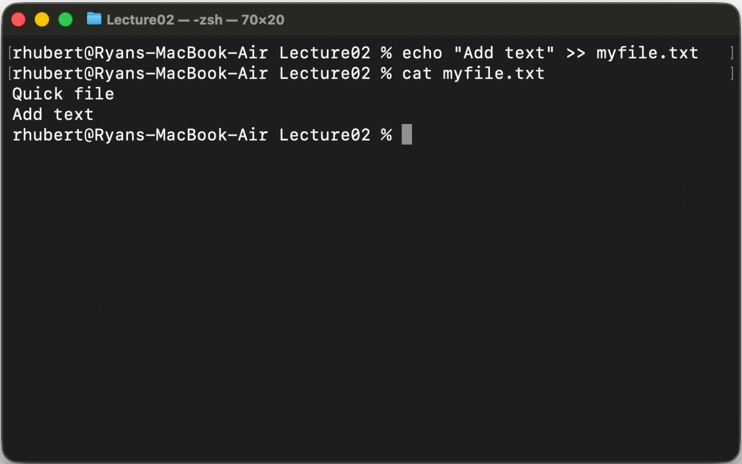
▷ Later, we will do it with R

Creating digital data



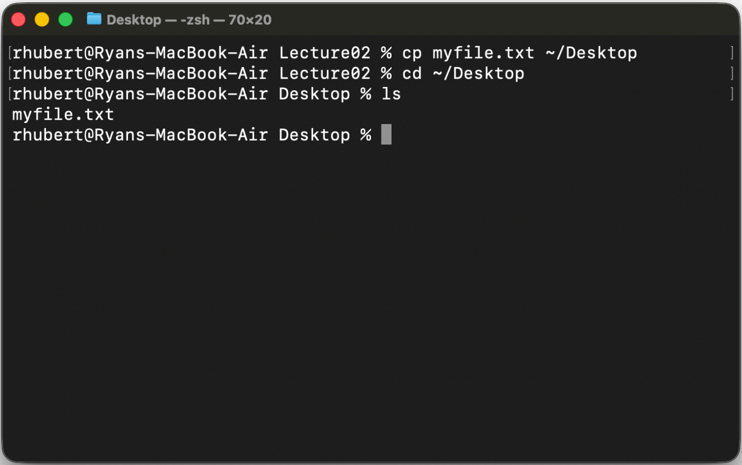
```
rhubert@Ryans-MacBook-Air ~ % cd LSE-MY472-AT25
rhubert@Ryans-MacBook-Air LSE-MY472-AT25 % mkdir Lecture02
rhubert@Ryans-MacBook-Air LSE-MY472-AT25 % cd Lecture02
rhubert@Ryans-MacBook-Air Lecture02 % echo "Quick file" > myfile.txt
rhubert@Ryans-MacBook-Air Lecture02 % ls
myfile.txt
rhubert@Ryans-MacBook-Air Lecture02 % cat myfile.txt
Quick file
rhubert@Ryans-MacBook-Air Lecture02 %
```

Creating digital data



```
rhubert@Ryans-MacBook-Air Lecture02 % echo "Add text" >> myfile.txt
rhubert@Ryans-MacBook-Air Lecture02 % cat myfile.txt
Quick file
Add text
rhubert@Ryans-MacBook-Air Lecture02 %
```

Creating digital data



```
rhubert@Ryans-MacBook-Air Lecture02 % cp myfile.txt ~/Desktop
rhubert@Ryans-MacBook-Air Lecture02 % cd ~/Desktop
rhubert@Ryans-MacBook-Air Desktop % ls
myfile.txt
rhubert@Ryans-MacBook-Air Desktop %
```

Deleting (stored) digital data

When you "delete" a file:

- ▷ Device marks the location of the stored bits as **stale**
- ▷ As you store new data, stale bits (eventually) get overwritten

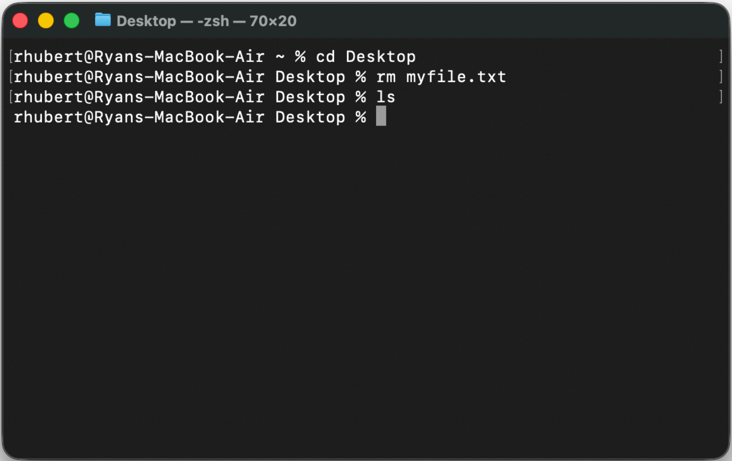
(There are some additional technical wrinkles for flash drives because they do not *directly* overwrite stale bits)

This has data security implications!

- ▷ "Deleted" data can be recovered if it hasn't been overwritten
- ▷ Use **secure erase** (especially on magnetic hard drives), and/or encrypt your storage drive (e.g., built-in Apple SSDs)

We'll discuss encryption a bit later in the course

Deleting (stored) digital data



Desktop — -zsh — 70×20

```
[rhubert@Ryans-MacBook-Air ~ % cd Desktop                              ]
[rhubert@Ryans-MacBook-Air Desktop % rm myfile.txt                     ]
[rhubert@Ryans-MacBook-Air Desktop % ls                                ]
 rhubert@Ryans-MacBook-Air Desktop % 
```

Outline

Digital data in R

We will usually work with digital data within R, where it is contained in **objects**

- R objects are **immutable**: once created, they cannot be changed in place
- R uses a **copy-on-modify** logic – if you change an object, a new object is created in memory

Other OOP languages are different in this respect

- E.g., in Python, some objects are immutable while others are mutable (and can be changed in place)

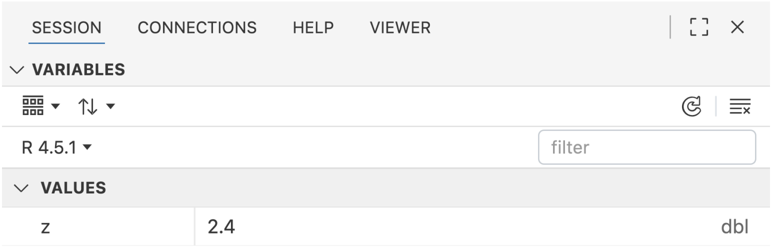
Interesting aside: flash drives use a **copy-on-write** logic and that's why you can't overwrite old data

Creating digital data in R

We can make an object named `z` that is the number 2.4:

```r
z <- 2.4
```

This object is now in R's **working environment** (it's occupying memory), as you can see in VARIABLES panel of Positron:



```
z    2.4    dbl
```

Creating digital data in R

More programmatic way to see what's in R's working environment:

```r
ls()
```

```
[1] "z"
```

The `{pryr}` package has some memory-management tools

```r
# install.packages("pryr") # if needed
pryr::mem_used()
```

```
48.8 MB
```

Side note: you can use functions from R packages without loading the entire package with `PACKAGE_NAME::FUNCTION_NAME`

Creating digital data in R

We can use {pryr} to see how copy-on-modify works:

```r
pryr::address(z) # memory address for `z`
```

```
[1] "0x11596c588"
```

```r
z <- 1.2
pryr::address(z) # Copy-on-modify
```

```
[1] "0x1284eb860"
```

```r
x <- z
pryr::address(x) # `x` is an alias for `z`
```

```
[1] "0x1284eb860"
```

```r
x <- 7
pryr::address(x) # `x` is now its own object
```

```
[1] "0x1284e9978"
```

Basic object types in base R

Objects `z` and `x` were **numeric** objects, but there are many other kinds of objects in R

R	Example
Numeric	`x <- 3.14`
Integer	`x <- 42L`
Character	`x <- "hello"`
Logical	`x <- TRUE`
Numeric vector	`x <- c(1, 2, 3)`
Character vector	`x <- c("1", "2", "3")`
List	`x <- list(1, "a")`
Named List	`x <- list(a=1, b=2)`

Note: these objects are not “tabular” data objects. (Next week!)

Basic object types in base R

Find out object type in Positron VARIABLES pane, or:

```r
str(x)
```

```
 num 7
```

```r
y <- c(1,2,3)
str(y)
```

```
 num [1:3] 1 2 3
```

```r
my.list <- list(a=1, b=2)
str(my.list)
```

```
List of 2
 $ a: num 1
 $ b: num 2
```

Clearing digital data from R

You can also clear objects from memory in R using:

▷ `rm()` – deletes names from R environment, removing bindings between symbols and objects

▷ `gc()` – triggers R's "garbage collector," reclaiming memory used by objects without names after `rm()` (usually not needed)

Do it for an individual object in your environment:

```
rm(z)  # removes name `z` from environment
gc()   # frees memory from unreachable objects
```

Or for *all* named objects:

```
rm(list = ls()) # removes all names from environment
gc()            # frees memory from unreachable objects
```

Reading digital data into R

Examples so far were making digital data on the fly

You can also read stored data into R’s working environment

There are several ways to do this, but we will mostly use the `{readr}` package in this class

- ▷ `{readr}` is part of the “tidyverse” suite of packages

Reading digital data into R

```r
library("readr")
my.path <- "~/LSE-MY472-AT25/Lecture02/myfile.txt"
my.data <- read_file_raw(my.path)
print(my.data) # raw bytes of data
```

```
 [1] 51 75 69 63 6b 20 66 69 6c 65 0a 41 64 64 20 74 65 78 74 0a
```

The `my.data` object contains the raw bytes of data in the file, expressed in **hexadecimal (hex) form** (read more here)

You can also view the text

```r
my.data.readable <- rawToChar(my.data) # convert bytes to text
print(my.data.readable)
```

```
[1] "Quick file\nAdd text\n"
```

Reading digital data into R

You can also see the bits, using this function:

```r
hex.to.bits <- function(raw){
  rawToBits(raw) |>
    as.integer() |>
    rev() |>
    paste(collapse = "")
}
```

To use it on the `my.data` vector of hex code bytes

```r
my.data.bits <- sapply(my.data, function(x) hex.to.bits(x))
print(my.data.bits)
```

```
 [1] "01010001" "01110101" "01101001" "01100011" "01101011" "00100000"
 [7] "01100110" "01101001" "01101100" "01100101" "00001010" "01000001"
[13] "01100100" "01100100" "00100000" "01110100" "01100101" "01111000"
[19] "01110100" "00001010"
```

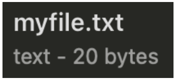
Reading digital data into R

File should be as big as the number of bytes:

```r
length(my.data.bits)
```

```
[1] 20
```

In the computer file browser, verify the file is 20 bytes



myfile.txt

text - 20 bytes

Note: you can also see bits of readable text using the `bits()` function from `{pryr}`

```r
pryr::bits("LSE")
```

```
[1] "01001100 01010011 01000101"
```

Reading digital data into R

What is going on here?

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

When you read data in R

- ▷ Under the hood, it is "interpreting" it for you
- ▷ When you use `read_file_raw()`, you're asking for minimal interpreting—just return **hex codes** for the bytes

We manually converted hex codes to "readable text" and it worked

- ▷ Million £ question: how did R know how to convert ***those*** bytes into ***that*** readable text?

Outline

Characters and strings

A **character** is the "smallest component of written language that has semantic value"

- ▷ See `https://unicode.org/glossary/#character`
- ▷ For example, "A", "ü", "2", etc.

A **(character) string** is a sequence of characters

- ▷ Strings are at the core of all of data science
- ▷ We've already worked with them quite a bit
- ▷ R refers to strings as "character" objects, which is somewhat imprecise

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
- These can be expressed in multiple formats

Encodings tell computer how to render text when given bytes

There are different encodings, using differing numbers of bits to represent characters, e.g. 7-bit, 8-bit, 16-bit, etc.

The origins of encoding: ASCII

ASCII: the original 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 an encoding standard that is now commonly used

- ▷ Has several encoding formats, e.g., **UTF-8**, **UTF-16**, **UTF-32**

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-16 and UTF-32 involve **null bytes** (`00000000`)

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

- ▷ Accommdates vast number of characters very efficiently

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		
ሊ	U+120A	11100001	10001000	10001010	
🫠	U+1FAE0	11110000	10011111	10101011	10100000

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		
ሊ	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
ሊ	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 R

Newer versions of R and Positron (and most modern software) use UTF-8 as their default character encoding

So, any digital text data created on the fly is UTF-8 encoded

- Also: all text displayed in the console or in the editor panel is also UTF-8 encoded

You can verify this using the `{pryr}` package

```r
pryr::bits("Ü") # this demonstrates UTF-8 encoding
```

```
[1] "11000011 10111100"
```

Character encoding in R

You can change the encoding of characters created on the fly:

```r
u16 <- iconv("ü", from = "UTF-8", to = "UTF-16LE", toRaw = TRUE)
print(u16[[1]]) # shows hex codes for bytes, not text
```

```
[1] fc 00
```

But you shouldn't do this without a good reason!

- ▷ You won't be able to print it as readable text in R/Positron:

```r
rawToChar(u16[[1]])
```

```
[1] "\xfc"
```

- ▷ More importantly: you'll create problems if you write to a file

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 minium, a human-readable file must be *text-based*

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 R

We already read in `myfile.txt` as raw bytes

We can use the `read_file()` to *directly* get the strings

```
my.data <- read_file("~/LSE-MY472-AT25/Lecture02/myfile.txt")
print(my.data) # a character string
```

```
[1] "Quick file\nAdd text\n"
```

But this only reliably works for files with characters that were encoded as UTF-8

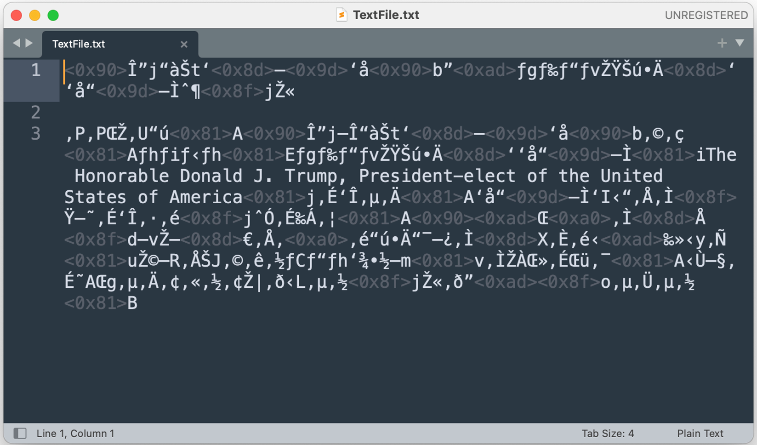
```
unk <- read_file("~/LSE-MY472-AT25/Lecture02/TextFile.txt")
print(unk)
```

```
[1] "\x90Δj\x93\xe0\x8at\x91\x8d\x97\x9d\x91\xe5\x90b\x94\xad\x83g\
```

This is mojibake: R assumes the wrong encoding

File encodings in R

Text editor also assumes wrong encoding and shows mojibake



```
<0x90>Î”j“àŠt‘<0x8d>—<0x9d>‘å<0x90>b”<0xad>ƒgƒ‰ƒ“ƒvŽŸŠú•Ä<0x8d>‘
‘å“<0x9d>—Ìˆ¶<0x8f>jŽ«

,P,PŒŽ,U“ú<0x81>A<0x90>Î”j—Î“àŠt‘<0x8d>—<0x9d>‘å<0x90>b,©,ç
<0x81>Aƒhƒifƒ‹ƒh<0x81>Eƒgƒ‰ƒ“ƒvŽŸŠú•Ä<0x8d>‘‘å“<0x9d>—Ì<0x81>iThe
 Honorable Donald J. Trump, President-elect of the United
States of America<0x81>j,É‘Î,µ,Ä<0x81>A‘å“<0x9d>—Ì‘I‹“,Å,Ì<0x8f>
Ÿ—˜,É‘Î,·,é<0x8f>jˆÓ,É‰Á,¦<0x81>A<0x90><0xad>Œ<0xa0>,Ì<0x8d>Å
<0x8f>d—vŽ—<0x8d>€,Å,<0xa0>,é“ú•Ä“¯—¿,Ì<0x8d>X,È,é‹<0xad>‰»‹y,Ñ
<0x81>uŽ©—R,ÅŠJ,©,ê,½ƒCƒ“ƒh‘¾•½—m<0x81>v,ÌŽÀŒ»,ÉŒü,¯<0x81>A‹Ù—§,
É˜AŒg,µ,Ä,¢,«,½,¢Ž|,ð‹L,µ,½<0x8f>jŽ«,ð”<0xad><0x8f>o,µ,Ü,µ,½
<0x81>B
```

File encodings in R

What can you do?

First, ask R to try to guess the encoding:

```r
guess_encoding(charToRaw(unk))
```

```
# A tibble: 2 x 2
  encoding     confidence
  <chr>             <dbl>
1 Shift_JIS          1
2 windows-1252       0.23
```

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

File encodings in R

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

```
x <- parse_character(unk, locale = locale(encoding = "Shift-JIS"))
print(x)
```

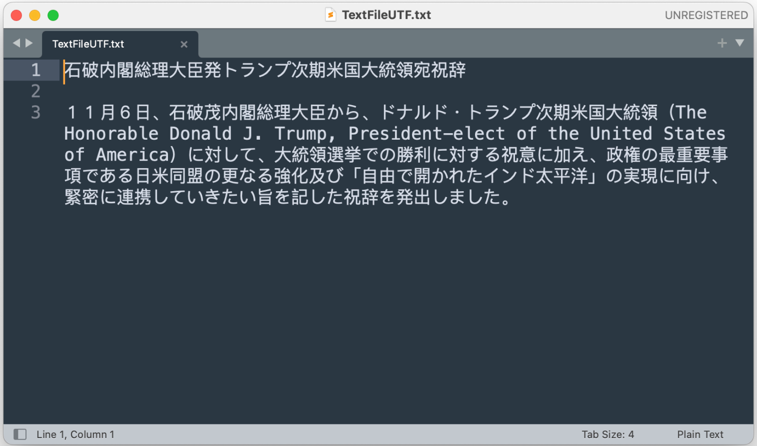
```
[1] "石破内閣総理大臣発トランプ次期米国大統領宛祝辞\n\n１１月６日、石破茂内閣総理
大臣から、ドナルド・トランプ次期米国大統領（The Honorable Donald J. Trump, Pr
esident-elect of the United States of America）に対して、大統領選挙での勝利
に対する祝意に加え、政権の最重要事項である日米同盟の更なる強化及び「自由で開かれた
インド太平洋」の実現に向け、緊密に連携していきたい旨を記した祝辞を発出しました。"
```

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

```
write_file(x, "~/LSE-MY472-AT25/Lecture02/TextFileUTF8.txt")
```

Note: since `x` is UTF-8 encoded, writing to file produces UTF-8 encoded bytes

File encodings in R



```
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 J. Trump, President-elect of the United States of America）に対して、大統領選挙での勝利に対する祝意に加え、政権の最重要事項である日米同盟の更なる強化及び「自由で開かれたインド太平洋」の実現に向け、緊密に連携していきたい旨を記した祝辞を発出しました。
```

Outline

Strings in R

Once you read a string into R, you can use the `{stringr}` package (again, in the tidyverse), to do things with your string

For example, with `{stringr}` you can

- ▷ find patterns, e.g., `str_detect()`
- ▷ “mutate” strings, e.g., `str_replace()`, `str_to_lower()`
- ▷ combine or split strings, e.g., `str_c()`, `str_split()`
- ▷ clean strings, e.g. `str_squish()`, `str_trim()`, `str_pad()`

Note: `str_c()` is equivalent to `paste()`/`paste0()` in base R

```r
paste(my.data, "Cool cool cool") # add characters
```

```
[1] "Quick file\nAdd text\n Cool cool cool"
```

Regular expressions

Searching for patterns in strings is very common in data science

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. Positron) 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

Caveat: R (mostly) does not use escaping, so `\.` is `[.]`, etc.

Regular expressions: syntax

Consider: “Ryan Hübert teaches on 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[0-9]+`
- `MY.{3}`
- `MY[^A-z]{3}`

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

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:

- `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

Strings in R

You can use regex patterns in many of these functions

```r
library("stringr")
rh <- "Ryan Hübert teaches on 472 and 459."
print(rh)
```

```
[1] "Ryan Hübert teaches on 472 and 459."
```

```r
str_replace_all(rh, "([^A-z ]{3})", "MY\\1")
```

```
[1] "Ryan Hübert teaches on MY472 and MY459."
```

Can do some of this in base R too

- E.g., `grep()` and `grepl()` are very common for finding elements of character vectors

```r
my.charv <- c("Ryan Hübert", "Beyoncé", "Bad Bunny")
grepl("ü", my.charv)
```

```
[1]  TRUE FALSE FALSE
```