

# Unit 01:

## Neville Longbottom and the mtcars dataset

### Wissenschaftliches Rechnen - R

Ferdinand Ferber and Wolfgang Trutschnig

Paris Lodron Universität Salzburg

2025-10-07

# Table of contents I

- 1 A primer on dataframes
- 2 A primer on data wrangling
- 3 A primer on ggplot2
- 4 A primer on tidymodels

## Course organization

- This course will be held as a UV, a combination of exercise and lecture.
- Coding needs to be learned hand-on, so little exercises await you within every session which we will do together.
- There will be weekly exercise sheets (and a nuudle).
- The attendance in this course is obligatory. You can miss at most one session without an excuse.
- @Grading: Maximal score: 100 points;  $\geq 61$  for passing
  - 40 points for solving exercises (and presenting solutions)
  - 20 points for active participation (Mitarbeit)
  - 40 points for the final exam/project



## Course philosophy

- Data-centric view
  - Exploring and understanding the data
  - Standard tools to analyze data
  - Handling challenging data (missing data, noise, label errors, high dimensionality, Big Data)
  - Forecasting
  - Clever feature engineering
- The course will be adjusted if necessary (depending on existing or lacking background knowledge).

## Course overview (to be adjusted)

- Units 1-4: R crash course, data visualization with `ggplot2` and modeling with the `tidymodels` framework
- Units 5-9: Challenging data
- Units 10-11: Feature engineering
- Units 12-14: Timeseries forecasting

# Neville Longbottom and the mtcars dataset



AI generated image using the prompt “Young Neville Longbottom programming in front of a computer wearing his Hogwarts school uniform and holding a toy car in one of his hands.”

# Neville Longbottom and the mtcars dataset

The Ministry of Magic collected a dataset on flying cars and published it in *The Quibbler* magazine. Neville Longbottom, who got a copy from Luna Lovegood, develops a fascination for flying cars and delves into Muggle Data Science to learn more about it.

# What kind of pics AI may produce



AI generated image (via Bing and DALL · E 3) using the prompt  
“Studying AI in Salzburg.”







# Atomic vectors

- atomic vector: simplest main data structures in R.
- It contains an ordered sequence (=vector) of values of basic types (integers, floating point numbers, complex floating point numbers, characters, logicals and raw bytes).

```
vec_int <- c(-2, -1, 0, 1, 2)
vec_num <- c(0.1, 5.2, 3.9e+4)
vec_cplx <- c(-1 + 3i, 4.7i, 2.1e-3 + 7.3e+4i)
vec_char <- c("Hello", "World", "!")
vec_raw <- as.raw(c(0xC0, 0xFE, 0xEE))
```

## Creating atomic vectors

- Create atomic vectors using the *combine* function `c()` as in the previous slide.
- There exist other functions to generate specific sequences as well:

```
x <- 1:10  
y <- seq(from = 1, to = 10, by = 3)  
z <- rep("bla", times = 5)
```

# Creating atomic vectors

- Notice, that the combine function does not nest vectors.
- It combines them into one vector.

```
c(c(1, 2), c(3, 4))
```

```
[1] 1 2 3 4
```

- Only identical types can be combined.

# Generating random numbers

- Important for statistical computing: generate samples of random variables.
- R provides a lot of probability distributions from where we can sample.

```
# Uniform distribution
runif(100, min = 0, max = 1)

# Normal distribution
rnorm(100, mean = 0, sd = 1)

# Poisson distribution
rpois(100, lambda = 4)

# Binomial distribution
rbinom(100, size = 10, prob = 0.5)
```

## Vectorized operations

- Numeric vectors support *vectorized* operations.
- Almost all operations in R are *immutable*, meaning that an expression like `2*x` will not modify `x`, but instead create a new object (which can be assigned a name):

```
x <- 1:10  
y <- 2 * x  
y
```

```
[1]  2  4  6  8 10 12 14 16 18 20
```

```
1 + y
```

```
[1]  3  5  7  9 11 13 15 17 19 21
```

- Vectorization is key for avoiding loops (speed).

# Indexing

- Multiple ways to do indexing/slicing in R:
- Positional: Given a vector of positive integers, the corresponding indices are returned in that order
- Exclusion: Given a vector of negative integers, all indices except the given ones are returned
- Logical: Given a vector of TRUE/FALSE values of the same length, the elements at the TRUE positions are returned

## Positional indexing

- When supplying a list of indices, R selects the corresponding indices and returns a new vector consisting of them in the given order.
- Indices can occur more than once, so the resulting vector can be larger than the original one:

```
x <- 1:10  
y <- c(2, 1, 1, 5)
```

```
x[y]
```

```
[1] 2 1 1 5
```

# Exclusion indexing

- When supplying a list of negative integers, R returns all indices *except* the given ones:

```
x <- 1:10  
y <- -c(2, 1, 1, 5)  
  
x[y]
```

```
[1] 3 4 6 7 8 9 10
```

# Logical indexing

- When supplying a list of TRUE/FALSE values of the same length, R returns all elements at the TRUE positions:

```
x <- 1:10  
y <- c(T,T,F,F,F,T,F,T,T,F)
```

```
x[y]
```

```
[1] 1 2 6 8 9
```

- This also works nicely with element-wise logical expressions:

```
x[x < 5]
```

```
[1] 1 2 3 4
```

## Getting and setting

- We can also overwrite one or several elements of a vector.  
Attention: This is a mutable operation.

```
x <- 1:10
```

```
x[3] <- 10
```

```
x
```

```
[1] 1 2 10 4 5 6 7 8 9 10
```

```
x[x<10] <- 0
```

```
x
```

```
[1] 0 0 10 0 0 0 0 0 0 10
```

## Exercise

- Neville wrote the following code in an imperative programming style.
- Can you use vectorization to improve readability and speed?
- Use, e.g., the `ifelse` command.

```
x <- runif(100)
for (i in 1:100) {
  y <- 2 * x[i] - 1
  if (y >= 0) {
    x[i] <- log(x[i])
  }
}
```

# Factors

- Factors contain categorical/discrete data. They are created with the `factor()` function:

```
x <- c("Jan", "Feb", "Jan", "Oct", "Feb", "May", "Apr")  
factor(x)
```

```
[1] Jan Feb Jan Oct Feb May Apr  
Levels: Apr Feb Jan May Oct
```

- Factors can be ordered as required (non lexicographic etc.)
- Will be useful in particular for plotting (in `ggplot2`)

## Coercing/converting data types

- R supports the usual data type coercion:

```
as.integer(c("1", "21"))
```

```
[1] 1 21
```

```
as.double(c("0.1", "5e-4"))
```

```
[1] 1e-01 5e-04
```

```
as.character(c(0.1, 5e-4))
```

```
[1] "0.1" "5e-04"
```

```
as.numeric(factor(c("a", "a", "b")))
```

```
[1] 1 1 2
```

# Dataframes

- Dataframes are potentially the most used data structures in R.
- Think of a dataframe (for the moment) as an Excel sheet or an SQL table.
- A dataframe can be created by providing a named list of columns, where each column is defined by an atomic vector<sup>1</sup>.

```
df <- data.frame(name = c("Alice", "Bob", "Charly"),  
                 age = c(30, 15, 55))  
df
```

|   | name   | age |
|---|--------|-----|
| 1 | Alice  | 30  |
| 2 | Bob    | 15  |
| 3 | Charly | 55  |

---

<sup>1</sup>We will see later (in the lecture *Albus Dumbledore and the nycflights13 dataset*) that we can put more involved/crazy stuff in a dataframe column.

# Extracting columns

You can extract the columns of a dataframe and get back the atomic vector:

```
df$name
```

```
[1] "Alice" "Bob"    "Charly"
```

Get the second entry of the column name:

```
df$name[2]
```

```
[1] "Bob"
```

# Parsing a dataframe

- Dataframes come usually in the form of a `.csv`, `.xlsx`, `.txt` file and we need to parse it first.
- There is a whole package dedicated to it in the tidyverse, the `{readr}` package.
- ...and there are many alternatives

```
df <- read_csv("some-tabular-data.csv", col_names=TRUE)
df <- read_delim("some-tabular-data.csv", col_names=TRUE)
df <- read.table("some-tabular-data.csv", head=TRUE)
```

# Exercise: Parsing a dataframe

- The Ministry of Magic collected of ATM withdrawals and Neville is keen to analyze it.
- Unfortunately, the dataset was provided as a .txt file.
- Download <http://www.trutschnig.net/ATM.txt> and parse it into a dataframe.

## Storing and reading a dataframe

- You don't want to store a dataframe in a .csv or .xlsx or .txt (unless you have to).
- You can save any R object (including dataframes) to disk and simply load it back in.
- Note, that .RData and .rds are R-specific (compressed) data formats.

```
# Some arbitrary R object
ATM <- read_delim("http://www.trutschnig.net/ATM.txt")

# Save it to disk (highly compressed)
saveRDS(ATM, "./ATM.rds")

# Load it back to memory
ATM2 <- readRDS("ATM.rds")
```



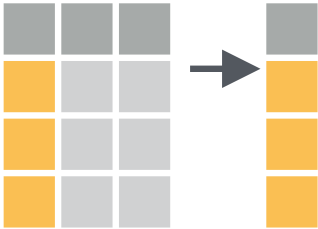








# Select<sup>3</sup>



- The `select` verb lets you select one or more columns from the input dataframe.

```
dplyr syntax: mtcars |> select(cyl)
```

---

<sup>3</sup>Image taken from the *dplyr cheat sheet*, Posit Software, PBC

# Select

- Let's use it on the `mtcars` dataset:

```
mtcars |>  
  select(cyl, hp)
```

|                   | cyl | hp  |
|-------------------|-----|-----|
| Mazda RX4         | 6   | 110 |
| Mazda RX4 Wag     | 6   | 110 |
| Datsun 710        | 4   | 93  |
| Hornet 4 Drive    | 6   | 110 |
| Hornet Sportabout | 8   | 175 |
| Valiant           | 6   | 105 |







# Rename<sup>4</sup>



- The rename verb lets you rename one or more columns from the input dataframe.

```
dplyr syntax: mtcars |> rename(num_cylinders = cyl)
```

---

<sup>4</sup>Image taken from the *dplyr cheat sheet*, Posit Software, PBC

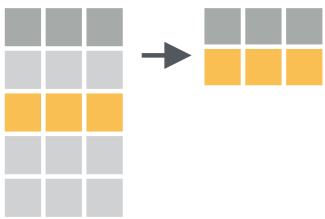
# Rename

- Here is an example on the mtcars dataset:

```
mtcars |> rename(num_cylinders = cyl,  
                 displacement = disp)
```

|                   | num_cylinders | displacement |
|-------------------|---------------|--------------|
| Mazda RX4         | 6             | 160          |
| Mazda RX4 Wag     | 6             | 160          |
| Datsun 710        | 4             | 108          |
| Hornet 4 Drive    | 6             | 258          |
| Hornet Sportabout | 8             | 360          |
| Valiant           | 6             | 225          |

# Filter<sup>5</sup>



- The `filter` verb lets you filter (=subset) rows of the input dataframe by testing them against a given predicate.

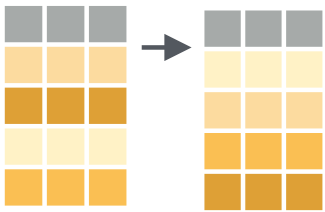
`dplyr` syntax: `mtcars |> filter(mpg < 15)`

---

<sup>5</sup>Image taken from the *dplyr cheat sheet*, Posit Software, PBC



# Arrange<sup>6</sup>



- The arrange verb lets you reorder the input dataframe based on one column or a combination of columns.

dplyr syntax: `mtcars |> arrange(mpg)`

---

<sup>6</sup>Image taken from the *dplyr cheat sheet*, Posit Software, PBC

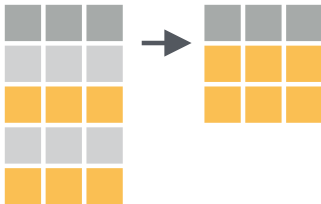
# Arrange

- An example on the `mtcars` dataset:

```
mtcars |> arrange(cyl, desc(dis))
```

|               | cyl | disp  | mpg  | hp  |
|---------------|-----|-------|------|-----|
| Merc 240D     | 4   | 146.7 | 24.4 | 62  |
| Merc 230      | 4   | 140.8 | 22.8 | 95  |
| Volvo 142E    | 4   | 121.0 | 21.4 | 109 |
| Porsche 914-2 | 4   | 120.3 | 26.0 | 91  |
| Toyota Corona | 4   | 120.1 | 21.5 | 97  |
| Datsun 710    | 4   | 108.0 | 22.8 | 93  |

# Slicing<sup>7</sup>



- There are several slice verbs that let you select a subset of rows of the input dataframe.

---

<sup>7</sup>Image taken from the *dplyr cheat sheet*, Posit Software, PBC

# Slicing

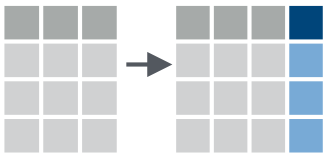
```
# Selects rows 5 to 10 (inclusive)
mtcars |> slice(5:10)
```

```
# Selects the first/last 5 rows
mtcars |> slice_head(n = 5)
mtcars |> slice_tail(n = 5)
```

```
# Selects 5 random rows
mtcars |> slice_sample(n = 5)
```

```
# Selects the 5 rows with the lowest/highest wt value
mtcars |> slice_min(wt, n = 5)
mtcars |> slice_max(wt, n = 5)
```

# Mutate<sup>8</sup>



- The mutate verb lets you modify or create new columns based on expressions of the existing ones of the input dataframe.

```
dplyr syntax: mtcars |> mutate(kW = hp * 1.3404)
```

---

<sup>8</sup>Image taken from the *dplyr cheat sheet*, Posit Software, PBC

























# Boxplots

- boxplot visualize univariate distributions.

```
df |> ggplot(aes(x = mpg)) + geom_boxplot()
```

