Leibniz Institute for Financial Research SAFE
Theodor-W.-Adorno-Platz 3 I 60629 Frankfurt am Main I Germany

LEIBNIZ INSTITUTE SAFE Sustainable Architecture for Finance in Europe

# Common errors in R

Ina Krapp
SAFE Research Datacenter*

Last Update: October 7, 2024

This tutorial gives an overview over common errors in R. It is intended to be run as a RMarkdown file in RStudio, which allows you to execute the code yourself. To run it, download the RMarkdown version here: RMarkdown version. Open it in a recent version of RStudio.

## Contents



## 1 Contact

If you encounter any difficulties or just want general information, do not hesitate to contact us.

SAFE Research Datacenter: datacenter@safe-frankfurt.de

More information about the SAFE Research Datacenter and further guides can be found here.

*krapp@safe-frankfurt.de

# 2 Prerequisites

R can be downloaded here.
We'll also use RStudio. RStudio is a user interface which makes working with R much more convenient. You can get it here.
If you want to follow along with the exercises, you'll need to download the files 'common_errors.xlsx' and 'common_errors.csv'. They can be found here: Excel file and here: csv file
Download these files and put them into the same folder that you have currently open in RStudio.

# 3 Aims and targets of this workshop

This workshops is about common errors in R. It shows reasons why they occur and ways to solve them. It aims to be beginner-friendly, but will skip over some details. So for complete beginners, the Introduction to R might still be interesting: Introduction to R

If you are interested in what you can do with R, other workshops are also available:
Cleaning data in R: An introduction to the tidyverse in R
Designing graphics with R: Introduction to ggplot2 – Create publication-ready graphs with R
Conducting a linear Regression in R: Linear regression with R
Conducting a Time Series Analysis in R: Time Series Analysis in R

# 4 About programming errors

Don't be afraid of getting an error. It's normal to get them all the time, and experienced R users don't necessarily get less errors - they just are quicker to solve them.

# 5 Typing mistakes

Typing mistakes are one of the most common sources of errors in R. Unfortunately, there is no single error message you can be sure to get if you just made a typing mistake.
However, there are a few frequent ones. Assume we have a few values we want to calculate the mean from. First, we can create an object in R to store them:

```
values = c(1000,2000,3000)
```

If we want to look at the object 'values' now, we can do so by writing its name:

```
values
```

```
[1] 1000 2000 3000
```

As you can see, the object contains the values we entered. If we do a typing mistake when trying to write the name of the object, we get an 'object not found' error.

```
valuess
```

```
Error: object 'valuess' not found
```

The same happens if we spell the initial name of the object wrong. Assume we have an object containing ages, but misspell the name:

```
agess = c(20, 25, 30)
```

Then, when we try to look at it, writing its intended name:

```
ages
```

```
Error: object 'ages' not found
```

The system will not find it. As you can see in the upper right corner, the age data is now stored in an object named 'agess', and to see the data in this object, we need to write 'agess'.

```
agess = c(20, 25, 30)
```

```
[1] 20 25 30
```

So if you get an 'object not found' error: 1. Check if you typed it wrong in the piece of code where the error occurred. 2. Check if you typed it wrong in the piece of code where you created it.
Likewise, this error can occur if you forgot how you named your object. For example, if you wrongly think that the object containing the values was named 'value':

```
value
```

```
Error: object 'value' not found
```

Once you have data in R, you will want to perform calculations with them. In this case, you apply a function to an object by writing the name of the function, and adding the object in brackets behind it. For example, we can use the 'mean' function to calculate the mean of the values in our 'values' object:

```
mean(values)
```

```
[1] 2000
```

A quite frequent error is not to close a parenthesis. R normally tries to close them automatically, but it still happens that people forget the closing bracket:

```
mean(values
```

```
Error: Incomplete expression: mean(values
```

As you can see from the error message, R treats this impression as incomplete - something, the final bracket, is missing.
In R, you can take parts of an object using its index. For example, the object 'values' contains three numbers, which have the index 1, 2, and 3. You can get the first two with the following code:

```
values[1:2]
```

```
[1] 1000 2000
```

If one of these square brackets is missing, the error message is the same that you get when a round bracket is missing:

```
values[1:2
```

```
Error: Incomplete expression: values[1:2
```

These errors are more complex to solve if the code contains several brackets. For example, assume we'd want the mean of the first two numbers stored in 'values'. Then, we can use this piece of code:

```
mean(values[1:2])
```

```
[1] 1500
```

In such a piece of code, the error depends on which bracket is missing. If the inner bracket is missing, R's error message treats the outer bracket as unfitting piece in the code, giving an 'unexpected ... in code' error:

```
mean(values[1:2)
```

```
Error: unexpected ')' in "mean(values[1:2)"
```

On the other hand, if the outer bracket is missing, you get the incomplete expression error again:

```
mean(values[1:2]
```

```
Error: Incomplete expression: mean(values[1:2]
```

This is related to how R processes commands: It does so from the inside out. So in the first step, it ignores the 'mean' command, and only takes the two first values from the 'values' object. In this process, it is expecting a closing square bracket at the end, but finds a round bracket - an unexpected object. On the other hand, if the outer bracket is missing, then the inner part of the code runs without problems. Only in the second step does R try to calculate the mean of the values it obtained in the first step. In this case, we are back to the error that R would expect the expression to go on because a part of it (the round bracket) is still missing - giving us the incomplete expression error.
Code with nested brackets can therefore quickly become confusing. There are ways to avoid it. The simplest is to create intermediate objects:

```
values_1_2 <- values[1:2]
mean(values_1_2)
```

```
[1] 1500
```

Within a function, if you spell the name of the object wrong, you get the same 'object not found' error you are already familiar with:

```
mean(valuess)
```

```
Error: object 'valuess' not found
```

But if you misspell the name of the function, you will get another error:

```
mea(values)
```

```
Error in mea(values) : could not find function "mea"
```

# 6 Loading packages and reading data

If you use code from a package, this error may also indicate that you're trying to use it when it is not loaded yet. For example, there is a package that allows you to load excel data into R. It contains the read_excel-function. If I apply the function without having loaded the package, I get the same 'could not find function' error as before:

```
common_error_data <- read_excel("common_error_data.xlsx")
```

```
Error in read_excel("common_error_data.xlsx") :
could not find function "read_excel"
```

I can load the package with the library command: (You may have to install it first)

```
library("readxl")
```

To avoid this type of issue, it is good practice to put the loading of libraries into the 'r setup' block at the beginning of the file.
Afterwards, I can load the excel data into R. Because of that, it is normally recommended to put your library commands at the start of a file, I'm only putting this one in the middle to demonstrate the error.
Note that read_excel-commands require the names of the datasets they access to be put into quotation marks. That's because R will search for any object whose name is not in quotation marks in the environment. However, the data we are searching for is not in the environment yet, so R will not find it there.

```
common_error_data <- read_excel(common_error_data.xlsx)
```

```
Error: object 'common_error_data.xlsx' not found
```

Instead, the read_excel-function searches the object with quotation marks in the folder where the R code runs. You can see this folder in the lower left in 'Files'. You should be able to find the Excel file you want to import there.

```
common_error_data <- read_excel("common_error_data.xlsx")
```

As you can see, the code now ran correctly. In the upper right corner, you now have an object called 'common_error_data' in your Environment.
Again, remember to close the quotation marks - like missing the final bracket, missing the final quotation mark gives an 'incomplete expression' error. You can use single or double brackets, but if you start with a single bracket, you must finish with a single bracket, and vice versa. The whole name of the file must be given, including the file ending (xlsx for Excel, csv for csv-files and so on). If you spell it wrong or forget the ending, you get the following error:

```
common_error_data <- read_excel("common_error_data")
```

```
Error: ‘path‘ does not exist: ‘common_error_data’
```

If you get an error like this, check that the file you try to import is visible in 'Files'. If it is, then you probably typed its name wrong. If it is not visible, then your code - or the file, depending on the point of view - is in the wrong folder.
Unfortunately, folder navigation can be tricky in R. Use 'Session -¿ Set Working Directory -¿ Choose Directory' if you think you're in the wrong folder.
The exact error message also depends on the type of file you aim to load. R supports a large variety of formats, including, for example, Excel and csv files. You can load data into R from a csv file with the following command:

```
common_error_data_csv <- read.csv("common_error_data.csv")
```

If you aim to get data from a csv file the computer can not find, you get a 'cannot open the connection' error.

```
common_error_data_csv <- read.csv("common_error_dat.csv")
```

```
Error in file(file, "rt") : cannot open the connection
```

Despite the different error descriptions, the underlying cause is usually the same: Either the name of the file was misspelled or the computer searches for it in the wrong folder.

# 7 Working with dataframes:

After we imported the Excel file, we have a new object in our environment. This object is a data frame. Data frames are tables. R can load them from a wide number of formats, including, but not limited to, Excel, CSV and Stata (.dta-files). When you work with R for research, you'll usually work with dataframes.
If you click on the name of the dataset, you'll see its table structure. It contains three columns and three rows. You can access individual columns of the dataframe using a '$' symbol:

```
common_error_data$Income
```

```
[1] 1000 2000 3000
```

A frequent mistake when trying to do that is misspelling the column name. Note that R is case-sensitive: 'I' and 'i' are different letters for R. Since a column named 'income' does not exist, we get the following message:

```
common_error_data$income
```

```
Warning: Unknown or uninitialised column: ‘income‘.NULL
```

This is a warning message, not an error message. As opposed to errors, which mean the code did not run, this piece of code did run. We do get an output. But in this case, the output is 'NULL' - since the column does not exist, the output is empty.
You can ignore warnings, pretend nothing happened, and continue to work with this. For example, calculating the sum of Incomes:

```
sum(common_error_data$income)
```

```
Warning: Unknown or uninitialised column: ‘income‘.[1] 0
```

But in some cases, the results stop making sense. Since R can not really access the income data, it instead calculates the sum of an empty object. This sum is, unsurprisingly, 0.
Such errors are easy to avoid: As soon as you put '$' behind a dataframe, R automatically suggests columns that are in this dataframe.

```
sum(common_error_data$Income)
```

```
[1] 6000
```

# 8 Unexpected inputs

R distinguishes different types of values, for example numbers and letters (letters are called 'characters' in R). Unsurprisingly, you can only calculate with numbers in R. Code as the one below, which aims to take the sum of names, gives an error.

```
sum(common_error_data$Name)
```

```
Error in sum(common_error_data$Name) :
invalid 'type' (character) of argument
```

You can look at the type of an object with the typeof command:

```
typeof(common_error_data$Name)
```

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

Because everything is an object in R, you can use practically everything as input to a function: Functions can take dataframes, dataframe columns, single numbers or even other functions as inputs.
Many functions even take different types of input (and behave differently depending on the type of input they got). This is called 'defining a function on a certain type of input'. It's part of what makes R so powerful, but since not every function works for every object, it also often leads to errors.
Another example:

```
sum(common_error_data)
```

```
Error in FUN(X[[i]], ...) :
only defined on a data frame with all numeric-alike variables
```

As you can see here, 'sum' could take a dataset as input - but only if all the values in the dataset were numbers. In this case, it would give out the sum of all these numbers. But since the dataset we use also contains words, it can not be used as input for 'sum'.
Signs like '+' or '$' are functions, too. '$' is typically used on dataframes, and can not be used on single numbers or vectors:

```
values$values
```

```
Error in values$values : $ operator is invalid for atomic vectors
```

In reality, this error typically occurs if an object is not what you think it is, so check if the object you tried to extract a column from really is a dataframe.
A related issue can occur when getting values from a vector. Remember you could take values from a vector with a square bracket:

```
values[1:2]
```

```
[1] 1000 2000
```

This is only possible for vectors or dataframes, not for functions:

```
mean[1:2]
```

```
Error in mean[1:2] : object of type 'closure' is not subsettable
```

# 9 Too few and too many arguments

Functions have compulsory and non-compulsory arguments. If a non-compulsory argument is not given in the code, R will use a default value instead. Look at the documentation of each function to see what the defaults are.
If a compulsory argument is missing in the code, R will give an 'argument is missing, with no default' error:

```
mean()
```

```
Error in mean.default() : argument "x" is missing, with no default
```

Vice versa, there is an 'unused_argument' error that will occur if too many arguments are given to a function. The typeof function is designed to give information about the type of exactly one object, and will fail with this error if two or more objects are given to it.

```
typeof(1, 2)
```

```
Error in typeof(1, 2) : unused argument (2)
```

It is not necessarily intuitive if R will give an error or not. For example, the mean function requires some object 'x' to take the mean from. But the function 'sum' does not strictly require values to take the sum from:

```
sum()
```

```
[1] 0
```

Although the code above is therefore technically valid, you'll want to avoid it - it could easily lead to unexpected results.

# 10 The equality sign

From mathematics, we are used to read '=' as 'is equal to' In programming, the meaning is slightly different. Code like the block below reads as 'create a variable with the value 3 and give it the name a'.

```
a = 3
```

Trying to assign a number to a number gives an error:

```
3 = 4
```

```
Error in 3 = 4 : invalid (do_set) left-hand side to assignment
```

This is because '3' is not a valid variable name. R interprets the statement above as 'Give 3 the value of 4', which is obviously nonsense.
If you want to compare two values, you need a double equality sign:

```
3 == 4
```

```
[1] FALSE
```

The code then returns 'TRUE' or 'FALSE'.

```
3 == 3
```

```
[1] TRUE
```

The '==' sign can also be used to compare columns with each other or with numbers. Below, it gives, for each value in the column 'Income', the information if this value is equal to 3000 or not.

```
common_error_data$Income == 3000
```

```
[1] FALSE FALSE  TRUE
```

If you only use a single equality sign, the code instead gives every row in the 'income' column the value 3000. So do not confuse that!

```
common_error_data$Income = 3000
common_error_data$Income
```

```
[1] 3000 3000 3000
```

# 11 Unwanted results

There are various results that are not errors, but often still unexpected to the user. An example:

```
mean(c(1,2,3,NA))
```

```
[1] NA
```

'NA' means 'not available'. It is how R calls missing data. Missing data is 'infectious': Any calculation with at least one NA value will most likely give you a NA value as a result. This is intentional and makes sense: If you want to take the mean value of four values, but don't know one of these values, you can not know the true mean, either. However, in reality, in such a situation, you may often want to calculate the mean of only the values you know.
For this, most R functions used for calculations have the option to exclude NA values by setting 'na.rm' to TRUE:

```
mean(c(1,2,3,NA), na.rm = TRUE)
```

```
[1] 2
```

An exception are linear regressions in R. They will automatically exclude NA values.

```
regression = lm(Income~Age, data = common_error_data) # We put the dependent
    variable to the left of the '~'
# and the independent variable(s) to the right and we tell R which dataset
   we are referring to.
summary(regression)
```

```
Warning: essentially perfect fit: summary may be unreliable
Call:
lm(formula = Income ~ Age, data = common_error_data)

Residuals:
         1          2          3
 2.625e-13 -5.251e-13  2.625e-13
```

```
Coefficients:
              Estimate Std. Error    t value Pr(>|t|)
(Intercept)  3.000e+03  2.304e-12  1.302e+15 4.89e-16 ***
Age         -1.575e-13  9.095e-14 -1.732e+00    0.333
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 6.431e-13 on 1 degrees of freedom
Multiple R-squared:    0.6,Adjusted R-squared:    0.2
F-statistic:   1.5 on 1 and 1 DF,  p-value: 0.4359
```

NA's may occur in R for a number of reasons. Often, they are already in the data when it is imported into R.
The code below creates a new column named 'Numbers'. It does so by attempting to turn the names from the 'Name' That would be necessary if R had wrongly treated a column of numeric values as words.
However, since the 'Name' column does not contain any numbers, the column 'Numbers' ends up filled with NA's instead.

```
1    common_error_data$Numbers = as.numeric(common_error_data$Name)
2    common_error_data$Numbers
```

```
Warning: NAs introduced by coercion
```

```
[1] NA NA NA
```