UNIVERSITÄT DUISBURG ESSEN

Open-Minded

Advanced R for Econometricians

Introduction

Preliminaries

Contact Information: Jens Klenke
Jens.Klenke@vwl.uni-due.de

Martin Arnold
Martin Arnold@vwl.uni-due.de

Slides, exercises and announcements will be provided on

https://moodle.uni-due.de/course/view.php?id=17174

The password for the course is "**Hadley**".

On Moodle you will find a link to Datacamp which will give you free access to all their content.

Grading

- There will be three graded assignments.
- You will have to work on a final group project which focuses on one or more of the topics discussed in the course.
- The final grade is a weighted average of the grades for the group project (50%), its presentation (20%) and the assignments (30%).
- You have to pass each part (assignments, project and presentation) separately to pass the course.

Outline

About R

- R is a language and environment for statistical computing and graphics.
- R is highly extensible. There is a massive set of packages for statistical modelling, machine learning, visualisation, as well as importing and manipulating data. Researchers in statistics and machine learning often publish an R package to accompany their articles.
- R is free and open source.
- R provides tools for communicating your results. R packages make it easy to produce html or pdf reports, or create interactive websites.
- R is designed to connect to high-performance programming languages like C, Fortran, and C++.
- R also has some disadvantages:
  - It is not very fast.
  - It has an inconsistent syntax.

Sources

- Advanced R by Hadley Wickham
- R for Data Science by Hadley Wickham and Garrett Grolemund
- The Art of R Programming by Norman Matloff
- The Use R! Series by Springer
- The R Series by CRC Press

Packages

- In R the fundamental unit of shareable code is the package.
- Chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package.

The main sources for packages are

- CRAN (Comprehensive R Archive Network) with as of now ca. 15.000 available packages
- GitHub (that is the place where packages are developed before getting published on CRAN)
- Bioconductor

Installing Packages

How to install packages from CRAN?

- Use `install.packages("MASS")` to install the package `MASS` from CRAN.
- Or using RStudio: Tools → Install Packages → `MASS`

How to install packages from Github?

- Install the package `devtools` from CRAN.
- E.g., use `devtools::install_github("tidymodels/broom")` to install the `broom` package from Github.
- If you want to find out how a package works, search for
  - a vignette by the package authors

    ```r
    library(broom)
    vignette("broom")
    ```

  - a blog post about the package, e.g. on R Bloggers

Getting Help

If you have specific questions about your code or an error message look on

- StackOverflow
- RStudio Community (if it is about RStudio or packages developed by RStudio)
- the GitHub repository of the package maintainer

The chances are high that somebody else has already asked the same question and it has already been solved.

If you don't find an answer you can ask a question yourself. What sounds easy is harder then one would think. Most importantly you have to provide a reproducible example.

Rstudio

- RStudio is an integrated development environment (IDE) for R.
- It provides some convenient functionality compared to the IDE shipping with R.

Rstudio Projects

- To work in a specific directory the base R way would be to use `setwd()`. This is problematic if you want to share your code or work from different machines on the same project.
- Use projects instead.

Workflow:

- Instead of opening a new R session and use `setwd()` you open the project. The working path is automatically set to the folder where the `.Rproj` file lives in.

Some Shortcuts

Shortcut	**Action**
Alt + Shift + k	list all shortcuts
Cmd/Ctrl + Shift + F10	restart R
Cmd/Ctrl + Shift + S	rerun current script
F1	help page
Ctrl + l	clears the console

Rstudio supports:

- package development
- version control via git and svm
- creation and compilation of dynamic documents
- connections to external data bases
- ...

R Basics

- We start by recapitulating some basics you should be familiar with when working with R.
- An in-depth treatment of this material can be found in the first chapter of Advanced R by Hadley Wickham.

R Basics: Data Types and Data Structures

We assume that you know about the basic data structures in R.

	Homogeneous	**Heterogeneous**
1d	Atomic Vector	List
2d	Matrix	Data Frame
nd	Array	

You should also be familiar with the basic data types (character, double, integer, logical).

Quiz:

1. How is a list different from an atomic vector?
2. How is a matrix different from a data frame?
3. How do you find out the type of an object?
4. How can you check if an object is of type character?

R Basics: Subsetting

You should know how to use the common subsetting operators `[`, `[[` and `$`.

Have you ever encounterd `@`?

Quiz

1. What is the result of subsetting a vector with positive integers, negative integers, a logical vector, or a character vector?
2. What is the difference between `[`, `[[`, and `$` when applied to a list?
3. When should you use `drop = FALSE`?

R Basics: Control Flows

The basic control flows are conditionals (choices) and loops.

Quiz

1. What is the difference between `if` and `ifelse()`?
2. What will be the value of `y` in the following code if `x` is `TRUE`? What if `x` is `FALSE`? What if `x` is `NA`?

```
y <- if(x) 3
```

3. What is returned by the following code?

```
switch("x", x = , y = 2, z = 3)
```

4. Name three kind of loops that can be implemented in R! How do they differ?

R Basics: Functions

- Write a function that takes a number `x` and returns the square of `x`.

Scoping

- An important concept and also a source of trouble is scoping, the act of finding the value associated with a name.
- R uses lexical scoping: it looks up the values of names based on how a function is defined, not how it is called.
- Example for lexical scoping:

```r
f1 <- function() x
x  <- "global"
f2 <- function(){
  x <- "local_f2"
  f1()
}
f2()
```

Scoping Rules

R’s lexical scoping follows four primary rules:

- Name masking
- Functions versus variables
- A fresh start
- Dynamic lookup

Scoping Rules: Name Masking

- What happens here?

```r
x <- 3
square_1 <- function() x^2
square_1()
```

- And here?

```r
x <- 3
square_2 <- function(){
  x   <- 2
  foo <- function() x^2
  foo()
}
square_2()
```

- Names defined inside a function mask names defined outside a function.

Scoping Rules: Functions versus Variables

- Let's make things more confusing. Can you predict the result?

```
x <- 3

square_2 <- function(){
  x   <- 2
  foo <- function() x^2
  foo()
}

square_3 <- function(x){
  square_2 <- 5
  square_2()
}

square_3()
```

- When a function and a non-function share the same name, R ignores non-function objects in a function call.

Scoping Rules: A Fresh Start

What is the result of the first call to `square_4()`? What will happen the second time?

```r
x <- 2
square_4 <- function(){
  (x <- x^2)
}

square_4()
square_4()
```

- Every time a function is called a new ***environment*** is created to host its execution.
- A function has no way to tell what happened the last time it was run; each invocation is completely independent.

Scoping Rules: Dynamic Lookup

Predict the result of the two calls to `f1()`. Compare it to the main rule of lexical scoping.

```r
x  <- 1
f1 <- function() x
f1()

x <- 2
f1()
```

- R looks for values when the function is run, not when the function is created.

Lazy Evaluation

What could cause an error here?

```r
lazy_function <- function(x){
  10
}

lazy_function()
```

Why doesn't it cause an error?

- Function arguments are lazily evaluated; they’re only evaluated if accessed.

```r
lazy_function <- function(x){
  x
}

lazy_function()
```

Style Guide

Object names

- Variable and function names should be lowercase.
- Use an underscore (_) to separate words within a name (as an alternative you can use camel case, but be consistent).
- Variable names should be nouns.
- Function names should be verbs.
- Strive for names that are concise and meaningful (this is not easy!).
- Where possible, avoid using names of existing functions and variables.

```
    # Good
day_one
day_1
```

```
# Bad
first_day_of_the_month
DayOne
dayone
djm1
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

Spacing

- Place spaces around all infix operators (`=`, `+`, `-`, `<-`, etc.) and `=` in a function call.
- Always put a space after a comma, and never before.

```r
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

- An exception to this rule are `:`, `::` and `:::`.

```r
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

Spacing

- Extra spacing (i.e. more than one space in a row) is ok if it improves alignment of equal signs or assignments (`<-`).

```r
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

- Do not place spaces around code in parentheses or square brackets (unless there's a comma, in which case see above).

```r
# Good
if (debug) do(x)
diamonds[5, ]

# Bad
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

Spacing

- An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it's followed by else. Always indent the code inside curly braces.

```r
# Good
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Bad
if (y < 0 && debug) {
message("Y is negative") }

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

Spacing

- It's ok to leave very short statements on the same line:

```
if (y < 0 && debug) message("Y is negative")
```

- Strive to limit your code to 80 characters per line.
- If you find yourself running out of room, this is a good indication that you should encapsulate some of the work in a separate function.

Indentation

- When indenting your code, use two spaces and never use tabs or mix tabs and spaces.

Assignment

- Use `<-`, not `=`, for assignment.

```
    # Good
x <- 5
# Bad
```

Commenting guidelines

- Comment your code.
- Each line of a comment should begin with the comment symbol and a single space: `# `.
- Comments should explain the why, not the what.
- Use commented lines of `-` and `=` to break up your file into easily readable chunks.

```
# Load data ---------------------------

# Plot data ---------------------------
```

Exercises

1. Combine all elements of `df_list` to one data frame. The result should be only a single line of code.

```
df_list <- lapply(0:4, function(j) data.frame(c1 = (1:5) + 5 * j,
                                              c2 = letters[(1:5) + 5 * j]
                                              )
                  )
```

2. Fix each of the following common data frame subsetting errors:

```
    mtcars[mtcars$cyl = 4, ]
    mtcars[-1:4, ]
    mtcars[mtcars$cyl <= 5]
    mtcars[mtcars$cyl == 4 | 6, ]
```

3. What does `df[is.na(df)] <- 0` do?

4. How would you randomly permute the columns of a data frame?

Exercises

5. Write a `while` and a `repeat` loop doing the same as

```
for(i in LETTERS[1:10]){
  print(i)
}
```

6. Write a `for` loop doing the same as

```
count <- 0
repeat{
  x     <- sample(1:6, 1)
  count <- count + 1
  if(x == 6) break
}
print(count)
```

Exercises

7. What is the problem here?

```r
# Number of apples
i <- 100
for (i in 1:3) {}
paste("The number of apples is", i)
```

8. What is the problem here? Can you debug it?

```r
df <- mtcars

lin_mod <- function(){
  lm(y ~ . ,data = df)
}

simple_lin_mod <- function(data, y, x){
  df <- data[ ,c(y, x)]
  names(df) <- c("y", "x")
  fast_lm()
}

simple_lin_mod(df, "mpg", "cyl")
```