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

Compiled from session02.tex @ 2024-04-26 16:57:48+02:00.

Day 2: Getting started with R

Andreï V. Kostyrka
20th of September 2023

UNIVERSITÉ DU LUXEMBOURG

Quick recap

We learned:

- How computers ‘memorise’ things and ‘compute’ results, and what they sacrifice in the process
- What features exist out there in popular statistical packages
- How some things feel in object-oriented programming languages, and how there is no royal road

Today, we do the first systematic steps.

Safety first: we start by learning how things ought to be organised in general.

Presentation structure

1. Best practices in programming
2. R and its ecosystem
3. Working without and with interface
4. Basic operations and data types in R
5. Getting help

Best practices in programming

Good problem-solving practices

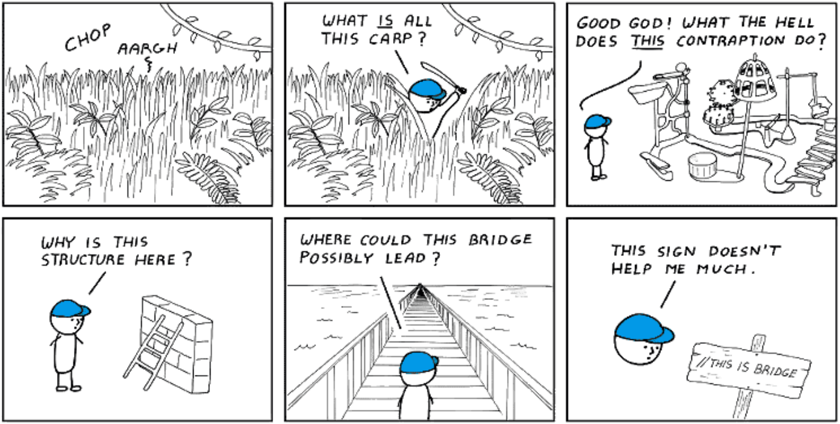
- No partial band-aid solutions, no kludges
- Dig to the bottom: If something is fundamentally flawed, address the core issue, be prepared for going down the rabbit hole
- In economic research, the problem is rarely superficial or solvable with cosmetic changes
  - ‘my grid is not fine enough’, ‘my bandwidth is too small’, ‘I don’t like how these 3 observations look like outliers’ = bad science!
  - Could be a bug in your code or the R package (we learn how to dig deeper in Session 4)

Rubbish in, rubbish out

- Problems can appear at **any** stage of research
- Constantly monitor the analysis workflow
  - Check the visuals (first and foremost!) and descriptives

Bad coding practices

- Spaghetti: unstructured, messy, disorganised code
- Obfuscation: using strange and unusual tricks or abusing language features, creating cryptic and error-prone code
  - Check out IOCCC submissions
- Monolithic, inflexible goliaths
  - No ‘too big to fail’; a straw **will** break the donkey’s neck
  - However, dozens of one-line functions are also bad
- Not doing checks and tests
  - Especially important in economic data sets: `education` (in years) can be 0, ..., 21 (or so); the values ‘999’ or ‘-1’ mean ‘something is wrong!’



‘I hate reading other people’s code!’

Credit: Abstruse Goose (2012).

Code comments

- More comments ≠ better: comprehensiveness is the enemy of comprehensibility
- Comments contain bugs and may become obsolete

```
# Checks that the sample size is at least 80
if (n >= 100) {
    # <apply a simplified formula>
}
```

- There are tests, parsers, linters etc. for code – but nothing for comments
- Comments can lie, but code cannot

Documentation ≠ comments

- Write documentation: high-level architecture
- How one should use the code, which principles and methods are employed
- Say what a function represents, expectation of input and output types, reasons why it may fail
    - *Example:* 'If $n > 1000$, may take minutes to run. Reduce the search tolerance for a speed up.'
- **In R:** Roxygen (only a brief mention)

Comments are still useful

- Comment non-trivial parts

```
# This hard-coded lookup works 3x faster in most cases
# and returns results accurate up to sqrt(doubleeps)
# https://stackoverflow.com/questions/42617883/
# why-sampling-matrix-row-is-very-slow
```

- If you got inspiration or borrowed code or a popular algorithm from somewhere else, refer to the source

```
# ISO calendar algorithm from
# https://webspace.science.uu.nl/~gent0113/calendar/
# isocalendar_text_5.htm
```

- If one needs human language to describe your code, maybe they should write more human code

Do not try to be funny in code or comments

- The person who reads your code knows nothing about the entire project
- Respect the person reading the code (and their time)
  - Do not write obscenities

Infamous fast $1/\sqrt{x}$ (by Gary Tarolli?) in C (simplified):

```
float Q_rsqrt( float x ) { // <...>
  // evil floating point bit level hacking
  i  = * ( long * ) &x; //
  i  = 0x5f3759df - ( i >> 1 ); // what the f*ck?
  y  = * ( float * ) &i;
  return 0.5F * y * ( 3.0F - x*y*y );
}
```

KISS principle

‘Keep it simple, stupid!’ (1960, K. Johnson, Lockheed)

‘Keep it short and simple!’ (1938, Minneapolis Star)

A jet aircraft must be repairable by an average mechanic in the field under combat conditions with few tools.
Stupid = relationship between the way things break and the sophistication available to repair them.

- DRY: don’t repeat yourself (Hunt & Thomas, ‘The Pragmatic Programmer’, 1999)
- Fail fast instead of failing silently (stops error propagation)

KISS example

> *I once won the first place in my university's poker AI competition. We had 2 hours to build a bot.*
> *I was a freshman and had no idea what I was doing.*
> *My algorithm was literally 2 lines.*

```
if isMyTurn:
  goAllIn()
```

> *It broke all the other bots, who started folding every single time.*
>
> `versaceblues` *on Reddit*

Not justification for obfuscation, skipping input checks, or extreme golfing! Simple ≠ unsafe or haphazard.

UNIX design philosophy

- Make each programme / function do one thing well
  - Make it easy to test and run programmes
- Write programmes to work together / handle streams
  - Expect the programme output to become input to another (unknown) programme
- Do not hesitate to throw away and rebuild clumsy parts
- Use tools in preference to unskilled help, even if you have to build them for a single run
  - Your inputs may change at any time, requiring re-calculation

Distinguishing similarities

- Like all adequate programming languages, R is **case-sensitive**: `var`, `VAR`, and `VaR` = 3 distinct objects
  - Case-insensitive languages are inadequate (1950s punchcard legacy, fewer holes to punch)
  - Windows file paths are not case-sensitive – on other systems, luckily, they are
- Many varieties of quotation marks:
  `'` ≠ `"` ≠ “ ≠ ” ≠ „ ≠ ‘ ≠ ’ ≠ `` ` ``
  - Attention when copying and pasting formatted text (text processors, pretty Web pages)

Line wrapping

Each new expression that represents a separate operation / instruction should start from a new line.

In rare cases, single-line statements with semicolon separation could be acceptable:

```
f <- function(x)
 {colnames(x) <- c("Time", "Unit"); return(x)}
```

However, the use of ';' is highly discouraged. This is the recommended way:

```
f <- function(x) {
  colnames(x) <- c("Time", "Unit")
  return(x)
}
```

Nested-structure indentation

Nested structures look better when formatted with a different number of spaces (like physical nested boxes):

```
n <- 10                 # Outer level
for (i in 1:n) {        # 1st nest
→cat("=== Counting up to", i, "===\n")
→for (j in 1:i) {  # 2nd nest
→→cat("i=", i, ", j=", j, "\n", sep = "")
→→s <- i + j
→→f <- qf(0.95, df1 = i, df2 = j)
→→cat("Their sum is", s,  # 3rd level
→→→"and 95% F crit. val. is", f, "\n")
 }
}
```

Two styles of indentation

Style 1: consistent spacing (each new level = 2, 4, or 8 spaces, or 1 tab) – easy to achieve with styling tools:

```
cat("Their sum is", s,  # Using 2 spaces
  "and 95% crit. val. is", f, "\n")  # everywhere
```

Style 2: arguments aligned to logical parts (requires more context-dependent user input):

```
cat("Their sum is", s,                # Comments are
    "and 95% crit. val. is", f, # aligned, too
    "\n")
```

In either case, do not nest too deeply (unless writing in Python, where it is a 'feature').

Object naming rules in R

- Names may contain letters, digits, ., and `_`
  - In Python, `df.head()` applies the `head()` function to `df`. In R, `.` makes no call (one would write `head(df)`)
- Names can start only with a letter
  - `x`, `x1`, `x.y`, `x.1`, `x_1` are valid
  - `x@`, `2x`, `_x` are invalid; `1e5` is a number (100 000)
  - Be consistent: `my.cool.name` vs. `their.ugly_name`
- Names starting with a dot (not followed by a digit) make objects **hidden**
  - Do not use `.x` unless you are a developer
- Dots in names are *occasionally* used to call S3 methods (wait until Session 4)

Naming conventions

No standard guidelines, but...

google.github.io/styleguide/Rguide.html

- Name non-functions and functions differently: `my.data` vs. `prepData()`
  - In these slides, functions have brackets and highlighting
- Name iterators `i`, `j`, `k` (or `ii` etc.)
  - But do not forget which ones are already in use
- Do not name objects `temp`, `xx`, `res`, `asdfghjkl` etc.

Give short names to frequently used objects, give human-readable names to create self-documenting code.

A bit ahead of pace: Why naming matters

Badly named objects may get overwritten unintentionally.

```
i <- 101 # Number of regions in France
t <- 8   # Years in the data set
d <- read.csv("french-census.csv")

# <..Iteratng from the baseline model..>
mod <- lm(mortality ~ winepercapita, data = d)
for (i in 1:5) mod <- coolMethod(data = d,
                         init.val = coef(mod))

for (k in 1:i) { # And now -- loop over regions
  print(sum(d$region == k)) # Oversight!
}
```

The variable `i` is now equal to 5!

Borrowing others’ code

- Online tutorials, StackExchange, ChatGPT etc. are amazing resources for getting help with a fragment of your research
- However, it is easy to borrow others’ bugs or mistakes
- Your research is as accurate (at most) as the most inaccurate and error-prone part of the code
- Give credit where it is due
  - Sometimes, one cannot borrow due to licence restrictions / patents / copyright
  - Reuse popular methods, but to not plagiarise others’ research

Any questions on good programming practices?

Office hours revisited

- Office hours, as per the syllabus, seem outdated in the context of this technology-focused course
- Moodle does not encourage information exchange
- Group chats are a reasonable way for all participants to share their experience, ask questions, get answers, and everybody see those Q&A

Should we create a dedicated group chat for R code / problem exchange for your research, or is it unnecessary?

R and its ecosystem

R console interface

- Big scientific calculator
- Scroll history with ↑ / ↓
- Auto-complete with ⇥ (TAB)

R workspace

- The workspace is your current R working environment
- Includes any user-defined objects (vectors, matrices, data frames, lists, functions)
- Global environment: built-in functions
- The entire environment can be saved as an image (all objects)

Packages

Confusing terminology:

1. *'package'* as in *'statistical package'* (e. g. Stata, Matlab)
2. *'package'* as in *'module'*, *'library'*, *'extension'*, *'framework'* (we use this definition)

Package / library: a set of functionalities written by someone else that can be imported and reused.

Popular programming languages have lots of libraries in centralised repositories (repos). TEX: 6500 (CTAN), R: 20 000 (CRAN), Rust: 120 000 (crates.io), Perl: 215 000 (CPAN), Python: 475 000 (PyPi), JavaScript: 2 500 000 (npm)

Where to get R packages

- The Comprehensive R Archive Network (CRAN) – default
  - `install.packages("packagename")`
- From other people's code repositories
- From downloaded compressed or archives
- Or even write your own ones!

Versions of R

- 3-part semantic version: x.y.z
  - x = major release (huge, potentially breaking changes), y = minor release, z = patch
- Multiple versions of R can be installed at the same time
  - One can easily set a specific version as the default one
  - R is relatively stable; 3.5.3 from 2019 works fine in 99% cases

R user packages

- 20 000 as of the moment of writing
  - Stand-alone or families (e. g. tidyverse)
  - Even more if unofficial repositories are counted
- Most written in **base R**, some require compilation
  - Windows users get pre-compiled binaries or can choose to compile them with **Rtools** (compilers + libraries)
  - Linux and Mac users compile from source automatically (potentially better performance with processor instruction set optimisation)
- Are often called 'libraries' (cf. `library(foreign)` vs. `update.packages()`

How to enable package capabilities

Installing: `install.packages("AER")` fetches it from the web repository and writes onto the hard drive (or click *Tools* → *Install Packages* in RStudio). **Do it only once!**

`AER` provide the `ivreg()` function. By default, upon fresh start, no packages are loaded into the memory; simply running `ivreg()` after installation produces an error.

Loading: to use `ivreg()` from `AER`, one has to ‘activate’ the package after it has been installed:

```
library(AER)
```

Then, `ivreg()` becomes available.

Alternative: without `library()`, write `AER::ivreg()`.

Package versions

- By default, the most recent version of a package is installed
- Package developers do not adhere to the x.y.z versioning scheme
- Whenever a *minor release* of R is out (e. g. 4.2.3 → 4.3.0), the packages are installed anew
  - *Example:* with R 4.2.1, the packages are installed to the local folder '4.2'
  - Upgrading R 4.2.1 to 4.2.3 preserves all locally installed packages, but 4.3.0 will create a new folder, '4.3', and cannot use anything installed for 4.2.z

Package installation paths

Default: subfolder in user's home (can be changed).

- Get the paths by running `.libPaths()` in R

Windows:

`C:\Users\YOURNAME\Documents\R\win-library\4.3`

Mac:

`~/Library/R/x86_64/4.3/library`

Linux (including the Uni.LU HPC):

`~/R/x86_64-pc-linux-gnu-library/4.3`

Here, ~ denotes the user home folder (`/Users/YOURNAME` on Mac, `/home/YOURNAME` on Linux).

Getting the package version

After installation, the package will stay at the same version, and will not be updated automatically.

However, on a different system, a later installation of a package may introduce discrepancies in the output.

Check the package version:

```r
packageVersion("AER")
```

Check the versions of all currently loaded packages:

```r
sessionInfo()
```

File paths on Windows (1/2)

Windows separates file paths with \ (backslash) instead of / (slash).

- A notorious deviation from the widespread convention in programming

Use forward slashes on all systems (Windows, Mac, Linux):

```
# Windows
d <- read.csv("C:/Users/avk/Desktop/db.csv")
# Mac, Linux, UNIX-like
d <- read.csv("~/Desktop/db.csv")
```

Possible but bad: **escaping** backslashes on Windows.

```
d <- read.csv("C:\\Users\\avk\\Desktop\\db.csv")
```

File paths on Windows (2/2)

DOS legacy build-up since the 1980s: Windows may print the **8.3 file path** (8 name characters, abbreviated via ~, dot, and 3 characters of extension).

```
Sys.getenv("R_HOME")
#> C:\PROGRA~1\R\R-43~1.0
#> Should be C:\Program Files\R\R-4.3.0
```

Most languages / systems are **case-sensitive**: `a.txt`, `A.txt`, `A.TXT`, `A.TxT` are different files.

Not on Windows, which is **case-insensitive** (these 4 files cannot co-exist in the same directory).

- Had a Dropbox ‘Case conflict’ with a Mac collaborator?

Why is Windows workflow so different

- Researchers have traditionally been working on UNIX systems (AT&T, Bell Labs)
  - Designed as a convenient platform for programmers developing software
  - Adopted in academic circles, experts sharing good tools
  - macOS core (Darwin) and Linux are UNIX-compatible
- Windows is still relying on MS-DOS conventions
  - Compatibility with 1979 86-DOS (QDOS) for 8086 kits
  - Backslash: 1970 TOPS-10 mainframe quirks (reserved /) + 1981 IBM model F keyboard ergonomics + 1983 MS-DOS 2.0
- Pragmatism in 2023: big-data frameworks (servers, CPU clusters, supercomputers) are only UNIX-compatible
  - Even on MS Azure servers, Linux dominates (since 2019)

Crucial aspect of R: vectors

In R, everything is a **vector**. Vectors can be

- Atomic (a set of elements of the same type)
- Lists (a collection of heterogeneous objects)

A vector is created with a one-letter command: `c()`.

Type the following into the console:

```
c(1, 2, 5)
```

Like in mathematics by default, it is assumed to be a **column** vector (even if it is printed as a line).

Examples of atomic vectors

We talk about data types later, but for now, observe:

- An integer sequence from 1 to 100:
  `1:100`
  (the `:` operator creates integer sequences)
- A character sequence:
  ```
  c("Als", "Gregor", "Samsa", "eines", "Morgens",
    "aus", "unruhigen", "Träumen", "erwachte")
  ```
- A logical vector: `c(FALSE, TRUE, TRUE, FALSE)`

Vectors and concatenation

The symbol **‘c’** stands for con**c**atenation.

Vectors can be combined (i. e. concatenated):

```
c(1:3, 12:15)
```

It is not a mistake to wrap a vector into a vector (the redundancy will be simplified):

```
1:3
c(1:3) # Same
c(c(c(c(c(1:3))))) # Ugly, but not a mistake
```

Installing from official CRAN repositories

Now that we know how to use vectors…

Install one or many packages at once:

```
install.packages("plm")  # For panel models
install.packages(c("DEoptim", "hydroPSO"))
# For Session 6, numerical optimisation
```

Installed something heavy? Remove it!

```
remove.packages(c("DEoptim", "hydroPSO"))
```

Package updates

Update everything installed: `update.packages()`

- If it ain't broken, do not fix it
  - *Real case:* `readxl()` from `tidyverse` threw an error and broke the entire R installation (!)
  - It required `rlang >= 1.0.6` when `1.0.2` was installed; upgrade failed because the DLL was locked
- If a full upgrade fails due to package A:
  - Try upgrading A manually: `update.packages("A")`
  - If the DLL is in use, reboot the computer and retry
  - Remove the leftover `LOCK` directory: `C:\Users\YOURNAME\Documents\R\win-library\4.1\00LOCK...`

Installing from the CRAN archive

Real example: I wanted to install the np package (for **n**on-**p**arametric methods), but it depends on `cubature`, and I could not compile `cubature` v. 2.0.4.6 on one specific machine. *(Suppose that v. 2.0.4 works.)*

Solution: **all** versions of **all** CRAN packages are archived! cran.r-project.org/src/contrib/Archive/cubature

```
install.packages("devtools")
devtools::install_version("cubature",
    version = "2.0.4")
```

Find the last version that works for you (assuming that the old version interacts well with other dependencies).

Installing from GitHub

The `devtools` package has a convenient function to install from GitHub.

If the repo is `https://github.com/sachit27/greenR`, the command to install the package is

```
install.packages("devtools")
devtools::install_github("sachit27/greenR")
```

If prompted to update, type the number and press Enter.

(If you had previously installed `devtools`, there is no need to run the `install.packages("devtools")` command again.)

Installing from GitLab

Similarly, there is a different command in the `devtools` package to install from GitLab.

If the repo is `https://gitlab.com/jimhester/covr`, the command to install the package is

```
devtools::install_gitlab("jimhester/covr")
```

(Assuming that `devtools` is installed.)

The tidyverse ecosystem

tidyverse: a collection of custom packages for data manipulation and visualisation extending the functionality of R greatly and providing new definitions.

```
# install.packages("tidyverse")  # Do not run yet
```

- Can be good for data cleaning and transformation
  - Can be faster, can be slower; can hang or halt with large data if more general good practice is neglected
- Syntax different from base R (is a dialect / custom framework)
- Introduces a specific structure called **tidy data**
  - Most packages cannot handle ‘tidy data’ = extra work converting back and forth

Tidyverse

Packages Blog Learn Help Contribute

R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

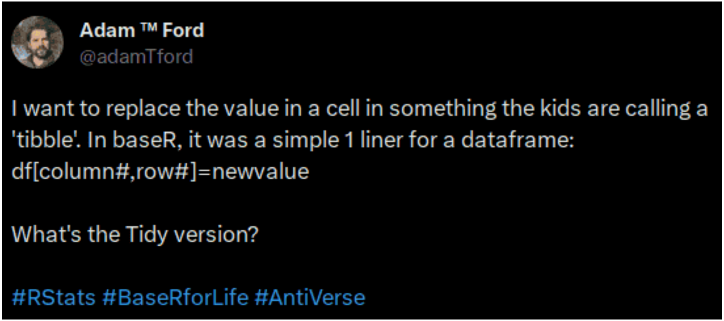
tidyverse drawbacks

- Frequent breaking changes and deprecations
  - Good for experimentation, but code rot is bad for research
  - Changelog for `dplyr` is all 'Breaking', 'Deprecated', 'Experimental', 'Removed', 'Superseded'
- Meta-layer that is hard to debug (too abstract)
  - Unnecessarily complicated for a narrow set of operations
  - Beginners make the most mistakes ⇒ counterproductive
  - Hard to craft an error-free chain in the first place
- Tutorials de-emphasise vectors and lists – the real strength of R!
  - Very limiting ('a thousand gadgets') without base R skills
  - Hurts other high-quality or high-performance, or specialised packages (jack of all trades = master of none)

Real-world problems are too serious

- Converting data with Arabic text from XLS to Stata DTA
  - One needs a good tool to handle right-to-left writing before encoding factors
  - Impossible on Windows before R 4.2: legacy code pages
- Processing downloaded web pages en masse, parsing, and scraping
  - Too many page layout designs even on the same site ⇒ custom-depth list traversal requires interactive checks
- Combining multiple fragmented data sets into a complete one
  - The structure is so different, *tidy data* is powerless
- Transformations with overlapping time windows
  - `data.table` memory issues on large (1m+) data sets

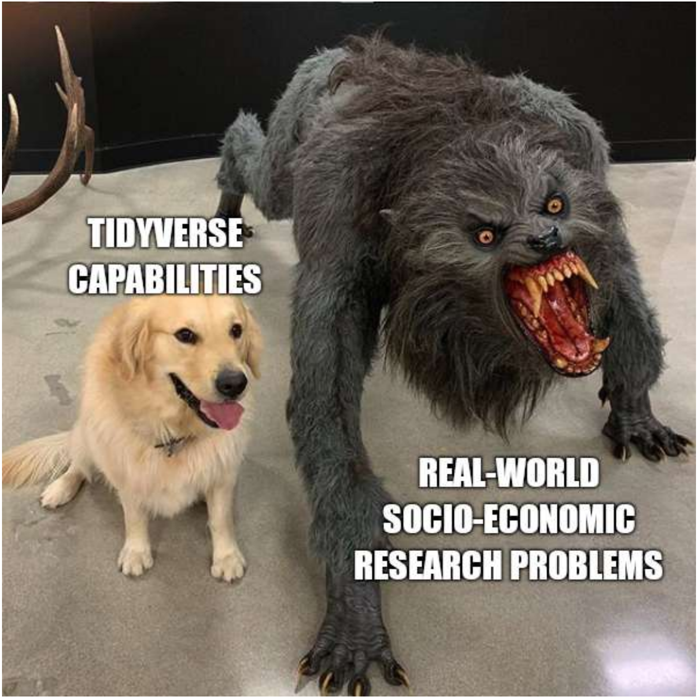
Learning full R, not just tidyverse



No good ‘tidy’ way to do this fundamental operation.

A ‘tidy’-only approach to R teaches learners to do much less with R than a standard R course in the same time.


Why is it, when something happens, it is always you three?
DATA.TABLE SUBSETTING
GGPLOT2 ARGUMENTS AND TIDY DATA
BROKEN MAGRITTR PIPES
imgflip.com


TIDYVERSE CAPABILITIES
REAL-WORLD SOCIO-ECONOMIC RESEARCH PROBLEMS

Dirty tidy code

General programming principles > tidy syntax subset.

```r
ifelse(grepl("cohort.infra", term) & grepl("pd_category", term) & grepl("gain", term) == 0, "cohort.infra.pd",
  ifelse(grepl("cohort.small", term)& grepl("pd_category", term) & grepl("gain", term) == 0, "cohort.small.pd",
    ifelse(grepl("cohort.growth", term)& grepl("pd_category", term) & grepl("gain", term) == 0, "cohort.growth.pd",
      ifelse(grepl("cohort.value", term)& grepl("pd_category", term) & grepl("gain", term) == 0, "cohort.value.pd",
        ifelse(grepl(":", term) == 0 & grepl("pd_category", term), "bond.pd",""
)))))))))))))))))))))))))))))) %>% filter(Type != "")

  ols.broom.age = ols.broom %>% mutate(three.factor.ols  = estimate) %>%
    filter(grepl(".pd", Type) == 0) %>% dplyr::select(AgeFactor, three.factor.ols, Type)
```

This is a real code example from an online appendix to a 2021 article published in JoFi.

base R: 

⇒ 

tidyverse:  ⇒ 

Those kids who used to play with bricks, grow up and build transformers. Those who used to play with transformers, grow up and build bricks. *A. Lebedev.*

Can packages conflict?

Masking: overlapping of functions with the same name from different packages. Multiple masked functions may co-exist without overwriting.

- `lag()` from base `stats`, panel `lag()` from `plm`

Namespace: a special environment for all the functions of a package.

- In C++, `std::cout` with `iostream`

In case of function name conflict, call masked functions via `pkgName::funName`.

```
stats::lag(...)
plm::lag(...)    # Clearly different functions
```

Should I always use package X?

- For complex operations, **yes**
- For simple operations, search 'how to X **in base R**' first
- Often, search and tutorials suggest `tidyverse` solutions, but it may (on many occasions) break things
  - Avoid `tidyverse` (including `ggplot`) in favour of simpler alternatives **wherever possible**
  - `data.table::fread()` reads CSVs blazingly quickly, but it enforces a custom class (`data.table`) that is often incompatible with standard syntax (e. g. subsetting) and requires conversions
  - Most popular libraries (LASSO, XGBoost, random forests etc.) require *pure numeric matrix inputs* – nothing else (no lists, `data.table`'s, `tibble`'s, or other syntactic fluff)

Do not over-rely on packages

- The diversity of the R ecosystem should not distract one from implementing their own functions
  - Spend 5 minutes writing a solution instead of 5 hours searching for somebody else's solution online
- There is no guarantee that what others wrote works well or even correctly
- Cautionary tale: an 11-line package broke the Internet
  - TL;DR: all because many programmers decided to import a dependency that contained only one function that was doing an extremely simple thing (and inefficiently!)
- Try breaking down the question into an algorithm (recall Prof. Fortran peeling potatoes)

Different packages = different syntax

- Different implementations may expect different input types – keep the number of others' functions to an absolute minimum
- Always read the examples for a function from a new package that you are about to use
- Example: `lm()` relies on formulæ, but `glmnet()` from `glmnet` requires numeric matrices

```
lm(mpg ~ cyl + hp + wt, data = mtcars)

glmnet::glmnet(x = mtcars[, c("cyl","hp","wt")],
    y = mtcars$mpg)
```

Transition from other packages

Different software may have different syntax for the same function – requiring a change of one's mindset.

Stata: `ivreg y x_incl (endog = x_excl)`

R (AER package):

```
ivreg(y ~ endog + x_incl | x_excl + x_incl)
```

R (`momentfit` package, dedicated to GMM):

```
mod <- momentModel(y ~ endog + x_incl,
                     ~ x_excl + x_incl,
                   vcov = "MDS")
tsls(mod)
```

Any questions on R extension packages?

Working without and with interface

Software to install

IDE: **i**ntegrated **d**evelopment **e**nvironment; software that highlights, checks code, helps in debugging and browsing.

1. Base R itself
   - `https://cran.r-project.org/`
   - Linux: better compile than fetch from repositories (stability + performance)
2. RStudio (recently merged with Posit) download:
   - `https://posit.co`
3. RTools
   - Windows: install the binary
     - `cran.r-project.org/bin/windows/Rtools`
   - Mac: install Xcode command-line tools
   - Linux: no need if R is compiled, otherwise `r-base` and `r-base-dev`

R sessions

Session: a running instance of R with the loaded packages and data

- Sessions can use sub-processes (see later with parallelisation)
- Sessions might crash under heavy loads (usually when dozens of packages are loaded)
    - *Solution 1:* try from console / batch
    - *Solution 2:* work with smaller objects, do not load packages copy-pasted from other projects

Once a package is loaded, **it stays loaded** unless the user detaches it or removes its namespace.

Scripts are the daily bread

- **Script:** plain text containing runnable instructions
  - No need to click 100 GUI buttons – carpal tunnel syndrome
- R is an interpreted language
  - Cannot output executables, but supports interactive applications (e. g. HTML pages with clickable elements)
  - Optimisations via the built-in just-in-time (JIT) compiler
  - R itself is written in C and Fortran ⇒ most functions wrap the input arguments nicely for low-level functions
- Comments start with `#`
- R Markdown: a combination of output text and source code (good for presentations), similar to Jupyter
  - Can be rendered to PDF or HTML via `knitr`

Working directory

Working directory: the base path from which all relative paths are constructed.

Always set the working directory! Compare:

```
d1 <- read.csv("C:/Users/Grzegorz Brzęczyszczykiewicz/Desktop/thesis/data/raw/base.csv")
cb <- read.csv("C:/Users/Grzegorz Brzęczyszczykiewicz/Desktop/thesis/data/raw/codebook.csv")
r2 <- read.csv("C:/Users/Grzegorz Brzęczyszczykiewicz/Desktop/thesis/data/raw/round2.csv")
```

versus

```
setwd("C:/Users/Grzegorz Brzęczyszczykiewicz/Desktop/thesis/data/raw")
d1 <- read.csv("base.csv")
cb <- read.csv("codebook.csv")
r2 <- read.csv("round2.csv")
```

When preparing a project for sharing, put everything in the same directory and use relative paths (comment out the `setwd(...)` line).

Live demonstration time!

R is a great calculator – try typing this

```
2 + 2  # 4
exp(1) # 2.718282

sqrt(2)  # 1.414214
2^0.5    # Same
2^(1/2)  # Same

sqrt(2)^2 - 2
#> 4.440892e-16 -- kind of expected

pi # 3.141593 -- has 16 accurate digits
sin(pi)
#> 1.224647e-16
sin(pi/4 + 3*6)
#> -0.06411375
```

Mathematical functions in R

Arithmetic operations: `+ - * / abs(x)`

Power-related: `sqrt(2)`, `2^3`, `2^(1/3)`, `exp(2.718)`, `log(8)`, `log(8, base = 2)`

Division remainder (modulo): `107 %% 20` (remainder 7), integer division: `107 %/% 20` (the result is 5)

Rounding: `floor(x)`, `ceiling(x)`, `round(x)`, `trunc(x)`

Trigonometric: `sin(x)`, `cos(x)`, `tan(x)`, `atan(x)`, ...

Special: $\Gamma(x)$ = `gamma(x)`, $\binom{k}{n}$ = `choose(n, k)`

Accurate log and exp for small inputs

For small inputs ($x \approx 0$): `log1p(x) = log(1 + x)`, `expm1(x) = exp(x) - 1`, but numerically stable.

```
x <- 10^-c(11, 13, 15, 17)
cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))

#>     x  log(1+x)  log1p(x)  exp(x)-1  expm1(x)
#>1e-11 1.000e-11 1.000e-11 1.000e-11 1.000e-11
#>1e-13 9.992e-14 1.000e-13 9.992e-14 1.000e-13
#>1e-15 1.110e-15 1.000e-15 1.110e-15 1.000e-15
#>1e-17 0.000e+00 1.000e-17 0.000e+00 1.000e-17
```

Reason: internally, to compute non-linear functions, computers calculate their polynomial approximations; the latter break down for tiny inputs (recall Session 1).

Statistical functions in R: `xdist`

- First letter: replace `x` with
  - `d` for density
  - `p` for probability distribution
  - `q` for quantile
- Replace `dist` with
  - `norm` for normal (Gaussian), `chisq` for χ^2
  - `t` for Student, `f` for Fisher
  - `binom` for binomial (including Bernoulli)
  - `unif` for uniform, `exp` for exponential
  - `gamma`, `beta`, `geom`, `hyper`, `lnorm`, `weibull`, `nbinom`

`dnorm(0)` = normal density at 0 ($\approx$0.3989)

`pchisq(5.99, df = 2)` = $\mathbb{P}(\chi^2_2 \le 5.99) \approx 0.95$ (PDF)

`qf(0.9, df1=2, df2=48)` = 90% F crit.val. w/ 2 & 48 DoF

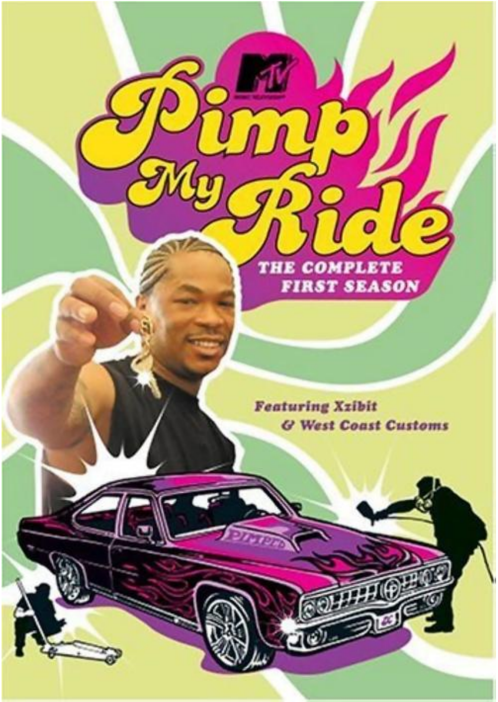
Function arguments

Notice that `log(10)` returns the natural logarithm (base *e*), while `log(8, base = 2)` = $\log_2 8 = 3$.

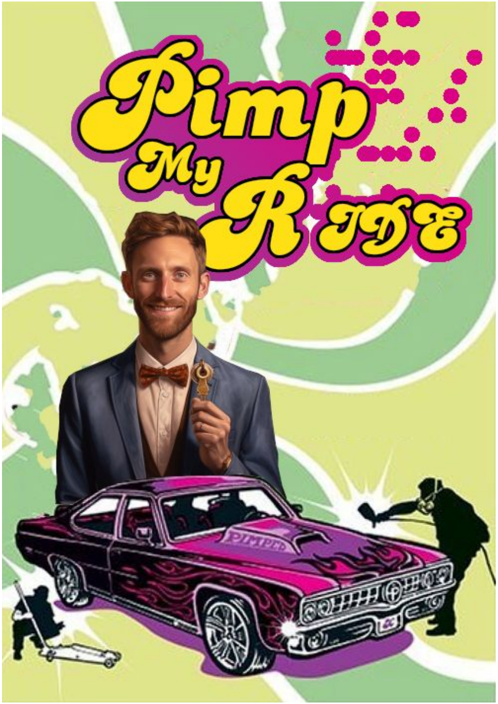
We spend more time on this later, but most functions in **R** are *customisable*: they can change their default behaviour to custom user-requested one.

Functions may work with 1 argument, 2 and more arguments, some of which can be mandatory or optional.

```
qnorm(0.95) # Default: 0.95-qntl of std. Gaussian
qnorm(0.95, sd = 2) # 0.95-qntl of N(0, var=4)
qchisq(0.9, df = 1) # 0.9-qntl of chi2 w/ 1 DoF
qchisq(0.9) # ERROR: no default dgr of freedom!
```



Credit: MTV (2004).


Pimp
My
RIDE
PIMPED

Pimp my R IDE

Go to Tools → Global Options.

- *General* tab: Disable 'Restore `.RData` into workspace', 'Save workspace' → 'Never'
- *Appearance* tab: Save your eyes by enabling a dark colour scheme (Ambiance, Chaos, Cobalt, Dracula...)

Working with IDE

Standard layout:

- Code editor
- Console with an interactive session
- Environment explorer
- Viewers for help, plots, files

Bad: click buttons to load data, set parameters in RStudio menus

Good: write a script to do all the job without requiring any supervision

Most useful features of RStudio

- Auto-save of modified script files
- Previews of tables and plots
- Auto-suggestion and auto-completion based on the existing objects: press ⭾
- Mass comment/uncomment: ctrl/⌘ + ⇧ + C
- Run the next line/fragment: Ctrl + Enter

Remember to disable workspace save / load first.

Code execution in RStudio

- ctrl/⌘ + ⏎ to run the current line or expression
  - If a line is long, or ambiguous, or has unmatched brackets, it will patiently wait for complete output – press ctrl + ⏎ to continue evaluation or Esc to break
- If a fragment of code is running for too long, terminate the session
  - If it does not help, open the task manager (Actvity Monitor on Mac, `top` or anything you like on Linux) and kill the `rsession` process(es)
  - RStudio usually auto-saves, but copy the code to a different editor before killing the session because RStudio will restart

Auto-saving in RStudio

RStudio auto-saves:

- Unnamed files that have not been 'Saved as'
- Modified files if the R session crashes

However, upgrading RStudio may 'kill' all unsaved unnamed files. 'Save as' everything before RStudio updates!

NB. Keep only one RStudio instance. Auto-saving is not guaranteed if multiples copies of RStudio are open.

Best practices for the workflow

- Make sure that your script can run without user input
- Make sure that the extra libraries are listed in the file
  - Write a nice informative message indicating any non-standard system-wise dependencies to be installed manually (e. g. `imagemagick`, `ffmpeg`, Java runtime)
  - Possible to check existing (`rownames(installed.packages())`) and install new CRAN packages automatically
  - Installing packages without any confirmation is considered harmful to the user

Existing features in the environment

Before one creates anything, the following is always available in the environment upon start:

- Built-in package functions
  - Functions from not-yet-loaded packages can still be accessed
- Working directory
- Options and graphics parameters
- Data sets (`mtcars` is useful for quick testing)

Handy existing objects

- Constant π: `pi`
- Built-in character vectors: `letters` and `LETTERS` (a, b, … and A, B, …)
- English month names: `month.name`, `month.abb`

`.Last.value` returns the value of the last top-level evaluated expression.

Parameters and platform variables

- `.Platform` has system information
  - Use `.Platform$OS.type` to detect is the user is using Windows or not; wait until Session 4 (parallelisation)
- `.Machine` has numerical characteristics: `.Machine$double.eps` = machine ε
- `sessionInfo()` for running instance summary, `Sys.info()` for OS summary, `version` for for installation summary, `last.warning` (obvious)
- `par()` for graphic parameters, `options()` for R settings (both can be re-defined; more on that later)

Saving results – advice

Good: keep the input data and the script that does the full job without any interaction and saves the necessary results.

Good: create `RData` files after computationally costly operations and load them in further re-runs. Do not read XLSX with extra packages every time: read XLSX, save in R native format, use henceforth.

Bad: save the full environment into a file (clutter!).

Bad: create multiple R scripts and go back and forth running selected chunks between them in the order that only one person knows.

Functions for saving and loading

Use `save()` to store multiple objects of any type (data, variables, models etc.):

```
a <- 1;  b <- qnorm(0.95)
mod <- lm(mpg ~ hp, data = mtcars)
save(a, b, mod, file = "myName.RData")
```

NB: `save()` inputs are object names; the target file name is a mandatory named argument `file="myName.RData"`.

To exchange files with other users, save data sets as CSV:

```
write.csv(mtcars, file = "myData.csv")
```

To load RData data saved with `save()`, use `load()`:

```
load("myStuff.RData", verbose = TRUE)
```

Example workflow (imaginary)

```
load("myPreviousStuff.RData", verbose = TRUE)
d1 <- read.csv("mydata.csv")
d2 <- openxlsx::read.xlsx("my.xlsx", sheet = 3)

data.clean <- merge(myStuff, d1, d2, ...)

# <Analysis goes here>
f <- formula(...)
mod1 <- estimateModel(f, data = data.clean)
mod2 <- estimateModel(f,
          data = data.clean[subsample, ])
summmary1 <- summary(mod1, robust = TRUE)
summmary2 <- summary(mod2, robust = TRUE)

# <Code for plotting -- wait until Session 5>

save(data.clean, mod1, mod2, summary1, summary2,
     file = "myFinal.RData")
```

Any questions on the interface and workflow?

Basic operations and data types in R

Object types

- Dozens of types, impractical to list them all
- User data: **everything is a vector**
  - Logical (boolean), integer, factor, real (numeric), character
  - Vectors can be wrapped into arrays and matrices
  - Abstraction: lists (including data frames); a list is still a vector (i. e. has length)
  - Custom object types provided by additional packages (e. g. `data.table`)
  - Expressions, formulæ, relationships for automatic transformations
  - Non-standard inputs: `Inf`, `NA`, `NaN`, `NULL`

Objects are everywhere

- Everything that exists is an object
- Everything that happens is a function call

A variable, a function, a data set etc. are objects.

Objects of various **classes** support various **methods** applied to them. One function may be tailored to process different objects differently.

Example: plotting is a **function call** (there in no 'plot' object) to show a representation of the data object(s) somewhere (we shall discuss it later).

Variables

Variable: a container that holds a value that can be created, modified, or deleted.

Assignment: `x <- 1` is better than `x = 1`.

Variable type need not be declared (dynamic typing).
In C/C++, one would type `int x = 1;` R does not care if `x <- 1` or `x <- "ABC"`.

Recall that R is case-sensitive:

```
y1 <- 1; Y1 <- 2
```

creates two different variables.

Multiple ways of assigning values

Recommended: `x <- 3`. Possible: `x = 3`.

Distinction: `x = x+1` makes little sense conceptually, but `x <- x+1` is clearly an overwriting assignment. Also: `x=1` works, but `1=x` fails.

Assign multiple values at once: `a <- b <- 10`.

There are two more ways:

- `assign("x", 4)` is the ‘full’ version of ‘`<-`’ and supports extra arguments (e. g. assigning values to a different environment; useful within nested function)
  - See `?assign` for more details
- `3 -> x` is cursed

NB: spacing matters! `x < -3` checks if $x \le 3$!

Functions

Function: a black box that takes some input(s) and returns some output (always a single return).

Functions have **arguments**. Arguments can be unnamed or named; if their order matches the function definition, argument names can be skipped.

Arguments can have optional default values:
`log(8) = log(8, base = exp(1))`
`= log(base = exp(1), 8)` (see function help).

Functions are so convenient, the entire Session 4 is dedicated to functions in R. They are the backbone of efficient workflow.

Objects with identical names

Functions from packages can share names with user objects.

```
mean
#> function (x, ...) UseMethod("mean")
#> <environment: namespace:base>
mean <- mean(1:4)
mean          # 2.5
mean(10:13)   # 11.5
```

Now, `mean` refers to the number, while `mean()` can still be used to compute averages.

This is safe, but can be confusing for the reader.

Removing objects and cleaning memory

Any user object can be deleted with the `rm()` command. It can be useful if there is the RAM is low.

```
a <- 10
b <- 1:100
a
#> 10
rm(a, b)
a
#> Error: object 'a' not found
```

Sometimes, the memory is not freed – use garbage collection (not needed unless the system hangs):

```
gc()
```

Vector type

In R, everything can be a vector, and some things, can only be vectors.

The function `c()` creates a vector.

The most popular data types created by the user is logical, numeric, character, and factor.

Vectors can be grown and combined:

```
a <- c(1, 2)
b <- c(3, 4)
c(8, 9, a, 5, b, b)
#> 8 9 1 2 5 3 4 3 4
```

Vectorised operations

Most operations in R are vectorised.

Those with a C/C++ background, **forget loops**. Loops can be useful, but in most cases, the best way to carry out vector operations in R is *without loops*.

- Operations between scalars and vectors are term-to-term
- Operations between vectors of equal length are term-to-term

```
1:10 + 100 # Returns 101, 102, ..., 110
c(2, 4, 8) + c(100, 300, 500) # c(102, 304, 508)
```

More on loops later (Session 3).

Creating vectors

Tools: `c()`, `rep()` (any type), `:`, `seq()` (numeric).

The colon operator has precedence over all other ones!

```
1:10+100     # Treated as (1:10) + 100
1:(10+100) # Treated as 1:110
```

If in doubt, extra brackets will not hurt.

```
(1:4)*100  # 100 200 300 400
```

Repeating elements

The `rep()` function creates **rep**eated elements. It comes in various flavours:

```
x <- c(3, 7)
rep(x, times = 2) # 3 7 3 7
rep(x, 2)         # 3 7 3 7
rep(x, times = c(2, 4))  # 3 3 7 7 7 7
rep(x, c(2, 4))          # 3 3 7 7 7 7
rep(x, each = 2)  # 3 3 7 7
rep(x, each = 2, times = 3)
#> 3 3 7 7  3 3 7 7  3 3 7 7
rep(rep(x, each = 2), 3)     # Same
rep(x, times = 3, each = 2) # Same
```

NB: Named argument order does **not** matter!

Creating sequences

Arithmetic sequence from 1 to 11 in steps of 3:

```
seq(1, 11, 3) # 1, 4, 7, 10
seq(from = 1, to = 11, by = 3)
# Note that 11 is not included
```

Useful to generate uniform grids for evaluation:

```
seq(0, 1, 0.01)  # Same as (0:100)/100
```

Sequence from/to a number with step size and/or length:

```
seq(1, 2, length.out = 26)
# 1st argument = 'from', 2nd = 'to'
# Same as seq(1, 2, by = 0.04)
seq(to = 6, by = -0.5, length.out = 10)
# 10.5 10.0 9.5 ...
```

Vector recycling

If two interacting vectors have different lengths, the shorter one is **recycled** (repeated until lengths match).

```
1:4 + c(10, 20, 30)
#> 11 22 33 14
#> Warning message: In 1:4 + c(10, 20, 30) :
#>  longer object length is not a multiple
#>  of shorter object length
```

Since `1:4` has length 4, `c(10, 20, 30)` was recycled to `c(10, 20, 30, 10)`.

Recycling is the default behaviour; it may save time, but requires awareness. Usually, a warning is printed. **Do not ignore console warnings!** Investigate warnings like errors.

Application of rep: creating panel data sets

id	period	wage
1001	2019	10.4
1001	2020	10.3
1001	2021	9.9
1001	2022	9.3
2001	2019	8.1
2001	2020	7.9
2001	2021	8.4
2001	2022	8.4
3001	2019	5.3
3001	2020	4.7
3001	2021	6.0
3001	2022	4.8

```
ids <- c(1001, 2001, 3001)
ps  <- c(2019, 2020,
         2021, 2022)
id     <- rep(ids, each = 4)
period <- rep(ps, 3)
```

Alternatively:

```
ids <- (1:3)*1000 + 1
ps  <- 2019:2022
```

Useful when there are too many units or periods.

Combining vectors

Vectors can be thought as sets with well-defined operations of set intersection, union, and difference.

```
x <- c(1, 4,  7,  8, 11, 11)
y <- c(3, 4, 12, 14, 15)
intersect(x, y) # 4
union(x, y)     # 1  4  7  8 11 3 12 14 15
setdiff(x, y)   # 1  7  8 11
setdiff(y, x)   # 3 12 14 15
```

NB: These operations discard duplicated observations. Use with care. The vector `x` still contains the duplicated 11.

Object names

Elements of vectors can have names (labels).

names is an **attribute:** extra tagged information that is ignored during calculations, but can be passed as ballast.

```
x <- 1:4
names(x) <- c("Adam", "Borys", "Cyryl", "Dymitr")
print(x)
#>   Adam   Borys   Cyryl Dymitr
#>      1       2       3      4
attributes(x)
#> $names
#> "Adam"    "Borys"   "Cyryl"   "Dymitr"
attr(x, "names")
#> "Adam"    "Borys"   "Cyryl"   "Dymitr"
```

Deleting names

Attributes in general can be recovered later or used to enhance readability in the console. One can delete attrbitues, too.

Drop names by removing the attribute or calling a function that deletes names:

```r
names(x) <- NULL
```

Or (same result):

```r
x <- unname(x)
```

Object length

Calculate the number of elements in any object:

```
x <- seq(0, 1, 0.01)
l <- length(x)
l  # 101
```

Vectors can have length 0. What if we compute the set difference between a small and a large vector?

```
a <- 1:3
b <- 1:4
d <- setdiff(a, b)
d
#> integer(0)
length(d)  # 0
```

Chaining operations

Multiple computations can be done in the same line:

```
l <- length(seq(0, 1, 0.01))
# is the same as
x <- seq(0, 1, 0.01)
l <- length(x)
# But keep the number of objects manageable
```

Whenever a '<-' assignment is made, it returns the assigned object. New objects can be created anywhere:

```
x <- seq(a <- 0, b <- 1, d <- 0.01)
print(c(a, b, d))
#> 0.00 1.00 0.01  # Now a, b, and d exist!
```

However, this makes the code slightly less readable.

Matrices

- Matrices are vectors stacked together (column by column)
- Matrices have two dimensions and have respective functions to visualise them
- Can be created by column (default) or by row

Matrix from a vector

Recall the discussion that any array can be wrapped in a 1D vector. The opposite is possible:

```
a <- 1:12
m <- matrix(a, nrow = 3, ncol = 4)
m
#>   1    4    7   10
#>   2    5    8   11
#>   3    6    9   12
```

Since the vector length is known, the second dimension can be omitted:

```
matrix(a, nrow = 3)  # Same
matrix(a, ncol = 4)  # Same
```

Matrix by row and transpose

Matrices can be written like text (left to right, then top to bottom). Use the `byrow` argument:

```
m <- matrix(1:6, nrow = 2, byrow = TRUE)
m
#>   1     2     3
#>   4     5     6
```

Transpose a matrix with the `t()` function:

```
t(m)
#>   1     4
#>   2     5
#>   3     6
```

This is why ‘c’ and ‘t’ are bad object names (see Slide 78).

Care with requested matrix dimensions

Since R recycles shorter vectors until all vector lengths match, make sure that the length of the matrix vector is divisible by the requested number of rows and columns.

```
matrix(1:20, ncol = 4)  # Works normally
```

```
matrix(1:20, ncol = 6)  # 20 is not divisible by 6
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    1    5    9   13   17    1
#> [2,]    2    6   10   14   18    2
#> [3,]    3    7   11   15   19    3
#> [4,]    4    8   12   16   20    4

#> Warning message: In base::matrix(...) : data
#> length [20] is not a sub-multiple or multiple
#> of the number of columns [6]
```

Getting matrix dimensions

Get the number or rows and columns of a matrix:

```
a <- 1:12
m <- matrix(a, nrow = 3, byrow = TRUE)
nrow(m)  # 3
ncol(m)  # 4
dim(m)   # c(3, 4): rows, then columns
```

However, the *length* of a matrix is the number of elements in the vector that makes it up. Recover the original vector:

```
length(m)  # 12 -- do not confuse with nrow!
v <- as.vector(m)
v
#> 1 5 9 2 6 10 3 7 11 4 8 12
```

Matrix ≠ vector

Although matrices are wrapped long vectors, matrices have non-empty dimensions, whilst vectors are dimensionless (not a $k \times 1$ matrix; no such default).

```
v <- 1:3
m <- matrix(v, ncol = 1)
print(v)
print(m)  # Formatting difference
dim(v)    # NULL
dim(m)    # c(3, 1)
t(v)      # Can still be transposed
t(t(v))   # Becomes a 3x1 matrix; esoteric
```

Some functions really expect $k \times 1$ matrix inputs – read the function documentation, convert if necessary.

One function to measure the length

Luckily, there are capitalised versions, `NROW()` and `NCOL()`, that would return the correct dimension for both matrices and vectors.

```
v <- 1:3
m <- matrix(1:6, ncol = 3, nrow = 2)
c(NROW(v), NCOL(v))  # 3 rows, 1 column
c(NROW(m), NCOL(m))  # 2 rows, 3 columns
```

Matrix dimension names

Matrices can have row names and column names.

```
a <- 1:12
m <- matrix(a, nrow = 3, byrow = TRUE)
rownames(m) <- c("Andor", "Botond", "Csaba")
colnames(m) <- c("Jan", "Apr", "Jul", "Oct")
m
#>        Jan Apr Jul Oct
#> Andor    1   2   3   4
#> Botond   5   6   7   8
#> Csaba    9  10  11  12
```

NB: `names()` ≠ `rownames()`! Check this:

```
names(m)
#> NULL
```

Joining (connecting) matrices

- Matrices with equal # of rows ($n \times a$, $n \times b$) can be bound horizontally (by column) into a $n \times (a + b)$ matrix
- Matrices with equal # of columns ($k \times m$, $l \times m$) can be bound vertically (by column) into a $(k + l) \times m$ matrix

The `cbind()` and `rbind()` functions do exactly this:

```r
a1 <- matrix(1:12,     nrow = 3, ncol = 4)
a2 <- matrix(1:8+100, nrow = 2, ncol = 4)
a3 <- matrix(1:6+200, nrow = 3, ncol = 2)
rbind(a1, a2)  # Result: 5x4
cbind(a1, a3)  # Result: 3x6
cbind(a1, a2)
#> Error: number of rows of matrices must match
```

Binding with names

A vector can be connected into a matrix without any conversion as long as their lengths are equal and they match `nrow()` of the matrix.

The arguments or `rbind()` and `cbind()` can be named – the supplied names become row / column names.

```
a <- matrix(1:12, nrow = 3, ncol = 4)
b <- cbind(1:3, 4:6, 7:9, 10:12) #  Same
cbind(a, 101:103)
cbind(a, Extra = 101:103)
```

Some rows / columns may remain unnamed – their names are set to `""` (empty string) – more details in Session 3.

A couple of matrix functions

If `m` is a matrix, use:

- `rowSums(m)`, `colSums(m)` to compute the sum in rows / columns, respectively
- `rowMeans(m)`, `colMeans(m)` to compute the means in rows / columns

We shall learn how to compute arbitrary quantities from a matrix (e. g. standard deviation by column, % of missing variables per observation) in Session 4.

Any questions on vectors, matrices, and dimension names?

Getting help

Sources of help

- Every user-level function is documented
  - Type `?myFunctionName` to open the help page
  - The documentation quality depends on the function author
- Type just `function` in the console (without any brackets) to read the source code of the function
- Google search and StackExchange is invaluable
- Many tutorials online
  - Various resources offer different solutions with different packages
  - If one does not work, try another; compare and benchmark

Documentation

- Base R: `https://cran.r-project.org/manuals.html`
- Every package: a CRAN page with source code and documentation (at least rudimentary)
  - Every function has an offline help page
  - All default argument values, definitions, description, example
- Some packages: vignettes highlighting the most popular features and giving a deeper overview
  - Offline vignettes: `vignette(all = TRUE)`
- R Journal, Journal of Statistical Software...

Getting function help

To get help about the `log()` function, type in the console:

```
?log           # Help on logarithms
help("log")    # Same
```

Some topics are dedicated to groups of functions:

```
?Distributions   # Help on built-in distributions
?InternalMethods # Most basic low-level commands
```

Keywords (`if`, `else`, `while`, `function`, `for`, `in`, `break`, …) and non-text characters must be put in backticks:

```
?for     # Does not work! Press Esc to cancel
?`for`   # Works -- help on loops
?`:`     # Help on the colon for sequences
```

Getting topic help

To search for a topic by keywords, use two question marks.

```
??quantile  # Evrythg that mentions 'quantile'
?quantile   # Help on the quantile() function
```

Use backticks or `help.search()` to search phrases with spaces:

```
??`maximum likelihood`
help.search("logistic regression")
```

Look up all package functions:

```
??stats
??survival
```

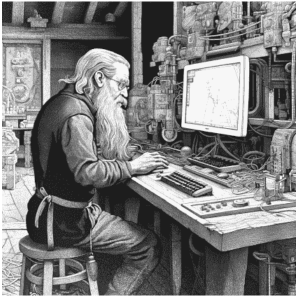
Getting package vignettes

Some packages are accompanied by beautiful small manuals, or **vignettes**.

```
install.packages("glmnet")
vignette(all = TRUE)  # Show all vignettes
vignette("glmnet")    # Open vignette
```

Using digital assistants

- Chat bots and neural networks can hallucinate a reasonable answer or a viable first step
- The accuracy of digital assistants depends on the task complexity
- **Always perform rigorous tests of the tool-generated solutions before incorporating them**
  - Try changing the inputs



Signor Chiatti Geppetto (ChatGPT) at work

‘A computer can never be held accountable, therefore, a computer must never make a management decision.’ (1979, IBM)

Can ChatGPT help with R?

- I have a data set `mtcars`
- I want to estimate the linear relationship between `mpg` and regressors (`cyl`, `disp`, `hp`, `gear`, `carb`)
- Write a function to estimate this linear regression in R

```
Sure, I can help you with that! You can use the
built-in `lm()` function in R to perform <...>
```

```r
# Function to estimate linear regression
perform_linear_regression <- function() {
  # Load the dataset
  data("mtcars")
  # Define the formula for the linear regression
  formula <- mpg ~ cyl + disp + hp + gear + carb
  # Fit the linear regression model
  regression_model <- lm(formula, data = mtcars)
  # Return the regression model
  return(regression_model)
}
# Call the function to perform linear regression
regression_result <- perform_linear_regression()
# Print the summary of the regression model
summary(regression_result)
```

- Correct result
- Too wordy
- Many redundancies
- Wrong standard errors

Stress-testing ChatGPT: robust SE

'How do I compute robust standard errors for the estimates?'

```
To compute robust standard errors for the parameter
estimates in R, you can use the sandwich package,
which provides <...>
```

```
library(sandwich)
perform_robust_regression <- function() {
  # <..same things as earlier..>
  robust_se <- sqrt(diag(vcovHC(regression_model)))
  result <- data.frame(Coefficients =
   ↪  coef(regression_model), Robust_SE = robust_se)
  robust_regression_result <-
   ↪  perform_robust_regression()
  return(result)
}

robust_regression_result <-
↪  perform_robust_regression()
print(robust_regression_result)
```

- Correct result
- Right package
- Adjusts the display style
- Human-readable explanation

Stress-testing ChatGPT: bootstrapping

‘How can I bootstrap the coefficients of this model to get bootstrap confidence intervals?’

```
Bootstrapping is a resampling technique that can be
used to estimate <...>
```

```
# Load the required packages
library(sandwich)
library(boot)
perform_robust_regression <- function() {
  # <..same things as earlier..>
}
perform_bootstrap <- function(data, num_bootstraps =
↪  1000) {
  boot_results <- boot(data,
  ↪  perform_robust_regression, R = num_bootstraps)
  boot.ci(boot_results, type = "bca")
}
data("mtcars")
# Call the function to perform bootstrap
bootstrap_result <- perform_bootstrap(mtcars)
```

```
Error in statistic(data,
 ↪  original, ...) : unused
 ↪  argument (original)
```

- Produces a non-trivial error
- There is no ‘original’ argument in the code

Stress-testing ChatGPT: fixing the error

‘This code does not work. I am getting this: `"Error in statistic(data, original, ...) : unused argument (original)"`. Fix the error.’

`I apologize for the confusion. It seems I made a mistake in the code. Here's the corrected version that should work:`

```
perform_robust_regression <- function(data, indices)
↪ {
  subset_data <- data[indices, ]
  # <...>
}

bootstrap_result <- perform_bootstrap(mtcars)

Error in t.star[r, ] <- res[[r]] :
  incorrect number of subscripts on matrix
```

- Still does not work
- Still produces a non-trivial error

Stress-testing ChatGPT: second error

‘This code does not work. I am getting this error: <...>’

`I apologize for the confusion. It seems there's still an issue in the code. The "incorrect number of subscripts on matrix" error is usually caused by incorrect indexing when saving the bootstrap results.`

```
perform_robust_regression <- function(data, indices)
↪  {
  subset_data <- data[indices, ]
  formula <- mpg ~ cyl + disp + hp + gear + carb
  regression_model <- lm(formula, data = subset_data)
  robust_se <- sqrt(diag(vcovHC(regression_model)))
  result <- coef(regression_model)
  return(result)
}

perform_bootstrap <- function(data, num_bootstraps =
↪  1000) {
  <...>
}

bootstrap_result <- perform_bootstrap(mtcars)
```

- No error
- Very slow (3.5 s)
- Computes robust SEs and never uses them

Speeding ChatGPT code up

‘This is too slow. Keep the bootstrap, but remove the code related to robust standard errors.’

```
Certainly, <...> In this version of the code, I
removed the calculation of robust standard errors
<...>
```

```r
# Load the required package
library(boot)
# Function to estimate linear regression coefficients
perform_regression <- function(data, indices) {
  subset_data <- data[indices, ]
  formula <- mpg ~ cyl + disp + hp + gear + carb
  regression_model <- lm(formula, data = subset_data)
  result <- coef(regression_model)
  return(result)
}
perform_bootstrap <- function(data, num_bootstraps =
↪  1000) {
  # <..same as before..>
}
# <..same as before..>
print(bootstrap_result)
```

- No error
- Much faster (1.5 s)
- However, it displays the confidence interval only for the intercept

Questioning ChatGPT competence

'For which coefficient is this confidence interval?'

```
I apologize <...> The code <...> calculates the
bootstrap confidence intervals for the first
coefficient <...> which corresponds to the intercept.
```

(Correct. But it gave incorrect code that returned **`NULL`***.)*

'Why am I getting NULL? I want confidence intervals for the 5 variables in the model.'

```
I apologize <...> Here's the corrected version <...>
```

(The new code still returns **`NULL`***.)*

Attempting to lead the blind

'No, I am still getting `NULL`. Maybe you could use `` `sapply' `` and call the `` `boot.ci' `` function with every index corresponding to a position in the coefficient vector?'

```
In this version of the code, we use sapply() to
iterate through the indices corresponding to each
coefficient <...>
```

```
Error in object$coefficients : $ operator is invalid
 ↪  for atomic vectors
```

'I am getting <error text>.'

```
I apologize for the continued issues <...>
```

(Still returns non-functioning code.)

Continuing to lead the blind

'Okay, if this is not working, I need simple R code that uses LASSO to pick 3 out of 5 of these variables for the same model.'

```
Certainly! You can use the glmnet package <...>
```

(Returns working code, but uses a weird and very restrictive parameter grid and selects 6 variables instead of 3.)

'There are 6 variables in the output. I need only 3. Modify the code that chooses only the 3 most relevant predictors.'

(Returns the code with 3 variables largest in absolute value.)

Giving up leading the blind

‘Modify the code so that it returns the variables that first appear on the LASSO selection path, not the ones largest in absolute value.’

(Goes back to the initial version.)

‘Wrong. It selects 6 variables. In addition, the custom lambda sequence is weird. Use the default one.’

(Returns a completely wrong result.)

What we expected – LM with robust SE

```r
library(sandwich)
library(lmtest)
f <- mpg ~ cyl + disp + hp + gear + carb
mod <- lm(f, data = mtcars)
print(coeftest(mod, vcov. = vcovHC))

#> t test of coefficients:
#>             Estimate     SE       t        p
#> (Intercept) 21.1076  8.9124  2.3684  0.02558 *
#> cyl         -0.2571  0.8079 -0.3183  0.75283
#> disp        -0.0186  0.0090 -2.0696  0.04856 *
#> hp          -0.0063  0.0208 -0.3034  0.76399
#> gear         2.6974  1.6915  1.5947  0.12286
#> carb        -1.4801  0.7158 -2.0676  0.04876 *
```

What we expected – bootstrap CIs

```r
library(boot)
getCoef <- function(d, i) coef(lm(f, data = d[i, ]))
set.seed(1)
b <- boot(mtcars, getCoef, R = 999)
bci <- sapply(1:6, function(i) boot.ci(b, type =
 ↪  "bca", index = i)$bca[4:5])

colnames(bci) <- names(coef(mod))
rownames(bci) <- c("Lower", "Upper")
print(bci, 3)

#>    (Intrcpt)   cyl     disp      hp  gear   carb
#> Lower   6.0 -1.91 -0.03822 -0.0491 0.265 -2.785
#> Upper  37.8  1.18  0.00164  0.0329 5.812 -0.122
```

What we expected – variable selection

```r
library(glmnet)  # 3 lines for estimation
f   <- mpg ~ cyl + disp + hp + gear + carb
m   <- model.frame(f, data = mtcars)
mod <- glmnet(x = m[, -1], y = m[, 1])

# And 4 lines for extracting 3 variables
b <- as.matrix(coef(mod))[-1, ]
gt3 <- apply(b, 2, function(x) sum(x!=0) >= 3)
bs <- b[, which(gt3)[1]]
rownames(b)[bs != 0]

#> "cyl"  "disp" "hp"
```

Do not shoehorn yourself

A: I have created an example for Session 4. It shows how a solution that takes 43 s to run can be transformed into a more memory-efficient one that takes less 1 s.

B: Yeah? I have been using R since 2014, and I do everything with `data.table`.

A: I do not see any obvious solution. Please teach me.

(45 minutes later after numerous errors in 2 code lines.)

B: Okay, there still should be some way to invoke it with `data.table` – let's ask ChatGPT.

Result: *A* and *B* are none the wiser.

So where do I learn?

ChatGPT 3.5 cannot go beyond the simplest examples.

- Incorrect answers to research questions, even after guiding rectifications (maybe v4 is better?)
  - However, some examples would take hours for new users without any assistance – **use it for hints, not full solutions**
- If the machine is learning, the user is not

Reliable sources:

- Examples from the built-in help (at the bottom)
- Blog posts with hands-on examples written by experienced R users (Rob J. Hyndman, Achim Zeileis, Peter Dalgaard, Sebastian Krantz, Bruno Rodrigues etc.)
- Researchers' pages – many publish online appendices
- Books with examples from the field

Further reading

- Variable naming style in R
- Test: big-data tool or Pokémon name?
- International Space Station switches to Linux in 2013

Thank you for your attention!