Empirical research using R:
in economics, finance, and management
Essentials, real examples, and troubleshooting

Compiled from session03.tex @ 2024-04-26 17:16:40+02:00.

Day 3: Special object types in R

Andreï V. KOSTYRKA

25th of September 2023

UNIVERSITÉ DU LUXEMBOURG

Quick recap

We learned:

- How to write comprehensible code
- How to run R within RStudio, install packages, and call functions
- How to work with vectors and matrices

Today, we learn how to manipulate other fundamental data types used in empirical analyses.

Presentation structure

1. Logical operations, conditions, loops
2. Loading and subsetting main data types
3. Special values, dates, structures
4. Text manipulation
5. Devilish details and pitfalls

Logical operations, conditions, loops

Logical class

In programming, one of the basic variable types is boolean: `TRUE` or `FALSE`.

- On / off, 1 / 0, flowing electric current / no current...

In R, the shorthands are `T` for `TRUE` and `F` for `FALSE`. One cannot name a variable `TRUE` or `FALSE` (protected keywords):

```
T             # Returns TRUE
T <- 327      # Works
TRUE <- 327 # Error
```

Logical operations

- AND `&`, OR `|`, NOT `!`
- Arithmetic: `<`, `<=`, `>`, `>=`, `==`, `!=`
- Special functions for checking if an object is of a certain type: `is.matrix()`, `is.character()`, `is.finite()`, `is.na()`, `%in%`...
  - `is.logical()` exists, too
  - `exists()` also exists: check if an object with this name is available: `x <- 3; exists("x"); exists("xx")`

Logical operation examples

```
TRUE & FALSE    # FALSE
TRUE | FALSE    # TRUE
!TRUE           # FALSE

2 < 3           # TRUE
1:4 < 3         # T, T, F, F
1:4 <= 3        # T, T, T, F

is.matrix(1:3)     # FALSE
is.vector(1:3)     # TRUE
is.character(1:3) # FALSE: it is "numeric"
is.character(as.character(1:3)) # TRUE

1:10 %in% 3:6  # F F T T T T F F F F
```

Checking conditions with `if` and `else`

```
x <- -0.5

if (x > 0) print("You rolled a positive number")
# Should print nothing

if (x > 0) {
  print("You are lucky today.")
} else {
  print("Woe upon ye.")
}
```

NB: The condition length must be 1!

NB: if the action contain only 1 instruction, the `{...}` braces can be omitted.

Checking multiple conditions

Use `else if` to check alternatives:

```
x <- -0.6
if (x > 0) {
  print("You are lucky, here is your regression:")
  print(lm(mpg ~ hp + wt, data = mtcars))
} else if (x < -0.01) {
  print("You are unlucky, here is your summary:")
  print(summary(mtcars))
} else { # When -0.01 <= x <= 0
  print("Your tiny negative number is ")
  print(x)
}
```

Nested conditions

Conditions can have multiple levels: it may be necessary to basic checks ('is it numeric?', 'are there ≥ *n* observations?', 'are there values < 0?') before context-dependent checks.

```
x <- 1:3
if (is.numeric(x)) {
  if (length(x) > 1) {
    print(sum(x))
  } else {
    print("Need at least 2 points.")
  }
} else print("Non-numeric x, cannot do maths.")
```

Try with `x <- 1` or `x <- "Lux"` (`sum("Lux")` → error).

Condition-checking specifics

- Do not write `if (x > 2 == TRUE)`: this is redundant at best and may cause errors at worst. Just `if (x > 2)`
- `if()` takes arguments of length 1. For checking vector conditions, use `ifelse()`

```
if (c(TRUE, TRUE)) print("Double TRUE!")
#> Error: the condition has length > 1

ifelse(1:20 %% 2 == 0, "Even", "Odd")
#>  "Odd"  "Even"  "Odd"  "Even" ...
```

`ifelse(cond1, val1, ifelse(cond2, val2, ...))` can be nested, but it is considered **bad practice**.

Condition check location

Think of the 'if-else' construct as of a function that returns an object that is immediately used. Save space and do not write redundant code:

```
a <- 2
if (a < 0) x <- 1:10 else x <- 11:20
```

can be changed without any loss of clarity to

```
a <- 2
x <- if (a < 0) 1:10 else 11:20
```

The check can be placed at any depth:

```
lm(mpg ~ hp,
   data = mtcars[if (a < 0) 1:10 else 11:20, ])
```

Playing an all-or-nothing game

Check if **all** elements of a logical vector are `TRUE` or any (at least one) is `TRUE`:

- `all(x) = x[1] & x[2] & x[3] & ...`
- `any(x) = x[1] | x[2] | x[3] | ...`

```
x <- 1:5
all(x < 8)  # TRUE
all(x < 3)  # FALSE
any(x < 3)  # TRUE
any(x < 0)  # FALSE
```

To check if all elements of x are `FALSE`, use `if (!any(x))`:

```
if (all(is.finite(x)) & !any(x <= 0))
  print("All elements of x are positive numbers.")
```

Switch statement

`switch()` is a handy shortcut to avoid multiple if-else's.

```
if (i == 1) doStuff()
  else if (i == 2) doFluff()
    else if (i == 3) dandruff() else ...
# can be replaced with
switch(i, doStuff(), doFluff(), dandruff(), ...)
```

`switch()` checks equality to a character string and returns the matching alternative. If there is no match, returns `NULL` without an error (more on `NULL` later).

```
x <- "cat"; y <- "fox"
x.snd <- switch(x, cat="meow", frog="croak")
x.snd   # "meow"
y.snd <- switch(y, cat="meow", frog="croak")
y.snd  # NULL
```

Loops

There are two types of loop in R: the **for** loop and the **while** loop. The former is more popular:

```
for (i in 1:10) {
  s <- i^2
  print(s)
}
```

Loops can iterate over any vector:

```
for (n in c("Asger", "Brynhildr", "Canute"))
  cat("God kveld,", n, "!\n")
```

NB: if the loop contains only 1 instruction, the `{...}` braces can be omitted.

‘While’ loops

‘While’ loops are useful when the termination condition depends on something happening inside the loop:

```
s <- 0
set.seed(1)
while (s < 100) {
  s <- s + rnorm(1, mean = 10)
  cat("The sum is", round(s, 2), "\n")
}
```

```
#> The sum is 9.37
#> The sum is 19.56
#> <...>
#> The sum is 101.32
```

Loops with conditions

Loops can be combined with conditions:

```
for (i in 1:100) {
  if (i %% 10 == 0) print(i)
}
#> 10 20 30 ...
```

Conditions can define anything – including iteratiors.

```
quick <- FALSE   # Try changing to TRUE
ii    <- if (quick) 1:10 else 1:100
for (i in ii) print(i)
```

A golfier (and slightly uglier) solution:

```
for (i in if (quick) 1:10 else 1:100) print(i)
```

Any questions on the logic control?

Loading and subsetting main data types

File types

In principle, R can read any file, but meaningful parsing requires format support.

- Built-in: plain text, native binary, or most popular versions of proprietary ones
  - Built-in `foreign` package for older versions of Minitab, SAS, SPSS, Stata, Weka
  - Libraries `readstata13` or `haven` for newer format specifications
- Can read genome data, images as arrays of pixel values etc. (outside the scope of this presentation)
- Can read and evaluate external R scripts: `source()`

How to store data for exchange?

- Simplest: plain-text CSV (universal)
  - **Best practice:** by default, exchange data with the rest of the world in plain-text / CSV format because *any software* can read them, even after decades
- Most non-R-user-friendly: write XLSX (appending sheets is possible)
  - Formulæ lost, only the values are preserved
- Most compact: native compressed format
- Complex structures: convert to XML, JSON, or YAML
- Or write a function that creates the desired layout (e. g. LaTeX formatted according to a specific style)

Data input

Multiple ways to load data into R:

- Text data: load lines as strings via `readLines()`
- Plain-text data separated with delimiters: `read.table()`, `read.csv()`
- Own binary format with efficient `gzip` or `xz` compression: `load()`, `readRDS()` (RData)
- Libraries for reading external formats (Excel, Stata DTA, SPSS SAV)
- Interfaces to connect to data bases (SQL)

Naming data sets

- On one hand, descriptive names are good
- But typing the same thing over and over hurts
  - My slide space is limited

Loading plain-text data

```r
# Reading a CSV table: ,-separated w/ decimal .
d  <- read.csv("mtcars.csv")

# CSV table in French locale: ; with decimal ,
d2 <- read.csv2("mtcars2.csv")
# A custom table with tab separators
d3 <- read.table("mttab.txt", sep = "\t")

save(d2, d3, file = "mydata.RData")
load("mydata.RData") # Loads d2 and d3
load("mydata.RData", verbose = TRUE)

# Text input from a text file (ASCII or Unicode)
aotm <- readLines("wsj-abreast-20070117.txt")
```

Loading Excel data

If there is rubbish in XLS(X) files (images, macros, external links etc.), some libraries may fail to load XLSX files – try a different package. If nothing helps, save XLSX as CSV.

```
install.packages("openxlsx") # Run only once
library(openxlsx)
r <- read.xlsx("remittance2017.xlsx") # 0.1 s

install.packages("readxl")
library(readxl)
r2 <- read_excel("remittance2017.xlsx")
# Takes 7 s! Non-standard return class (tibble)!

install.packages("xlsx")
library(xlsx)
r3 <- xlsx::read.xlsx("remittance2017.xlsx",
  sheetIndex = 1) # Takes 8 s + depends on JDK
```

Loading foreign data formats

```
install.packages("readstata13") # Run only once
# Comment the line above after the initial run
library(readstata13)
ab <- read.dta13("abdata.dta")

# Two packages to load SAS data
install.packages(c("sas7bdat", "haven"))
library(sas7bdat)
d1 <- read.sas7bdat("suivia.sas7bdat")
# Fails due to the input format quirks
library(haven)
d2 <- read_sas("suivia.sas7bdat")
d2 <- as.data.frame(d2)
```

NB. haven functions return **tibbles** – non-standard tables not understood by other functions! Convert them to DFs.

Data-loading speed

Some packages are faster: `data.table::fread()` is an improvement over the base R `read.csv()`.

`s02-large.csv` has 32 000 rows and 11 columns:

```
install.packages("data.table")
library(data.table)
system.time(read.csv("s02-large.csv")) # 140 ms
system.time(fread("s02-large.csv"))    #  22 ms
```

Recall the load time differences between `openxlsx::read.xlsx()` and `xlsx::read.xlsx()`. Always compare the speed if multiple implementations are available.

Classes

Before we proceed to data transformation and research problems, we need to learn the specifics of data classes.

Every object in R belongs to a class (object-oriented programming).

Based on the class, certain functions (methods) behave differently.

Example: `summary()` prints summary statistics (min, mean, median, ...) for every variable if used on a data frame and a single table (with standard errors, p-values, ...) is used on a linear model.

Numeric class

The most popular class: simply write a number!

Numbers can be written with exponential notation, especially when they are large:

```
n <- 10000000  # 10^7, but hard to read
n <- 1e7       # Much better
n <- 10^7      # Worse: requires ^ evaluation
```

Integer class

Numbers with no decimal part so save space / simplify calculations. Created by postpending the letter 'L' or using `as.integer()` (the decimal part is dropped, not rounded).

```
c(class(4), class(4L))  # "numeric", "integer"
as.integer(c(3, 3.7))   # 3 3
```

Not all numbers can be represented as integers: we have 64 bits ⇒ gaps appear ⇒ large integers are approximated!

```
.Machine$integer.max                # 2147483647
class(.Machine$integer.max)         # "integer"
class(.Machine$integer.max + 1)     # "numeric"
x <- 1000^7;   y <- x + 1           # x = 1e21
y - x  # 0, but by now, you are not surprised
```

Integers are beneficial for big data

- An integer takes up 4 bytes of memory, a real number: 8 bytes of memory – cut down memory use
- Floating-point operations can be slower than integer ones – speed up calculations (especially matching)

Suppose that you are handling German census data.

```
object.size(ai <- as.integer(1:1e8))  # 0.4 GB
object.size(an <- as.numeric(1:1e8))  # 0.8 GB
# Real machines have these limits!
bi <- rep(1:1e4, 1e4)
bn <- as.numeric(bi)
setdiff(bi, ai) #  7.6 seconds on average
setdiff(bn, an) # 12.5 seconds on average
```

Logical to integer using which()

Find the positions of `TRUE` in a logical vector using `which()`.

```
x <- c(-2, 5, -7, -9, 0, 1)
a <- x > 0      # F T F F F T
p <- which(a)   # c(2, 6)
```

Useful to find matches (equality so something):

```
x <- c("Anna", "Bohumila", "Cyntia",
       "Dobroslav", "Emil")
which(x == "Emil")  # 5
```

Find matrix elements by row and column (array indices):

```
which(matrix(1:12, nrow = 4) == 7, arr.ind = TRUE)
# row col
#   3   2
```

Mathematical operations with logical

Since logical values `TRUE` and `FALSE` can be re-interpreted as 1 and 0 in mathematical context, apply functions to count the number / proportion of `TRUE` elements:

```
x <- c(T, F, T, T, F)
sum(x)   # 3
mean(x)  # 0.6
```

This is useful when something must be done based on proportions. Example: dropping the variables with more than 10% missing values involves checking the condition `colMeans(is.na(data)) >= 0.1` (because `is.na(data)` is a matrix of logical values, and its column means are between 0 and 1).

Character class

Character: a letter, digit, or symbol displayed as a single unit. **String:** an array of characters treated as a group.

Characters are created via quotation marks (`"` or `'`) or `as.character()`. R fully supports Unicode (UTF-8):

```
x <- c("D", "Δ", "Д", "DΔД")
nchar(x)  # 1  1  1  3
```

Save scripts in UTF-8 via *File – Save with encoding*.

- If Windows uses a locale with a funny code page, the text will appear broken if the editor assumes a different one
- Windows 10 (2022 update) would not allow R 4.1 to display accented or non-Latin letters (‘naïveté’, ‘ΔY’)

Matrices and arrays

Last time, we talked about matrices – arrays are their generalisations.

Array: regular multi-dimensional 'hypercube' of data, or a vector wrapped in multiple dimensions.

```
a <- matrix(1:12, nrow = 3, ncol = 4)
b <- array(1:24, dim = c(4, 3, 2))
b
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]   13   17   21
#> [2,]   14   18   22
#> [3,]   15   19   23
#> [4,]   16   20   24
```

List: umbrella vector class

List: collection of objects of any type and size, assembled together.

```
a <- list(1:3, matrix(1:12, nrow = 3),
          lm(mpg ~ hp + cyl, data = mtcars),
          "A string of ponies")
```

List elements can have names that can be set at creation:

```
list(boyheights  = c(176, 182, 175, 196),
     girlheights = c(159, 169, 172, 166))
```

We shall learn how to handle names very soon.

Subsetting

Subsetting: selecting or extracting parts of objects, or using indices to re-assemble objects into new objects.

Three types of subsetting in R:

1. Subsetting by integer index
2. Subsetting by logical vector
3. Subsetting by character name

Indexing in R starts with **1**, unlike languages with 0-indexing (e. g. in C, stst element is obtained via `x[0]`).

The function that extracts elements by index is `[` – square bracket, followed by the indices, followed by the closing `]`.

Subsetting by index

`[` accepts integer indices; commas separate dimensions.

- One-dimensional vectors: `x[1:3]`
- 2D matrices / data frames: `x[1:3, 5:6]`
  - Put nothing to select all sub-dimensions: `x[1:3, ]` selects rows 1–3 and all columns of x
  - If x is a vector, `x[1:3, ]` will produce an error
  - If a single row or column of a matrix is chosen, the dimensions are dropped (the object is simplified)
- Multi-dimensional objects: if x is a 4D array, `x[1:3, , 2, c(4, 9)]` selects rows 1–3, all columns, 3rd slice and elements 4 and 9 from the 4th dimension

Dimension dropping in arrays

If an index of length 1 is requested, the result is flattened.

Selecting only one matrix row/column (`x[, 3]` or `x[8, ]`) yields a vector with `NULL` dimensions. If a matrix with one row/column is required, add '`, drop = FALSE`':

```
x <- matrix(1:12, nrow = 4)
y <- x[, 3] # Vector
y[1:2, ]  # Error; only y[1:2] will work
w <- x[, 3, drop = FALSE] # Matrix with 1 col
w[1:2, ]  # Works
```

Arrays: 4D `x[1:3, , 2, c(4, 9)]` → 3D. The 3rd dimension is dropped: only one index (2) is requested. Use `x[1:3, , 2, c(4, 9), drop = F]` to keep it 4D.

Subsetting by name

If an object dimension has names, elements can be extracted with a character vector. Integer and name indices can be used simultaneously:

```
x <- matrix(1:12, nrow = 3)
colnames(x) <- c("SP500", "DAX", "MSFT", "GE")
x[1:2, c("SP500", "DAX")] # Same as x[1:3, 1:2]
```

Best practice: use meaningful names and do not hardcode indices! Names are immune to place changes.

```
m <- lm(mpg ~ hp + cyl)
coef(m)[3]       # Unclear; the EViews way
coef(m)["cyl"]   # The human way
# Even if m <- lm(mpg ~ cyl + hp) [swapped order],
# coef(m)["cyl"] will return the same thing
```

Use case for `drop = FALSE` with names

The user may select 1 or more columns in a matrix. (Maybe they are using a variable-selection procedure.)

```
good.vars <- c("mpg", "cyl")
rowSums(mtcars[, good.vars])  # Works
good.vars2 <- "mpg"
rowSums(mtcars[, good.vars2])
#> Error: 'x' must be an array of at least two dimensions
```

This `drop = FALSE` argument ensures that the object always has dimensions (even with 1 column) and matrix functions are applicable:

```
rowSums(mtcars[, good.vars2, drop = FALSE])
```

Subsetting by logical vector

If a logical vector has the same length as a dimension of x, subsetting with it keeps the elements where it is **TRUE**.

Keep only the columns that have sum > 20:

```
x <- matrix(1:12, nrow = 3)
s <- colSums(x)  # Sum by column: 6 15 24 33
s > 20  # FALSE FALSE TRUE TRUE
x[, s > 20]  # Keeps columns 3 and 4
```

If the logical vector has wrong length (e. g. 4 columns but 3 logical elements), it will be *silently* recycled – beware:

```
x[, c(T, F, T)]
# ncol(x) = 4, hence, it returns columns 1, 3, 4
# because TFT if recycled to length 4: TFTT
```

Subsetting with functions

Use functions to compute appropriate indices.

Return the last element of x and the last column of y:

```
x[length(x)]
y[, ncol(y)]
```

Return the last 3 columns of the matrix `m`. Use the fact that `ncol(w) = 50` ⇒ `ncol(w) - 2:0 = 50 - 2:0 = 48:50`.

```
w[, ncol(w) - 2:0]
```

Return every 4^{th} row of `m` starting from the 2^{nd}:

```
a[seq(2, nrow(a), by = 4), ]
```

Reversing the order

Use `rev()` to reverse a vector:

```
rev(1:10)  # 10 9 8 7 6 5 4 3 2 1
```

It can be done with `x[length(x):1]`, too.

`rev()` is not good with other data types:

- `rev()` unwraps a matrix into a long vector, losing dimensions
- `rev()` reverses the order of data-frame columns

For matrices and DFs, `x[NROW(x):1, ]` is the solution.

Fun fact: amazingly, `rev(x)` is slightly slower (nanoseconds, *do not worry in real applications!*) than `x[length(x):1]` because it checks if the length is non-zero first.

Getting the first / last part

The `head()` and `tail()` functions return the beginning / end of vectors (elements) or data frames (rows).

Return the first 10 elements of the vector x and the last 5 rows of the data frame y:

```
head(x, 10)
tail(y, 5)
```

NB. The `head()` and `tail()` functions are **slow** compared to direct subsetting. For various lengths of x (10, 100, 1000), `tail(x, 1)` is 20× slower than `x[length(x)]`, and `tail(x, 5)` is 7× slower than `x[length(x) - 4:0]`.

Subsetting lists

A list is a vector. Two main functions are applicable:

- `[` returns certain elements and preserves the outer list
- `[[` returns *only one* element and 'unpacks' it from the list

To select a sub-list of a list x, use `x[3:5]`. To 'extract' the element of a list, use `x[[3]]`.

- Unnamed elements are selected by index in brackets: `x[3:5]`, `x[[1]]`
- Named elements are selected by name / index or via the `$` operator: `x[["dog"]]`, `x$dog`, `x[c("dog", "cat")]`

Difference between [and [[

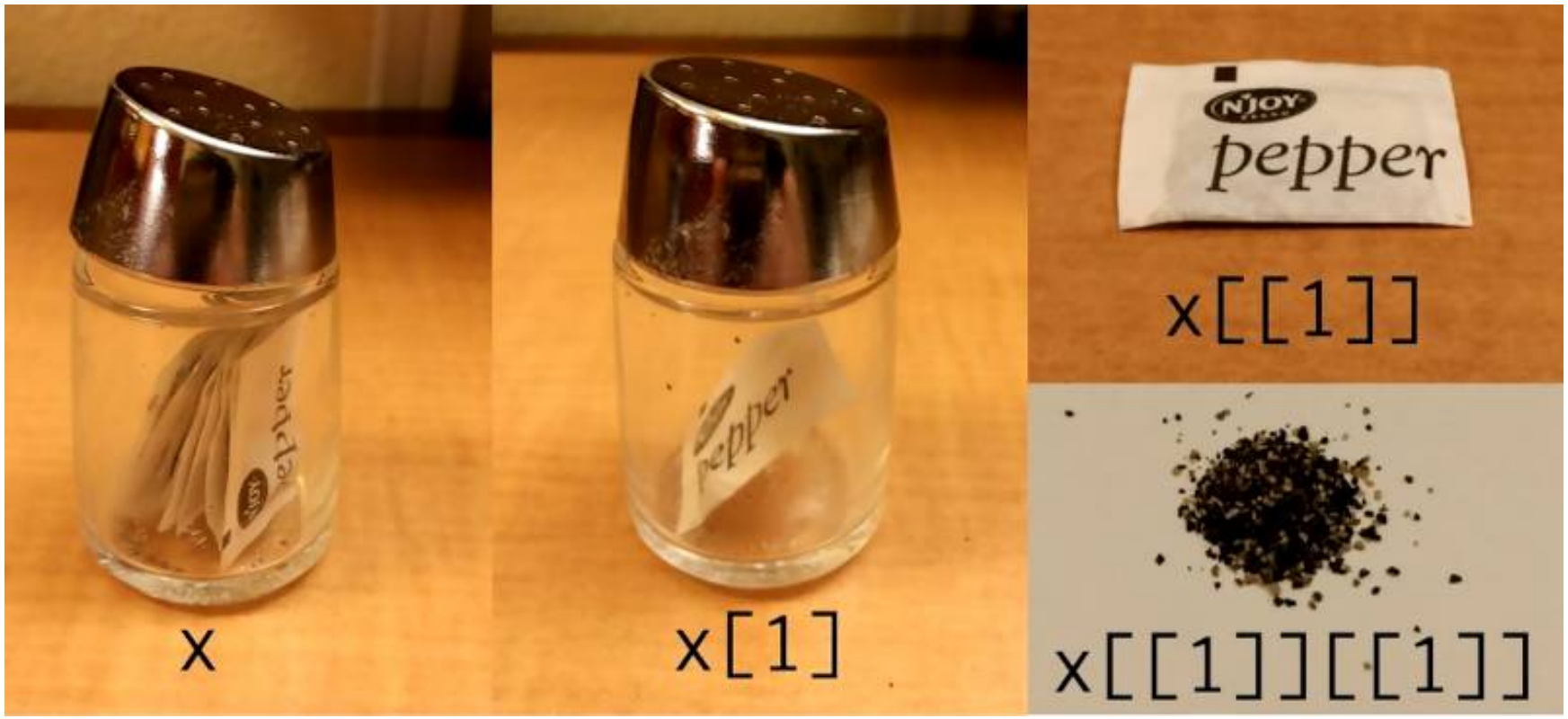
- [selects certain elements of a list; the result is a list
  - x[1] is still a list with one element; x[1:2] is a list, too
- [['unpacks' list elements; the result may have a different class

```
x <- list(1:2, letters[6:8], LETTERS[1:5])
#>  [[1]]      [[2]]                [[3]]
#>   1 2    "f" "g" "h"    "A" "B" "C" "D" "E"

x[1]   # A list with 1 element

#>   1 2
x[[1]] # A numeric vector
#> 1 2
c(length(x), length(x[1]), length(x[[1]]))
#> 3 1 2
```

Difference between [and [[, visualised



Accessing list elements via $

Use $ to get the elements of a list (no quotation marks).

```
y           <- list(1:3, c("Alex", "Bill"), mtcars)
names(y) <- c("Numbers", "Names", "My data")
```

```
y$Numbers
#> 1 2 3
```

```
y$Names
#> "Alex" "Bill"
```

```
y[["Names"]]  # Same
y[[Names]]    # Does not work; error
```

If a name contains spaces or other characters breaking the names, use backticks:

```
y$`My data`
```

Factor class

Factor: a word with multiple meanings, but in R, it represents a categorical variable (unordered or ordered), usually with not too many distinct values.

In reality, any finite set can be mapped bijectively into a set of positive integers:

$$\{🌐, \propto, ♣, \leqslant, \Theta, ₥, \propto\} \mapsto \{4, 3, 1, 2, 5, 6, 3\}$$

Internally, a factor variable is an integer with a label attached to it – nothing else. Factors are great for interactions, multi-level models, and saving memory.

Creating a factor

Call `factor()` with an existing character vector:

```
x <- c("Low", "Low", "Medium", "High", "Medium")
y <- factor(x)
#> Low    Low    Medium High   Medium
#> Levels: High Low Medium
```

By default, the labels are assigned alphabetically (H, L, M), which means that the y vector is internally confusing. Override them with the `levels` argument:

```
as.integer(y)  # 2 2 3 1 3
as.numeric(y)  # Same
levels(y)
w <- factor(x, levels = c("Low", "Medium", "High"))
as.integer(w)  # 1 1 2 3 2
```

Converting factors

Factor-to-character conversion is lossless (but the resulting variable uses more memory):

```
x <- c("Carol", "Bob", "Bob", "Carol", "Alice")
f <- factor(x)
f2 <- as.character(f)  # Back to character
f2[f2 == "Bob"] <- "Bill"
f2  # "Carol" "Bill"  "Bill"  "Carol" "Alice"
```

This is why factors may be a nuisance at the data-transformaton stage (but will be incredibly handy in analysis).

Repeating and dropping values

Multiple instances of the same value can be indexed, which will repeat it in the output:

```
x <- c("A", "B", "C")
x[c(1, 3, 1, 2)]  # A C A B
```

Negative indices can be used to drop elements:

```
m <- matrix(1:18, nrow = 3, ncol = 6)
m[-1, -1]  # Same as m[2:3, 2:6]
m[, -ncol(m)]  # Same as m[, 1:5]
m[, -c(2, 5)]  # Same as m[, c(1, 3, 4, 6)]
```

Either only negative or only negative indices should be used: something like `x[c(1, -3)]` produces an error.

Changing parts of a vector

Use subsetting with `<-` to assign values to a subset:

```
x <- c("A", "B", "C")
x[2] <- "Bull"  #  x becomes  A Bull C
m <- matrix(1:12, nrow = 3, ncol = 4)
m[1, ] <- 99  # Set the first line equal to 99
m[2:3, 3:4] <- matrix(-11:-14, ncol = 2)
#> 99  99   99   99    # Now m is this
#> 2    5  -11  -13
#> 3    6  -12  -14
```

For lists, use `mylist[[i]] <- value`.

The `[` function gets elements, the `[<-` function sets or replaces elements; they are different, but their invocation looks similar: `x[3]` vs. `x[3] <- 1`.

Adding new values to vectors / matrices

Vectors can be grown through *concatenation*. If the target index is not contiguous, NAs are added in the middle:

```
x  <- 11:13
x1 <- c(x, 99) # 11 12 13 99
x1[100] <- -3   # Same as c(x, rep(NA, 95), -3)
```

Grow matrices with `cbind()` or `rbind()`, or embed small matrices into larger ones.

```
m <- matrix(1:9, ncol = 3)
cbind(m, NA, NA, 6)
# Result: 3 full cols, 2 NA cols, 1 cols of 6s
mbig <- matrix(NA, ncol = ncol(m), nrow = 20)
mbig[1:nrow(m), ] <- m
mbig[nrow(mbig), ] <- -1
mbig  # 3 full rows, 16 NA rows, 1 row of -1s
```

How not to add new values

The ‘bad’ way – to assign non-existent indices – may work for vectors and lists; the values at the skipped indices are filled with `NA` (for lists, `NULL`):

```
x    <- 1:3
x[6] <- 99  # x becomes 1 2 3 NA NA 99
y <- list(1:3, LETTERS, mtcars)
y[[6]] <- "w-('U')-w  Kilroy was here"
y[4:5]  # NULL NULL
y[[6]]  #  w-('U')-w  Kilroy was here
```

However, it does not work for matrices:

```
m <- matrix(1:9, ncol = 3)
m[5, 5] <- 99
# Error: subscript out of bounds
```

Renaming objects

There is no dedicated ‘rename’ function in R for objects in the environment; vector elements and array dimensions can be renamed.

Renaming objects is achieved via copying with a different name and deleting the old instance:

```
new.objname <- old.and.bad.object.name
rm(old.and.bad.object.name)
```

Removing `old.and.bad.object.name` is necessary to free up memory (the only situation where memory usage does not go up is when neither object is modified).

Data frame

Data frame: a *list* of vectors of the same length assembled like a matrix (by column).

- Looks like a matrix, can be subset like a matrix
- Can be subset like a list via $
- Can have data of various types (character, logical, factor, numeric)

Some functions cannot take data frames and require matrices instead. Be careful when converting data frames to matrices.

Example: LASSO from `glmnet` requires a numeric matrix, which takes one more line to prepare via `model.matrix`.

Favourite test data frame: `mtcars`

R has many built-in data frames. This course makes many illustrations with `mtcars`.

Check the DF summary statistics and classes:

```
summary(mtcars)
class(mtcars$mpg)
```

Produce scatter plots of all variable pairs:

```
plot(mtcars)
```

Test functions with vector or matrix inputs:

```
mean(mtcars$mpg)  # The mean of miles per gallon
var(mtcars[, c("mpg", "wt")]) # Variance-covar.
cor(mtcars$mpg, mtcars$wt)    # Correlation
```

Data frame subsetting

- Column subsetting via indices, logical vector, or names
  - Use `drop = FALSE` to keep the length-1 dimension
- Nevertheless, a row of a data frame is always a data frame (list), with dimensions not dropped
  - `class(mtcars[, 1])` is `numeric`, but `class(mtcars[1, ])` is `data.frame` because the elements of a row may have different classes – simplification not posible

Naming with and w/o dimensions

Vectors and matrices have different naming methods:

```
x <- 1:4
names(x) <- c("Adam", "Borys", "Cyryl", "Dymitr")
x    # dim(x) is NULL
#>  Adam  Borys  Cyryl Dymitr
#>     1      2      3      4

m <- matrix(1:8, nrow = 2, byrow = TRUE)
rownames(m) <- c("Andor", "Botond")
colnames(m) <- c("Jan", "Apr", "Jul", "Oct")
m    # dim(m) is c(2, 4)
#>        Jan Apr Jul Oct
#> Andor    1   2   3   4
#> Botond   5   6   7   8
```

Dimension names

- If x is a vector and not an array (`dim(x)` is `NULL`), its names are stored in the `names` attribute (vector)
- If x is an array (`dim(x)` has length at least 2), its names are stored in the `dimnames` attribute (list)

From the last example:

```
dimnames(m)  # Is a list of length 2
#> [[1]]
#> "Andor"  "Botond"
#> [[2]]
#> "Jan" "Apr" "Jul" "Oct"
```

Changing only some names

Just like value vectors, name vectors can be changes only in certain positions using indexing. In this example, we rename some variables of `mtcars`.

Rename one column from `drat` to something descriptive:

```
d <- mtcars
colnames(d)[colnames(d) == "drat"] <- "axle.ratio"
```

Rename columns 3 and 4:

```
colnames(d)[3:4] <- c("displcmt", "horsepwr")
```

Array names

Arrays can be created with dimension names via `array(..., dimnames = ...)` if `dimnames` are passed as a list of character vectors of dimensions-matching lengths. The `dimnames` attributes can be assigned at any time:

```
a <- array(1:24, dim = c(4, 3, 2))
dimnames(a) <- list(c("Jan", "Apr", "Jul", "Oct"),
  c("Goog", "MS", "Meta"),  c("Train", "Test"))
a
#> , , Train
#>
#>     Goog MS Meta
#> Jan    1  5    9
#> Apr    2  6   10
#> Jul    3  7   11
#> Oct    4  8   12
#> , , Test
#>
#>     Goog MS Meta
#> Jan   13 17   21
#> Apr   14 18   22
#> Jul   15 19   23
#> Oct   16 20   24
```

Using $ or name index

In data frames, it is more efficient to extract vectors via $:

```
class(mtcars)  # "data.frame"
mtcars[, "mpg"] - 2*mtcars[, "vs"]
mtcars$mpg - 2*mtcars$vs  # Same but shorter
```

However, the $ accessor is not available for matrices – only [with column names:

```
m <- as.matrix(mtcars)
class(m)        # "matrix" "array"
m$mpg - 2*m$vs
#> Error in m$mpg :
#>   $ operator is invalid for atomic vectors
m[, "mpg"] - 2*m[, "vs"]  # Works
```

Data frames vs. matrices

- A data frame is a list of vectors of identical length; these vectors can be of various types, i. e. data frames are collections or *heterogeneous* variables
- A matrix is an atomic vector (all values have the same type) wrapped into a 2D array

In R, there are many functions that operate on lists and return lists (especially in parallelisation); therefore, data frames are better handled as lists. Despite the fact that matrices contain data of the same type, they are rarely faster than DFs.

Benchmark the speed and try both in large-scale applications!

Data frame as a list of vectors

```
x <- mtcars
class(x)  # data.frame
x$Name <- rep(c("Alice", "Bob"), 16)
str(x)
#> 'data.frame':        32 obs. of  12 variables:
#>  <...>
#>  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
#>  $ Name: chr  "Alice" "Bob" "Alice" "Bob" ...
```

Converting x into a matrix will coerce all columns into one type – *character*:

```
as.matrix(x)
#>                    mpg    cyl disp    hp    # ...
#> Mazda RX4          "21.0" "6" "160.0" "110" # ...
#> Mazda RX4 Wag      "21.0" "6" "160.0" "110" # ...
```

Data frame from a matrix

Unlike `data.frame` → `matrix`, the reverse is lossless.

```
m <- matrix(1:8, nrow = 2, ncol = 4)
colnames(m) <- c("USA", "Japan", "EU", "Rest")
rownames(m) <- c("Train", "Test")
m$Japan  # Does not work
d <- as.data.frame(m)
d$Japan  # Works
c(is.list(m), is.list(d))  # FALSE, TRUE
```

Since a data frame is a list, its column names can be set with `colnames()` or `names()`:

```
names(d)[1] <- "U.S."
names(d)
#> "U.S."  "Japan"  "EU"  "Rest"
```

Data frame vs matrix speed

Q: Should I use matrices or data frames?

A: It really depends on the application.

Myth: matrices are faster than data frames because the data are of the same type. *We test this statement in a later section (if we have time)!*

Test results: data frames are approx. 2× faster than matrices for row subsetting, and hundreds (!) of times faster for column subsetting ⇒ **use data frames** for data manipulation!

NB: 'toy' data sets do not necessarily reflect the reality – test on real data sets if possible.

Use dedicated functions

If there is a specialised built-in function, use it instead of a hand-made one.

```
m <- matrix(runif(1e6), nrow = 1e3)
microbenchmark::microbenchmark(
  colSums(m), apply(m, 2, sum))
#> colSums: 1.1--1.4 ms, apply: 8.6--16 ms
```

The `Rfast` package has highly optimised fast functions that help process large data sets.

```
microbenchmark::microbenchmark(
  Rfast::colMedians(m), apply(m, 2, median))
#> Rfast: 10--11 ms, apply: 49--59 ms
```

Do not re-compute large logical vectors

Do not subset big DFs many times with the same condition:

```
for (i in 1:100) {  # This is slow
  di <- d[d$id == i & d$income > 100, ]
} # Operations with di inside
```

If an operation is to be done **by** a certain group:

- **Best:** split a data set into a list using `split()`, process each element of the list, and `unsplit()`

```
ds <- split(d, f = d$id)   # Work with ds
d <- unsplit(ds, f = d$id)
```

- **Next best:** extract a subset, process it, insert it back

```
dsmall <- d[d$id == 1, ]   # Work with dd
d[d$id == 1, ] <- dsmall
```

Do not subset unnecessary matrices

Task: extract the first 100 elements of the variable x from the data frame d.

Common mistake: `d[1:100, ]$x`. This is bad because it subsets a huge sub-DF (first 100 rows of d) and then, extracts a vector.

Best solution: `d$x[1:100]` because column extraction from DFs is blazingly fast, and subsetting vectors is faster than subsetting DF rows.

If some operation is done for multiple variables in a data frame, select columns first, observations last.

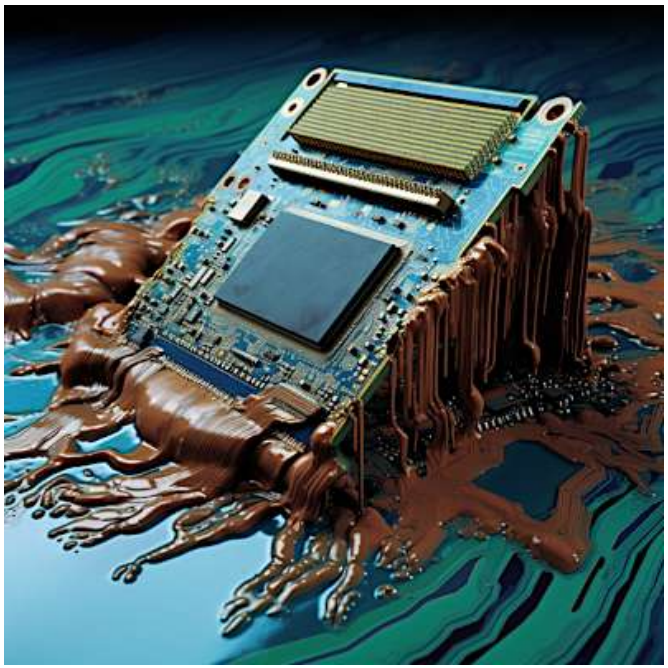
Good practice: use memory sparingly

Do not create many copies of large data frames with minuscule changes: chains like

```
d2 <- d1[d1$age > 17, ]
d3 <- d2[d2$age < 66, ]
```

devour RAM.

Hard to keep track, easy to forget which copy is the relevant one.



Make sure that overwriting does not corrupt the data upon rerun: `d <- d[d$age > 17, ]` is safe (idempotent), `d <- d[1:(nrow(d)-5), ]` is not (always drops 5 rows).

Automatic type conversion

Recall Session 1: R is a dynamically typed language.

Storage capacity: character > numeric > integer > logical.

- Characters can store any symbols, numbers with arbitrary precision etc. – literally anything
- Numerics are limited to 64 bits for real numbers
- Integers have no decimal part
- Logicals are the least informative

Generally, R does not like losing data, which is why it 'bumps up' mixed data types.

Type casting example

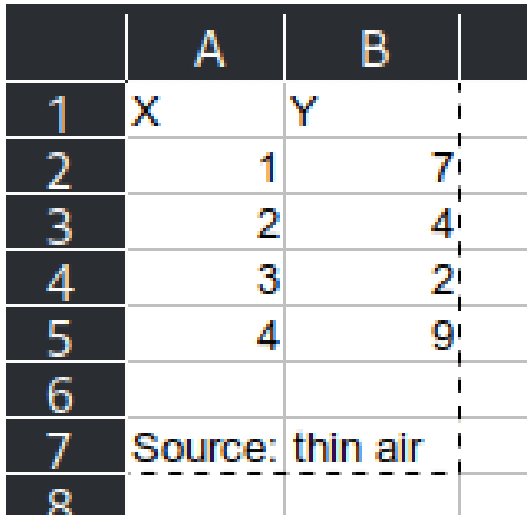
```
class(c(1, 10, 3))          # numeric
class(c("a", "b", "y"))     # character
c(10, "b", "y")
#> "10" "b"  "y"
class(c(10, "b", "y"))      # character
```

Numbers saved as character require conversion to numeric to enable arithmetic operations:

```
c("10", "4") + 2
#> Error: non-numeric argument to binary operator
as.numeric(c("10", "4")) + 2
#> 12  6
as.numeric(c("10", "a")) + 2
#> 12 NA  # Because as.numeric("a") is NA
```

Recasting char → num in practice

Spreadsheets may contain footnotes or remarks. If a dirty source contains mixed data, the values are read as **character**. It requires explicit conversion into numeric. Check the variable classes after loading – recast if needed!



	A	B
1	X	Y
2	1	7
3	2	4
4	3	2
5	4	9
6		
7	Source: thin air	
8		

```
d <- openxlsx::read.xlsx("mixed.xlsx")
d$X # "1" "2" "3" "4" "Source: thin air"
d <- d[1:4, ]
mean(d$X) # NA
sapply(d, class) # X is still char; Y is num
d$X <- as.numeric(d$X)
mean(d$X) # 2.5
```

Factors from numeric

Since a factor is an integer variable with character labels and alphabetically sorted levels, straightforward conversion from numeric to factor and back is lossy because only the order is saved:

```
x <- c(1, 1, 4, 10, 4, 7, 7, 10, 1)
f <- factor(x)
as.numeric(f)
#> 1 1 2 4 2 3 3 4 1
```

To recover the original numeric values, read as character first (to use the label information):

```
as.numeric(as.character(f))
#> 1  1  4 10  4  7  7 10  1
```

Converting classes

If a codebook has '1' = 'Male', '2' = 'Female', '9' = 'Other', '999' = non-response, assign labels to increasing values (internally, the input is mapped to positive integers: $\{1, 2, 9, 999\} \mapsto \{1, 2, 3, 4\}$).

```
g <- c(1, 2, 2, 999, 9, 1)
f <- factor(g, labels = c("M", "F", "O", NA))
f  # M    F    F    <NA> O    M
#>   Levels: M F O <NA>
as.integer(f)  # Recoded to 1 2 2 4 3 1
```

Character-to-numeric is lossless for valid numbers:

```
as.numeric(c("1", "-2", "3.5", "1e1", "A", "R2D2"))
#> 1.0 -2.0  3.5 10.0   NA   NA
#> Warning message: NAs introduced by coercion
```

Sorting arrays

Sorting: sub-setting a vector with a permutation that rearranges it into (a/de)scending order. `order()` yields these indices, `rank()` yields the ranks, and `sort(x)` does the same as `x[order(x)]`.

```
set.seed(1); x <- runif(10)
rbind(x, rank(x), order(x), sort(x))
```

```
#> 0.27 0.37 0.57 0.91 0.20 0.90 0.94 0.66 0.63 0.06 -- x
#>    3    4    5    9    2    8   10    7    6    1 -- rank
#>   10    5    1    2    3    9    8    6    4    7 -- order
#> 0.06 0.20 0.27 0.37 0.57 0.63 0.66 0.90 0.91 0.94 -- sorted
```

Sorting a data frame `d` by the variable `x` is as simple as `d[order(d$x), ]`. To sort by multiple keys (e.g. panel data with ID and time), use `d[order(d$id, d$time), ]`.

Any questions on the variable classes?

Special values, dates, structures

Special values: NULL, NA, NaN, Inf

There are special values to represent real phenomena:

- `class(NULL) = "NULL"` (for length-0 objects)
- `class(NA) = "logical"` (to represent the absence of knowledge of a particular value)
- `class(NaN) = class(Inf) = "numeric"`
  - not a (real) number – could be a complex number but the user did not use complex values
  - infinity represents anything larger than the largest number representable with 64 bits

Empty vector: NULL

`NULL` is an object of zero length representing the absence of an element.

Changing some elements of a vector to `NULL` effectively removes those elements.

Useful in debugging: a function can return meaningful output of length ≥ 1 in case of success and `NULL` in case of failure.

Creating variable-length vectors with NULL

When **NULL** is put into a vector, it is ignored:

```
c(1, 2, NULL, 4)  # 1 2 4
```

Depending on a condition, an element may exist or be skipped:

```
act <- 4  # Try changing to 5
c("Tybalt", "Romeo",
  if (act < 5) "Juliet" else NULL, "Mercutio")
```

If `act == 5`, there will be no `"Juliet"` element.

Not-a-number and infinity

Some mathematical operations are undefined, and some expressions are indeterminate. Recall limits in undergraduate maths classes: $0/0$, 1^∞, $0 \cdot \infty$...

```
sqrt(-1)    # NaN + warning   1 / 0       # Inf
log(-4)     # NaN + warning   -1 / 0      # -Inf
sin(Inf)    # NaN + warning   Inf + 5     # Inf
Inf / Inf   # Silent NaN      Inf + Inf   # Inf
```

`Inf` are useful to define large penalties in numerical optimisation (some solvers require numeric values only).

- `-Inf` exists, too, and represents certain limits (e.g. `log(0)`).

`NaN` is usually a *consequence* of 'forbidden' maths (maybe input checking / domain restrictions are needed)

Undefined / missing value: NA

To check if some elements of x are `NA`, use `is.na()` (it also react positively to `NaN`s).

NB: `x == NA` does not work!

To drop `NA`'s from a vector, use logical conditions or `na.omit()`.

```
a <- c(1, log(-1), NA, 1/0, 9)  # 1 NaN NA Inf 9
a[!is.na(a)]  # 1 Inf 9
na.omit(a)
#> 1 Inf 9
#> attr(,"na.action")
#>  2 3
```

Uncertainty propagation

`NA` has type `"logical"` and interacts with logical values:

- `TRUE | NA` is `TRUE`, but `TRUE & NA` is `NA`
- `FALSE & NA` is `FALSE`, but `FALSE | NA` is `NA`

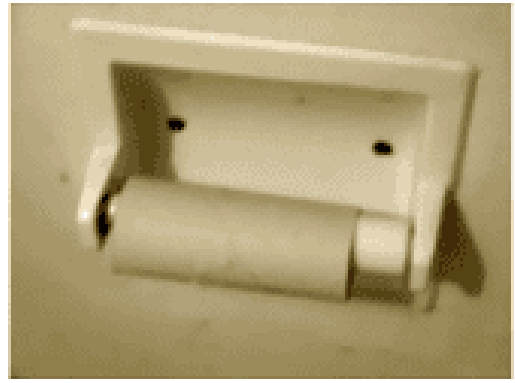
`NA` addles computations: when one of the inputs is undefined, the output is by default undefined.

- Some statistical functions (`mean()`, `sd()`, `quantile()`, ...) accept the `na.rm = TRUE` argument to ignore them: if `x <- c(1, 2, NA, 6)`, then, `mean(x)` is `NA`, but `mean(x, na.rm = TRUE)` is 3
- Some functions (e.g. `lm()`) drop rows with `NA`
- Some functions return `NA`, and some throw an error (prompting the user to get rid of `NA`s somehow)

Non-standard values, visualised

Non-zero value

0



NULL

NA or error

Credit: David Armendáriz.

Special values are not ‘bad’

- Standards (like IEEE 754) exist for a reason, and these ‘not numbers’ help in error handling
- Special values prevents such absurd errors as Stata’s ‘`(x > 100) = TRUE`’ if `x == .` (is missing)
  - In Stata, ‘`.`’ is represented internally by a large number
- Safety regulations are written in blood
  - Failing is necessary in programming to let the user know that something went wrong
  - Failure to fail can be fatal (as in the case with Therac-25: 6 people died)
- Do not ignore `NA`s, or replace them with zeros, or impute!

Checking if a value is numeric

```
a <- c(1, NA, NaN, Inf)
class(a)      # "numeric"
is.finite(a)
#> TRUE FALSE FALSE FALSE

if (any(!is.finite(a)))
  stop("Dark times, non-finite values.")
```

Note that this method filters out 'good' numbers, whereas `na.omit()` leaves infinite values:

```
na.omit(a)  # 1 Inf
```

NB: `is.numeric(a)` will not work because it checks the **class** of the entire object, not elements. Numeric vectors can have non-finite values.

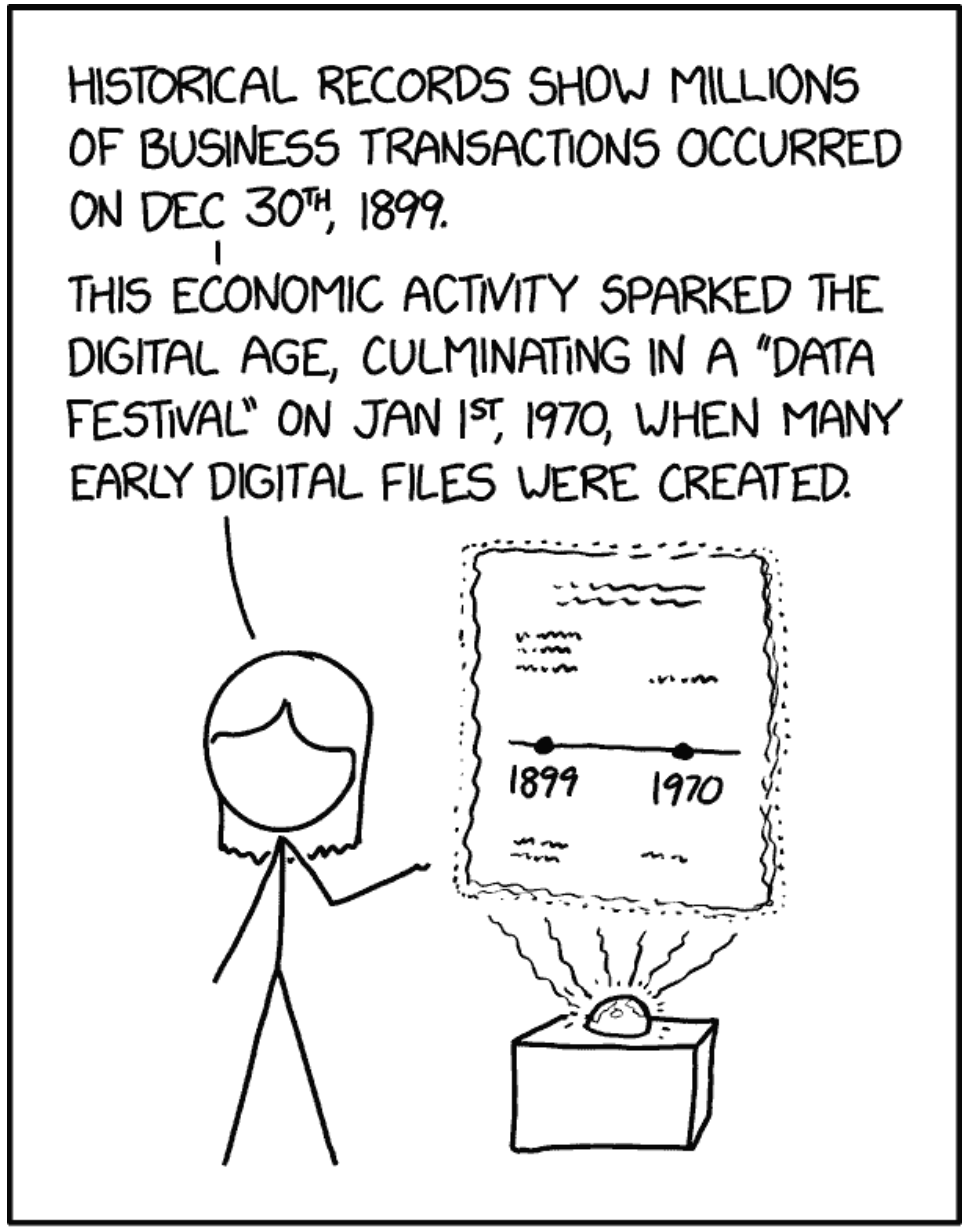
Date

Date: a time stamp measured in the number of days elapsed since the 1st of January, 1970 (UNIX epoch). (MS Excel uses the 30th of December, 1899, as the origin.)

Special operations apply to dates: extracting weekday or part of date (month or year), adding days, re-formatting etc.

ISO 8601 date format: `YYYY-MM-DD` (or '/'). No `MM-DD-YY` (USA style)! Bonus: sorting by name that starts with `YYYY-MM-DD` = sorting by date!

```
as.Date(c("2023-12-31", "1980-05-04"))
as.integer(as.Date("2023-10-01"))      # 19631
as.Date(19631, origin = "1970-01-01") # 2023-10-01
```



Credit: xkcd. Technical explanation.

Reading dates from files

If a text/CSV file contains ISO dates, the column with hyphens is read as character – use `as.Date()`:

```
d <- read.csv("ibm-iso.csv") # "2000-01-04" ...
d$date <- as.Date(d$date)
```

If an XLSX file has broken dates (typical Excel), convert from integer starting on 1899-12-30:

```
d <- openxlsx::read.xlsx("IBM.xlsx")
# d$date is 36528, 36929, ... -- WHAT
d$date <- as.Date(d$date, origin = "1899-12-30")
# Or use readxl -- but convert from tibble
d <- as.data.frame(readxl::read_excel("IBM.xlsx"))
```

Use the `format` argument to read dates in any notation (see `?as.Date`): `as.Date("12/31/99", "%m/%d/%y")`.

Date sequences

The familiar `seq()` function changes its behaviour if the class of its `from` and/or `to` argument is `Date`.

The `by` argument can be `"day"`, `"week"`, `"month"`, `"quarter"`, or `"year"`. It can be preceded by a positive or negative integer + space, or followed by `"s"`.

```
seq(as.Date("1991-01-01"), by = "month",
    length.out = 12*30) # Jan 1991 -- Dec 2020
seq(as.Date("1910/1/1"), as.Date("1999/1/1"),
    "year") # Jan 1910 to Jan 1999 by year
seq(as.Date("1910/1/1"), as.Date("1999/1/1"),
    "3 years") # 01-01-10, 01-01-13, ...
seq(as.Date("2000/1/1"), as.Date("2003/12/1"),
    by = "quarter") # 16 quarters
```

Advancing time

Since dates are measured in days, arithmetic operations on them are well-defined:

```
x <- as.Date(c("2019/1/26", "2019/2/26"))
x + 1    #  2019-01-27  2019-02-27
x - 100  #  2018-10-18  2018-11-18
```

However, months have a different number of days:

```
x + 3  # 2019-01-29 2019-03-01
```

The `lubridate` package has great functionality:

```
library(lubridate)
x %m+% months(1)
#> 2019-02-26  2019-03-26 -- cf. x + 31
```

Manipulating dates

```
x <- seq(as.Date("2019-01-01"),
         by = 5, length.out = 100)
```

- Get the week days: `weekdays(x)` (Tue, Sun, Fri...)
- Extract the month only: `format(x, "%m")`
- Get the day as a number (1–366): `format(x, "%j")`
- Get the week number (1–53): `format(x, "%V")`

See `?strptime` for all formats.

Other units: time, month, quarter

R supports POSIX time (calendar or local). The time is defined as the (real) number of seconds since 1970-01-01.

```
as.POSIXct("2023-09-12 02:32:03")
```

Time stamps are useful for benchmarking.

```
tic0 <- Sys.time()
Sys.sleep(1)
difftime(Sys.time(), tic0, units = "mins")
# Time difference of 0.01668883 mins
```

The `zoo` package has custom classes for months and quarters: `zoo::as.yearqtr()` and `zoo::as.yearmon()` (internally, it converts dates to the first date of a month/quarter). It can be useful in some applications.

Examining object structure

A quick object overview is often necessary to understand what it represents. Structure display is especially useful when one is connected to a remote server with CLI only.

```
x <- 1:4
m <- matrix(1:8 + 0.1, nrow = 2,
  dimnames = list(c("A", "B"), month.abb[1:4]))

str(x)
#> int [1:4] 1 2 3 4

str(m)
#> num [1:2, 1:4] 1.1 2.1 3.1 4.1 5.1 6.1 7.1 8.1
#> - attr(*, "dimnames")=List of 2
#>  ..$ : chr [1:2] "A" "B"
#>  ..$ : chr [1:4] "Jan" "Feb" "Mar" "Apr"
```

Structure of lists

Since R functions can return only one value, many methods return a lists of outputs / diagnostics.

Example: `lm()` (linear model) returns a list with OLS estimates, residuals, predicted values etc.

```
l <- lm(mpg ~ hp + wt, data = mtcars)
str(l)
```

```
List of 12
 $ coefficients : Named num [1:3] 37.2273 -0.0318 -3.8778
  ..- attr(*, "names")= chr [1:3] "(Intercept)" "hp" "wt"
 $ residuals    : Named num [1:32] -2.572 -1.583 -2.476 0.135 0.373 ...
  ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" ...
 $ rank         : int 3
 $ fitted.values: Named num [1:32] 23.6 22.6 25.3 21.3 18.3 ...
  ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" ...
 $ call         : language lm(formula = mpg ~ hp + wt, data = mtcars)
 $ model        :'data.frame':        32 obs. of  3 variables:
  ..$ mpg: num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
  .. <...>
 - attr(*, "class")= chr "lm"
```

Quick analysis of lm structure

```
str(l <- lm(mpg ~ hp + wt, data = mtcars))
```

- OLS residuals and fitted values preserve observation names (Mazda, Datsun etc.)
- The coefficient vector has variable names
- There is a fully preserved copy of the data set used in estimation under `l$model`
  - Other methods and packages use it; `plot(l)` (more precisely, `stats:::plot.lm(l)` extracts data for diagnostic plots), `sandwich::vcovHC(l)` uses the model matrix for robust variance estimation
  - This data set copy may consume a lot of memory; consider saving only `l$coefficients` and minimum necessary objects in large-scale simulations

Special data types

In empirical analysis, time-series data and panels are quite common.

Time series: basic regular time series with `ts()`, extending the basic capabilities with packages: `zoo`, `xts`, `lubridate` etc. Any aggregation command can be applied (disaggregation: the `td` package).

Panels: the `plm` package (excellent vignette) and some others; panel data can be read as ordinary rec

Time series

- Many ways to represent time series as objects; different types = different methods
- Can be index- or time-based (the user decides how to treat them)
  - Multi-firm stock return data with gaps = which type?
  - Weeks of the year = which type? How many?
- Default: `ts` (index-based with frequency), extensions: `zoo`, `xts` (allows irregular series)
- Some analyses can be carried out without any TS attributes (e.g. `sandwich::vcovHAC` assumes ordered residuals and returns a consistent variance estimator for stationary data)

Panel data

The panel functionality is provided by custom packages.

- `plm`, `panelr`, `pglm`, `gnm`, `mclogit`, `FENmlm`, `bife`, `PanelCount` packages
  - Fixed effects, first differences, dynamic panels (GMM) with valid inference, conditional logit, non-linear and multi-level panels
- Separate GMM packages for general estimation (`gmm` or the newer `momentfit`)
- Panel transformations and aggregation by groups are supported by the data transformation packages
- Some may work quicker, some may be slower (especially in the computation of marginal effects)

Model object types

R has an agreed-upon model format. It is usually a **list** of

- Model matrix
- Estimated coefficients
- Fitted values + residuals
- Other model-specific information

Depending on the model type, extra methods are applicable to models (covariance matrix, plotting, classification, forecasting etc.).

However, it all depends on the exact implementation in custom packages.

Clearing environment

Use `rm(list = ls())` to remove all user objects.

This will **not** clear all loaded packages. To unload one package (e.g. `dplyr`), use

```
detach("package:dplyr", unload = TRUE)
```

This is rarely used, but saves time if it will take too long to restart the R session. However, the cleanest way is to start a new session.

Assignment submission: start RStudio from scratch – **your script must run without an error**. Same goes for code sharing.

Text manipulation

Text manipulation

- R is fully Unicode-compatible on Linux and Mac, on Windows starting from 4.2.0
- Create and modify text according to arbitrary rules
- Full Perl-compatible regex support
  - **Reg**ular **exp**ressions: special syntax supported by many programming languages that enables text pattern matching and replacement
- Enables en-masse file manipulation because file names are strings that can be obtained via `file.exists()` and modified

Pasting

Use `paste0()` to merge scalars or vectors no separator or `paste(..., sep = "whatever")` (default: space).

```
x <- c("A", "B", "C")
paste0(x, ".", 1:3)  # A.1 B.2 C.3
paste(x, 1:2)  # "A 1" "B 2" "C 1"
paste(x, 1:2, sep = "!")  # "A!1" "B!2" "C!1"
```

Use the `collapse` argument to merge a character vector with the chosen delimiter (when there is more than one input, `sep` is still added, unless `paste0()` is called):

```
paste(x, collapse = "!")       # "A!B!C"
paste(x, 1:3, collapse = "!") # "A 1!B 2!C 3"
paste(x, 1:3, sep = "", collapse="!") # "A1!B2!C3"
paste0(x, 1:3, collapse="!")          # "A1!B2!C3"
```

grepl: find logical matches

`grepl(what, x)` returns a logical vector (same length as x): `TRUE` if the `what` string was found in the element of x.

The matching is case-sensitive. Use `ignore.case = FALSE` or search for character ranges with square brackets.

Find the countries containing the letter **b**:

```
cn <- c("Belgium", "France", "Germany", "Italy",
        "Greece", "Ukraine", "Kosovo", "Sweden",
        "Norway", "Albania", "EU27", "EU14")
grepl("b", cn)  # F ... F T F F
cn[grepl("b", cn)]  # Only alBania
cn[grepl("b", cn, ignore.case = TRUE)]
cn[grepl("[bB]", cn)]  # Belgium Albania
```

Operations with grepl output

`grepl()` is useful where logical variables make sense: `mean(grepl("[bB]", cn))` counts the share of elements of `cn` containing the letter **b** or **B** (1/6 in this example).

The output of `grepl()` can be negated with `!` to invert a logical vector (i. e. `TRUE` where the match was not found)

Example: suppress part of the output. Show only the coefficients that do not have 'factor' in their names:

```
l <- lm(mpg ~ factor(cyl) + wt + qsec, data = mtcars)
b <- l$coefficients
names(b)  # (Intercept)  factor(cyl)6  ...
b[!grepl("factor", names(b))]
#>  (Intercept)          wt        qsec
#>   25.9180659  -3.8887560   0.5034263
```

grep: find indices of matches

`grep(...)` returns the indices of pattern matches, i. e. acts like `which(grepl(...))`.

```
txt <- c("arm", "foot", "lefroo", "bafoobar")
grep("foo", txt)  # 2 4
```

If there are no matches, `grepl` returns a vector of all FALSE, and `grep`, a vector of length 0 (`NULL`).

Depending on the application, `grep` or `grep` may be more convenient.

Special search patterns

Inside a search pattern for `grep(l)`, the `^` symbol denotes the beginning of a string, and `$`, the end of string.

The square brackets + hyphen denote a character range:
`[A-Z]` = any symbol between **A** and **Z** in the Unicode table,
`[^A-Z]` = any symbol except for the range.

```
cn <- c("Belgium", "France", "Germany", "Italy",
        "Greece", "Ukraine", "Kosovo", "Sweden",
        "Norway", "Albania", "EU27", "AG")
cn[grep("^G", cn)]  # Germany, Greece but not AG
cn[grep("e$", cn)]
# France, Greece, Ukraine but not Sweden
grepl("^EU", cn)   # What begins with EU
grepl("[0-9]", cn) # Any digit between 0 and 9
grepl("[3-5]", cn) # Any digit between 3 and 5
```

gsub: substitution in strings

`gsub()` performs **g**lobal **sub**stitution in strings using patterns. It takes 3 arguments: what to search for, with what to replace, and where.

```
# Remove all non-letter characters:
x <- "3.7 k, 3.6 k, 3.7 k, 15 m, 12/h, Encyclong"
gsub("[^A-Za-z]", "", x)  # "kkkmhEncyclong"
gsub(",", ".", "3,9")  # Decimal comma -> dot
gsub("[№*='’%éè/&]+", "", x) # Delete characters

f <- "input.xlsx"
d <- openxlsx::read.xlsx(f)
write.csv(d, gsub("\\.xlsx", "\\.csv", f))
```

NB. The symbols `.()[]*?+` must be escaped with `\\` for a literal match.

Creating formulæ by pasting

Very useful for big data! Imagine that the data set `d` has the following column names:

```
colnames(d) <- c("id", "return5", "volX", "dvol2",
                 "vol5", "vol10", "vol15", "spread")
```

A formula for regressing `return5` on all regressors *starting* with `vol` and followed by a digit (but not a letter) can be created automatically.

```
regs <- x[grep("^vol[0-9]", x)]
rhs  <- paste0(regs, collapse = " + ")
f    <- formula(paste0("return5 ~ ", hrs))
print(f)  #  return5 ~ vol5 + vol10 + vol15
lm(f, data = d)
```

Subsetting DFs by column name match

Data frame columns can be selected based on patterns in column or row names. Same for vectors and their names.

```
dlag <- d[, grep("^lag_", colnames(d))]
```

NB. Consider adding `, drop = FALSE` in case the match has length one but the matrix structure must be preserved.

Select the columns from 1 till the one that goes right before the first column whose name starts with **W**:

```
d[, 1:(grep("^W", colnames(d))[1] - 1)])
```

NB. Consider writing an exception if there might be no match and `grep()` returns `NULL` (to avoid the error).

Substitution in column names

Replace `"V"` with `"Quarter"`:

```
a <- b <- as.data.frame(matrix(1:12, nrow = 3))
colnames(a)  # V1 V2 V3 V4
colnames(a) <- gsub("V", "Quarter", colnames(a))
```

Convert quarters to months by elminating characters, performing arithmetic operations on the numbers, and converting back to character:

```
quarter <- as.numeric(gsub("V", "", colnames(b)))
month       <- (quarter-1)*3 + 1
colnames(b) <- paste0("M", month)
#> M1 M4 M7 M10
```

Text substrings

Extract sequences of characters in fixed positions with `substr()`:

```
cities <- c("Paris", "London", "Kyiv",
            "Athens", "Stratford-upon-Avon")
substr(cities, 1, 3) # Extract chars 1--3
#> "Par" "Lon" "Kyi" "Ath" "Str"
substr(cities, 6, 8) # Produces empty strings
#> ""    "n"   ""    "s"   "for"
```

Printing to console

R has two useful functions to output material to the screen.

- `cat()` concatenates the supplied arguments into one character vector
- `print()` is a generic function that prints objects depending on their class (i. e. is a method)

Printing with cat()

`cat()` offers more granular control over printing.

- Accepts an arbitrary number of input arguments, coerces them to a *character vector*
- Requires a newline escape sequence '\n' at the end of the line; without it, prints to the same line.

Useful for printing custom single lines:

```
set.seed(1)
for (i in 1:10) cat("Iteration", i,
  "-- have a random number:", runif(1), "\n")
```

The default separator is a space; use `sep` to change it:

```
for (i in 1:10) cat("Iter ", i,
  ", proper comma spacing ensured.\n", sep = "")
```

Printing with print()

`print` is a generic method that will display a meaningful representation on an object depending on its type.

- Vectors: wraps them in the window, adds line numbers
  - Unlike `cat()`, shows only 1000 first elements by default
- Matrices / data frames: shows row and column names
- Model objects: shows the coefficients and/or other relevant diagnostic messages

Whenever one types an object name in an interactive console, R calls `print()` automatically:

```
m <- lm(mpg ~ hp, data = mtcars)
m  # Same as print(m)
```

print() vs. cat()

`print()` respects the object structure: `cat()` will print an array (e. g. matrix) as an unwapped vector – print will keep the array intact.

```
m <- matrix(1:12, nrow = 3)
cat(m)   # 1 2 3 ... 11 12
print(m) # Rectangular
for (i in 1:nrow(m)) cat(m[i, ], "\n")  # Same
```

Unlike `cat()`, `print()` adds embellishments. This is where '`[1]`' comes from: position index. Try in a narrow console:

```
|print(10000:10005)                |
|                                  |
|[1] 10000 10001 10002 10003       |
|[5] 10004 10005                   |
```

cat() for basic TEX tables

In research, many tables are highly customised. More often than not, there is no package or function to produce output ready for copy-pasting into a TEX document.

Use `cat()` with a custom column separator. To produce \\ at the end, escape both backslashes:

```
set.seed(1)
a <- matrix(runif(12), ncol = 3)
for (i in 1:nrow(a))
  cat(round(a[i, ], 2), " \\\\\n", sep = " & ")
# The " \\\\\n" is really " \\ \\ \n"
```

80% there (except for the extra & and inconsistent digits) – we can improve this example later.

Rounding numbers

`round()` applies mathematical rounding rules and returns a numeric.

```
round(0.453, digits = 2)  # 0.45
round(0.453, 4)           # Still 0.45
```

Negative `digits` argument to round to powers or 10:

```
round(453, -2)            # 500
```

NB: unlike mathematical rounding, `round()` rounds to the nearest **even** digit (IEEE 754 standard)!

```
round(c(0.45, 0.55), 1)  # 0.4 0.6, not 0.5 0.6!
x <- seq(-1.5, 4.5, 0.5) # -1.5 -1.0 ...
round(x)  # -2 0 0 2 2 4 4
```

Formatting numbers

In many programming languages, `sprintf()` is the ultimate formatter: add leading or trailing zeros, convert to scientific notation, round etc.

- `f` = decimal (**f**loat), up to *n* places

```
sprintf("%.4f", c(pi, 0.45))}  # 3.1416 0.4500
sprintf("%1.0f", pi)  # Rounding to integer: 3
```

- `e` = exponential

```
sprintf("%.2e", -0.00001234)   # -1.23e-05
```

- `d` or `i` = integer, `0` = add leading zeros

```
sprintf("%06d", 3)      # 000003
sprintf("%06.2f", pi)   # 003.14
```

  - Last line: 'return 6 positions in total, starting with zeros, use 2 decimal places after the dot'

Formatting to significant digits

Significant digits (sigdigs): reliable digits starting with the 1st leftmost non-zero digit up to the accuracy limit.

Sometimes, relative accuracy > equal length (=absolute accuracy).

Use `formatC()` to print decimals up to *n* sigdigs.

```
x <- pi * 10^(-5:4)
formatC(x, digits = 4)
#> 3.142e-05 0.0003142 0.003142 0.03142 0.3142
#> 3.142     31.42     314.2    3142    3.142e+04
```

In scientific notation, they would be `3.142e-5 ... 3.142e+4`.

Avoiding accuracy loss with sigdigs

- Do not report too few digits for the sake of alignment
  - Relative accuracy is usually more important than equal number of digits; mind the order of magnitude
  - Exponential notation guarantees identical accuracy: `1.23e-3` vs. `1.23e+3` (but slightly harder to read)
- If the numbers enter a ratio, add an extra digit because of precision loss in division
  - A table with 'coef (SE) = 0.02 (0.01)' could imply any t-statistic from $\frac{0.02499}{0.00501} = 4.99$ to $\frac{0.01501}{0.01499} = 1.00$

Approximation: in most cases, with a sample size of n the change of results due to an extra observation is of the order $1/n$. Sample size 100 = expect a change in the 2^{nd} sigdig.

Significant-digit reporting rules

- In text and tables, provide 2–3 sigdigs, e. g. ‘a 53% increase’ or ‘p = 0.0123’, not ‘a 53.1% increase’
- In appendices or simulation results, use 4 sigdigs: ‘in 10 000 Monte-Carlo experiments, $\bar{X}$ = 1.234, SD = 1.432’
- Do not write meaningless sigdigs **(false precision)**
  - $\hat{\beta}$ = 3.6870228, SE = 0.4976992 is nonsense

Accuracy (how stable the results are) ≠ precision (number of digits written to represent a quantity).

Rule of thumb: drop the last observation, re-compute the results. In which digit the numbers changed = accuracy limit ⇒ precision should not exceed that accuracy.

Date manipulation as text

Certain date changes can be done if the date is represented as numeric or text.

Problem: convert the end-of-month dates `2023-01-31`, `2023-02-28`, `2023-03-31`... to the beginning of month.

Four solutions:

- **Hard:** calculate the number of days in a month `k <- c(31, 28, 31, ...)`, subtract $k - 1$ from dates
- **Obfuscated:** add 1 day and subtract 1 month
  - Prone to errors: what if the last date is the last *trading* date, e. g. `2023-09-29`?
- **Unattractive:** convert to `zoo::as.yearmon()` and back
- **Easy:** replace the last 2 digits of `YYYY-MM-DD` with `01`

Converting end of month to beginning

Convert date to character, take a substring from the 1st till the 8th character, postpend "01":

```
x  <- as.Date(c("2023-01-31", "2023-02-28",
                "2023-03-31", "2023-04-30"))
ym <- substr(as.character(x), 1, 8)
paste0(ym, "01")

#> "2023-01-01" "2023-02-01"
#> "2023-03-01" "2023-04-01"
```

Regular expessions

Regex: special syntax for text pattern matching and replacement, i. e. search by symbolic mask.

Regexes are supported by many programming languages. Learning them has positive externalities. *(Huge topic, deserves its own course!)*

Credit: xkcd. Technical explanation.

Examples of regular expressions

- Find a lowercase Latin letter followed by 1–4 digits (the first digit must be 1 or 2)
  `[a-z][12]\d{0,3}`
- Check for duplicated words (2 in a row)
  `(\b\w+\b)(?=.*\b\1\b)`
- Replace the USA MM/DD/YYYY with internatonal (ISO) YYYY-MM-DD date format, where MM or DD can be either with (/01) or without (/1) leading zeros
  `/([01]?\d)\/([0-3]?\d)\/([12]\d{3})/$3-$1-$2/`
- Is it a valid Visa credit card number
  `^4[0-9]{12}(?:[0-9]{3})?$`

Shall we learn regular expressions?

According to the survey **so far**, this topic ranks 8/20, which is why we *may* touch upon it.

Even if the voting results change...The literature on regular expressions is vast, and there are many useful resources.

- Read tutorials with examples online: regular-expressions, RegexLearn, CodeCademy, RegexOne, RexEgg
- Validate regexes and see what they are doing, step by step: regex101.com, regexr.com, regextester.com

Regex in IDEs / editors

How to choose a good text / code editor / IDE: do not use an editor / IDE if it does not support regex. *(Same goes for programming languages!)*

RStudio: yes, it supports them.

Suppose that you want to find parts of code where the `ds1`, `ds2`, ... variables are located, but you want to avoid matches such as perio**ds**, in**ds**, fol**ds** (i. e. not at the beginning).

Solution: ctrl + F, enable 'Regex', search for `\bds` (here, `\b` stands for the word **b**oundary).

Any questions on the special values and text operations?

Devilish details and pitfalls

Is rm(list = ls()) unnecessary?

Some users hate `setwd()` and `rm(list = ls())`.
Those users usually write packages for other package devs. Their views may be too radical for Eco / Fin / Mgmt.
Do as you feel comfortable, but:

- Do not write paths more than once
- Do not restore `.RData` images
- Make sure your scripts works without extra clicking
  - Did you manually load extra packages or data files?

Setting working directories for many users

If an R script is used by multiple people, it is possible to semi-hard-code a flexible solution *(sic!)* for the group:

- If there are 2 users and *A* uses Windows, *B* uses Mac, check `.Platform$OS.type`
- Use the HOME environment variable or `Sys.info()` to distinguish between multiple users / machines

The next example shows how to auto-detect paths for 2 users with 3 machines.

Directory auto-detection

John uses Linux, Matilde has 2 machines (Mac and Windows) with identical user names.

```
user <- Sys.info()["user"]
os <- Sys.info()["sysname"]
if (user == "John") setwd("~/Dropbox/research/1")
if (user == "Matilde") {
 switch(os,
  Darwin  = setwd("/Users/Matilde/Google One/project"),
  Windows = setwd("C:/Users/Matilde/OneDrive/article"))
}  # Do nothing otherwise
```

Consider writing a function (Session 4) with `tryCatch()` that would return `NULL` in case `setwd()` is unsuccessful.

Should one load packages?

If a package is loaded, all of its exported functions are accessible from the workspace.

One may want to invoke a function directly using the double colon, `::` (assumes that `data.table` is installed):

```
data.table::fread("s02-large.csv")
```

- Being concise and precise
  - In text: '...where the robust matrix is computed with `sandwich::vcovHC(..., type="HC1")`' instead of the verbose 'we use the `sandwich` package function `vcovHC()` with the `type="HC1"` option ...'
- Dealing with masked functions / avoiding masking

Packages may hide secrets

Some functions in packages are not exported (cannot be simply called by typing their name) – they are accessible via the triple colon, `:::`.

```
plm::lagt.pseries()   # Fails
plm:::lagt.pseries()  # Works
```

Usually, functions are hidden from the user for a good reason (package developers rely on them, but they are not essential for the user).

However, some packages have bugs ⇒ something is not working as expected ⇒ one needs to go deeper.

Enforcing length-1 conditions via && ||

There are **long** AND and OR operators: if the condition length is not 1, they thrown an error (useful for checks even before invoking `if`, thus eliminating the need for checking `if (length(x) == 1)`).

```
x <- -2:2
(x >= 0) && (x <= 0)
#> Error: 'length = 5' in coercion to 'logical(1)'

(x >= 0) & (x <= 0)
#> FALSE FALSE  TRUE FALSE FALSE
```

This behaviour is **new in R 4.3.0**. In versions 4.2 and below, `&& ||` returned the first element (caution!).

Order of operations

AND is evaluated before OR. To change the order, use brackets: `T|T&F = T|(T&F) = T|F = T`, but `(T|T)&F = T&F = F`.

Logical operations have the lowest precedence (use brackets to make them evaluate first): `T&F*5` is evaluated to `T&(F*5) = T&0 = F` (because in the logical context, 0 is typecast to `F`).

Use brackets for legibility if the order is non-obvious: `!1:10 %in% 2:3 = !(1:10 %in% 2:3)`, but the latter makes the condition vector stand out.

Do not over-rely on ==

Despite `==` being called the 'equality test' operator, it may behave unexpectedly in many cases:

- `0.7-0.4-0.3 == 0` is `FALSE` because it is `-5.6e-17`
  - `all.equal(0.7-0.4, 0.3, tolerance=1e-12)`
- Equality check via `==` is vectorised: `1:3 == 3:1` is `c(F, T, F)`. To compare two objects, use `identical()`
- Zero-length or `NA` checks are special:
  - `is.null()` and `is.na()` must be used
  - `if (TRUE[FALSE]) print("Yes")` throws an error because `TRUE[FALSE] = NULL` has length zero – it might be worth checking `if (length(...) > 0)`
- For checking classes, there are specialised functions `is.data.frame()`, `is.factor()`, `is.Date()`, ...

Checking equality to TRUE

When x not logical or has length 0, `if(x)` fails:

```
ids <- setdiff(1:3, 1:4) # Length 0
if (ids[1] < 10) print("First id is small")
#> Error: missing value where TRUE/FALSE needed
if ("Truthy") print("Feeling truthy today")
#> Error: argument is not interpretable as logical
```

Check if something is really **TRUE** (logical, has length 1 and not **NULL** or **NA**) via `isTRUE()` (same for `isFALSE()`).

Use case: `all.equal()` returns either **TRUE** or some string ('Mean difference', 'Lengths differ' etc.):

```
all.equal(1:3, 4:1) # "Numeric: lengths (3, 4) differ"
if(!all.equal(1:3, 4:1)) print("Check") # Error
if(!isTRUE(all.equal(1:3, 4:1))) print("Check") # Works
```

Limitations of isTRUE

- `isTRUE()` does not catch errors: if the condition checking returns an error, evaluation will stop
  - `x <- "ABC"; isTRUE(sum(x) > 0)` fails because `sum(x)` throws an error for character x, not `NA` / `NaN`
- Sometimes, `isTRUE()` is redundant because there is a better condition-checking function that does the job
  - To get non-null dimensions even from vectors, one can use `NCOL()` instead of `ncol()` + `NULL` check:

```
if (isTRUE(ncol(x) > 1) & length(b) == 1)
  b <- rep(b, ncol(x))   # Redundant
if (length(b) == 1) b <- rep(b, NCOL(x))
```

  - If x is a vector, `NCOL(x) = 1`, `rep()` does nothing

Inconveniences of 'while' loops

1. Extra diagnostic code required to keep track of the number of steps

```
s <- i <- 0
set.seed(1)
while (s < 100) {
  s <- s + rnorm(1, mean = 10)
  i <- i + 1
}
cat("Took", i, "iters to reach s =", s, "\n")
#> Took 10 iters to reach s = 101.322
```

2. What if the loop never terminates?

```
x <- 1
while (x > 0) x <- x + 1  # HANGS
```

   - Always cap iterations; equivalent to a 'for' loop + break

Loop iterator variable creation

The loop iterator variable is created in the current environment that **remains after the termination**:

```
rm(list = ls())
exists("j")  # FALSE
for (j in 1:10) print(j)
exists("j")  # Now TRUE
```

Recall Session 2 ('Why naming matters'): if `i` already exists in the current environment, it is overwritten.

```
i <- function(x) print(x^2) # Horrible func name
i(8)  # Prints 64
for (i in 1:5) print("Doing stuff 5 times")
i(8)  # Error: could not find function "i"
```

Breaking out of loops

Loops can be terminated prematurely based on a condition:

```
for (i in 1:10) {
  ret <- i^2
  if (ret > 50) break
}
# Execution resumes after the loop

cat("Stopped at i =", i, "for ret =", ret, "\n")
#> Stopped at i = 8 for ret = 64
```

The objects that existed within the loop at the moment of breaking remain as they were at the moment of breaking in the current environment.

Logical and numeric

`TRUE` and `FALSE` are converted to 1 and 0 respectively when used in calculations:

```
3 + TRUE      # 4
2 + FALSE*5   # 2
as.numeric(c(TRUE, FALSE)) # c(1, 0)
c(TRUE, FALSE) + 0  # Same but a dirty hack
as.logical(0:1)     # c(FALSE, TRUE)
```

If a real number is forced as a logical, `0` is treated as `FALSE`, and any non-zero number as `TRUE`:

```
as.logical(c(-100, -0.1, 0, 0.7-0.4-0.3, 1))
#> T T F T T because 0.7-0.4-0.3 = -5.6e-17
```

Emerging empty names

If a vector is selectively named, `NA` names are assigned:

```
a <- 1:3
names(a)[2] <- "Second"
print(a)
#>  <NA> Second   <NA>
#>     1      2      3
```

However, matrix rows / columns cannot partially named:

```
a <- matrix(1:12, nrow = 3, ncol = 4)
colnames(a)[2] <- "Second"
#> Error: length of 'dimnames' [2]
#> not equal to array extent
```

The name vector length for matrices must be exact:

```
colnames(a) <- rep("", 4)
colnames(a)[2] <- "Second"
```

Duplicated matrix dimension names

R works fine with duplicated matrix dimension names, despite it being *bad practice* leading to errors:

```
a <- matrix(1:12, nrow = 3, ncol = 4)
colnames(a) <- LETTERS[1:4]
b <- a+100
all.equal(colnames(a), colnames(b))
d <- cbind(a, b)
colnames(d)  # A B C D A B C D
```

Why it is bad: selecting by column name will select only the first column with a matching name!

```
d[, "A"]
#> 1 2 3  # Column 5 (101, 102, 103) omitted!
```

Data frame name de-duplication

Unlike `cbind()`, `data.frame()` adds `.1`, `.2`, ...if some elements have duplicated names (to avoid confusion):

```
a <- matrix(1:12, nrow = 3, ncol = 4)
colnames(a) <- LETTERS[1:4]
b <- a+100
d <- data.frame(a, b)
colnames(d)
>#  A  B  C  D A.1 B.1 C.1 D.1
```

However, the nothing is stopping the user from enforcing duplicated names, which is to be avoided:

```
colnames(d)[c(1, 5)] <- c("A", "A")
colnames(d)  # A B C D A B.1 C.1 D.1
d$A  # 1, 2, 3 -- same as before
```

Name propagation

Not all elements may be named – in this case, the 'umbrella' names are stretched with a numeric sufffix.

```
y <- c(N = 100, stats = c(0, 1, -99))
#>      N stats1 stats2 stats3
#>    100      0      1    -99
```

If new names are added during concatenation, the original names of the internal object are prefixed:

```
x <- c(Mean = 3, SD = 2)
z <- c(N = 100, stats = x)
#>      N stats.Mean   stats.SD
#>    100          3          2
```

Row / column subsetting test framework

Create a reasonably sized data frame `d` by repeating `mtcars` 1000 times vertically and 30 times horizontally and a numeric matrix `m` (because `d` has only numeric data and no non-numeric classes):

```
d <- do.call(rbind, # Creating 32k rows
       replicate(1000, mtcars, simplify = FALSE))
d <- do.call(cbind, # Creating 330 cols
       replicate(30, d, simplify = FALSE))
colnames(d) <- paste0("X", 1:ncol(d))
m <- as.matrix(d)
```

Then, we test subsetting with these data.

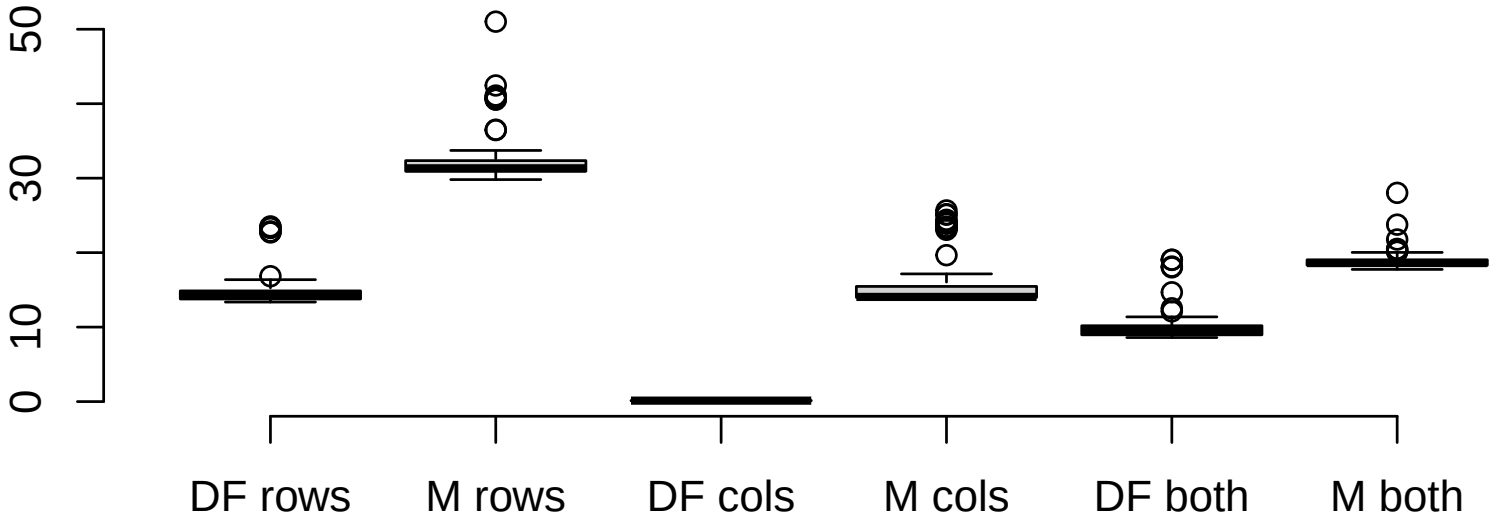
Row / column subsetting code

Run 3 tests: randomly subset 10 000 rows / 200 columns / both. We create 2 functions, `rRows()` and `rCols()`, to generate random row / column indices, and run the code 100 times using the `microbenchmark` package.

```
mb <- microbenchmark::microbenchmark
rRows <- function(n = 1e4)
  sample(1:nrow(d), size = n, replace = TRUE)
rCols <- function(n = 200)
  sample(1:ncol(d), size = n, replace = TRUE)
mr <- mb(d[rRows(), ], m[rRows(), ],
         d[, rCols()], m[, rCols()],
         d[rRows(), rCols()], m[rRows(), rCols()],
         unit = "milliseconds")
```

Row / column subsetting times

Timing in milliseconds (less = better):



- Data frame row or r+c subsetting *seems* 2× faster
  - But why ‘seems’, not ‘is’?
- Data frame column subsetting is 500× (!) times faster

Data frame subsetting perks

Why is resampling columns faster than resampling rows?

- In data frames, columns are short individual vectors (as list elements); manipulating them one by one is blazingly fast

Why is resampling DF rows faster than matrix rows?

- To re-sample matrix rows, sub-vectors are selected and concatenated from the huge full-matrix $r \times c$ vector

Row and column operation speed

Matrix operations by column are *slighty* faster than by row

- Matrix columns are made of *sequential* elements of the large $r \times c$ vector); getting elements `1:3` is marginally (nanoseconds) faster than `c(1, 1001, 2001)`

Compare the speed of row vs. column summation for a 1000×1000 matrix (nanoseconds add up):

```
m <- matrix(runif(1e6), nrow = 1e3)
microbenchmark::microbenchmark(
  colSums(m), rowSums(m))
#> colSums: 1.1--1.4 ms, rowSums: 2.2--3.2 ms
```

Numeric matrices from mixed data

Create a fully numeric matrix by recoding characters, factor, dates etc. as *integers* (is *fully reversible* if one saves the levels, i. e. the 'codebook' approach).

- factor: save levels → `as.integer()`
- character: → factor → `as.integer()`
- date: → `as.integer()` (days since 1970-01-01)

```
dn <- d # Creating a copy
char.vals <- levels(factor(d$char))
fac.levs  <- levels(d$clr)
dn$char   <- as.integer(factor(dn$char))
dn$date   <- as.integer(dn$date)
dn$clr    <- as.integer(dn$clr)
m <- as.matrix(dn)  # Fully numeric matrix
```

Restoring mixed data from numeric

From integer: subset saved levels with integer indices to get character vectors (recode if needed).

```
dr       <- as.data.frame(m) # Matrix copy
dr$char <- char.vals[dr$char]
dr$date <- as.Date(dr$date)
dr$clr  <- factor(fac.levs[dr$clr])
all.equal(d, dr) # TRUE
```

This arcane manner of handling data is ubiquitous.

- Censuses, surveys, admin. data rely on codebooks:
  1 = married, 2 = divorced, ..., 99 = non-response
- Many data sets for SPSS / SAS suffer from this
  - Saving characters (not integers) = waster of space; some developers are too lazy to implement compression
  - Enforced backwards compatibility with 1990s workflows

Empty arrays and data frames

When nothing matches the logical condition in array subsetting or the index is `NULL`, the complementary array dimensions are saved. The same is valid for data frames.

```
x <- matrix(1:8, nrow = 2)
x[, colSums(x) < 0] # colSums(x) are positive...
#> [1,]
#> [2,]
dim(x[, colSums(x) < 0])  # 2 0
x[rowSums(x) < 0, ]
#>      [,1] [,2] [,3] [,4]
x[NULL, ]  # [,1] [,2] [,3] [,4]
mtcars[NULL, ]
#>  [1] mpg  cyl  disp hp   drat wt   qsec ...
#> <0 rows> (or 0-length row.names)
```

round ≠ format

- `round()` accepts what to round as the 1st argument, and #digits as the 2nd – `sprintf()` does the opposite
- Rounding with `sprintf()` works differently:

```
round(0.45, 1)                          #  0.4
sprintf("%.1f", 0.45)                   # "0.5"
sprintf("%.1f", 0.44999999999999999)    # "0.5"
sprintf("%.1f", 0.4499999999999999)     # "0.4"
```

Problems with factor value substitution

```
x <- c("Carol", "Bob", "Bob", "Carol", "Alice")
f <- factor(x);  print(f)
#> Carol Bob   Bob   Carol Alice
#> Levels: Alice Bob Carol
```

Simple value substiution does not work if the assigned value does not exist in the factor:

```
f[f == "Bob"] <- "Bill"
#> Warning: invalid factor level, NA generated
# Carol <NA>  <NA>  Carol Alice
```

The levels can be renamed via direct assignment:

```
levels(f)[2] <- "Bill"
f
#> Carol Bill  Bill  Carol Alice
```

Matrices from mixed-class data frames

A data frame naturally holds vectors of any type.

```
set.seed(1)  # For reproducibility
d <- mtcars
d$char <- sample(LETTERS, size=nrow(d), replace=TRUE)
d$date <- seq.Date(as.Date("2021-01-01"),
                   length.out = nrow(mtcars), by = "month")
d$clr  <- factor(sample(c("Red", "Green", "Blue"),
                        size = nrow(d), replace = TRUE))
sapply(d[, c("mpg", "char", "date", "clr")], class)
#>         mpg        char    date         clr
#>   "numeric" "character"  "Date"  "factor"
```

Some applications require only numeric matrices (nothing else). What can one do? Use **formulæ** to carry out meaningful conversion to numeric!

Shorter object names with 'with'

R supports multiple data frames with arbitrary data types. Yet, there is a certain elegance to the restrictions in certain software packages: one data set = shorter invocation.

`with()` creates interprets object names as list element names. It saves time if the list/DF name is too long.

```
cor(mtcars$mpg, mtcars$wt)
with(mtcars, cor(mpg, wt)) # Identical

clean.data <- mtcars
clean.data$sum <- with(clean.data, mpg + cyl + am)
# clean.data$mpg + clean.data$cyl + clean.data$am
```

Drawback: no auto-complete with Tab in RStudio!

Formula type

Formula: an expression used to formalise a relationship that can be used to transform the data.

Usually looks like `y ~ x1 + x2` (without ""s).

- Allows one to generate transformed variables on the spot without tedious preparations
- Respects the variable class

Formulæ are not universal (e. g. cannot specify an EViews-like dynamic relationship) or reversible (e. g. $Y \mapsto \log Y$, modelling $\widehat{\log Y}$, and automatically getting $\exp(\widehat{\log Y})$). **But:** certain packages can handle dynamic terms (`plm`) or transformations (`forecast`).

Formula term

We shall discuss formulæ for models in Session 7:

- Multiple regression: `y ~ x1 + x2`
- Product: `x1:x2`
- All interactions: `x1*x2`
- No incercept: `-1`
- Dummies: `factor(x2)`
- Transform: `y ~ I(f(x1))`
- Everything else: `y ~ .`

Formulæ can be generated with string manipulation (no need to write the entire expression manually).

Today, we only use formulæ to convert factors → dummies.

Factors to binary matrices

- Factor variables can be losslessly converted into a set of dummy indicators
- Since `factor()` is a mapping into a set of integers, if there are k distinct factor levels, creating k dummies (one for each level) is its lossless numeric equivalent

Factors to dummies sans the baseline

Let `clr` denote the colour of a car: red (baseline), green, or blue. An economist is estimating fuel efficiency (miles per gallon) as a function of horsepower, weight, and car colour.

$$\text{mpg} = \beta_0 + \beta_1 \text{hp} + \beta_2 \text{wt} + \gamma' \text{clr} + U$$

`clr` is a factor, there is an intercept in the model ($\beta_0 \cdot 1$) $\Rightarrow$ green cars would be compared to red cars (baseline) $\Rightarrow$ the red category is omitted. Estimable equation:

$$\text{mpg} = \beta_0 + \beta_1 \text{hp} + \beta_2 \text{wt} + \gamma_2 \mathbb{I}_{\text{clr=green}} + \gamma_3 \mathbb{I}_{\text{clr=blue}} + U$$

Factors to dummies with the baseline

Interpreting the differences compared to the baseline is sometimes inconvenient (why should red be the baseline?): there may be no good baseline). In this case, a full set of dummies is required.

clr	$\mathbb{I}_{\text{red}}$	$\mathbb{I}_{\text{green}}$	$\mathbb{I}_{\text{blue}}$
Red	1	0	0
Green	0	1	0
Blue	0	0	1

$$\text{mpg} = \beta_1 \text{hp} + \beta_2 \text{wt} + \gamma_1 \mathbb{I}_{\text{clr=green}} + \gamma_2 \mathbb{I}_{\text{clr=green}} + \gamma_3 \mathbb{I}_{\text{clr=blue}} + U$$

Factors to sets of binary indicators

Recall the data set from slide 154. Use a formula *without the intercept* to create a matrix:

```
f  <-  ~ factor(clr) - 1
mm <- model.matrix(f, data = d)
head(mm)
```

By default, the names start with '`factor(...)`', which can be ugly (but can be useful). Change them if needed:

```
colnames(mm) <- levels(d$clr)
d <- cbind(d, mm)
```

NB: R chose factor level ordering 'Blue, Green, Red' because it is alphabetical.

Creating interactions via model matrix

Variable changes such as interactions and functional transformation can be created in a model matrix.

If an economist wants to estimate a *linear* model

$$\text{mpg} = \beta_0 + \beta_1 \text{am} + \gamma' \text{cyl} + \delta' (\text{am} \times \text{cyl}) + \log \text{wt} + U,$$

the computer would be crunching the matrix

$$\begin{pmatrix} 1 & \text{am} & \mathbb{I}_{\text{cyl=6}} & \mathbb{I}_{\text{cyl=8}} & \text{am} \cdot \mathbb{I}_{\text{cyl=4}} & \text{am} \cdot \mathbb{I}_{\text{cyl=6}} & \text{am} \cdot \mathbb{I}_{\text{cyl=8}} & \log \text{wt} \end{pmatrix}$$

```
f <- mpg ~ am*factor(cyl) + I(log(wt))
model.matrix(f, data = d) # It is that simple
```

Special escape sequences

Escape sequence: a combination of symbols that is not interpreted literally (but rather differently). Starts with \

```
cat("a\ta")     # TAB
cat("a\na")     # New line
cat("a\r\na")   # New line on Windows
cat("a\bg")     # Believe it or not, backspace!
cat("\a")       # Alert: make your PC beep
```

The backslash \ needs escaping, too. Recall Session 2, 'File paths on Windows': a path is a character → \ activates special sequences → de-activate \ by preceding it with \: `cat("C:\\boot.log")` → OK; `cat("C:\Users")` → error.

Escaping of quotation marks characters

Quotation marks: " and ' are identical. To write a string with literal ", wrap it in ' (or vice versa). If are present in the same string, precede them with \ to **escape** (de-activate) the QM and make it literal:

```
cat('Restaurant "Paris"', "Café 'Londres'")
cat("Restaurant \"Paris\"")
cat("Restaurant "Paris"")  # Error
```

Backticks (grave accents) are special, too, because they allow creating non-standard names (e. g. names with spaces):

```
cat("A tick \` on my back\n")
?`\``  # Get help on the backtick
```

Further reading

- Organising scripts into projects
  (so that Posit developers do not set your PC on fire)
- Therac-25 and its deadly false zero error code

Thank you for your attention!