```
# Description ------------------------------------------------------------

# In this script, we study how to read data from external sources into R.
# We then learn how to "tidy up" that data and combine it with other data.
# This means quite a bit of selecting and filtering and juggling with
# rows and columns.


# Getting SNB data on exchange rates -------------------------------------------------------

# To start, go to the data portal of the Swiss National Bank.
# Make Sure to switch to ENGLISH!
# Choose table selection and then foreign exchange market.
# Choose data from January 2001 to present, in csv format.


# Create a folder "Data" within the directory where you run your
# R project for this part of the course ("Data Science Fun/Programming").
# Put the csv file into that folder.
# The file name will be something like snb-data-devkum-en-selection-20170901_1430.csv
# Make sure to get the English version!

# Open the csv file first with Excel.
# In my case it looks pretty ugly.
# This is because my regional settings misinterpret the data formats.

# Now open the data with a text editor.
# Try to understand how this data is organized.



# Packages ---------------------------------------------------------------------



# We need
library(tidyverse)

# We also need the packages stringr. This is part of the
# tidyverse, but is not loaded automatically with the
# above command.
library(stringr)



# Assignment: Read SNB data ----------------------------------------------------------------


# YOUR TURN: Read the beginning of Ch. 8 (print) and try to get the data into R



# Reading data into R ----------------------------------------------------------------
```

```r
# Discussion and solution of above

# Good idea to always first clear all data that are
# still in the working space to start with a clean sheet:
rm(list = ls())

# First idea to read data:
read_csv("snb-data-devkum-en-selection-20171002_1430.csv")

# Second idea:
read_csv("Data/snb-data-devkum-en-selection-20171002_1430.csv")

# Third idea:
read_csv("Data/snb-data-devkum-en-selection-20171002_1430.csv", skip = 3)

# the argument skip refers to the number of rows that are not read into R.
# Check why this is important!

# Btw., you can get help by typing into the console
# ?read_csv or ?read_csv2 and hitting enter.
# The "commands" read_csv are in fact FUNCTIONS with ARGUMENTS!

# Fourth idea:
read_csv2("Data/snb-data-devkum-en-selection-20171002_1430.csv", skip = 3)

# Fifth idea
xRates0 = read_csv2("Data/snb-data-devkum-en-selection-20171002_1430.csv", skip = 3)

# Note: which function works (best) may differ from system to system. What works best
# for me may not work for you since I have US regional settings.

typeof(xRates0)
class(xRates0)

# So here we go again with a tibble!
# It's the usual data type for spreadsheet data.

# Check whether this compares well with how it looks in Excel!

# Alternative versions
test = read_delim("Data/snb-data-devkum-en-selection-20170901_1430.csv", ";", skip = 3)


# Remove test
rm(test)

# Briefly discuss here why R projects are important for reading data
# Close RStudio and reopen it. load the tidyverse and try to run
# xRates0 = read_csv2("Data/snb-data-devkum-en-selection-20170901_1430.csv", skip = 3)
# Does it work? If not, choose the Programming project from the upper-right corner.
# Does it work now?

# Another common situation: Close RStudio and reopen it. Run immediately
```

```r
# xRates0 = read_csv2("Data/snb-data-devkum-en-selection-20170901_1430.csv", skip = 3)
# Does it work? Why not?
# Believe me, these situations arise quite often ;-)


# Assignment: Read more data ----------------------------------------

# Use google to find the OECD data platform.
# Play around with the platform and try to understand how it works.
# Download some data that you find interesting.

# SOLUTION

# The data portal is at http://stats.oecd.org/.
# I selected data on Production and Sales (MEI). I got the data set
# MEI_REAL_29102017172534360.csv.

# In my case, it is comma-separated, there is a column header, but no lines to skip.
# oecd = read_csv("Data/MEI_REAL_29102017172534360.csv")


# Selecting rows and columns in a data frame ------------------------------------------------------------

# We go back to the SNB data on exchange rates. Have a look at the data.

# We want to select only Euro and USD, get rid of all the other exchanges rates.
# Then we want to bring the data into a "tidy" format.

# Read about manipulating and tidying tibbles/dataframes
# in the book Chapters 3 and 9.

# We have already carried out some important steps with
# the shopping list in 1_IntroCoding. But there is more to learn!
# You can read about manipulation of dataframes in Chapters 3 and 9.
# But it's a bit complicated, so we first do some important steps together.



# What columns do we have in our dataframe?
names(xRates0)

xRates0 %>% distinct(D0)

# A different way to get the same without the pipe is
distinct(xRates0, D0)

# Can  you see how the right-hand side of the pipe
# specifies the pricipal argument of the function?

# Another very useful function that dates back to
# before the tibble and tidyverse era is
unique(xRates0$D0)

# Note: unique() is not compatible with the new tidyverse functions.
```

```r
# The following does not work:
# xRates0 %>%
#   unique(D0)
#
# xRates0 %>%
#   unique(.$D0)


# Internally, xRates0 is a special type of list. Lists
# are objects that contain other objects. A tibble or
# data frame contains, as sub-objects, the vectors that
# make the columns of the data frame (tibble).
# These columns can have names. If they do, you can refer
# to them by <list name>$<column name>.
# You will not see this in the book, but it is nevertheless
# very useful in some instances.


# D0 is a very boring column that does not contain valuable
# info (it means that the data contains monthly averages.)
# So we just drop it...

# We go on using the pipe, so you get used to it.

# Drop D0
xRates <- xRates0 %>%
  select(-D0)

# Read about select on p. 51 - 53 in the book.

# What values are in D1?
xRates %>% distinct(D1)

# What is an alternative way to inspect what
# values we have in D1?




# Now we want to get rid of most of the rows.
# We choose to work only with the rows that belong
# to EUR and USD observations (the 1 in EUR1 etc. means
# that the exchange rate is *1* EUR per 1 CHF).

# For choosing rows, we use the filter() function.
# See p. 45 in the book.
xRates = xRates %>%
  filter(D1 %in% c("EUR1", "USD1"))

# The %in% should be understood in the sense of
# set theory. The idea is to choose only the rows
# the values of which are elements in the set
# {"EUR1", "USD1"}. See p. 48 in the book.
```

```r
# From long to wide ------------------------------

# In a tidy data frame, columns represent "variables" in the
# statistical (rather than programming) sense.
# Recap: What is a statistical variable?
# What is a variable in programming?


# Our data is NOT tidy! One single column
# mixes quite some info that we have a separate interest
# in: We are SEPARATELY interested in EUR and USD exchange
# rates, we do not want them "stacked" on top of each other.

# Rather, we want to have separate columns for EUR and USD
# and this means the data will get "wider" and only half as
# "long." Let's see how we get there (a bit tricky).


# The function that we need is spread()
# Read about spread in the book on p. 154-157.
xRatesWide = xRates %>%
  spread(key = "D1", value = "Value")

# The key argument is the column that contains the "labels"/names
# for the new columns that are going to represent *statistical*
# variables. Here, column D1 contains EUR1 and USD1

# The value argument indicates which column
# contains the values that should enter the
# respective rows of the new variables. Here,
# this column is already called "Value" (we already
# got the data with this name from the SNB).



# Strings, numeric date format ----------------------------------------

# Read about strings in Ch 11.

class(xRatesWide$Date)
# We have Dates as characters.

# Since we want to be able to use the
# dates for some numerical calculations, we convert them into
# numbers. For later use, we want to keep info about years and
# months separate.

# Use the function mutuate to create a new variable
# (and we continue using the pipe operator)
xRatesWide = xRatesWide %>%
  mutate(year = as.integer(str_sub(Date,1,4)))
```

```r
# The second line does a whole bunch of things at once.
# To get a better idea what it actually does, rip
# it apart and inspect the pieces. Unfortunately,
# this does not work well with the pipe, so we
# better use more traditional dataframe syntax.

str_sub(xRatesWide$Date,1,4)

class(   str_sub(xRatesWide$Date,1,4)    )
class(   as.integer(str_sub(xRatesWide$Date,1,4))   )

# So this works as intended.
# Note, you can type '?str_sub' in the console
# and hit enter to get help info on this function.


# Let's do the same with months (YOU do it!)

# The solution is:
xRatesWide = xRatesWide %>%
  mutate(month = as.integer(str_sub(Date,6,7)))

class(xRatesWide$year)
class(xRatesWide$month)

# Let's calculate a new column containing
# date information in a single numerical format

xRatesWide = xRatesWide %>%
  mutate(DateNum = year + (month-1)/12)

# Why do we subtract 1 from month??



# A first plot ------------------------------------------------------------

# Read about plotting with ggplot in Chapter 1 of the book.
# The below code looks exactly like on p.5 of the book.

ggplot(data = xRatesWide) +
  geom_point(mapping = aes(x = DateNum, y = EUR1))

# Very mean... but something like this may happen to you
# more often. Why does it not work? (Or does it, in your case?)

# Check the type of EUR1...
class(xRatesWide$EUR1)

# And change it (if necessary)

xRatesWide = xRatesWide %>%
  mutate(EUR1 = as.double(EUR1)) %>%
  mutate(USD1 = as.double(USD1))
```

```r
# Note: Double is the most precise
# numeric format (and integer is the least
# precise). Double makes sense for a variable
# like exchange rates. You can read more about
# the double format on p. 294 of the book.


ggplot(data = xRatesWide) +
  geom_point(mapping = aes(x = DateNum, y = EUR1))


# It's maybe more fun and interesting to plot
# the two exchange rates at once. For this, we
# use geom_line()

# Note: we are omitting "data = " and "x = "
# It still works.
ggplot(xRatesWide) +
  geom_line(mapping = aes(DateNum, EUR1)) +
  geom_line(mapping = aes(DateNum, USD1))

# This looks a little boring. Let's make it more colorful
ggplot(data = xRatesWide) +
  geom_line(mapping = aes(DateNum, EUR1), color = "red") +
  geom_line(mapping = aes(DateNum, USD1), color = "green")

# We will explore plotting im more depth in 3_plotting.R.


# Getting data on exports ----------------------------------------

# We want to analyze the relationship between exports and exchange rates
# Go again to the data portal of the SNB, choose "Economic Data" and
# "Foreign trade by goods category". Make sure to download data from the
# English site, so we all get the same data. Choose again data from Jan 2001
# until present in csv format. Download the data and put them again in the
# same data folder. In my case, the file is called snb-data-ausshawarm-en-selection-20170921_0900.csv.

# Read
trade0 = read_csv2("data/snb-data-ausshawarm-en-selection-20170921_0900.csv", skip = 3)




# Functions ------------------------------------------------------

# We need again transform the date info into a numeric format as above.
# Suppose we want to download even more data sets where we need to
# make this same calculations. It would be quite silly to every time
# do this with copy/pasting the respective code. Why?

# The programming structure that allows you to get a lot more efficiency
# is a FUNCTION. It's a flexible template of code that you can tweak
```

```
# to your specific use with the help of "arguments".

# so let us use a function to generate a variable with a numeric date info.


# Prepare the construction of a function.
# The arguments in functions need generic names that are
# "place holders" for the names of the variables that you actually want to use.
# It is common to use df as a placeholder for any dataframe
# that you may use as an argument of a function.

# Look at this code: We assign our dataframe
# trade0 the generic name df. (In a way, that's what
# happens if you pass trade0 as an argument into a function.)
df = trade0

# And we assign a particular column that we are
# interested in a generic name col.
col = df[["Date"]]
# We introduced this in the context of lists
# at the end of 1_IntroCoding.

# Now we simply use the code that we developed above:
df = df %>%
  mutate(year = str_sub(col,1,4),
         month = str_sub(col,1,4),
         DateNum = as.integer(year) + (as.integer(month)-1)/12)


# A function let's us to use the very same code
# as a template for any argument that may
# take on the role of df.


calc_numDate = function(df, col){

  col = df[[col]]

  df = df %>%
    mutate(year = as.integer(str_sub(col,1,4)),
           month = as.integer(str_sub(col,6,7)),
           DateNum = as.integer(year) + (as.integer(month)-1)/12)
  return(df)
  # In this case, this is not strictly necessary. But
  # You will see below that it sometimes is... It tells
  # the function to "give back" df as the output of the
  # function. Put differently, it means: "please give back the *entire*
  # data frame df, and not just the new column Date (or so).
}

# So this is our template that we can use
# with any suitable data frame. Let's
# check it out:
```

```r
trade = calc_numDate(trade0, "Date")

# Inspect the trade dataframe to see what happened.


# Here is an even better way to specify
# this function. We assign the col-argument
# a default value.

calc_numDate = function(df, col = "Date"){

  col = df[[col]]

  df = df %>%
    mutate(year = as.integer(str_sub(col,1,4)),
           month = as.integer(str_sub(col,6,7)),
           DateNum = as.integer(year) + (as.integer(month)-1)/12)

  return(df)
}

# Note: You can see this very often in R functions.
# Check out ?read_csv. Can you see the many default
# arguments in the help info?

# The idea of a default is that it can be
# overwritten if the argument is provided
# explicitly.

# Check it out: We only need one single argument now:

test.Trade = calc_numDate(trade0)

test.xRates = calc_numDate(xRates0)


# Read more about functions in Chapter 15.


# Delete the test objects
rm(test.Trade, test.xRates, df, col)


# Assignment: Your first function ------------------------------------------

# Write a function that converts a column of a dataframe into a
# a double format. (The old column get's replaced and keeps the
# same name.) Apply it to the colum "Values" of the trade dataframe.
# Give the function a telling name. Use "Value" as a default argument
# for the column name.

# Hint: First figure out how you change the values of
# a column using the [[]] syntax of data frames. Do not
# use pipes, and do not use mutate()! (With pipes and mutate, the
```

```r
# problem is more difficult ;-)

# The solution is just one line, wrapped by the function syntax!

# Those with substantial background knowledge can try to do it also with "mutate".

# Here goes my SOLUTION (and the steps to develop it).

class(trade$Value)

#col = "Value"; df = trade

colToNum_DNW = function(df, col = "Value"){

  df[[col]] = as.double(df[[col]])
}

# DNW stands for "Does Not Work" (as intended) ;-)

colToNum = function(df, col = "Value"){

  df[[col]] = as.double(df[[col]])

  return(df)
}

test.colToNum_DNW = colToNum_DNW(trade)

test.colToNum = colToNum(trade)

# Can you see the difference?
# In the DNW version, the function only returns
# the column that we changed. This behavior is
# somewhat "buggy", but every language has some
# buggy behavior that is not always easy to anticipate.

# Now let's use the proper function

trade = colToNum(trade)

class(trade$Value)

# And let's clean up. Let's delete everything that starts with "test"

# Try to google this! I googled
# <"r how remove all objects starting with"> and used the first hit
# http://r.789695.n4.nabble.com/remove-multiple-objects-starting-with-same-name-td4418694.html

rm(list = ls(pattern = "test"))

# You can also remove functions
rm(colToNum_DNW)
```

```
# Loops via sapply() ------------------------------------------------------

# Now we go on with our trade data set (data on exports).
# Let's try to figure out what type of information we have in there.

# Specifically, let's try to figure out what the letters in the columns mean
trade %>% distinct(D0)

# Go to the German version of the SNB site where you downloaded the data...
# After some detective work, you may get a clue...

# We want to give these {E, A, H} better names.
# Note that this is a combination of a "conditional" task
# and a loop.

# Here is the conditional part. It's a bit like
# If the value is "E", replace it with (say) "Imp".
# If the value is "A", replace it with "Exp". Etc.

# And here is the problem from the loop perspective:
# Loop through all letters in D0. For the first letter,
# change the value to "Imp", for the second, change it to "Exp" etc.

# It would be quite cumbersome to programm such a thing explicitly.
# So R has some special functions that do it for us.

# The "conditional" part is done by the switch function below.
# the loop is handled by the sapply function.

# Here is the code:


trade <- trade %>%
  mutate(D0 = sapply(trade$D0, switch,
                     E = "Imp",
                     A = "Exp",
                     H = "TradeBal") )

# The mutate(D0 = ...) function tells R to change the values of the column D0;
# sapply(trade$D0, ...) tells R to loop through all the values of
# trade$D0 (i.e. E, A, H) and apply the function SWITCH to these.
# Finally, the function switch contains the information about how
# to change values.

# This is very dense coding!! The first time you see this, it
# feels pretty inpenetrable. But reread the above paragraph,
# and one day it will look pretty straightforward! :-)


# You can read more about switch on p.278 (Chapter 15).
# The book advises us to use "map" insteady of sapply,
# but it does not work. This is a bug of the new tidyverse.
```

```r
# To better understand sapply, consider the following:


switch("E", E = "Imp",
           A = "Exp",
           H = "TradeBal")


switch(c("H", "E", "A"), E = "Imp",
       A = "Exp",
       H = "TradeBal")

#It is exactly because the previous does not work that
# we need to use sapply. It "loops"  through the list (character vector).
sapply(c("H", "E", "A"), switch,
       E = "Imp",
       A = "Exp",
       H = "TradeBal")


# map_chr (suggested in the book), does not work here:
map_chr(c("H", "E", "A"), switch,
       E = "Imp",
       A = "Exp",
       H = "TradeBal")

# sapply() is a somewhat advanced function that feels overwhelming for
# beginners. I suggest you just try to understand the idea of the
# above code and then use it via copy/paste for similar cases.
# In other words, just use it like a recipe!


# Grouping ----------------------------------------------------------------


# Let's turn back to our data and proceed with the other columns

trade %>% distinct(D1)

# Get the meaning of those values from the
# Excel version that you can downloaded from the SNB data portal.
# It involves quite some detective work!

# There are, weirdly, two values of TBS, i.e. TBS0 and TBS1
# Let's see, whether there is any important difference:

trade %>% group_by(D0) %>%  distinct(D1)
# You can read more about group_by in Chapter 2.


# What can you infer from this?
```

```
# Changing values of trade$D1 ---------------------------------------



# Our first reaction is certainly to copy and paste
# our sapply code from above and modify the labels:
#
# trade <- trade %>%
#   mutate(D1 = sapply(trade$D1, switch,
#                      T = "All",
#                      MAE = "Machines",
#                      PUB = "Precision",
#                      C = "Chemistry",
#                      TBS0 = "Clothing",
#                      #F = "Vehicles",
#                      M = "Metals",
#                      U = "Watches",
#                      P = "Precision (narrow)",
#                      TBS1 = "Clothing"
#                      ) )

# It does not work :-( Well, at least not if we include the F...

# To see the problem, run this...
switch(c("F"), F = "Vehicles")

# and this...
sapply("F", switch,
       F = "Vehicles")

# and compare it to this...
sapply("G", switch,
       G = "Parking slots")

# Can you guess why R has a problem here?
# Note that T works... So this is somehow a little buggy behavior...

# WELCOME TO REAL-WORLD MESSY DATA-PREPROCESSING!!! ;-)

# FIRST RERUN CODE until to before the section
# "Changing values of trade$D1".
# Comment that part of the code out and then continue
# below.


# So let's first change the F
trade <- trade  %>% mutate(D1 = replace(D1, D1 == "F", "Vehicles"))

# the replace() function allows us to specify a condition
# for the replacement.


# Now, the sapply code from above should work...
```

```
trade <- trade %>%
  mutate(D1 = sapply(trade$D1, switch,
                     T = "All",
                     MAE = "Machines",
                     PUB = "Precision",
                     C = "Chemistry",
                     TBS0 = "Clothing",
                     Vehicles = "Vehicles",
                     M = "Metals",
                     U = "Watches",
                     P = "PrecisionNarr",
                     TBS1 = "Clothing"
  ) )


# Finally, let's inspect the D2 column

trade %>% distinct(D2)

# We could change it with the below code. But
# For the moment we just leave it, so this code is
# out-commented.


# trade <- trade %>%
#   mutate(D2 = sapply(trade$D2, switch,
#                      WMF = "Level (nominal)",
#                      N = "Nominal change (in \u0025)",
#                      R = "Real change (in \u0025)"
#   ) )
#

# The reason we do not use this code now is that it would
# be an obstacle below when we change the data from
# long to wide. It would lead to excessively long
# column names.



# Assignment: Trade data from long to wide ------------------------------------------

# Again, we want to bring the trade data in a format such
# that the info in one column corresponds to a
# variable in the statistical sense. The goal is to have
# one column for every possible combination of values in
# the columns D0/D1/D2!

# Take a sheet of paper and try to figure out which combinations
# these are!

# Consider the following
unite(trade, key, D0, D1, D2)
```

```r
# Use this to bring the trade dataframe into a tidy format
# (YOU do it!)

tradeWide = trade %>%
  unite(key, D0, D1, D2) %>%
  spread(key = "key", value = "Value")



# Join tradeWide and xRatesWide ------------------------------------------------

# We want to analyze how exchange rates affect exports.
# For this, it is convenient to have exchange rates and
# trade data in a single dataframe.

# For this, we use the inner_join() function


D <- tradeWide %>%
  inner_join(xRatesWide, by = c("Date", "year", "month", "DateNum"))

# You can read about joins in Chapter 10.

# D is a TIDY data set!!!
# Read about tidy data in Chapter 9, pp. 148-151.



# Let's get rid of the 1 ins USD1 and EUR1
# (Note, these are now column names, no longer
# values in the data frame!)

D = rename(D, USD = USD1, EUR = EUR1)



# Changes in Xrates ------------------------------------------------------

# You may have been relieved that, finally, we have all
# the data that we need in one data set, and the data is tidy!
# But there is still a little more work to do.

# When we look at exchange rates and exports, we are often not interested
# in how the level of exchange rates (e.g. 1.60 CHF/EUR) relates to the level
# of exports (e.g. 15 bn CHF). Rather, we want to know how a
# (percentage) CHANGE in exchange rates is associated with a
# (percentage) CHANGE in exports.

# For this, we calculate rates of changes. In the
# econ-stat slang, this is often called a "growth rate".

# R has a lag() function. To check what it does consider

(test.y = 2001:2017)
```

```
(lag(test.y))

# check also this
lag(1:10)



D <- D %>%
  mutate(EUR_L = lag(EUR),
         gEUR = (EUR / EUR_L-1)*100)

# Since we need this more often, let's write a function
# that converts the column of a data frame into growth rates

# YOU DO THIS!

# Here is my solution

# Just for developing the function
colName = "EUR1"; df = D

toGrowth = function(df, colName){

  lagged = lag(df[[colName]])
  gRate = (df[[colName]] / lagged -1)*100

  return(gRate)

}

# Quiz: If we apply this function to a column in a
# data frame:
# 1) How many new columns are created?
# 2) What are the names of these columns

D = D %>%
  mutate(test = toGrowth(D, "EUR"),
         gUSD = toGrowth(D, "USD"))

D$test = toGrowth(D, "EUR")

D$gUSD = toGrowth(D, "USD")

# remove variables that we do not want
D = select(D, -test, -EUR_L)




# Export/Save ------------------------------------------------------------

write_csv(D, "data/TradeEx_tidy.csv")
save(D, file = "data/TradeEx_tidy.RData")
```