Getting Started with R

Sven Otto

April 16, 2024

Table of contents

Welcome

This tutorial aims to serve as an introduction to the software package R. Other excellent and much more exhaustive tutorials can be found at the following links:

- An interactive R-package for learning R: swirl (highly recommended for beginners).
- Interactive R courses at Datacamp and Coursera (free, but registration required).
- Learn R in 30 minutes: link.
- Video series by Nick Huntington-Klein: Introduction to R for Economists.
- The official introduction and reference cards for basic R and time series analysis.
- Some excellent books:
  - Hands-On Programming with R (for absolute beginners)
  - R for Data Science (R and the tidyverse)
  - Advanced R (improve your programming skills)
  - R Codebook (proven recipes for data analysis)
  - Forecasting: Principles and Practice (time series analysis in R)
  - R Packages (write your own R package)
  - HappyGitWithR (version control with RStudio)

Why R?

- R is **free** of charge. On the R project webpage cran.r-project.org, you can download R for Windows, Mac OS, or Linux. Windows users can also directly follow this link: cran.r-project.org/bin/windows/base/
- You can use R via a terminal or install an IDE, which is much more convenient. The celebrated IDE **RStudio** for R is also **free** of charge. Download RStudio here: posit.co/download/rstudio-desktop/. Make sure that you install R before installing RStudio.
- Within RStudio, you can use **Quarto**, which provides an authoring framework to export your R code/outputs/plots together with LaTeX formulas and text as a PDF file or website in an appealing way. Have a look here. This website is also built with Quarto. You may want to use Quarto for your assignments, term papers, or thesis.
- R is equipped with one of the most flexible and powerful graphics routines available anywhere. Check out these repositories with examples of appealing and informative R graphs: Clean Graphs, R Graph Catalog, Publication Ready Plots.

- One of the best features of R are the large number of contributed packages from the statistical community. You find R packages for almost any statistical method out there and many statisticians provide R packages to accompany their research.
- R is the de-facto standard for statistical science.

Matrix algebra

R is a matrix-based programming language. Matrix algebra provides an efficient framework for analyzing and implementing econometric methods. To refresh your matrix algebra skills and to learn how to use it in R, please check out my **Crash Course on Matrix Algebra in R**.

Accompanying R scripts

All R codes of the different sections can be found here:

- rintro-sec1.R.
- rintro-sec2.R.

Comments

Feedback is welcome. If you notice any typos or issues, please report them on GitHub or email me at sven.otto@uni-koeln.de.

1 Base R

1.1 Short Glossary

Let's start the tutorial with a (very) short glossary:

- **Console**: The thing with the `>` sign at the beginning.
- **Script file**: An ordinary text file with suffix `.R`. For instance, `yourfilename.R`.
- **Working directory**: The file directory you are working in. If no directory is explicitly specified when loading data, then R assumes that the data is located in the *working directory*. Useful commands: with `getwd()`, you get the location of your current working directory, and `setwd()` allows you to set a new location for it.
- **Workspace**: This is a hidden file (stored in the working directory as *.RData*) where all objects you use (e.g., data, matrices, vectors, variables, functions, etc.) are stored. When you close RStudio, you will be asked if you want to save or delete the session's *workspace*. If you save it, it will be loaded automatically with the next R session, provided you start R in the corresponding working directory. Useful commands: `ls()` shows all elements in our current workspace, and `rm(list=ls())` deletes all elements in our current workspace.

1.2 First Steps

A good idea is to use a script file like **myscipt.R** to store your R commands. You can send single lines or marked areas of your R code to the console by pressing the **CTRL+RETURN** (STRG+ENTER) keys.

To start with baby steps, we do some simple calculations:

```r
2+2 # addition
```

```
[1] 4
```

```r
2*2 # multiplication
```

```
[1] 4
```

```r
2/2
```

```
[1] 1
```

```r
2-2
```

```
[1] 0
```

```r
2^3 # exponentiate
```

```
[1] 8
```

Note: Anything written after the `#` sign will be ignored by R, which is very useful for commenting on your code.

The **assignment operator** `<-` will be your most often-used tool. Here is an example of creating a **scalar** variable:

```r
x <- 4
x
```

```
[1] 4
```

```r
4 -> x # possible but unusual
x
```

```
[1] 4
```

```r
x = 4
x
```

```
[1] 4
```

Note: The R community loves the `<-` assignment operator. Alternatively, you can use the `=` operator.

1.3 Vectors and functions

And now a more interesting object - a **vector**:

```r
y = c(2,7,4,1)
y
```

```
[1] 2 7 4 1
```

The command `ls()` shows the total content of your current workspace, and the command `rm(list=ls())` deletes all elements of your current workspace:

```r
ls()
```

```
[1] "has_annotations" "x"               "y"
```

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

```
character(0)
```

Note: RStudio's **Environment** pane also lists all the elements in your current workspace. That is, the command `ls()` becomes a bit obsolete when working with RStudio.

Let's try how we can compute with vectors and scalars in R.

```r
x = 4
y = c(2,7,4,1)

x*y # each element in the vector y is multiplied by the scalar x
```

```
[1]  8 28 16  4
```

```r
y*y # a term-by-term product of the elements in y
```

```
[1]  4 49 16  1
```

The term-by-term execution, as in the above example, `y*y`, is a main strength of R. We can conduct many operations **vector-wisely**:

```r
y^2
```

```
[1]  4 49 16  1
```

```r
log(y)
```

```
[1] 0.6931472 1.9459101 1.3862944 0.0000000
```

```r
exp(y)
```

```
[1]    7.389056 1096.633158   54.598150    2.718282
```

```r
y-mean(y)
```

```
[1] -1.5  3.5  0.5 -2.5
```

```r
(y-mean(y))/sd(y) # standardization
```

```
[1] -0.5669467  1.3228757  0.1889822 -0.9449112
```

Element-wise operations are a central characteristic of matrix-based languages like R (or Matlab). Other programming languages often have to use **loops** instead:

```r
N = length(y)
1:N

y.sq = rep(0,N)
y.sq

for(i in 1:N){
  y.sq[i] = y[i]^2
  if(i == N){
    print(y.sq)
  }
}
```

The `for()`-loop is the most common loop, but there is also a `while()`-loop and a `repeat()`-loop. However, loops in R can be relatively slow. Therefore, try to avoid them!

Useful commands to produce **sequences** of numbers:

```
1:10
-10:10
?seq # Help for the seq()-function
seq(from=1, to=100, by=7) # sequence generation
rep(0,10) # replicate elements
```

The `[]`-operator **selects elements** of vectors:

```
y[c(2,4)]
```

```
[1] 7 1
```

Element selections can be made on a more **logical** basis, too. For example, if you want only the elements of the vector `y` that are strictly greater than 2:

```
y[y>2]
```

```
[1] 7 4
```

```
# Note that this gives you a boolean vector:
y>2
```

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

Note: Logical operations return so-called **boolean** objects, i.e., a `TRUE` or a `FALSE`. For instance, if we ask R whether `1>2`, we get the answer `FALSE`.

1.4 Further Data Objects

Besides the classical data objects like scalars and vectors, there are many other objects in R:

1.4.1 The matrix

A `matrix` is a rectangular array of numbers.

```r
mymatrix = matrix(data=1:16, nrow=4, ncol=4)
mymatrix
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

Matrices are extremely useful for theoretically analyzing statistical methods and implementing them practically.

> Matrix Algebra in R
>
> To refresh your matrix algebra skills with implementations in R, check out my **Crash Course on Matrix Algebra in R**.

1.4.2 The list

In `lists`, you can organize different kinds of data. E.g., consider the following example:

```r
mylist = list(
  "Some_Numbers" = c(66, 76, 55, 12, 4, 66, 8, 99),
  "Animals"      = c("Rabbit", "Cat", "Elefant"),
  "My_Series"    = c(30:1)
)
```

A very useful function to find specific values and entries within lists is the `str()`-function:

```r
str(mylist)
```

```
List of 3
 $ Some_Numbers: num [1:8] 66 76 55 12 4 66 8 99
 $ Animals     : chr [1:3] "Rabbit" "Cat" "Elefant"
 $ My_Series   : int [1:30] 30 29 28 27 26 25 24 23 22 21 ...
```

1.4.3 The data frame

A `data.frame` is a `list` object with more formal restrictions (e.g., an equal number of rows for all columns). As indicated by its name, a `data.frame` object is designed to store data:

```
mydataframe = data.frame(
  "Credit_Default"   = c( 0, 0, 1, 0, 1, 1),
  "Age"              = c(35,41,55,36,44,26),
  "Loan_in_1000_EUR" = c(55,65,23,12,98,76)
)
```

The `data()` command lists all sample data sets available in R. Let us have a look at the dataset `mtcars`. It is a `dara.frame` object and contains data on several aspects of 32 automobiles from 1974.

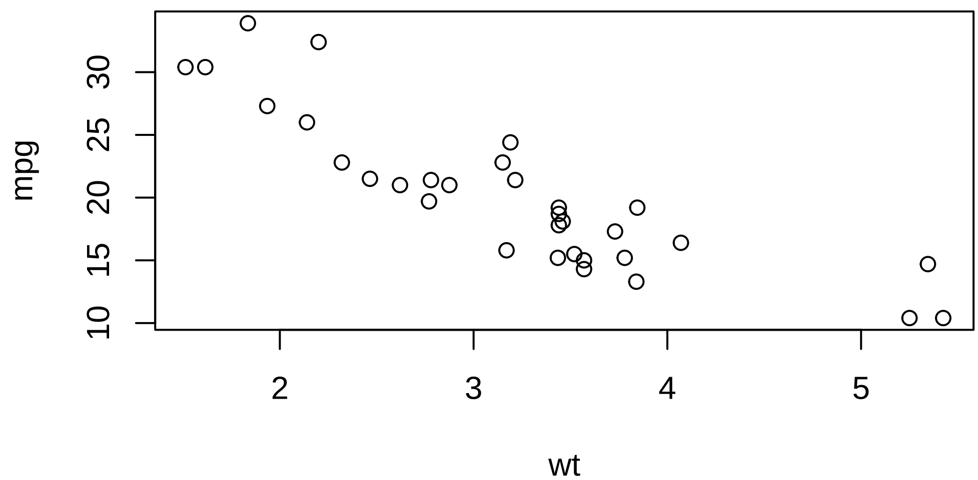
```
mtcars
```

```
                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
```

```
Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

With the function `subset` we can select variables and subsets of a dataframe. Let's create a scatterplot of the variables `mpg` (miles per gallon) and `wt` weight (in 1000 lbs).

```r
plot(subset(mtcars, select = c(wt, mpg)))
```



A `data.frame` is also useful in a time series context. Since time series data typically include a calendar date for each observation, the observation and date can be stored together as a `data.frame`. R provides the class `Date` for calendar dates, which can be generated with the function `as.Date()`.

```r
d = as.Date("2021-04-01") # a data object to store dates
class(d) # to get the object class
```

```
[1] "Date"
```

```r
myseries = c(16,17,18,16,15,19)
mydates = seq.Date(as.Date("2021-04-01"), by=1, length.out = 6)
mytimeseries = data.frame(mydates, myseries)
mytimeseries
```

```
     mydates myseries
1 2021-04-01       16
2 2021-04-02       17
3 2021-04-03       18
4 2021-04-04       16
5 2021-04-05       15
6 2021-04-06       19
```

1.4.4 The ts object

A `ts` (time series) object is tailored explicitly to time series with a yearly time basis and an equidistant observation horizon, such as annual, quarterly, and monthly data. It assigns a specific year/quarter/month to each vector entry.

```r
myts = ts(c(66, 76, 55, 12, 4, 66, 8, 99), start = 2020, frequency = 4)
myts
```

```
     Qtr1 Qtr2 Qtr3 Qtr4
2020   66   76   55   12
2021    4   66    8   99
```

```r
anothertimeseries = ts(1:50, start = 2015, frequency = 12)
anothertimeseries
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2015   1   2   3   4   5   6   7   8   9  10  11  12
2016  13  14  15  16  17  18  19  20  21  22  23  24
2017  25  26  27  28  29  30  31  32  33  34  35  36
2018  37  38  39  40  41  42  43  44  45  46  47  48
2019  49  50
```

```r
# The window() command selects the time series observations for a given subperiod
window(anothertimeseries, start=2015.5, end=2017.5)
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2015                          7   8   9  10  11  12
2016  13  14  15  16  17  18  19  20  21  22  23  24
2017  25  26  27  28  29  30  31
```

The `data()` command lists all sample data sets available in R. Let us have a look at the dataset `AirPassengers`. It is a `ts` object and contains data on monthly totals of international airline passengers from 1949 to 1960.

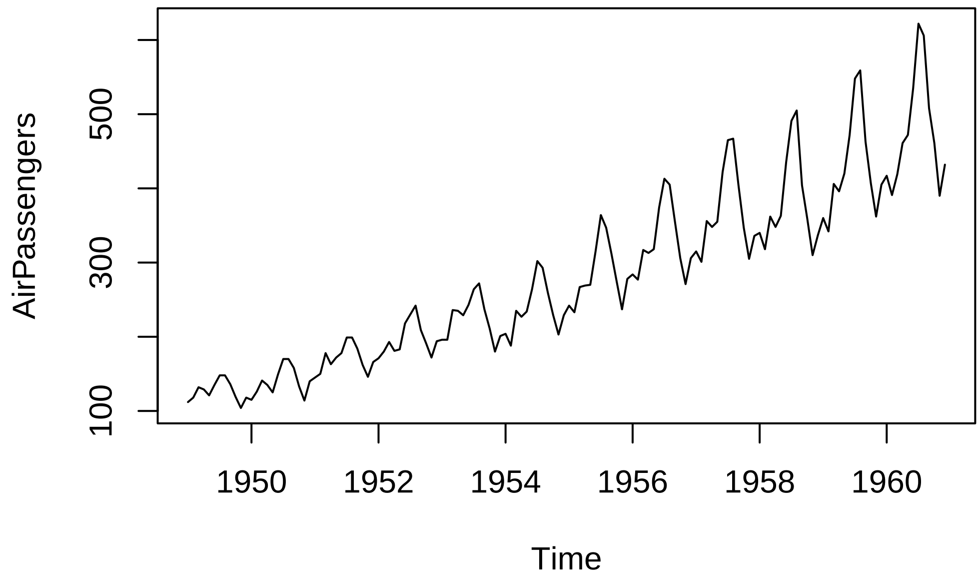
```r
data() # lists all datasets currently loaded in the R environment
?AirPassengers # get more information about the dataset
AirPassengers
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949 112 118 132 129 121 135 148 148 136 119 104 118
1950 115 126 141 135 125 149 170 170 158 133 114 140
1951 145 150 178 163 172 178 199 199 184 162 146 166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
1955 242 233 267 269 270 315 364 347 312 274 237 278
1956 284 277 317 313 318 374 413 405 355 306 271 306
1957 315 301 356 348 355 422 465 467 404 347 305 336
1958 340 318 362 348 363 435 491 505 404 359 310 337
1959 360 342 406 396 420 472 548 559 463 407 362 405
1960 417 391 419 461 472 535 622 606 508 461 390 432
```

```r
class(AirPassengers)  # AirPassengers is a ts object
```

```
[1] "ts"
```

```r
plot(AirPassengers)
```


AirPassengers
100
300
500
1950
1952
1954
1956
1958
1960
Time

2 Packages

One of the best features of R are the large number of contributed packages from the statistical community. The list of all packages on CRAN is impressive! Take a look at it here. You find R packages for almost any statistical method out there. Many statisticians provide R packages to accompany their research. Some packages also provide additional functionality for R or include datasets.

2.1 The `xts` package

Let us look at a time series specific package: the `xts` package. It can be installed using the `install.packages()` function.

```
install.packages("xts")
```

The `xts` package provides the class `xts`, which has certain advantages over `ts`. A `ts` object can specify the frequency of a time series only as a portion of a year (1 for yearly, 4 for quarterly, 12 for monthly data). This scheme is convenient for regular macroeconomic time series but impractical for daily data (leap year problem), high-frequency data, or irregularly collected data. In an `xts` object, we are much more flexible and manually assign a specific time index to each observation in the time series.

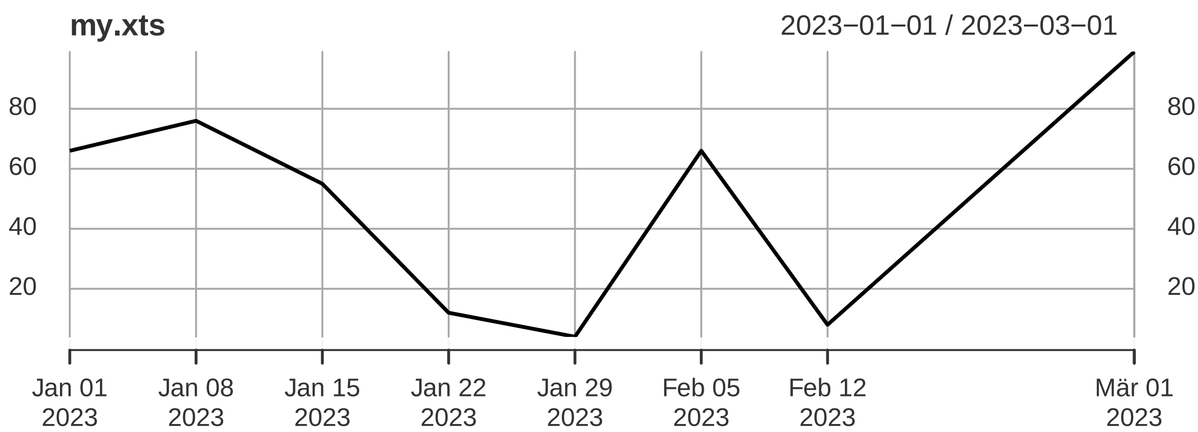
Once installed, the package only has to be loaded at the beginning of a new R session, which is done with the command `library(xts)`.

```
library(xts)
myts = ts(c(66, 76, 55, 12, 4, 66, 8, 99), start = 2020, frequency = 4)
as.xts(myts)  # convert a ts object into an xts object
```

```
        [,1]
2020 Q1   66
2020 Q2   76
2020 Q3   55
2020 Q4   12
2021 Q1    4
```

```
2021 Q2   66
2021 Q3    8
2021 Q4   99
```

```
# we may assign irregular time points:
dates = seq.Date(as.Date("2023-01-01"), by = 7, length.out = 7)
dates[8] = as.Date("2023-03-01")
my.xts = xts(myts, dates)
plot(my.xts)
```



2.2 Data packages

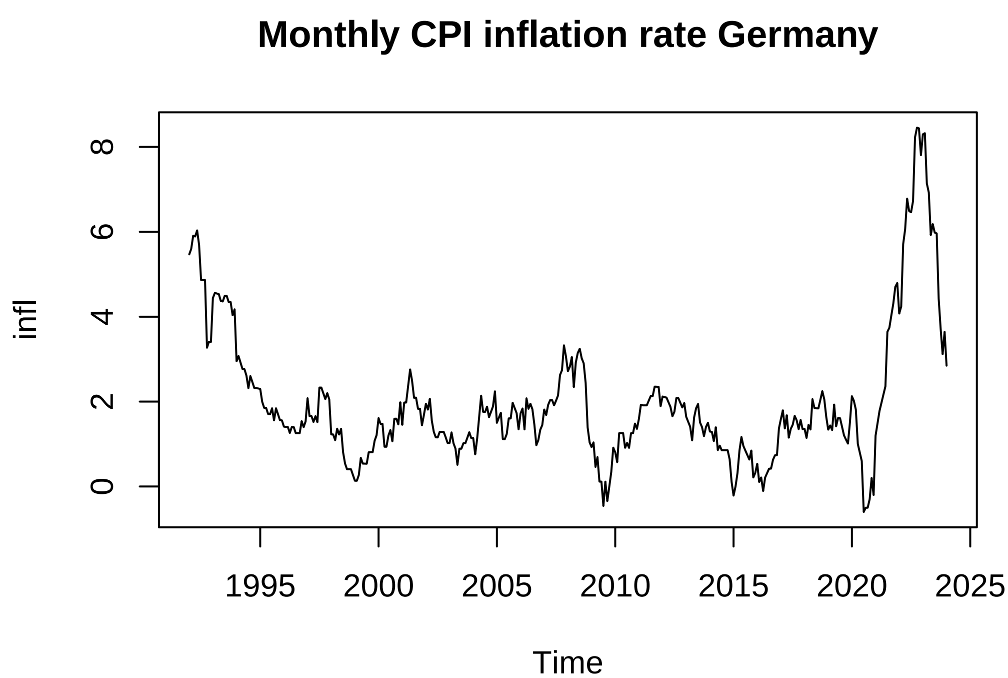
For teaching, I have created the package `teachingdata`, which contains some current datasets. The package is not available on CRAN (your package must meet specific quality standards and go through a review process to be accepted there), but I have created a GitHub repository to make it accessible. We need the package `remotes` and its function `install_github()` to install a package from a GitHub repository.

```
install.packages("remotes")
remotes::install_github("ottosven/teachingdata")
```

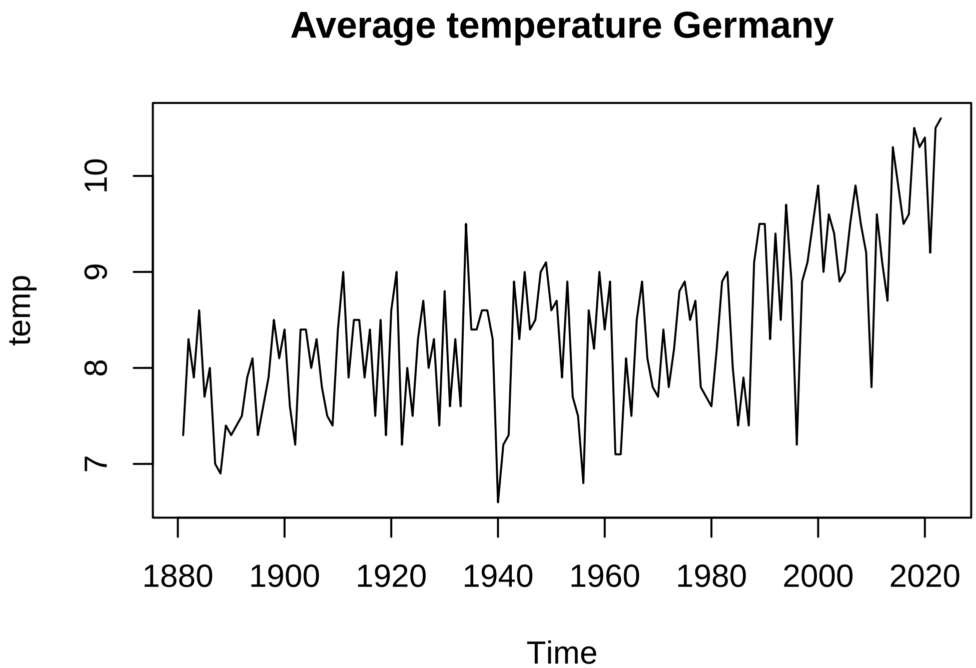
Let's have a closer look at the data from the `teachingdata` package.

```
library(teachingdata)
data(package = "teachingdata")
plot(gdp, main = "Quarterly GDP Germany")
```

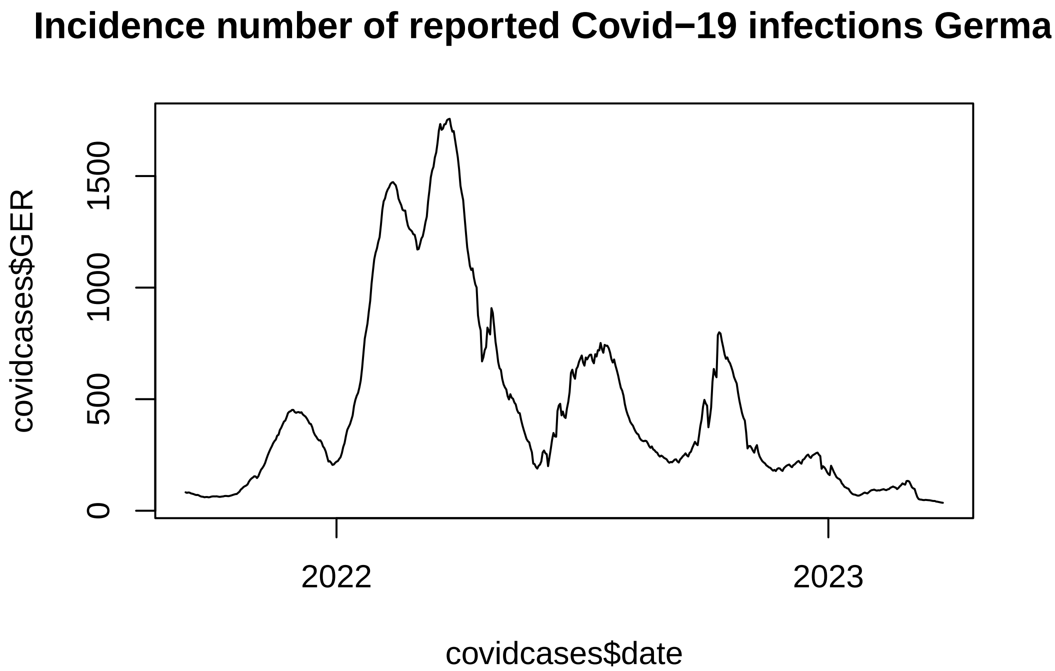

```r
plot(infl, main="Monthly CPI inflation rate Germany")
```



```r
plot(temp, main="Average temperature Germany")
```




```r
plot(covidcases$date, covidcases$GER, type="l",
     main="Incidence number of reported Covid-19 infections Germany")
```



2.3 The `tidyverse`

The `tidyverse` is a collection of packages that lets you import, manipulate, explore, visualize, and model data in a harmonized and consistent way.

Installing the `tidyverse` package:

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

In this lecture, we will mainly use R to theoretically understand the learned statistical and econometric methods and apply them illustratively. For this purpose, base R is entirely sufficient. However, `tidyverse` has become state of the art for applied work with large data sets and is especially recommended for data management and visualization.

To give you a flavor of the tidyverse, let us briefly discuss the `ggplot2` and `tibble` packages, which are part of the `tidyverse`.

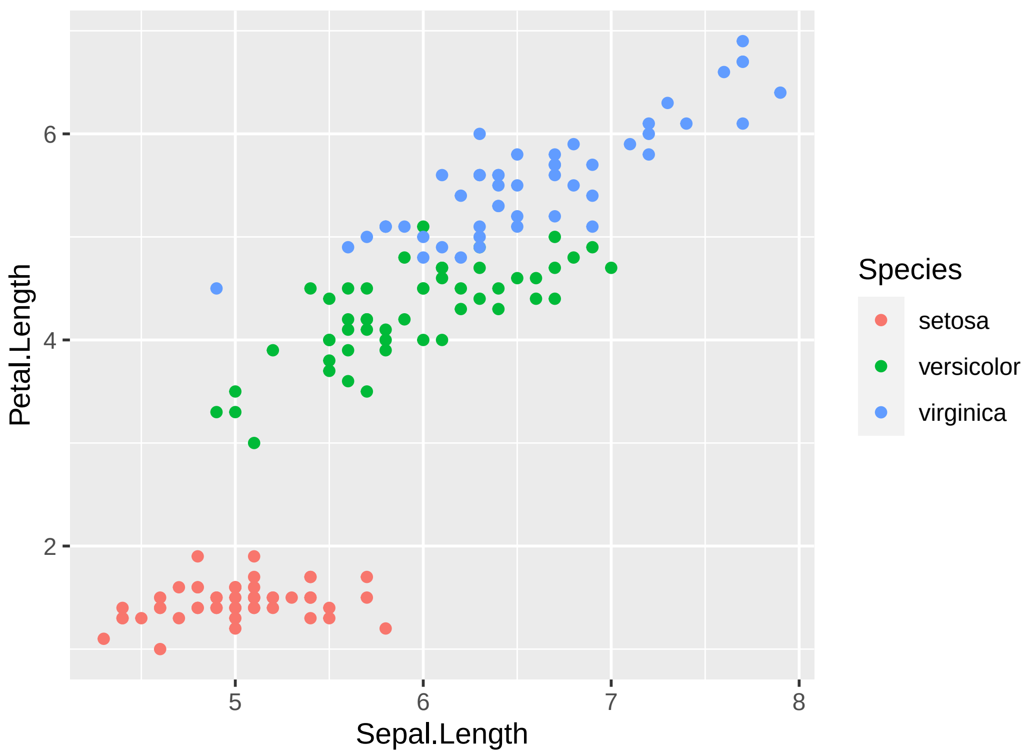
```
library(tidyverse)
```

Nice plots can be produced using the R-package `ggplot2`. Let's plot the `iris` dataset, which is contained in base R.

```r
class(iris) # iris is a data.frame
```

```
[1] "data.frame"
```

```r
iris |>
  ggplot(aes(x = Sepal.Length, y = Petal.Length, color = Species)) +
  geom_point()
```



A `data.frame` in the tidyverse is called `tibble`. A `tibble` is sometimes more flexible and convenient for manipulating and printing data. Let's transform the iris data frame into a tibble.

```r
iris.tbl = as_tibble(iris)
iris.tbl # iris.tbl is a tibble
```

```
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
```

```
 1          5.1          3.5           1.4          0.2 setosa
 2          4.9          3             1.4          0.2 setosa
 3          4.7          3.2           1.3          0.2 setosa
 4          4.6          3.1           1.5          0.2 setosa
 5          5            3.6           1.4          0.2 setosa
 6          5.4          3.9           1.7          0.4 setosa
 7          4.6          3.4           1.4          0.3 setosa
 8          5            3.4           1.5          0.2 setosa
 9          4.4          2.9           1.4          0.2 setosa
10          4.9          3.1           1.5          0.1 setosa
# i 140 more rows
```

As an extension, a `tsibble` object is a `tibble` with an additional time series structure. It contains a specific *index* variable corresponding to the observation's time index. Let us convert the `covidcases` data into a `tsibble`. To visualize a `tsibble` we also need the `fable` package.

```
library(tsibble)
library(fable)
```

In a `tsibble` object, we can define so-called *key* variables, which define the subjects or individuals measured over time. Key variables also allow easy processing of panel data in R.

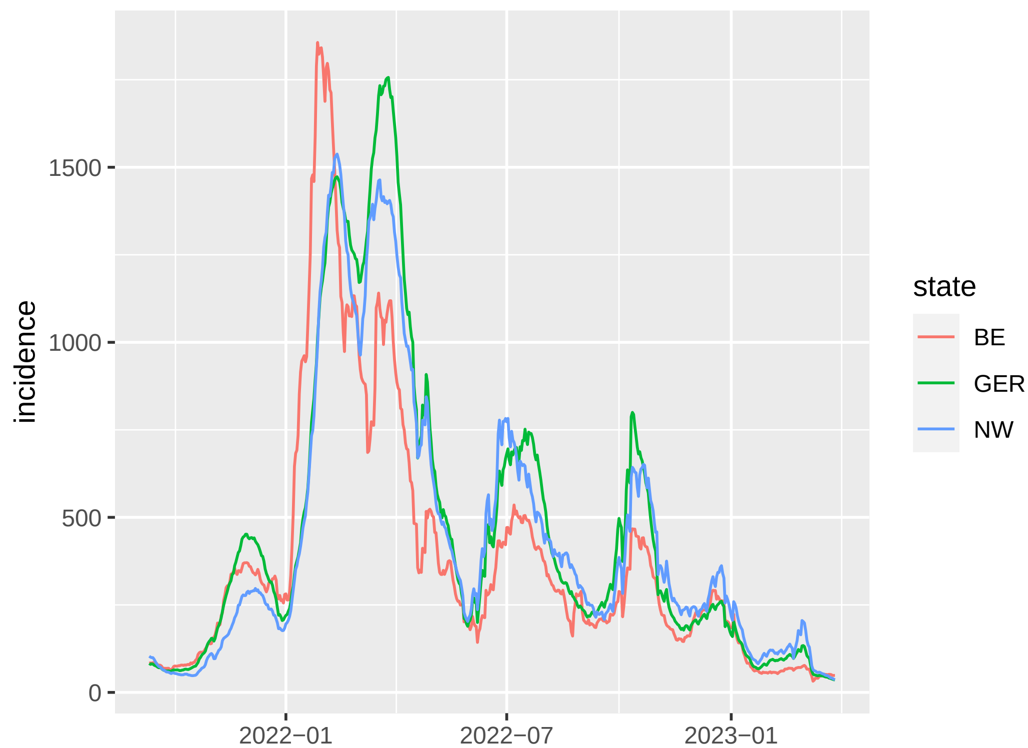
In the `covidcases` example, the key variables are the federal states, and the time series is the incidence numbers. Since a simultaneous display of the curves of all federal states would produce a very cluttered plot, we select only the total Germany, Nordrhein-Westfalen, and Berlin. The different steps can be represented in tidyverse as a sequence of multiple operations using the pipe operator `|>` (other pipes like `%>%`do a similar job).

```
covid.tsibble = as_tsibble(covidcases, index=date) |>
  pivot_longer(-date, names_to = "state", values_to = "incidence") |>
  filter(state %in% c("GER", "NW", "BE"))
covid.tsibble
```

```
# A tsibble: 1,689 x 3 [1D]
# Key:       state [3]
   date       state incidence
   <date>     <chr>     <dbl>
 1 2021-09-11 BE         83.5
 2 2021-09-11 NW        103. 
 3 2021-09-11 GER        82.7
 4 2021-09-12 BE         84.3
```

```
 5 2021-09-12 NW        101.
 6 2021-09-12 GER        80.1
 7 2021-09-13 BE         83.7
 8 2021-09-13 NW         99.3
 9 2021-09-13 GER        81.8
10 2021-09-14 BE         84.9
# i 1,679 more rows
```

```r
covid.tsibble |>
  autoplot(incidence) + theme(axis.title.x=element_blank())
```



For an introduction to the `tidyverse` and to learn more about the packages and functions used above, have a look at the book R for Data Science. To learn more about visualizing and analyzing time series data using the `tsibble` and `fable` packages, I recommend the textbook Forecasting: principles and practice.