EFR summary

Programming (AAC), FEM71003

2025-2026

Lectures 1 to 4

Weeks 1 to 4

Deloitte.

DeNederlandscheBank
EUROSYSTEEM

Details

Subject: Programming 2025-2026

Teacher: P Breuer

Date of publication: 22.11.2025

Koffietje doen?

Start jouw carrière bij BDO

Maak kennis met BDO Accountants & Adviseurs, de beste plek om als toptalent aan de slag te gaan. De koffie staat voor je klaar. Vertellen wij je over wat jij kan bijdragen, en jij ons over je ambities.

Scan de QR-code en plan jouw koffiemoment in.

werkenbijbdo.nl ▶

BDO

Programming - lecture 1

Learning objectives

You will learn how to:

- Write R scripts to import and explore data
- Use data tidying techniques and visualization tools
- Apply data transformation techniques
- Perform regression analysis and write basic loops and functions
- Apply string manipulation techniques and regular expressions
- Employ textual analysis techniques

How to leverage AI for learning programming:

- Code explanations
- Debugging assistance
- Practice problems
- NOT replace your own coding skills, AI is here to assist.

R and RStudio: coding basics

Learning to program is like learning a new language, it takes practice and patience. Programming skills are transferable to other analytics tools (tools like Java, SQL, Python etc.)

What is **R**?

- Programming language for statistical computing and graphics.
- Open source and free software

What is **RStudio**?

- Integrated development environment (IDE) for R.
- Open source and free software
- Helpful features
    - User-friendly interface, code autocompletion and exhaustive help on any object.

In the interface of RStudio there are 4 windows that are aligned on the screen as follows:

1. Script editor Enter commands and save to file	3. Workspace View active objects and history of commands executed
2. Console Enter commands and see output	4. Files/Plots/Packages/Help Open files, view plots, install and load packages or use the help function.

You can use R as calculator, by simply typing the calculation you want to perform.

- R also knows math functions like square root (=sqrt())
- R commands will be shown alongside their resulting outputs, if entered in the R console.

R packages

What is an R package?

- Collection of functions, data and documentation that explains the package
- It adds useful and extra functionality not included in base R

Download and install a package

- install.packages(), e.g., *install.packages("tidyverse")*
- You only need to install a package once

Load the installed package

- library(), e.g., *library(tidyverse)*
- You need to load a package each time you open RStudio
    - Makes all its functions available to use
    - Coding rule: Load the packages at the start of your R session or script

Objects

R can store information as an object. This can be everything from a whole dataset to a certain text (=string variable)

- Use the assignment operator (<-) to assign some value to an object.
- Object_name <- value
    - E.g. x <- 15 or y <- "Hi, welcome to the first programming lecture."
- The created object will appear in your workspace window.
    - Coding rule: Use informative names for files, variables, and functions.
- Assigning a new value to an existing object overwrites. The previous value.

R recognizes different classes of objects.

- Assigning object into classes allows R to perform class-specific operations
- Use class() to check the class of an object.
    - Classes can vary from "numeric", "character" to "function" (=e.g. sqrt)

Vectors
A one-dimensional collection of elements, all of the same data type

- They enable efficient data manipulation

Create vectors

- c() function, which stands for "concatenate"
- Commas separate different element of a vector
    - E.g. c(TRUE, FALSE, True, NA)

Types of vectors (typeof())

- logical vector: FALSE, TRUE, and NA (e.g. c(TRUE, FALSE, TRUE, NA))
- Numeric vector: Numbers (e.g. c(1, 2, 3))
- Character vector: Strings (e.g. c("NL", "BE", "DE"))

Access specific elements of a vector ("indexing")

- By using []
    - E.g. b[1] in vector b <- c(TRUE, FALSE, TRUE, NA) gives TRUE.

Combine multiple vectors into one by simply putting the vectors you want to add into a new vector.

Avoiding errors

Encountering errors is part of the learning process, you'll need to learn how to fix these errors.

- Error: object not found
    - R is case sensitive, so always check your spelling.
- Error: incomplete expression
    - When you don't finish an expression you'll get the + prompt.
    - Press ESC if you want to cancel the command being evaluated.

If you at some point don't know what to do for a specific function you can always use the help function.

- E.g. help(class())

Or use google or chatgpt for example.

Data import

The working directory is the location on your computer where R loads data from and saves data from.

- To display the current working directory: getwd()

You can set a working directory where to store all your data on the computer.

- Use: setwd()
- Relative paths are flexible if you move your project folder.
  - .. : parent directory (e.g., C:/Users)
  - . : current directory (e.g., C:/Users/Programming)
- Absolute paths specify the full location (e.g. C:/Users/Programming/Week 1)
- Use / as path separator

Tidyverse

There are many ways to write code in R that produce the same results

- The tidyverse is like a dialect or syntax style of R programming
- It provides a more efficient and elegant way to work with data
- It is build for data science tasks: importing, cleaning, and transforming

To load files in R you'll need the readr package (=part of tidyverse). For different filetypes you'll need different functions

- Txt. Files: read_delim()
- Csv. Files: read_csv() (comma delimited) and read_csv2() (semi-colon delimited)
  - The col_names argument is standard set at TRUE, this means that the first line of the file being read contains the variable names.
  - If this is not the case, you can use skip = n to skip the first n lines
  - Use comment = "#" to drop all lines that start with #

A **tibble** or **data.frame** object is a two-dimensional table with rows and columns, where each column can be of a different data type

- Use view() to view a table-like representation of *tibble* or *data.frame* objects.

Data export

Use the function write_csv() to save data as a CSV file

- E.g. *write_csv(name dataset, "Filename")*
- Use a new filename if you don't want to replace the original file

To view the first rows of data you can use: head()

- E.g. *head(name dataset)*
- If you only want to see the first n rows you can add it to the function
    - E.g. *head(name dataset, n)*

To view the last rows of data you can use: tail()

- E.g. *tail(name dataset, n)*

Summarizing data characteristics

To get insights into each variable's distribution and characteristics use summary()

- This helps in understanding variable types, ranges, and missing values
    - E.g. *summary(name dataset)*
- Check 1 specific column of the data by using $
    - E.g. *summary(name dataset$name column)*
- Check multiple columns of the data by using []
    - E.g. *summary(name dataset [1:5])*
    - This shows the summary of column 1 to 5.

Dataset structure

The dim() function checks the dimension of the dataset

- E.g. dim(name dataset)
- It returns 2 values
    - The number of observations (rows)
    - The number of variables (columns)

The str() function checks the structure of the dataset

- E.g. str(name dataset)
- It returns a summary of the dataset, such as:
    - Number of observations (rows) and variables (columns)
    - Data types of each column (e.g. integer, numeric, character, factor)
    - The first few values in each column.

R markdown

R markdown is a plain text file format that combines code, text, and results in a single document. It support various output formats, such as PDF and Word, and it generated high-quality, fully reproducible reports for easy sharing and collaborating. To create a new R Markdown document you do the following: File -> New File -> R Markdown. To generate reports in PDF, Word or HTML format you use the **knit** function in the toolbar.

An R Markdown document contains **3 types of content**

1. YAML header surrounded by: ---
    - Specifies title, author, date and output format.
2. Chunks of R code surrounded by: ```
3. Text with simple text formatting (#=first level header, ##=second level header)

Run code in R Markdown:

- Insert a code chunck: Ctrl/Cmd+Alt+I
- Run current code line: Ctrl+Enter
- Run current code chunk: Ctrl+Shift+Enter

Chunk options

- Echo=FALSE -> show results, not code in finished line
- Eval=FALSE -> don't run the code
- Include=FALSE -> run the code, but code & result aren't shown
- Message=FALSE -> prevent messages in finished line
- Warning=FALSE -> prevent warning in finished file
- Results='hide' -> hide printed output
- Fig.show='hide' -> hide plots

Lecture 2

Tidying data

A large portion of data analysis time is spent cleaning and preparing data. Tidy datasets are easy to model, visualize and organized in a specific structure. But the main goal of tidying data is to store data values in a uniform way to facilitate analysis.

There are **3 interrelated rules** which makes a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

All packages in the *tidyverse* are designed to work with tidy data. You can tidy up messy data with the *tidyr* package (is part of *tidyverse*).

Pivoting

Pivoting a table allows to change the form of your data without changing any of the values. You can pivot from wide to long: pivot_longer(), or from long to wide: pivot_wider().

Long

i	j	Grade
1	1	8
1	2	4
2	1	6
2	2	7

Wide

i	Grade1	Grade2
1	8	4
2	6	7

Wide to long

- Pivot_longer(data, cols, names_to="name", values_to="value")
  - Data: data frame
  - Cols: columns to pivot into a longer format
  - Names_to: name of the column created from the data stored in the column names
  - Values_to: name of the column created from the data stored in cell values.

Pivot_longer(table, c("2024","2025"), names_to="Year", values_to="Grade")

ID	2024	2025
1	8	4
2	6	7

ID	Year	Grade
1	2024	8
1	2025	4
2	2024	6
2	2025	7

Long to wide

- Pivot_wider(data, names_from="name", values_from="value")
  - Data: data frame
  - Names_from: name of the column to get the names of the output columns from.
  - Values_from: name of the column to get the cell values from.

Pivot_wider(table, names_from="Year", values_from="Grade")

ID	Year	Grade
1	2024	8
1	2025	4
2	2024	6
2	2025	7

ID	2024	2025
1	8	4
2	6	7

Separating and uniting columns

Sometimes it can happen that there's multiple data in one cell (e.g. lastname/ firstname) or if you want to combine two cells. Then you can use one of the two following codes:

- Separate(data, col, into, sep)
  - Data: data frame
  - Col: column to separate
  - Into: names of new columns
  - Sep: separator between columns
  - E.g. *separate(table, Name, into=c("Lastname", "Firstname"), sep="/")*
- Unite(data, col, columns, sep)
  - Data: data frame
  - Col: name of the new column
  - Columns: columns to unite
  - Sep: separator between values
  - E.g. *unite(table, col=Name, c("Lastname", "Firstname"), sep="/")*

Handling missing values

Datasets often have missing values (NA). If you ignore these missing values this can lead to biased analyses or broken visualizations.
To detect missing values, you can use 2 functions:

- Is.na()

    - This tests each element for missing values (Gives TRUE if it's missing)
- Sum(is.na())
    - Counts the number of missing values.

You can drop the rows with missing values using drop_na().
To replace missing values you can use replace_na()

Data visualization: Tables

Use the stargazer package to produce high-quality summary and regression tables. Tables can directly be exported to LaTeX, HTML or text formats. This is especially useful when working with R markdown.

Summary table
Descriptive statistics in text format, results can be seen in R console.

- e.g. *stargazer(as.data.frame(df), type="text", title="new title", digits = 1, out="table.txt")*

  To ensure that the LaTeX table code is directly inserted into the document without being altered you have to put: results = "asis" into the chunk.

Data visualization: Graphs

In R you can create graphs with the ggplot2 package (part of tidyverse).
To make a graph you use the following functions:

- Define the dataset to use in the graph
    - ggplot() -> e.g., ggplot(data=df)
- Specify which variables to map to the x and y axes
    - aes() -> e.g., ggplot(data=df, mapping=aes(x=income, y=experience))
- Define the plot to draw
    - geom_function() -> e.g., ggplot(data=df, mapping=aes(x=income, y=experience)) + geom_point()
- Save the graph to current working directory
    - ggsave() -> e.g., ggsave(filename = "Figure 1.png")

Common types of graphs:

- geom_point(): scatter plot
- geom_histogram(): Histogram
- geom_line(): lines/time series plot
- geom_bar(): bar plot
- geom_boxplot(): Box plot

- geom_density(): Density estimate
- geom_smooth(): fitted regression line

Example: Time series plot

The geom_line() function draws a connected line. You can also filter the data with the following function: filter(data, condition)

Advanced graphics

You can also ad some aesthetics to the graph. The most common are:

- color -> color for point/lines
- fill -> fill color for areas
- shape -> symbol for points (square, round, triangle etc.)
- linetype -> type of line (solid, dashed, dotted etc.)
- size -> sizo of points/lines

The labs() function allows you to set labels for multiple plot elements all at once.
It is also possible to work with multiple layers in your code. In this way you can show a scatterplot and a regression line in the same graph. To do so you can just type the codes on top of each other.

Global vs local aesthetics

- Global
    - Aesthetics are globally defined for the entire plot. Mappings for x, y and color apply to all layers in the plot.
    - *ggplot(data=df, mapping=aes(x=income, y=experience, color=age)) +*
      geom_point() +
      Geom_smooth()
- Local
    - Aesthetics are locally defined within each layer
    - *ggplot(data=df) +*
      geom_point(mapping=aes(x=income, y=experience, color=age)) +
      Geom_smooth(mapping=aes(x=income, y=experience))

Lecture 3

Data transformation

Transform data with the *dplyr* package, which is part of *tidyverse*
Basic grammar of data transformation

- Operation on **rows**
  - filter(): picks rows of a table based on their values
  - arrange(): orders rows based on values of a column(s)
- Operation on **columns**
  - select(): picks columns of a table based on their names
  - mutate(): adds new columns that are functions of existing columns
- Operation on **groups**
  - summarize(): aggregates multiple rows down to a single summary
  - group by(): divides a table into groups

Filter

filter(data, ….): picks rows of a table based on their values that meet a certain condition.

- data: data frame
- ….: Condition(s) that must be true to keep a row
- E.g. *storms_a<-filter(storms, wind >=50)*

Logical and Boolean operators to use with filter()

==	equal to	is.na()	is missing
!=	not equal to	!is.na()	is not missing
<	less than	%in%	in a set of specific values
>	greater than	|	or
<=	less than or equal	&	and
>=	greater than or equal		

Arrange

arrange(data, column1, column2, ….): orders rows based on values of a column

- Data: dataframe
- Column1, column2: Columns by which to arrange the data
- desc(): descending order (ascending is standard)
- E.g. *storms_b<-arrange(storms, wind)*

Select (same as filter function but now for rows, easy reminder select has a c of columns and Filter has a r of rows)

select(data, columns_to_select): picks columns of a table based on their names

- Data: data frame
- Columns_to_select: columns to keep
  - Direct column names: column names, separated by commas
  - Range of columns: **:** to select a range of columns
  - Column exclusion: **-** to exclude columns

- E.g. *storms_c<-select(storms, -Date)* (Here you remove the column date)

Mutate

mutate(data, ….): adds new columns that are functions of existing columns

- Data: data frame
- ….: expressions that define the new columns (new_column_name = expression)
  - E.g. *storms_d<-mutate(storms, ratio=pressure/wind)*

If else

if_else(): create conditional variable

- Condition: logical vector
- TRUE: where condition is TRUE, matching values from TRUE
- FALSE: where condition is FALSE, matching values from FALSE
- E.g. *storms_e<-mutate(storms, red=if_else(wind>100, 1, 0))*

Summarize and group by

Summarize(data, ….): aggregates multiple rows down to a single summary

- Data: data frame
- ….: One or more expressions that calculate summary statistics (new_column_name = summary_function(column_name))
  - Summary functions: sum(), mean(), sd()

Group_by(data, ….): divides a table into groups

- Data: data frame
- ….: columns to group by

The pipe operator

- %>%
- This function allows to combine multiple function without always having to repeat for example the data frame.
- E.g. (In this case you don't have to repeat "table1" all the time in the functions)
  table1 %>%
  Group_by(Year) %>%
  Summarize(Mean_Case=mean(Case))

Ungroup

The function ungroup() removes existing grouping structure created by the group_by() functions

- If you don't use the ungroup() function averages are calculated **within** each group, when you do use it, averages are calculated **across** all rows in the dataset.

Row-wise

Row-wise operations require grouping of a row

- Built-in functions: rowSums(), rowMeans()
- To select which columns to include use: across()
- To handle missing values add: na.rm=TRUE
    - E.g. *total_score=rowSums(across(c(Column1, column2)), na.rm=TRUE)*

Joining data

Combine data with the *dplyr* package, which is part of *tidyverse*

Basic grammar for combining data

Operation on tables

- Append columns (datasets have the same number of rows, but different columns)
    - Bind_cols(x,y): combines tables by adding columns side-by-side
- Append rows (datasets have same columns but different rows)
    - Bind_rows(x,y): combines tables by stacking rows on top of each other.
- Mutating joins
    - inner_join(x,y): keeps observations that occur in both x and y
    - left_join(x,y): keeps all observations in x
    - right_join(x,y): keeps all observations in y
    - Full_join(x,y): keeps all observations in either x or y
- Filtering joins
    - semi_join(x,y): keeps all observations in x that have a match in y (but you'll only keep the observations from x; no new column is added)
    - anti_join(x,y): drops all observations in x that have a match in y (You'll only get the observations that aren't in y)

Keys: variables used to connect a pair of data frames in a join

- **Primary key**: variable(s) that uniquely identifies each observation.
- **Foreign key**: variable(s) that corresponds to a primary key in another table.

Duplicate values

Duplicate values can appear when merging, importing, or appending data

- Identical values across entire rows or within specific columns

Detect duplicates: duplicated()
- You'll get a logical vector: TRUE for duplicates

Count duplicates: sum(duplicated())
- It will sum up the TRUE values.

Locate duplicates: which(duplicated())
- It will give the row number of the duplicate.

Remove duplicates: distinct()

Dates in R

To work with dates and times in R, you'll need the *lubridate* package, which is part of the *tidyverse* package.

Depending on how the date is written, you can convert character or numeric date values into date objects.
- Y -> year
- M -> month
- D -> day

Different functions
- ymd(20221101) -> "2022-11-01"
- dmy(01112022) -> "2022-11-01"
- mdy("November 1st, 2022") -> "2022-11-01"

It is also possible to calculate how much time elapsed between two dates
- entry_date <- ymd("2022-06-01")
  implementation_date <- ymd("2024-01-15")
  implementation_date – entry_date

Time difference of 593 days
- You can also use as.numeric(x), this converts X into a number
  - as.numeric(implementation_date – entry_date)
    [1] 593

Get individual components from a date variable by using $ (output is numeric!)
- Get the year: *df$year <- year(df$date)*
- Get the month: *df$month <- month(df$date)*
- Get the day: *df$day <- mday(df$date)*

Create a date variable from individual components with make_date()

- Input = numeric
- Output = date data type
    - E.g. *df$date_new <- make_date(df$year, df$month, df$day)*

Lecture 4

Regression analysis

A regression analysis is a statistical method for analyzing a relationship between two or more variables. In other words, one variable can be predicted or explained by using information on the other variables.

- Idea: fit a function that closely represents the trend in the data.

The most common form of regression analysis is the **linear regression.** This fits a straight line to the data.

- $Y = \alpha + \beta X + \varepsilon$
    - Y: Dependent, response or outcome variable
    - X: Independent, explanatory or predictor variable
    - α: Intercept
    - β: slope coefficient
    - ε: error term
- It uses the ordinary least squares

In R you can also do a linear regression by using the lm(formula, data) function.

- Formula: Y ~ X (+…)
- Data: data frame

You can do both a **univariate** and a **multivariate** linear regression

- Univariate linear regression (=simple linear regression)
    - E.g. *unireg <- lm(sales ~ adverstising, data=sales)*
- Multivariate linear regression
    - E.g. *multireg <- lm(sales ~ advertising + rd, data = sales)*

The summary() function shows then the regression output.

```
Call:
lm(formula = distance ~ avg_quality + fr_orig + de_orig + sc_orig,
    data = country_data)

Residuals:
     Min       1Q   Median       3Q      Max
-24.1910  -2.8133   0.0713   3.4919   9.3159

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    8.315      2.246   3.702 0.000251 ***
avg_quality   -8.620      1.096  -7.866 5.47e-14 ***
fr_orig1      -2.023      1.495  -1.353 0.176948
de_orig1       2.124      1.517   1.401 0.162273
sc_orig1       1.801      1.580   1.139 0.255339
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.703 on 325 degrees of freedom
Multiple R-squared:  0.2846,    Adjusted R-squared:  0.2758
F-statistic: 32.33 on 4 and 325 DF,  p-value: < 2.2e-16
```

Call: model specification

Residuals: The difference between the observed and predicted values (ideally this should be symmetrically distributed around the line)

Coefficients: Significance testing (standard error, t-value and p-value)

Residual standard error: std. dev. of residuals

The function stargazer() can make a table out of the regression output.

- E.g. *stargazer(unireg, multireg, type="text", title="sales")*

It is also possible that you use categorical variables. In this case you can use a **logistic regression**. This fits a regression curve when Y is a categorical variable. These variables have a fixed and known set of possible values.

- Binary logistic regression (A or B)
- Multinomial logistic regression (A,B, C)
- Ordinal logistic regression (A<B<C)

In R the glm(formula, family, data) creates a logistic regression model.

- Formula: Y~X (+…)
- Family: probability distribution (for logistic regression: family=binomial(link="logit"))
- Data: data frame

Generalized linear models allow for non-normally distributed dependent variables.

Before you can execute the logistic regression, you will have to do some preparatory steps. It is important to transform categorical variables to factors. You can do this by using as.factor().

The estimated coefficients are in log odds scale -> log $(\frac{p}{1-p})$

Fixed effects

Fixed effects control for unobservable confounding factors that are constant within categories. It is a variable that helps rule out alternative explanations for the

relationship between the outcome and the explanatory variable of interest. What is it in 3 points:

- Fixed effects are dummy variables for each category
- Used as covariates in regressions to account for unobservable factors
- Add category-specific constants

3 types of fixed effects

1. **Unit fixed effects**
    - Account for fixed differences across units (e.g. industries, countries)
    - Basically, characteristics that vary across units but do not vary over time.
    - By including state fixed effects, you control for these time-invariant characteristics.
2. **Time fixed effects**
    - Account for common time trends
    - Characteristics that vary over time, but do not vary across units.
    - By including year fixed effects, you control for these common time-varying factors.
3. **Two-way fixed effects**
    - Regression model including both unit and time fixed effects
    - Two-way fixed effects design accounts for confounding factors that are either:
        - Constant over time but vary across units.
        - Constant across units but vary over time.
    - Two-way fixed effects allow you to isolate the within-state relationship between the dependent and independent variable.

In R you can include fixed effects as well. For this you'll need the lfe package.
Basic syntax:
felm(regression formula, FE dimension, instrumental variable, SE clustering dimension, data frame)

Panel data

3 types

1. Cross-sectional data
    - One point in time (t=1), many observation units (i=1, …, n)
2. Time-series data
    - Many time periods (t=1, …, T), one observation unit (i=1)
3. Panel data

- Many observation units (i=1, …, n) over time (t=1, …, T)

Loops

Loops is a tool for reducing duplication of effort by automatically repeating a set of operations several times. There are 3 types of loops:

1. **for** loop
2. **while** loop
3. Repeat loop

e.g. merging 50 data files into one dataset.

for loop

Repeat a task a defined number of times.

Basic syntax

- for(element in vector){
  expression
  }
- e.g. for(i in 1:5) {
  print(i)
  }
  This line of code will print i that is each value in the vector, so it will give you the numbers 1, 2, 3, 4 and 5

while loop

It repeats a task until a specific condition is met.

Basic syntax:

- while(logical condition) {
  expression
  }
- e.g. i <- 0
  while(i<=4){
  i <- i + 1
  print(i)
  }
  This will give the output 1, 2, 3, 4 and 5.
- If you don't add the condition i <- i + 1, the loop will run indefinitely because i will always be 0 and thus meet the condition.

Functions

R is full of **built-in functions**. These are predefined functions that are available in R to perform common tasks or operations. R does also allow you to write your own function (=**user-defined function**). This function is a set of statements organized together to perform a specific task.

Basic syntax:

- f <- function(argument(s)) {
  statements
  }
- This creates a function with name f which takes certain arguments and executes the following statements.
- E.g. create a function that adds two numbers:
  Add_num <- function(a,b) {
  Result <- a+b
  Print(result)
  }
- Call the function:
  Sum <- add_num(3, 4)
  This will give the result 7