



Environment, Packages, Functions, Import/Export

Enrico Toffalini

Install packages... and more

Most of the data analysis tasks you will perform won't come from base R: you most often load additional **packages** for specialized functions

Install a package from CRAN

Simultaneously install multiple packages

```
install.packages("effsize")
```

```
install.packages( c("effsize","psych","ggplot2") )
```

For development or personal use, you may occasionally install packages from outside CRAN, such as from GitHub:

```
devtools::install_github("FilippoGambarota/filor")  
devtools::install_github("EnricoToffalini/toffee")
```

After installing, you need to load the packages using function **library**:

```
library(effsize)  
library(ggplot2)
```

Install packages... and more

call functions

After loading a package, its functions are directly callable throughout the R session:

```
library(psych)

fisherz(rho=0.5) # use a function from the "psych" package

[1] 0.5493061
```

you may directly call any function from any installed package, even without loading it, using “::”; this is especially useful when there is a risk of functions with conflicting names, or if you just don’t want to load an entire package for using a single function:

```
psych::fisherz(rho=0.5)

[1] 0.5493061
```

Some R packages that you will or may need in the future (1/3)

Package	Used for what	Examples of functions
<code>base</code> (base R)	Basic functions	<code>sum</code> , <code>mean</code> , <code>sqrt</code> , <code>abs</code> , <code>c</code> , <code>data.frame</code> , <code>summary</code> , <code>scale</code> , <code>plot</code> , <code>+</code> , <code>-</code>
<code>stats</code> (base R)	Basic statistical calculations and functions	<code>sd</code> , <code>cor</code> , <code>cor.test</code> , <code>t.test</code> , <code>lm</code> , <code>glm</code> , <code>AIC</code> , <code>rnorm</code> , <code>rbinom</code>
<code>graphics</code> (base R)	Basic statistical calculations and functions	<code>plot</code> , <code>boxplot</code> , <code>hist</code> , <code>barplot</code>

(You may actually use these “base” packages very often without even realizing that they are packages)

Some R packages that you will or may need in the future (2/3)

Package	Used for what	Examples of functions
<code>lme4</code>	Fitting (generalized) (non-)linear mixed-effects models	<code>lmer</code> , <code>glmer</code> , <code>ranef</code>
<code>performance</code>	Useful tools for models	<code>check_collinearity</code> , <code>r2_nagelkerke</code> , <code>icc</code>
<code>effects</code>	Display effects for various statistical models	<code>allEffects</code> , <code>effect</code>
<code>emmeans</code>	Estimate marginal means for various models	<code>emmeans</code>
<code>effectsize</code>	Compute or convert different effect sizes	<code>cohens_d</code> , <code>hedges_g</code> , <code>cohens_f</code> , <code>d_to_r</code>

Some R packages that you will or may need in the future (3/3)

Package	Used for what	Examples of functions
<code>ggplot2</code>	Create beautiful plots using The Grammar of Graphics	<code>ggplot</code> , <code>geom_point</code> , <code>geom_line</code> , ...
<code>lavaan</code>	Structural Equation Models (SEM)	<code>sem</code> , <code>cfa</code>
<code>semTools</code>	Useful tools for SEMs	<code>compRelSEM</code> , <code>measEq.syntax</code>
<code>metafor</code>	Perform meta-analysis	<code>rma</code> , <code>rma.mv</code> , <code>forest</code> , <code>funnel</code> , <code>regtest</code>
<code>brms</code>	Fitting practically any Bayesian model via MCMC with STAN	<code>brm</code> , <code>set_prior</code>
<code>blavaan</code>	Fitting Bayesian SEMs	<code>bcfa</code> , <code>bsem</code>

Reproducibility tip: `sessionInfo()`

When sharing your full R project, also share info on your **R version, OS, and loaded package versions** for reproducibility

```
sessionInfo()
```

```
R version 4.5.1 (2025-06-13 ucrt)
Platform: x86_64-w64-mingw32/x64
Running under: Windows 11 x64 (build 22631)
```

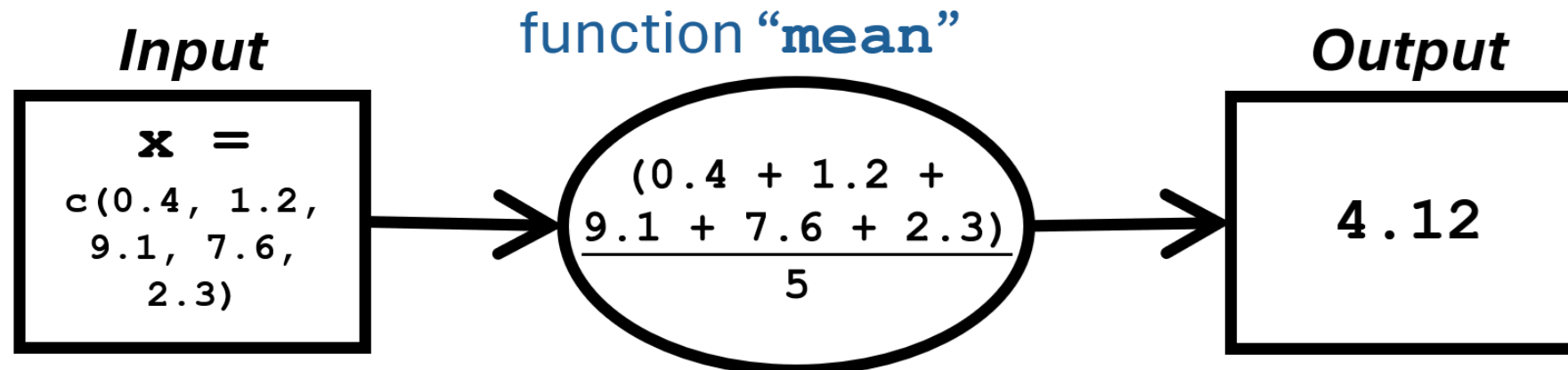
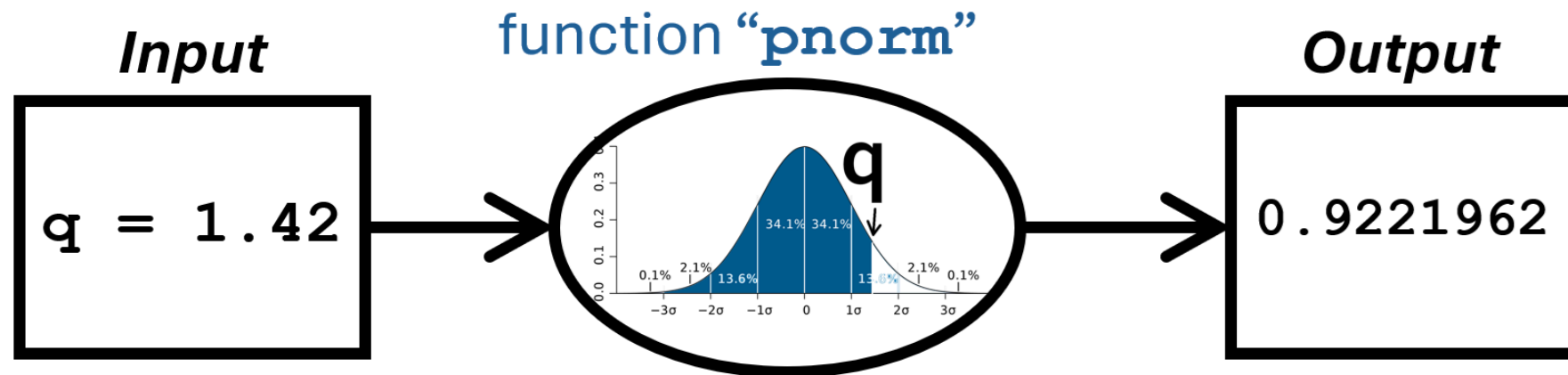
```
Matrix products: default
  LAPACK version 3.12.1
```

```
locale:
```

```
[1] LC_COLLATE=English_United Kingdom.utf8
[2] LC_CTYPE=English_United Kingdom.utf8
[3] LC_MONETARY=English_United Kingdom.utf8
[4] LC_NUMERIC=C
[5] LC_TIME=English_United Kingdom.utf8
```

Functions and arguments

Functions typically take some *input* parameters, known as *arguments*, process that, and yield some *output*/result(s)



Functions and arguments

arguments

- values or variables you pass to a function as input, or to control its behavior

for example, `seq()` generates a sequence of numbers; “from” and “to” are arguments: it will provide the integers between these two extremes:

```
seq(from = 3, to = 7)
```

```
[1] 3 4 5 6 7
```

`length.out` controls how many equally spaced numbers must be generated:

```
seq(from = 3, to = 7, length.out = 4)
```

```
[1] 3.000000 4.333333 5.666667 7.000000
```

alternatively, `by` defines the step size between numbers:

```
seq(from = 3, to = 7, by = 0.6)
```

```
[1] 3.0 3.6 4.2 4.8 5.4 6.0 6.6
```

Functions and arguments

arguments

- values or variables you pass to a function as input, or to control its behavior

`rnorm()` will generate “n” random numbers from a normal distribution with “mean” as the average and “sd” as the standard deviation:

```
rnorm(n = 5, mean = 100, sd = 15)
```

```
[1] 97.47999 119.33902 99.25235 88.08878 91.06610
```

Positional matching - arguments names may be omitted if placed in the correct order

```
rnorm(5, 100, 15)
```

```
[1] 105.85073 71.08416 102.59129 118.26543 106.36570
```

Functions and arguments

Default arguments - a function *might* still work even if some arguments are omitted, if it can use its own *default values* (in this case “`mean=0, sd=1`”)

```
rmnorm(n = 5)
```

```
[1] 0.02528938 -0.16551273 -0.79750249 -1.41128179 -0.84761314
```

Errors - however, omitting mandatory arguments will result in an *Error*

```
rmnorm(mean = 100, sd = 15)
```

Error in rmnorm(mean = 100, sd = 15): argument "n" is missing, with no default

Warnings - Some inputs may cause the function to produce *Warnings* and bad output, but do **not** stop code execution

```
rmnorm(n = 5, mean = 100, sd = -15)
```

```
Warning in rmnorm(n = 5, mean = 100, sd = -15): NAs produced
```

```
[1] NaN NaN NaN NaN NaN
```

Functions and arguments

HELP! see the documentation of a function

There are two ways to access documentation: using “?” and using `help()`

```
?rnorm # this will work  
help(rnorm) # this does the same
```

FilesPlotsPackagesHelpViewerPresentation

←→🏠🔍

R: The Normal Distribution - Find in Topic

Normal {stats}R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

<code>x, q</code>	vector of quantiles.
<code>p</code>	vector of probabilities.
<code>n</code>	number of observations. If <code>length(n) > 1</code> , the length is taken to be the number required.
<code>mean</code>	vector of means.
<code>sd</code>	vector of standard deviations.
<code>log, log.p</code>	logical; if TRUE, probabilities <code>p</code> are given as <code>log(p)</code> .
<code>lower.tail</code>	logical; if TRUE (default) probabilities are $P[X \leq x]$ otherwise $P[X > x]$

Set up Working Directory

The **Working Directory** (WD) is the location of the folder in your computer where R reads and saves files by default.

Let's see my own WD using the `getwd()` function:

```
getwd()
```

```
[1] "C:/Users/enric/Desktop/Basics R DataScience/Slides"
```

If you import/export anything (*data, figures, workspaces*, etc.) you need to know your WD!

Set up Working Directory

As a general rule:

- When you open R or the RStudio app, the default WD may be the `documents` folder (in Windows) or the `home directory` (e.g., `/home/username`; in Linux or macOS);
- This default may be reset at any time from inside RStudio on `Tools > Global Options... > General`;
- If you open a file (e.g., a `.R` script) using RStudio, the WD is set at that file location (unless the RStudio app was already open before);
- However, you can set a new WD at any time from within the R code, using the `setwd()` function, for example:

```
setwd( "C:/Users/enric/" )
```

Set up Working Directory

⚠ Windows users, be careful!

When you copy-and-paste a folder address, Windows will probably take “\” as path separator, but “\” is the escape character in R, so you incur an error:

```
setwd( "C:\Users\enric\" )
```

Error: '\U' used without hex digits in character string (<input>:1:12)

How to fix

```
setwd( "C:\\Users\\enric\\" )
```

or

```
setwd( "C:/Users/enric/" )
```


Absolute vs Relative paths

- **ABSOLUTE paths:** indicate the full path from the root, e.g., `"C:/Users/enric/"` (in Linux it might be `"/home/enric/"`; in macOS `"/Users/enric/"`)
- **RELATIVE paths:** indicate the path starting from (*relative to*) the current directory. Most often, you prefer this for import/export, so the same script works on any device; e.g., `"figures/"` or `../figures/`:
 - `png(filename = "figures/Fig1.png")` saves a file `Fig1.png` into `figures`, which is a subfolder **inside** the current WD;
 - `png(filename = "../figures/Fig1.png")` saves a file `Fig1.png` into `figures`, which is a folder **outside, one level up** (`../`) the current WD

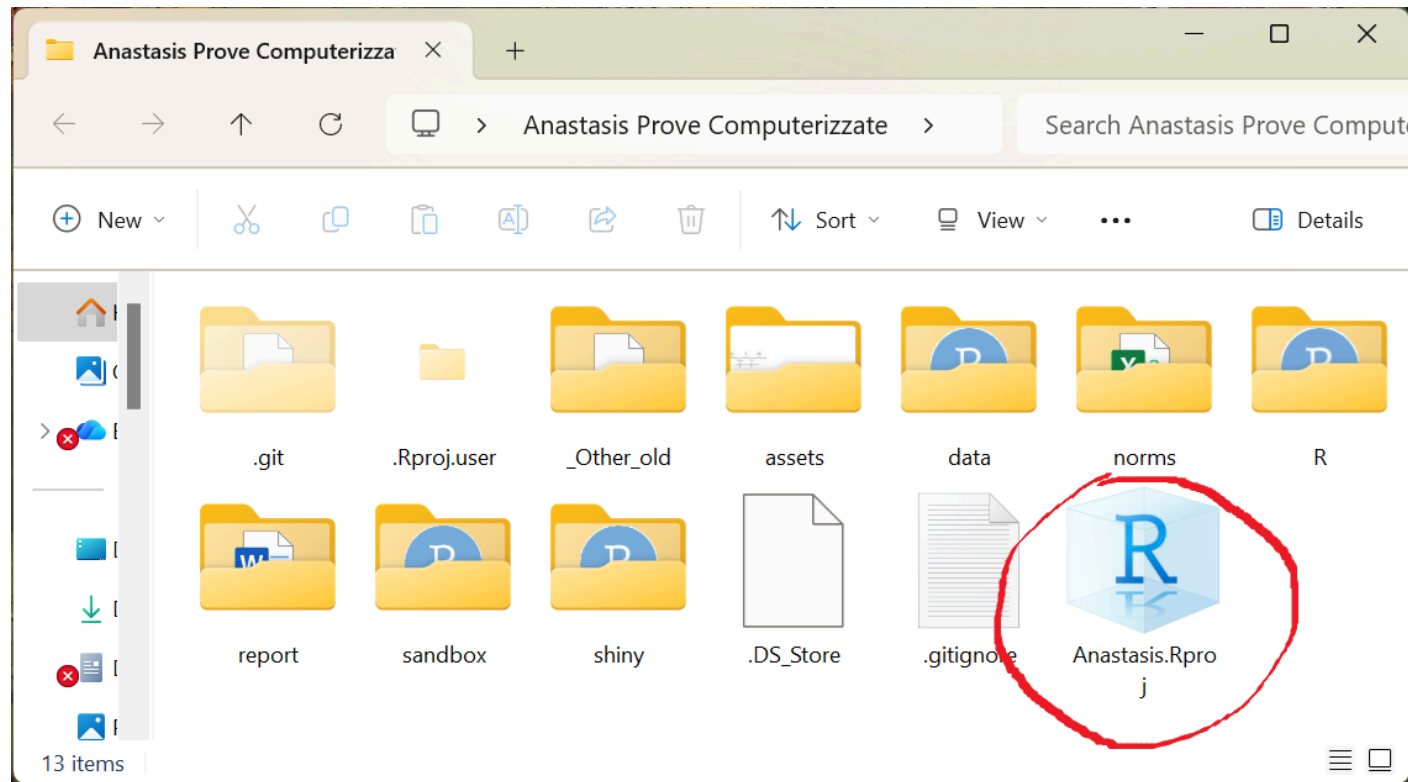
RStudio Projects

`.Rproj` may eliminate the need of using `setwd()` within scripts, so you can use only relative paths

- You can create a *new project* in RStudio with **File > New Project...** choose a specific folder
- Keep all materials of your project in the same folder as the newly created `.Rproj` file
- As you open the `.Rproj`, it will automatically start a new *RStudio* session with the WD set into that folder.

RStudio Projects

Opening the `.Rproj` file ensures that RStudio automatically sets the project folder as the working directory, so all files are managed relative to here:



Import/export

Now let's see how to perform **import/export** operations for:

- **The Workspace:** all objects that exist in your current R session, all results and computations stored so far (see them in the “*Environment*” panel or with `ls()`);
- **Data: SUPER IMPORTANT!** we will focus especially on tabular (Excel-like) data, that we treat as dataframes;
- **Figures:** save your plots for reports and more in `.pdf`, `.png`, and more formats.

Import/export

Workspace

All your R code (script) is generally stored in text files with a `.R` extension. But where do you save your results and objects?! Maybe you just don't...

However, you can export the entire *workspace* using `save.image()`

```
# Let's populate the workspace first
N = 100
x = rnorm(N,0,1)
y = 2 + 0.6*x + rnorm(N,0,0.8)
df = data.frame(x, y)
fit = lm(y ~ x, data=df)

# now Let's save it
save.image("myWS.RData")
```

Specifying `"myWS.RData"` is not mandatory but recommended, otherwise your file will simply be named `".RData"`. (By the way... *where* will it be saved?)

Import/export

Workspace

Alternatively, you can save just one or a few workspace objects using `save()`

```
# Let's populate the workspace first  
N = 100  
x = rnorm(N,0,1)  
y = 2 + 0.6*x + rnorm(N,0,0.8)  
df = data.frame(x, y)  
fit = lm(y ~ x, data=df)  
  
# now Let's save only two objects  
save(df, fit, file="myWS.RData")
```

This will save only objects `df` and `fit` into a newly created file named `myWS.RData`. This is useful when you have an overcrowded workspace and want to save only a few objects containing the final results

Import/export

Workspace

Once you open a new R session, you may load the previously stored workspace using the `load()` function, specifying `load("workspace_name.RData")`, like this:

```
# empty the workspace to make sure there's actually nothing!
```

```
rm(list=ls())
```

```
# now load the previously saved workspace
```

```
load("myWS.RData")
```

```
# make sure that the objects have been loaded
```

```
ls()
```

Import/export

Data

Arguably a **fundamental skill** for anyone working in data science!

Most people use *MS Excel* or similar software (e.g., *LibreOffice Calc*) for handling data, which produce their own file formats (e.g., `.xlsx`). That's perfectly fine. However... the **most versatile data format is `.csv` (comma-separated values)**, a simple text (no formatting, no licences required) file format for storing tabular data/dataframes.

- **Best practice:** Save data in `.csv` format from your software of choice before importing it in R.

Import/export

Data

Here's an example of using `read.csv()` for importing data:

```
# IMPORT csv data from a "data" subfolder, and store it in an object named "df"
df = read.csv("data/Performance.csv")

# OR if you want to be explicit on settings:
df = read.csv("data/Performance.csv", header=TRUE, sep=";", dec=".")

head(df) # have a look at the first few rows
```

	id	name	anx	acc	time
1	1	nydga	20	15	2.077932
2	2	bwknr	14	9	2.436858
3	3	sauuj	18	12	2.549814
4	4	vnjgi	27	15	4.386718
5	5	oueiy	21	11	5.248933
6	6	neebj	12	13	3.463094

⚠ in Italian Excel export settings, it is possible that *separator character* (`sep`) be “;” and *decimal point character* be “,” so be aware of your settings!

Import/export

Data

If you absolutely want to import your data directly from a **MS Excel** document (**.xlsx**), you may use function **read_excel()** from the package **readxl**:

```
library(readxl)
df = data.frame( read_excel("data/Performance.xlsx") )
# data.frame() forces it to be a dataframe, otherwise it's a tibble
head(df)
```

	id	name	anx	acc	time
1	1	nydga	20	15	2.077932
2	2	bwknr	14	9	2.436858
3	3	sauuj	18	12	2.549814
4	4	vnjgi	27	15	4.386718
5	5	oueiy	21	11	5.248933
6	6	neebj	12	13	3.463094

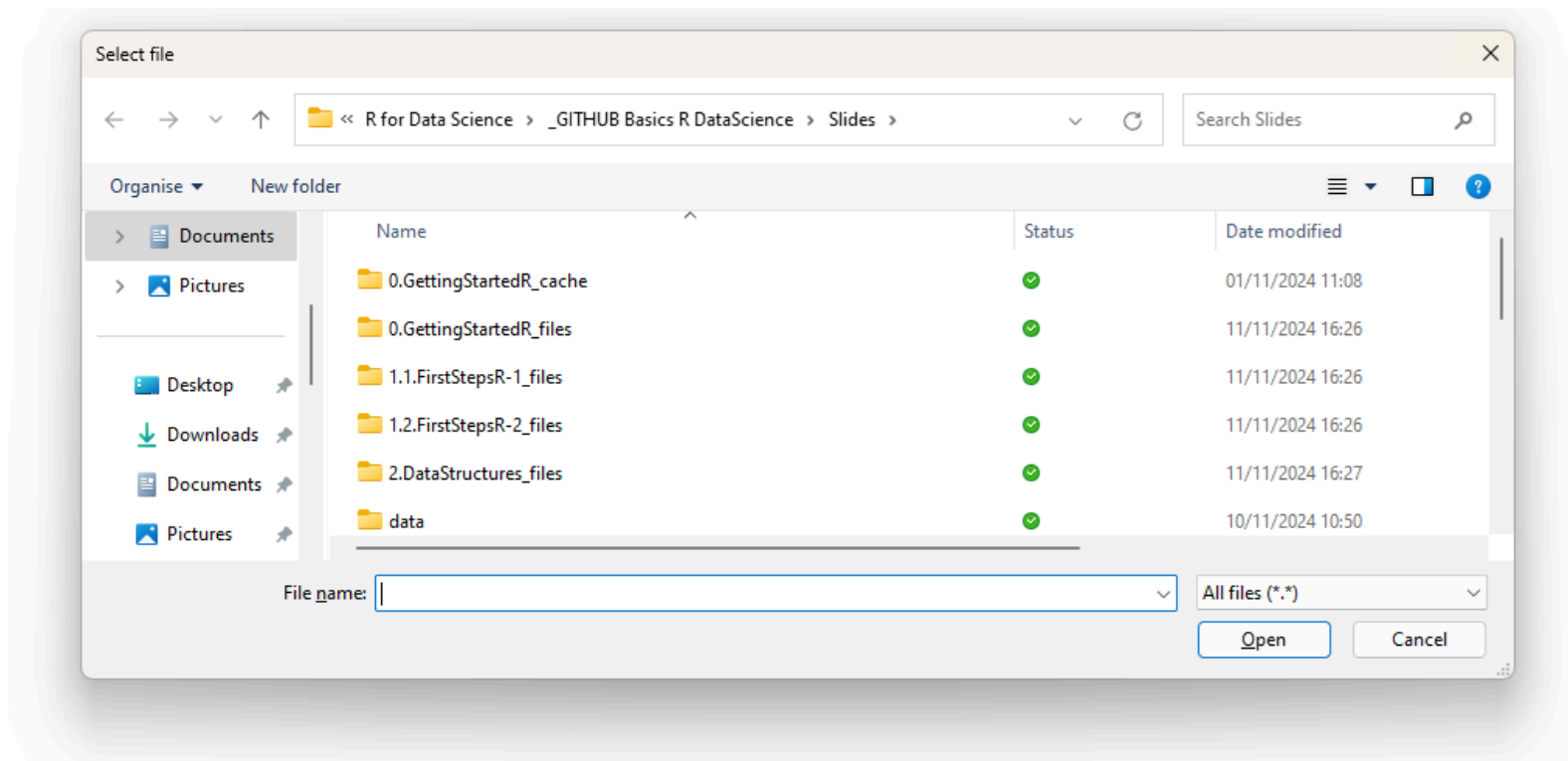
- You may even import data from an **SPSS** document (**.sav**) using the **read.spss()** function from the **foreign** package

Import/export

Data

A good trick if you don't want to specify any relative or absolute path, and want to manually select data each time, is using the `file.choose()` function:

```
df = read.csv( file.choose() )
```



Import/export

Data

Other “tricks” for importing data involve using the functions in the **RStudio menu**, particularly:

- File > Import Dataset > From text (base)...
- File > Import Dataset > From Excel
- File > Import Dataset > From SPSS...

However ... using these functions is not best practice, because they are specific to the RStudio IDE. **It's better to use code for reproducibility**

Import/export

Data

You have processed data with R, now... how to **export** it?

When collaborating with someone also using R, you may choose to exchange data directly by exporting the object or the entire workspace as a `.RData` file, using the `save()` or `save.image()` function respectively.

However, if you need to export a dataframe in a more universally readable tabular format, such as `.csv`, you may use `write.table()`:

```
# specify the dataframe to export (here named "df")  
# along with the desired file name, and other arguments  
write.csv(df, file="myExportedData.csv")  
  
# if you don't want row numbering / row names  
write.csv(df, file="myExportedData.csv", row.names=F)
```

Import/export

Figures

R has a collection of functions for exporting figures in different formats: `pdf()`, `png()`, `jpeg()`, `bmp()`, `tiff()`, `svg()`.

Here is an example using `png()` :

```
# set up a graphic output file named "MyFigure.png" with some settings  
png("MyFigure.png", height=1500, width=2000, units="px", res=300)  
  
# code for creating a simple boxplot  
boxplot(iris$Sepal.Width)  
  
# close the graphic output file and actually export the plot  
dev.off()
```