

# CODING PRACTICES

Life with R



Erik Kusch

[erik.kusch@au.dk](mailto:erik.kusch@au.dk)

Section for Ecoinformatics & Biodiversity  
Center for Biodiversity and Dynamics in a Changing World (BIOCHANGE)  
Aarhus University

25/03/2020

## 1 Quality of Life Improvements

- Coding in R
- Documenting Your Work

## 2 Code Structure

## 3 Sharing Work

- Reproducibility
- Functions & Sourcing

## 4 User-Friendliness

- Progress Bar
- Estimators
- Parallel Processing

# Objects

Assign **recognizable**, **concise** names to your objects!

Let's start with the object `MyData` - this can be anything. Let's assume it is a raster of NDVI values during the year 1982. How do we come up with a better name?

**Specificity** in your naming helps you keep track of your data in your code:

- NDVI1982

**Classes** of objects determine how they get handled. Add them as a suffix:

- NDVI1982\_ras

The same goes for **loop indicators**! Avoid single-letter indicators and instead use something like `Iter_Years` - an indicator *iterating* (`Iter`) over years.

Also, always use `<` – for assigning objects!

# Objects

Assign **recognizable**, **concise** names to your objects!

Let's start with the object `MyData` - this can be anything. Let's assume it is a raster of NDVI values during the year 1982. How do we come up with a better name?

**Specificity** in your naming helps you keep track of your data in your code:

- NDVI1982

**Classes** of objects determine how they get handled. Add them as a suffix:

- NDVI1982\_ras

The same goes for **loop indicators**! Avoid single-letter indicators and instead use something like `Iter_Years` - an indicator *iterating* (`Iter`) over years.

Also, always use `< -` for assigning objects!

# Objects

Assign **recognizable**, **concise** names to your objects!

Let's start with the object `MyData` - this can be anything. Let's assume it is a raster of NDVI values during the year 1982. How do we come up with a better name?

**Specificity** in your naming helps you keep track of your data in your code:

- NDVI1982

**Classes** of objects determine how they get handled. Add them as a suffix:

- NDVI1982\_ras

The same goes for **loop indicators**! Avoid single-letter indicators and instead use something like `Iter_Years` - an indicator *iterating* (`Iter`) over years.

Also, always use `< -` for assigning objects!

# Objects

Assign **recognizable**, **concise** names to your objects!

Let's start with the object `MyData` - this can be anything. Let's assume it is a raster of NDVI values during the year 1982. How do we come up with a better name?

**Specificity** in your naming helps you keep track of your data in your code:

- NDVI1982

**Classes** of objects determine how they get handled. Add them as a suffix:

- NDVI1982\_ras

The same goes for **loop indicators**! Avoid single-letter indicators and instead use something like `Iter_Years` - an indicator *iterating* (`Iter`) over years.

Also, always use `< -` for assigning objects!

# Objects

Assign **recognizable**, **concise** names to your objects!

Let's start with the object `MyData` - this can be anything. Let's assume it is a raster of NDVI values during the year 1982. How do we come up with a better name?

**Specificity** in your naming helps you keep track of your data in your code:

- `NDVI1982`

**Classes** of objects determine how they get handled. Add them as a suffix:

- `NDVI1982_ras`

The same goes for **loop indicators**! Avoid single-letter indicators and instead use something like `Iter_Years` - an indicator *iterating* (`Iter`) over years.

Also, always use `<` – for assigning objects!

# Objects

Assign **recognizable**, **concise** names to your objects!

Let's start with the object `MyData` - this can be anything. Let's assume it is a raster of NDVI values during the year 1982. How do we come up with a better name?

**Specificity** in your naming helps you keep track of your data in your code:

- NDVI1982

**Classes** of objects determine how they get handled. Add them as a suffix:

- NDVI1982\_ras

The same goes for **loop indicators**! Avoid single-letter indicators and instead use something like `Iter_Years` - an indicator *iterating* (`Iter`) over years.

Also, always use `< -` for assigning objects!



# Quality of Life Improvements

Use **spaces** to improve readability of your code:

```
paste0("Bad", "Spacing")  # bad  
paste0("Good", "Spacing") # good
```

Insert **line breaks** to improve workflow. I like to break multiple lines of one function call into logical blocks. For example, in `ggplot2`, the first line holds data, the second contains aesthetics, the third adds a layer, and so on.

To avoid side-scrolling, use the automatic **soft-wrap** function in `RStudio`.

Be **consistent** in your coding style. Found a `tidyverse` solution to your problem online but don't use `tidyverse` syntax in the rest of your document? Adapt the solution to be in the same style as the rest of your coding!

# Quality of Life Improvements

Use **spaces** to improve readability of your code:

```
paste0("Bad", "Spacing")  # bad  
paste0("Good", "Spacing") # good
```

Insert **line breaks** to improve workflow. I like to break multiple lines of one function call into logical blocks. For example, in `ggplot2`, the first line holds data, the second contains aesthetics, the third adds a layer, and so on.

To avoid side-scrolling, use the automatic **soft-wrap** function in `RStudio`.

Be **consistent** in your coding style. Found a `tidyverse` solution to your problem online but don't use `tidyverse` syntax in the rest of your document? Adapt the solution to be in the same style as the rest of your coding!

# Quality of Life Improvements

Use **spaces** to improve readability of your code:

```
paste0("Bad", "Spacing")  # bad  
paste0("Good", "Spacing") # good
```

Insert **line breaks** to improve workflow. I like to break multiple lines of one function call into logical blocks. For example, in `ggplot2`, the first line holds data, the second contains aesthetics, the third adds a layer, and so on.

To avoid side-scrolling, use the automatic **soft-wrap** function in `RStudio`.

Be **consistent** in your coding style. Found a `tidyverse` solution to your problem online but don't use `tidyverse` syntax in the rest of your document? Adapt the solution to be in the same style as the rest of your coding!

# Quality of Life Improvements

Use **spaces** to improve readability of your code:

```
paste0("Bad", "Spacing") # bad  
paste0("Good", "Spacing") # good
```

Insert **line breaks** to improve workflow. I like to break multiple lines of one function call into logical blocks. For example, in `ggplot2`, the first line holds data, the second contains aesthetics, the third adds a layer, and so on.

To avoid side-scrolling, use the automatic **soft-wrap** function in `RStudio`.

Be **consistent** in your coding style. Found a `tidyverse` solution to your problem online but don't use `tidyverse` syntax in the rest of your document? Adapt the solution to be in the same style as the rest of your coding!

# Comments

Don't be overly proud of your coding skills. Comment everything!

## Bad comments:

- ambiguous
- sparse

```
# Data Manipulation
```

## Good comments:

- specify (what is the code doing)
- justify (why is it being done)

```
# Z-Score calculation for comparability
```

Write comments as though you were **coding for someone else!**

# Comments

Don't be overly proud of your coding skills. Comment everything!

## Bad comments:

- ambiguous
- sparse

```
# Data Manipulation
```

## Good comments:

- specify (what is the code doing)
- justify (why is it being done)

```
# Z-Score calculation for comparability
```

Write comments as though you were **coding for someone else!**

# Comments

Don't be overly proud of your coding skills. Comment everything!

## Bad comments:

- ambiguous
- sparse

```
# Data Manipulation
```

## Good comments:

- specify (what is the code doing)
- justify (why is it being done)

```
# Z-Score calculation for comparability
```

Write comments as though you were **coding for someone else!**

# Comments

Don't be overly proud of your coding skills. Comment everything!

## Bad comments:

- ambiguous
- sparse

```
# Data Manipulation
```

## Good comments:

- specify (what is the code doing)
- justify (why is it being done)

```
# Z-Score calculation for comparability
```

Write comments as though you were **coding for someone else!**



# RMarkdown

Using `Rmarkdown` for your research comes with a multitude of advantages:

- 1 Entire **workflow in one program** (`RStudio`)
- 2 Great capabilities to **present code**
- 3 **Research** and reports **reproducible** at the click of **one button**
- 4 **Combines** `R` functionality and  $\text{\LaTeX}$  formatting (if desired)
- 5 **Consistent formatting**
- 6 **Clear presentation of code**
- 7 **Dynamic documents** (you can generate various output document types)
- 8 Applicable for **almost all document types** you may desire as an output (e.g. manuscripts, presentations, posters, etc.)

# Header

The **Head** is used as an information statement at the top of your code document that informs the user of the contents, author, and (sometimes) date of the last edit on said document:

```
# #####  
# PROJECT: Project Title  
# CONTENTS: What the code in your file is used for  
# AUTHOR: Who worked on it  
# EDIT: When the last edit was made  
# #####
```

Personally, I don't use the edit argument in my heads. Instead, I rely on *version control*.

# Header

The **Head** is used as an information statement at the top of your code document that informs the user of the contents, author, and (sometimes) date of the last edit on said document:

```
# #####  
# PROJECT: Project Title  
# CONTENTS: What the code in your file is used for  
# AUTHOR: Who worked on it  
# EDIT: When the last edit was made  
# #####
```

Personally, I don't use the edit argument in my heads. Instead, I rely on *version control*.

# Preamble

The **Preamble** is where you set up the most important parameters/guidelines for your coding script. Personally, I *highly* recommend to make your first line in the preamble read `rm(list=ls())`.

This is also where you load **packages**. I recommend doing so as follows:

```
install.load.package <- function(x) {
  if (!require(x, character.only = TRUE))
    install.packages(x, repos = "http://cran.us.r-project.org")
  require(x, character.only = TRUE)
}
```

This functions *checks* whether the package is already installed. If it is, it will be *loaded*. If it is not, it will be *installed* from CRAN and subsequently loaded.

You use it for multiple packages like this:

```
package_vec <- c("ggplot2", "foreach", "doParallel")
supply(package_vec, install.load.package)
```

# Sections

**Sections** are a powerful tool in RStudio!

They keep your code:

- *structured*
- *clear*
- *easy-to-use*

They do so by:

- *collapsing* code
- adding a *table of contents*

```

6
7 ▾ ##### Libraries #####
8 library(tidyverse)
9 library(fuzzySim)
10 library(vegan)
11

```

Expand /  
collapse  
sections  
by clicking  
the arrow  
on the left

```

6
7 ▸ ##### Libraries #####
13 ▸ ##### Import data #####
19 ▸ ##### Calculation of variables #####
20 ##### >> traits #####
60 ▸ ##### >> plots ----
61 plot_list <- plot_data %>% pull(plot_ID) %>% as.character()
62 # add mountain and elevation band columns

```

# Sections

**Sections** are a powerful tool in RStudio!

They keep your code:

- *structured*
- *clear*
- *easy-to-use*

They do so by:

- *collapsing* code
- adding a *table of contents*

```

6
7 ▾ ##### Libraries #####
8 library(tidyverse)
9 library(fuzzySim)
10 library(vegan)
11

```

Expand /  
collapse  
sections  
by clicking  
the arrow  
on the left

```

6
7 ▾ ##### Libraries #####
13 ▸ ##### Import data #####
19 ▾ ##### Calculation of variables #####
20 ##### >> traits #####
60 ▾ ##### >> plots ----
61 plot_list <- plot_data %>% pull(plot_ID) %>% as.character()
62 # add mountain and elevation band columns

```

# Sections

**Sections** are a powerful tool in RStudio!

They keep your code:

- *structured*
- *clear*
- *easy-to-use*

They do so by:

- *collapsing* code
- adding a *table of contents*

```

6
7 ##### Libraries #####
8 library(tidyverse)
9 library(fuzzySim)
10 library(vegan)
11

```

Expand /  
collapse  
sections  
by clicking  
the arrow  
on the left

```

6
7 ##### Libraries #####
13 ##### Import data #####
19 ##### Calculation of variables #####
20 ##### >> traits #####
60 ##### >> plots ----
61 plot_list <- plot_data %>% pull(plot_ID) %>% as.character()
62 # add mountain and elevation band columns

```

# Reproducibility

**Reproducibility** is **paramount** in science.

## Working directories

It is common to set a base directory at the beginning of your code and refer to sub-directories from there. We can *do better*:

- `getwd()` in a `project` will yield the directory of the `project` file
- the package `here` offers more functions to never have to set a directory path by hand

## Randomness

Random processes (e.g. `sample()`, `rnorm()`, etc.) in R are always *pseudo-random* (their calculation is based on your computer time). We can make sure the result of a random process is always the same between researchers by using:

- `set.seed()`



# Reproducibility

**Reproducibility** is **paramount** in science.

## Working directories

It is common to set a base directory at the beginning of your code and refer to sub-directories from there. We can *do better*:

- `getwd()` in a project will yield the directory of the project file
- the package `here` offers more functions to never have to set a directory path by hand

## Randomness

Random processes (e.g. `sample()`, `rnorm()`, etc.) in R are always *pseudo-random* (their calculation is based on your computer time). We can make sure the result of a random process is always the same between researchers by using:

- `set.seed()`

# Reproducibility

**Reproducibility** is **paramount** in science.

## Working directories

It is common to set a base directory at the beginning of your code and refer to sub-directories from there. We can *do better*:

- `getwd()` in a `project` will yield the directory of the `project` file
- the package `here` offers more functions to never have to set a directory path by hand

## Randomness

Random processes (e.g. `sample()`, `rnorm()`, etc.) in R are always *pseudo-random* (their calculation is based on your computer time). We can make sure the result of a random process is always the same between researchers by using:

- `set.seed()`

# Functions & Sourcing

*Big projects* lend themselves well to a **multi-coment workflow**:

## Functions

User-defined functions are a great way of *soft-coding* your analyses to repeat them for different input parameters and I highly recommend doing so all the time. Make sure your functions are:

- internally *consistent*
- *well-documented*
- easy to *understand*

## Sourcing

Splitting your code into multiple documents is a great idea to ensure your project remains *structured* and *easy to handle*. You can use the `source()` function to combine those documents. Keep in mind that you should:

- use *sensible file names*
- *sourced* functions still need to be *called*

# Functions & Sourcing

*Big projects* lend themselves well to a **multi-coment workflow**:

## Functions

User-defined functions are a great way of *soft-coding* your analyses to repeat them for different input parameters and I highly recommend doing so all the time. Make sure your functions are:

- internally *consistent*
- *well-documented*
- easy to *understand*

## Sourcing

Splitting your code into multiple documents is a great idea to ensure your project remains *structured* and *easy to handle*. You can use the `source()` function to combine those documents. Keep in mind that you should:

- use *sensible file names*
- *sourced* functions still need to be *called*

# Functions & Sourcing

*Big projects* lend themselves well to a **multi-coment workflow**:

## Functions

User-defined functions are a great way of *soft-coding* your analyses to repeat them for different input parameters and I highly recommend doing so all the time. Make sure your functions are:

- internally *consistent*
- *well-documented*
- easy to *understand*

## Sourcing

Splitting your code into multiple documents is a great idea to ensure your project remains *structured* and *easy to handle*. You can use the `source()` function to combine those documents. Keep in mind that you should:

- use *sensible file names*
- *sourced* functions still need to be *called*

# Progress Bar

A **progress bar** is a great way of keeping you updated on how your code is progressing so far. It is especially useful when your code involves *loops*:

```
Data_vec <- 1:100 # a vector on integers from 1 to 100
# creating progress bar
ProgBar <- txtProgressBar(min = 0, max = length(Data_vec),
                          style = 3)
# looping over contents of Data_vec
for(Iter_ProgBar in 1:length(Data_vec)) {
  setTxtProgressBar(ProgBar, Iter_ProgBar) # updating ProgBar
} # end of Data_vec loop
```



# Estimators

**Estimators** are a great way to know when to come back to your computer or server and check up on your data, code, and results. They are most useful in *loop* based approaches as they need a baseline for the estimation:

```
Data_vec <- 1:100 # a vector on integers from 1 to 100
T_Begin <- Sys.time() # record time
# looping over contents of Data_vec
for(Iter_Est in 1:length(Data_vec)){
  Sys.sleep(.1) # pause for .1 seconds
  # estimator produced on first iteration
  if(Iter_Est == 1){
    T_End <- Sys.time() # record time
    Duration <- as.numeric(T_End)-as.numeric(T_Begin) # time difference
    print(paste("Estimated time to finish:",
               as.POSIXlt(T_Begin + Duration*length(Data_vec),
                           tz = Sys.timezone(location=FALSE))
               ))
  } # end of estimator check
} # end of Data_vec loop

## [1] "Estimated time to finish: 2020-03-25 00:33:50"
```

Yes, I did work on this presentation past midnight.

# Parallel Processing

Got a big data approach where you carry out the same kind of analysis for many different species or data sets? It's taking forever to complete? **Parallel processing** has your back! By default, R only uses one of the cores in your computer or server for processing code. Use this, to make use of all the computational power:

```
library(doParallel) # for registering clusters
library(foreach) # for parallel processing
Cores <- detectCores() # identify the number of cores in your machine
cl <- makeCluster(Cores) # create virtual cluster
registerDoParallel(cl) # register cluster of cores
# parallel processing
foreach(Iter_Par = 1:length(Data_vec)) %dopar% {
  # your function here
} # end of parallel processing
stopCluster(cl) # stop cluster
```