Intro to Programming with R for Political Scientists

Session 2: Base R and Tidyverse Basics

Markus Freitag

Geschwister Scholl Institute of Political Science, LMU

2021-06-27

Overview

Trivia

- R was designed in 1993 by Ross Ihaka and Robert Gentleman
- Builds upon the S programming language by John Chambers
  - Named R as a play on S and bc of the first names of the authors
- There are 0 packages available on **CRAN** as of 2021-06-27.
- **R-Studio** $\neq$ **R-Core Team**; the former is a mix of a for-profit and a non-profit company; highly committed to produce free & open-source products; has some business solutions

Image source and more R-History trivia

Workflow- You forked and cloned the course repo.

- Navigate to `Session Scripts > Session 2` and open `Session_2_script.R`.
- You will see a pre-formatted Script containing some useful information on comments and formatting.
- The script is otherwise empty. Fill it with the stuff I will discuss on the slides.
- Make comments for yourself. Learn as you write. Explore. Stage, commit and push.
- Hopefully not you →
- If you have a second monitor, great! If not, split your screen.

CalculatoR

CalculatoR

```r
7 + 5 # [n] stands for the nth element printed to the console.
```

```
## [1] 12
```

```r
4 * 5 + 2 / 3^3 # Multiplication and division first, then addition and subtraction
```

```
## [1] 20.07407
```

```r
# Modulo Operators:

10 %/% 3 # Integer division
```

```
## [1] 3
```

```r
10 %% 3 # Remainder ("Rest")
```

```
## [1] 1
```

CalculatoR

```r
# Relational and logical operators

3 < 4
```

```
## [1] TRUE
```

```r
2 == 1 & 4 > 2 # == "equal to"; & "element
```

```
## [1] FALSE
```

```r
2 == 1 | 4 > 2 # | "element wise logical or
```

```
## [1] TRUE
```

```r
3 != 4 # != "not equal"
```

```
## [1] TRUE
```

```r
7 %in% 300:500 # %in% can be used to evaluc
```

```
## [1] FALSE
```

```r
# Take care about the order of precedence..
```

CalculatoR

```r
# Floating Points

0.1 + 0.2
```

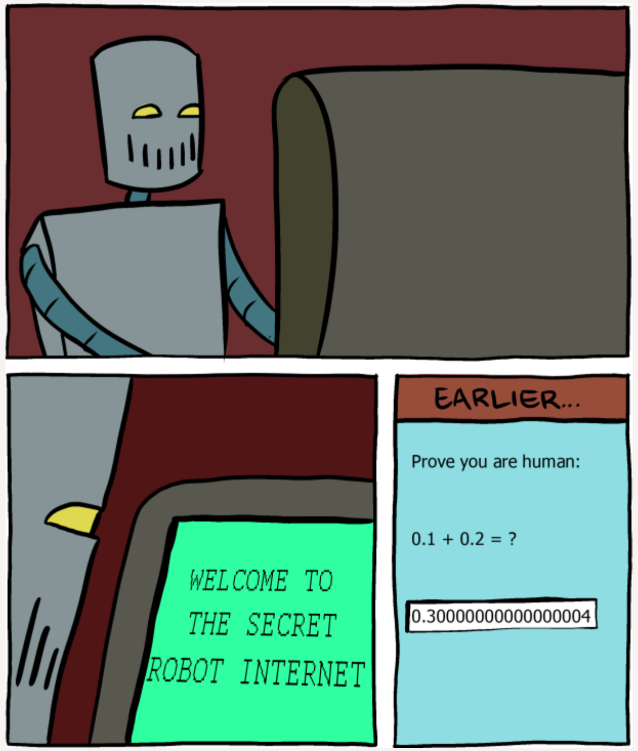
```
## [1] 0.3
```

```r
0.1 + 0.2 == 0.3
```

```
## [1] FALSE
```

Why?!

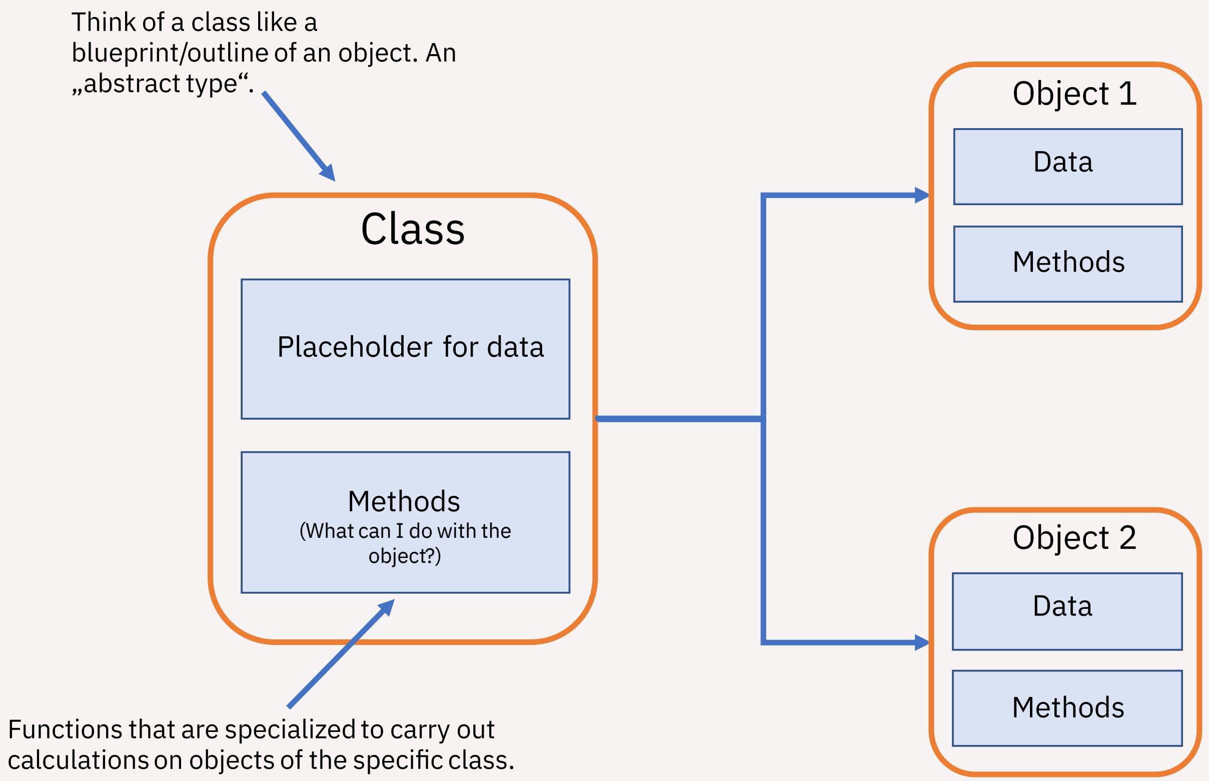
> **Because internally, computers use a format (binary floating-point) that cannot accurately represent a number like 0.1, 0.2 or 0.3 at all.**



A Primer on OOP ("Object Oriented Programming")

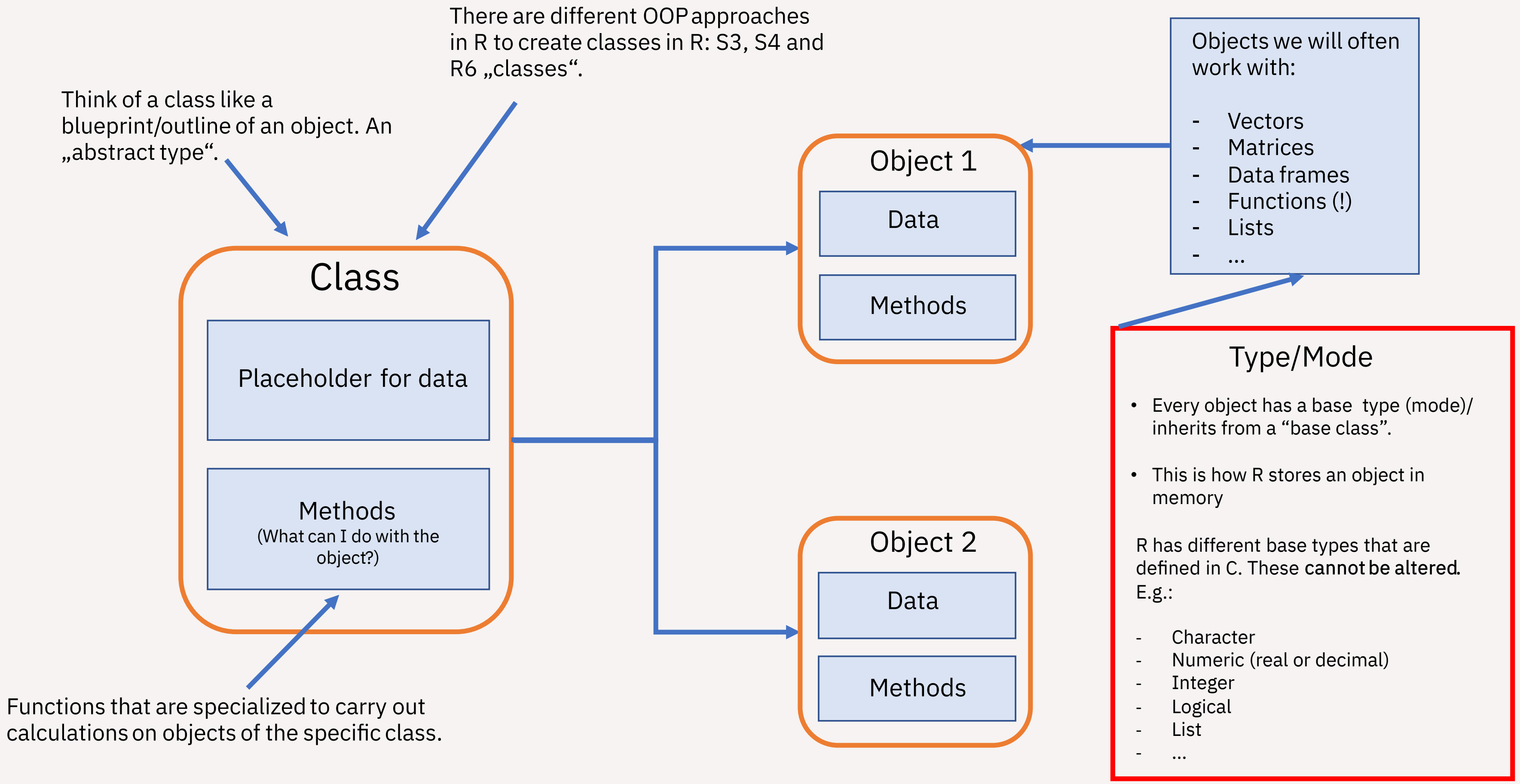
Object Oriented Programming

> Everything is an object and everything has a name.


Think of a class like a blueprint/outline of an object. An „abstract type“.
Class
Placeholder for data
Methods
(What can I do with the object?)
Functions that are specialized to carry out calculations on objects of the specific class.
Object 1
Data
Methods
Object 2
Data
Methods

Object Oriented Programming

> Everything is an object and everything has a name.



There are different OOP approaches in R to create classes in R: S3, S4 and R6 „classes“.

Think of a class like a blueprint/outline of an object. An „abstract type“.

Functions that are specialized to carry out calculations on objects of the specific class.

Objects we will often work with:

- Vectors
- Matrices
- Data frames
- Functions (!)
- Lists
- ...

Type/Mode

- Every object has a base type (mode)/ inherits from a "base class".
- This is how R stores an object in memory

R has different base types that are defined in C. These **cannot be altered.** E.g.:

- Character
- Numeric (real or decimal)
- Integer
- Logical
- List
- ...

Functions

- **Functions** are objects; we will discuss them in more detail in Session x.
- For now, just think of them in the usual mathematical sense, where we pass some argument and get back a value. E.g. $f(x) = \frac{2x+3}{\sqrt{3}}$.
- In R, one or multiple arguments get passed to the function body (where the function is defined) and you get back some results.
- For instance, to define the above function and call it, we specify the following:

```r
f <- function(x) {
  (2 * x + 3) / sqrt(3)
}

f(3)
```

```
## [1] 5.196152
```

Functions

- There are many functions in R, some are written by users and scientists and put into some package, some are built-in functions of "base" R.[1]
- A really helpful built-in function - fin the literal sense - is the... `help()` function.
- Gives you more information about the usage and arguments of a built-in/user-written function.

We can also call `help()` to get help about help:

```r
# help(help)
# Or for short:
?help
```

- But let's not get ahead of ourselves... what did this arrow (`<-`) thing do?

[1] **Fine Point:** The arithmetic, logical and relational operators we met are actually also function calls.

Making Objects: Assignment (Operators)

- You can use `<-` or `=` for assignment
- For instance,

```
a <- 3 # Or a = 3; under the hood, assignme
```

assigns the name `x` to an object of type/mode numeric holding the value 3. I.e. binds an object to a name.

Simplification:

> creates an object named `a`, containing the value `3`.

-Using

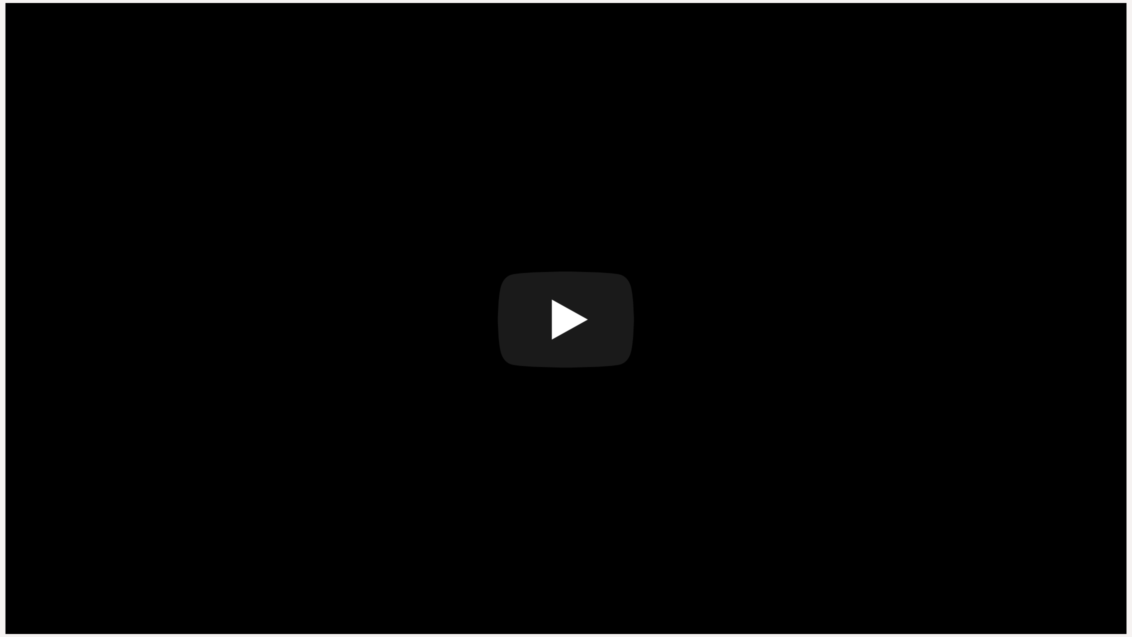
```
class(a)
mode(a)
```

gives you information about the class/type of the object. `class()` gives the class of the object from an OOP POV, `typeof()` (or `mode()`)the base type.

- In this case, this is not very interesting as we created an **atomic** numeric vector.
- What is the class/type of `b <- 3 > 2` and `c <- "string"`?

Making Objects: Assignment (Operators)

- Using `=` is legal as per the man, the myth, the legend Ross Ihaka himself:



- But there are also some sensible arguments to stick mostly to `<-` in casual settings (e.g. readability: easier to discriminate from function arguments)
- Bottom line: **be consistent** .

Naming Conventions

- For readability, we want names of/bound to objects to be meaningful.
- Pretty easy to do when working with "real" data; laziness pretty much the only obstacle ;)
- There are several naming conventions. I like `snake_case` and `camelCase` the most. We will use snake cases in this course.
- **Be consistent** .
- There are some **"forbidden"/"reserved" words** that cannot be assigned as names. E.g. `NA` (logical constant indicating missing values), `if`, `else`, `break`.

Workspace/Environment

- In contrast to Stata, R can hold multiple objects at a time.
- This is very convenient; you can copy, modify etc. R also copies **ALOT** internally (one reason why its sort of a slow language).
- The global environment is the interactive workspace you usually work with.
- In R-Studio you can inspect some objects by clicking on them (equivalent to calling `View()`)
- We won't go into the details of environments but see **here** for an advanced treatment.
- This will get more intuitive soon (e.g. when we take on different data structures on the next few slided)

Vectors

- Vectors are the most fundamental data structure in R.
- As vectors in R have to be of the same type (e.g. numeric), they are often called **atomic** vectors
- ***Technical Fine Point:*** Vectors have attributes, importantly dimension and class. With the dimension attribute, vectors can become arrays and matrices. With the class attribute, we can built an S3 object on top of the base type (see e.g. factor type).

We can build longer vectors by concenating shorter ones using the `c()` function:

```r
chr_var <- c("a", "b", "c", "d") # A 4-element atomic vector of type character
```

Indexing:

```r
chr_var[3] # Returns the third element of object "chr_var".
chr_var[1:3] # Returns the 1st three.
```

Lists

- Objects of type list are highly versatile. They are vectors but more "generic".
- I.e. elements of a list can have different types.[2]

For example:

```r
list_a <- list(
  5:7,
  "string",
  c(TRUE, TRUE),
  c(1.23, 4.20)
)
```

constructs a list using the function `list()`.

[2] **Fine Point:** A list does not store objects of different types, it references to them.

Lists

- Indexing:

```r
list_a[3] # This returns a list with elemer
```

```
## [[1]]
## [1] TRUE TRUE
```

```r
list_a[[4]] # We need double brackets to ir
```

```
## [1] 1.23 4.20
```

```r
# Index within a list element:
list_a[[4]][2]
```

```
## [1] 4.2
```

```r
# For named lists, we can index with $:
names(list_a) <- c("a", "b", "c", "d") # We
list_a$b
```

```
## [1] "string"
```

```r
list_a$d[1]
```

```
## [1] 1.23
```

```r
# Adding a new object-reference, i.e. an el

list_a$e <- c("R", "is fun")
```

Factors

- To represent categorical variables, we often use factor objects in R (e.g. makes it easier to get counts of all categories).
- They are build on top of integer vectors and come with a levels attribute.

```r
backgrounds_char <- c("none", "Stata", "Stata", "Stata", "R") # Student's prog. backgrounds
backgrounds_fac <- factor(backgrounds_char, levels = c("none", "Stata", "R")) # Or:
# backgrounds_fac <- factor(c("none", "Stata", "Stata", "Stata", "R"))
```

```r
class(backgrounds_fac)
```

```
## [1] "factor"
```

```r
typeof(backgrounds_fac)
```

```
## [1] "integer"
```

```r
table(backgrounds_fac)
```

```
## backgrounds_fac
##  none Stata     R
##     1     3     1
```

Data Frames

- Data frames are special lists: lists of evenly sized vectors.
- You likely already have a grasp for their structure from Stata or other software; they are crucial for data analysis --> the rectangle we love.

```r
set.seed(666)
df_langs <- data.frame(
  background = factor(c(
    "Python",
    "Stata", "Stata", "Stata",
    "R"
  )),
  skill = rnorm(5)
)
# View(df_langs)
head(df_langs) # print first few lines of an object (6 by
# Indexing
df_langs[2, 2] # Second row of second column
df_langs$background[1]
df_langs[df_langs$skill < 0,] # Combined with our operato
```

```
##   background      skill
## 1     Python  0.7533110
## 2      Stata  2.0143547
## 3      Stata -0.3551345
## 4      Stata  2.0281678
## 5          R -2.2168745

## [1] 2.014355

## [1] Python
## Levels: Python R Stata

##   background      skill
## 3      Stata -0.3551345
## 5          R -2.2168745
```

Data Frames

- So we have an n (num. of obs.) $\times$ k (num. of vars) matrix with observations $i \in \{1, \ldots, n\}$ and variables $j \in \{1, \ldots, k\}$.

```
dim(df_langs)
```

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

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,k} \end{bmatrix}$$

- `nrow(df_langs)` = `## [1] 5` ; `ncol(df_langs)` = `## [1] 2`
- `df_langs[1,2]`: $(i = 1)$ and $(j = 2)$

Data Frames

- When passing columns of a data frame (or lists etc.) we need need to take care that the named references to them "live" inside the data frame and not in the global environment.

```
# Throws an error:
lm(skill ~ background) # [3]; the lm object is of type list
```

```
m1 <- lm(df_langs$skill ~ df_langs$background) # Or: lm(skill ~ background, data = df_langs)
summary(m1) # generic function used to produce result summaries of the results of various model fitting functions
```

```
##
## Call:
## lm(formula = df_langs$skill ~ df_langs$background)
##
## Residuals:
##          1          2          3          4          5
##  1.986e-16  7.852e-01 -1.584e+00  7.990e-01 -3.827e-16
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
## (Intercept)                0.7533     1.3720   0.549    0.638
## df_langs$backgroundR      -2.9702     1.9403  -1.531    0.265
## df_langs$backgroundStata   0.4758     1.5843   0.300    0.792
##
## Residual standard error: 1.372 on 2 degrees of freedom
## Multiple R-squared:  0.7056,    Adjusted R-squared:  0.4113
## F-statistic: 2.397 on 2 and 2 DF,  p-value: 0.2944
```

Data Frames

- This is the time to also plug Vincent Arel-Bundocks' nice package **modelsummary**.
- Makes perfectly formated regression tables/coef. plots for HTML/MD/LaTeX/Word/etc. a one-liner.
- Also has nice data summary functions which we will use next session.

Data Frames

Result:

	Model 1
(Intercept)	0.753 (1.372)
Python (reference)	NA
R	-2.970 (1.940)
Stata	0.476 (1.584)
Num.Obs.	5
R2	0.706
R2 Adj.	0.411
AIC	20.8
BIC	19.2
Log.Lik.	-6.385
F	2.397

Type Conversion

In most cases, we can also convert an object to another type. Sometimes this has side-effects.

```r
df_langs_m <- as.matrix(df_langs) # convert data.
print(df_langs_m)
```

```
##      background skill
## [1,] "Python"   " 0.7533110"
## [2,] "Stata"    " 2.0143547"
## [3,] "Stata"    "-0.3551345"
## [4,] "Stata"    " 2.0281678"
## [5,] "R"        "-2.2168745"
```

```r
typeof(df_langs_m[2, 1])
```

```
## [1] "character"
```

- As matrices are atomic, an implicit coercion happened.[4]
- If stuff cannot get converted, you get NAs.

[4] **Fine Point:** Coercion order is character > double > integer > logical.

Loading/Installing Packages

Classic:

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

If you are lazy (...for CRAN packages; recommended only for smaller projects/problem sets):

```r
install.packages("pacman")

pacman::p_load(
  tidyverse,
  patchwork,
  rio,
  data.table
)
```

Loading/Installing Packages

- As mentioned, R comes with 0 packages on CRAN up to date.
  - This is both a blessing and a **curse**.
  - Many useful packages also live only on Github.
  - Escaping package/library dependency hell is a bit of a pain in R (much easier in Python and Julia).[5]
- To take it to another level, **Dockers** expand reproducibility by providing a full runtime environment (and they work well with R).

[5] **NOTE:** The **packrat** package is one option. **renv** is much better. You should definitely use it if you work on a larger project.

Loading/Installing Packages: Namespace Conflicts

- As we assign names to objects and there are as many packages (and many more functions) as a native english speaker has in his **vocabulary**, we naturally get some overlap.
- Diving deep into environments, package **namespaces**, etc. can take you down a **rabbit hole**.
- We will keep this very surface-level.
- R warns you about namespace conflicts when loading some package, e.g. by telling you that some objects are `masked from package: base`.
- The package loaded last "wins", i.e. function `a` of package 2 masks function `a` of package 1.
- If you only need the mask function a few times, use `package1::functiona()`. Else, overwrite it `functiona <- package1::functiona()`.

The Tidyverse

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

- Developed by **Hadley Wickham**, supported by RStudio
  - Packages that are easy to learn, that have a similar syntax, and are convenient
- There are alternatives... base R is still stable and useful; data.table is fast; **collapse**(brand new) is even faster
- In my opinion, the tidyverse is a bit verbose/wordy. Alot of functions doing one particular thing.
- You will always end up googling alot. But that's something you will do alot anyways (I do too!).

Next up: Data Wrangling