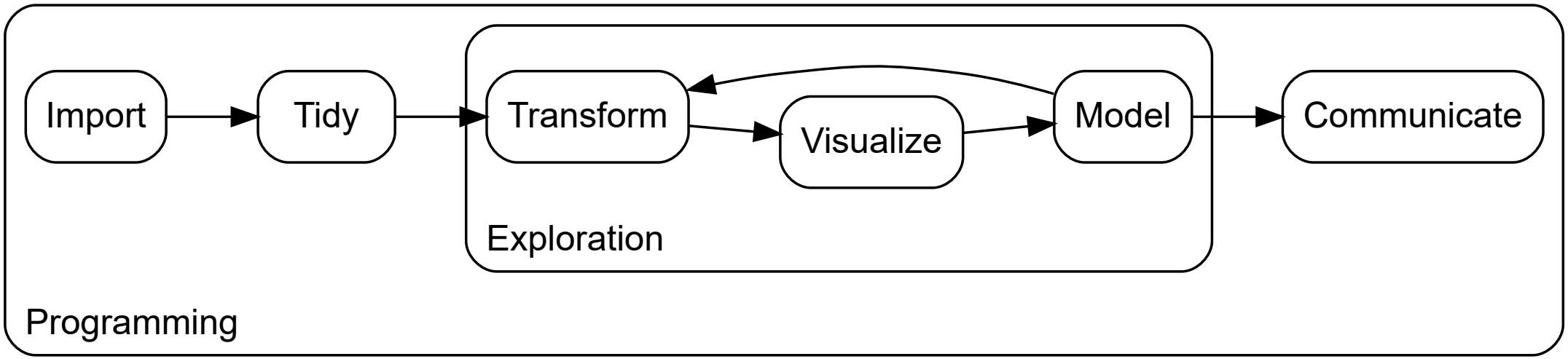
Data Analysis using R

Programming

Sven Werenbeck-Ueding

21.09.2023



Source: Wickham and Grolemund (2016)

Functional Programming

Functional Languages

- Programming languages with **first-class functions** are called functional languages
- First-class functions behave like other data structures, i. e. they can be
  - Assigned to variables
  - Passed on to other functions as arguments
  - Created inside functions
- Functions are often required to be **pure**, i. e. they satisfy:
  - The output is determined by its input so that it always returns the same output given the same input
  - There are no side-effects to a function so that it does not change the global environment, disk or displays output on the screen

- Not all functions in R are pure (e. g. `rnorm()` generates random numbers from a normal distribution), but they are first-class functions
- Yet, at its core, R is a functional programming language
- Functional style can be adopted by separating pure and non-pure functions

Functional Style

> It's hard to describe exactly what a [functional programming] style is, but generally [...] it means **decomposing a big problem into smaller pieces**, then solving each piece with a function or combination of functions.
>
> When using a functional style, you strive to **decompose components of the problem into isolated functions** that operate independently. Each function taken by itself is simple and straightforward to understand; complexity is handled by composing functions in various ways.
>
> Wickham (2019)

Functions in R

Why use functions in R?

- The code on the right creates a data frame with 50 observations and three variables (x, y, z) drawn from normal distributions $N(\mu, \sigma)$ with $\mu = 5$ and $\sqrt{\sigma} = 10$
- Each variable is standardized: $\frac{x - \bar{x}}{\sqrt{\sigma_x}}$
- This approach is bad practice → copy-pasting code is **prone to errors** and **hard to maintain**
- Does not improve readability and comprehensiveness of the code

Do not try at home!

```r
# Create a data frame with 50 observations
# and 3 normal distributed variables with
# mean 5 and standard deviation 10
df <- data.frame(x = rnorm(50, 5, 10),
                 y = rnorm(50, 5, 10),
                 z = rnorm(50, 5, 10))
```

```r
# Standardize each variable
df$x <- (df$x - mean(df$x)) / sd(df$x)
df$y <- (df$y - mean(df$y)) / sd(df$y)
df$z <- (df$z - mean(df$z)) / sd(df$z)
```

Why use functions in R?

- Instead, we can write a function with a proper name that **takes a numeric vector** as input and **returns the standardized vector**
- The function can then be applied to each column of the data frame that we have created in advance
- This function is more **readable** and **comprehensive**
- **Maintaining the code becomes easier** because altering the standardization requires only adjusting the function `standardize()`
- Writing a function should be considered when the same code is repeated several times
- Because R use first-class-functions, it is inherently made for using functional programming

```
standardize <- function(x) {
  mean_x <- mean(x)

  sd_x <- sd(x)

  (x - mean_x) / sd_x
}
```

```
df$x <- standardize(df$x)
```

Function Components

- There are three key components to functions:
  1. The **name** of a function is assigned using the `<-` operator
  2. The function **arguments** are provided in the parentheses of `function()`, separated by commas
  3. The **body** defines what is executed on the arguments and is placed the in curly brackets `{...}` after `function()`
- Binding a function to a name is not required → **anonymous function**
- The return value can be defined using `return()` or simply the last printed object (as in the `standardize()` function)

```
<NAME> <- function(<ARGUMENTS>) {
  <BODY>
}
```

Function Arguments

- Function arguments can generally be categorized in

    1. the **data** to compute the body on
    2. the **details** of the computation

- Example: `log()` takes `x` as the data argument and the details of the computation are defined by the argument `base`

- Default arguments can be specified in `function()`, e. g. the default value for `base` in `log()` is `exp(1)`

```
log
```

```
## function (x, base = exp(1))  .Primitive("log")
```

Function Arguments

Best Practices

- Data arguments are placed before details arguments
- Default values are set to their most common value (e. g. 0.95 for confidence level)
- Empty spaces around `<-` and `=`
- Match names of arguments in existing R functions for consistency:
  - `x`, `y`, `z`: vectors
  - `w`: weights
  - `df`: data frame
  - `i`, `j`: numeric indices (such as rows and columns)
  - `n`: number of rows (matrices and data frames) or vector length
  - `p`: number of columns (for matrices and data frames)

Checking Argument Values

- It is good practice to check arguments for their validity to avoid the function body to through errors
- Can use `if` statements to check whether values given for arguments satisfy conditions
- If an input does not satisfy these conditions, we can `stop()` the execution of the function early on
- We can pass a string to `stop()` to notify the user of what went wrong
- Example: `x` in our `standardize()` function has to be of type numeric

Function | Error

```
standardize <- function(x) {
  if(!is.numeric(x))
    stop("x has to be numeric")

  x_mean <- mean(x)

  x_sd <- sd(x)

  (x - x_mean) / x_sd
}
```

Checking Argument Values

- It is good practice to check arguments for their validity to avoid the function body to through errors
- Can use `if` statements to check whether values given for arguments satisfy conditions
- If an input does not satisfy these conditions, we can `stop()` the execution of the function early on
- We can pass a string to `stop()` to notify the user of what went wrong
- Example: `x` in our `standardize()` function has to be of type numeric

Function | Error

```
not_numeric <- c("This",
                 "is",
                 "not",
                 "numeric")

standardize(not_numeric)
```

```
## Error in standardize(not_numeric): x has to
```

Lexical Scoping

> R [...] looks up the values of names based on how a function is defined, not how it is called.
>
> "Lexical" here is not the English adjective that means relating to words or a vocabulary. It's a technical CS term that tells us that the scoping rules use a parse-time, rather than a run-time structure.
>
> Wickham (2019)

Lexical scoping means that R follows these rules:

1. **Name masking**
2. **Functions over variables**
3. **Independent invocation**
4. **Dynamic lookup**

Lexical Scoping

Name Masking | Functions over Variables | Independent Invocation | Dynamic Lookup

- Names defined inside a function mask names outside a function
- If R cannot find a name inside a function, it looks for the name outside the function (e. g. in the global environment)

```
y <- 5

some_function <- function() {
  print(y)
}

some_function()
```

```
## [1] 5
```

Lexical Scoping

- If a function and a variable residing in different environments share the same name, R chooses the function if the name is called
- Two objects should never be named the same, regardless of this behavior!

```
y_1 <- function(x) x + 5

y_2 <- function() {
  y_1 <- 5
  y_1(y_1)
}

y_2()
```

```
## [1] 10
```

Lexical Scoping

- Each function call is executed in a new environment
- Names defined during function call are not present, when executing the function call again
- You may have noticed that running `standardized()` in your environment does not create variables `x_mean` and `x_sd`

```
some_function <- function(x) {
  x + 5
}

some_function(5)
```

```
## [1] 10
```

```
some_function(5)
```

```
## [1] 10
```

Lexical Scoping

- R looks for names only when the function is executed, not when it is created
- Reassigning variables that are defined outside the function but called during the function execution, may change the function output

```
some_function <- function() {
  x + 5
}

x <- 10

some_function()
```

```
## [1] 15
```

```
x <- 15

some_function()
```

```
## [1] 20
```

Lazy Evaluation

- Arguments are only evaluated when they are accessed by R (**lazy evaluation**)
- If an argument of a function is not called inside the function, it is not evaluated by R, potentially saving resources when the evaluation would be computationally expensive
- Allows for defining default arguments in terms of other arguments

```
some_function <- function(x = 5,
                          y = 1,
                          z = x + y) {
  z
}

some_function()
```

```
## [1] 6
```

Ellipsis

- An **ellipsis** (`...`) is a reserved special argument that captures all arguments not otherwise matched by the function specification
- Useful when writing functions that primarily wrap other functions
- Example: Our `standardize()` wraps around `mean()` and `sd()`
  - If we call `standardize()` on a vector with missing values, `mean()` and `sd()` would throw errors
  - Since both `mean()` and `sd()` take the same argument, we may want to pass these arguments in `standardize()` down to these functions

Code | Output

```
standardize <- function(x, ...) {
  if(!is.numeric(x))
    stop("x has to be numeric")

  x_mean <- mean(x, ...)

  x_sd <- sd(x, ...)

  (x - x_mean) / x_sd
}

# Create a vector with one NA and two
# random values
vec_na <- c(NA, rnorm(2, 5, 10))

standardize(vec_na, na.rm = TRUE)
```

Ellipsis

- An **ellipsis** (`...`) is a reserved special argument that captures all arguments not otherwise matched by the function specification
- Useful when writing functions that primarily wrap other functions
- Example: Our `standardize()` wraps around `mean()` and `sd()`
  - If we call `standardize()` on a vector with missing values, `mean()` and `sd()` would throw errors
  - Since both `mean()` and `sd()` take the same argument, we may want to pass these arguments in `standardize()` down to these functions

Code | Output

```
## [1]         NA  0.7071068 -0.7071068
```

Apply Functions

- Writing functions is most useful when the execution of their body is needed multiple times in your code
- Example: we want to use `standardize()` on each column of `df`
- We could `for` loop through all columns of `df`

```
# Number of columns in df
df_ncol <- ncols(df)

# Loop through columns and standardize
for(i in 1:df_ncol) {
  df[,i] <- standardize(df[,i])
}
```

A better approach is to use `apply()` on `df`!

Apply Functions

```
?apply
```

- `apply()` returns a vector, matrix or list of values returned by applying a function to margins of a matrix
- *Very* useful for applying functions over rows or columns of data frames and matrices
- Takes a matrix `X` as an input and applies a function `FUN` over `MARGIN`
- `MARGIN` specifies whether `FUN` should be applied over rows (`=1`), columns (`=2`) or both (`=c(1, 2)`)
- Since `df` can be coerced to a matrix, we can apply `standardize()` over `MARGIN=2` to obtain a matrix with standardized columns

Code | Output

```
apply(X = df, MARGIN = 2, FUN = standardize)
```

Apply Functions

```
?apply
```

- `apply()` returns a vector, matrix or list of values returned by applying a function to margins of a matrix
- *Very* useful for applying functions over rows or columns of data frames and matrices
- Takes a matrix `X` as an input and applies a function `FUN` over `MARGIN`
- `MARGIN` specifies whether `FUN` should be applied over rows (`=1`), columns (`=2`) or both (`=c(1, 2)`)
- Since `df` can be coerced to a matrix, we can apply `standardize()` over `MARGIN=2` to obtain a matrix with standardized columns

Code | Output

```
##                x           y           z
##  [1,] -0.71993515 -1.22760040  0.65616285
##  [2,] -0.45507530 -0.43369018 -0.20819787
##  [3,]  0.62603921 -0.32675511  1.32617959
##  [4,]  0.34784529  1.21012800  0.49484340
##  [5,] -0.46435711  0.57186403  0.55235288
##  [6,] -0.45105333  0.28578766  0.56986976
##  [7,] -1.37864993 -1.84723467 -0.03024116
##  [8,]  0.07394353 -0.11853905  0.56936574
##  [9,] -1.08574976  0.07114682 -0.42503780
## [10,] -2.10704478  0.00604602  1.46647138
## [11,]  1.18790176  0.99548652  0.08880549
## [12,]  2.05919433  0.40881541 -1.38235809
```

Variants of Apply

There are several `*apply()` functions, each tailored to specific requirements for input and output types:

- `lapply(X, FUN, ...)`: applies `FUN` over a list
- `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`: wrapper of `lapply` that returns a vector or matrix
- `vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)`: has a pre-specified type of return value given by `FUN.VALUE`

Apply Anonymous Functions

Instead of applying our `standardize()` function, we could have defined an anonymous function inside `apply()` with the same result:

Code | Output

```
apply(df, 2, function(x) (x - mean(x)) / sd(x))
```

Apply Anonymous Functions

Instead of applying our `standardize()` function, we could have defined an anonymous function inside `apply()` with the same result:

Code | Output

```
##                 x           y           z
##  [1,] -0.71993515 -1.227600040  0.65616285
##  [2,] -0.45507530 -0.433690181 -0.20819787
##  [3,]  0.62603921 -0.326755116  1.32617959
##  [4,]  0.34784529  1.210128002  0.49484340
##  [5,] -0.46435711  0.571864030  0.55235288
##  [6,] -0.45105333  0.285787658  0.56986976
##  [7,] -1.37864993 -1.847234671 -0.03024116
##  [8,]  0.07394353 -0.118539050  0.56936574
##  [9,] -1.08574976  0.071146823 -0.42503780
## [10,] -2.10704478  0.006046019  1.46647138
## [11,]  1.18790176  0.995486516  0.08880549
## [12,]  2.05919433  0.408815406 -1.38235809
```

Composing Functions

Function Composition

- In `base` R, multiple functions can be composed by:
  1. Saving intermediate results
  2. Nesting functions
- Both approaches are rather flawed
- Saving intermediate results requires assigning function output, eventually floating the environment with many irrelevant objects
- Nesting functions may reduce readability and comprehensiveness of your code
- Example: Applying `stanardize()` over columns of `df` returns a matrix. We may want to coerce the resulting matrix back to a data frame using `as.data.frame()`

Intermediate Results | Nested Functions

```
df_std <- apply(df, 2, standardize)
df_std <- as.data.frame(df_std)
```

Function Composition

- In `base` R, multiple functions can be composed by:
  1. Saving intermediate results
  2. Nesting functions
- Both approaches are rather flawed
- Saving intermediate results requires assigning function output, eventually floating the environment with many irrelevant objects
- Nesting functions may reduce readability and comprehensiveness of your code
- Example: Applying `stanardize()` over columns of `df` returns a matrix. We may want to coerce the resulting matrix back to a data frame using `as.data.frame()`

Intermediate Results | **Nested Functions**

```
as.data.frame(
  apply(
    df,
    2,
    standardize
  )
)
```

magrittr

The magrittr package offers a set of operators which make your code more readable by:

- structuring sequences of data operations left-to-right (as opposed to from the inside and out),
- avoiding nested function calls,
- minimizing the need for local variables and function definitions, and making it easy to add steps anywhere in the sequence of operations.

The operators pipe their left-hand side values forward into expressions that appear on the right-hand side, i.e. one can replace f(x) with x %>% f(), where %>% is the (main) pipe-operator. When coupling several function calls with the pipe-operator, the benefit will become more apparent.

Bache and Wickham (2022)

What are "packages"?

- Packages are bundles of R code, data and documentation that are often shared via the **Comprehensive R Archive Network** (CRAN)
- R packages are often developed by small teams and made available via CRAN for anyone to download (open source)
- To install and load the `magrittr` package on your system, run the code below
- Content from loaded packages is only available in your current session → if you close and reopen R, you have to run `library(magrittr)` again but **not** have to re-install it

```
install.packages("magrittr")
library(magrittr)
```

The Pipe-Operator

- The pipe operator (`%>%`) is **immensely** useful for writing comprehensive code in functional programming style
- `%>%` does what the name implies: It pipes the value on the left side into an expression or function on the right side
- Pipes can be chained, e. g. `x %>% f_1() %>% ... %>% f_n()`
- Functions are sequentially executed on the output of the previous function
- Keyboard shortcut: `Ctrl + Shift + M`
- Example: Pipe `df` through `apply()` and `as.data.frame()`

```
df %>%
  apply(2, standardize) %>%
  as.data.frame
```

Features of the Pipe-Operator

1. Left-hand side (LHS) is piped into the right-hand side (RHS) as the first argument of the function
2. %>% can be used *inside* functions as well
3. The LHS can be explicitly called with a dot (`.`) if it is needed at a different argument position of the RHS (not to be confused with the dot in R formulas!)
4. Parentheses can be omitted when the LHS needs only a single argument that is at the first position (not recommended)

⚠ Base R's Pipe Operator

- As of version 4.1.0, base R supports the native pipe operator `|>`
- Version 4.2.0 added a placeholder for the LHS (_)
  - Works only if the argument is named
- Apart from the naming, this pipe operator essentially works the same as `magrittr`'s pipe operator (at least for simple operations)
- See this `tidyverse` blog entry for differences between both

```
set.seed(21903)

x <- rnorm(1000, mean = 0, sd = 1)

# Basic usage
x |>
  mean()
```

```
## [1] 0.007922553
```

```
# Using the LHS placeholder `_`
TRUE |>
  mean(c(x, NA), na.rm = _)
```

```
## [1] 0.007922553
```

purrr

> `purrr` enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. If you've never heard of FP before, the best place to start is the family of `map()` functions which allow you to replace many `for` loops with code that is both more succinct and easier to read.
>
> Wickham and Henry (2022)

map()

```
?purrr::map
```

Similar to `apply()`, `map()` applies a function over each element of an input list or vector and returns an object of the same length.

- Takes an input `.x` (list or vector) and transforms it using a function `.f`
- Arguments to be passed on to `.f` can be defined after `.f`, separated by commas
- Contrary to `apply()`, `purrr`'s `map_*()` function are type-consistent, e. g. `map()` always returns a list
- There are several variants of `map` that return pre-specified types: `map_int()` (integer), `map_dbl()` (double), `map_chr()` (character), `map_df()` (data frame)
- `walk()` executes `.f` on `.x` but returns `.x` (useful for side-effects of `.f` such as exporting data)

Example: `map_df()`

- Pipe `df` into `map_df()` to standardize each column and return a data frame with standardized columns
- Anonymous functions can be defined inside `map()` functions
- Lambdas can be used if the LHS should define an argument in the RHS function that is not placed at the first position
- Lambdas behave similar to pipes in that the LHS value is called via `.`
- See `vignette("base")` for more many more handy use cases

Named Function | Anonymous | Lambda

```
df %>%
  map_df(standardize) %>%
  head(n = 5) # Print top 5 rows
```

```
## # A tibble: 5 × 3
##         x      y      z
##     <dbl>  <dbl>  <dbl>
## 1 -0.720 -1.23   0.656
## 2 -0.455 -0.434 -0.208
## 3  0.626 -0.327  1.33
## 4  0.348  1.21   0.495
## 5 -0.464  0.572  0.552
```

Example: `map_df()`

- Pipe `df` into `map_df()` to standardize each column and return a data frame with standardized columns
- Anonymous functions can be defined inside `map()` functions
- Lambdas can be used if the LHS should define an argument in the RHS function that is not placed at the first position
- Lambdas behave similar to pipes in that the LHS value is called via `.`
- See `vignette("base")` for more many more handy use cases

Named Function | **Anonymous** | Lambda

```
df %>%
  map_df(function(x) {
    (x - mean(x)) / sd(x)
  }) %>%
  head(n = 5)
```

```
## # A tibble: 5 × 3
##         x      y      z
##     <dbl>  <dbl>  <dbl>
## 1 -0.720 -1.23   0.656
## 2 -0.455 -0.434 -0.208
## 3  0.626 -0.327  1.33
## 4  0.348  1.21   0.495
## 5 -0.464  0.572  0.552
```

Example: `map_df()`

- Pipe `df` into `map_df()` to standardize each column and return a data frame with standardized columns
- Anonymous functions can be defined inside `map()` functions
- Lambdas can be used if the LHS should define an argument in the RHS function that is not placed at the first position
- Lambdas behave similar to pipes in that the LHS value is called via `.`
- See `vignette("base")` for more many more handy use cases

Named Function | Anonymous | **Lambda**

```
df %>%
  map_df(~ standardize(x = .)) %>%
  head(n = 5)
```

```
## # A tibble: 5 × 3
##         x      y      z
##     <dbl>  <dbl>  <dbl>
## 1 -0.720 -1.23   0.656
## 2 -0.455 -0.434 -0.208
## 3  0.626 -0.327  1.33
## 4  0.348  1.21   0.495
## 5 -0.464  0.572  0.552
```

The `tidyverse`

`tidyverse`

> The tidyverse is a set of packages that work in harmony because they share common data representations and API design. The `tidyverse` package is designed to make it easy to install and load core packages from the tidyverse in a single command.
>
> Wickham, Averick, Bryan, Chang, McGowan, François, Grolemund, Hayes, Henry, Hester, Kuhn, Pedersen, Miller, Bache, Müller, Ooms, Robinson, Seidel, Spinu, Takahashi, Vaughan, Wilke, Woo, and Yutani (2019)

Installation and Use

- Packages included in the `tidyverse` meta package cover a wide range of applications covering almost any task required in data analysis projects:
  - Importing
  - Tidying
  - Transforming
  - Visualizing
  - Programming
- Loading the `tidyverse` package automatically loads a set of core packages, commonly used for these tasks (e. g. `magrittr` and `purrr` are part of the core packages)
- Over the course of the semester, you will become familiar with the fundamental packages and concepts of the `tidyverse`

```
# Install and load the tidyverse
install.packages("tidyverse")
library(tidyverse)
```

Tidyverse Principles

- `tidyverse` packages share a common Application Programming Interface (API), making it easy to follow a consistent programming style, when using its packages
- The tidy API follows these rules:
    1. Reuse existing data structures
    2. Compose simple functions with the pipe
    3. Embrace functional programming
    4. Design for humans
- See the tidy tools manifesto for more information

Reuse Existing Data Structures

- Existing data structures are re-used whenever possible
- Common data structures are **easier for users to get accustomed to** than than special-purpose structures
- R is mainly for statistical programming: Many packages work with rectangular data with observations as rows and variables as columns → using the `base` R data structure for rectangular data
- For lower level data structures such as vectors, use `base` R's atomic vector structure

Compose Simple Functions with the Pipe

- Reduce complex problems to simple tasks that can be chained
- Keep functions as simple as possible (but not too simple)
  - **Each function** should achieve **one task**
  - Rule of thumb: You should be able to explain the goal of your function in one sentence
- Avoid mixing side-effects and transformations: Each function should either transform an input and return the output *or* be used for their side effect
- Define function names as verbs (they *do* something)

Embrace Functional Programming

- Use what R is made for: Functions!
- Focus on immutable objects and copy-on-modify semantics
- Use `apply` or `purrr`'s `map` functions instead of loops and copy-pasting code

Design for Humans

- In the end, your **code must be written, read and understood by humans**
- Computer efficiency is not as important in data analysis as the main portion of your analysis is spent on thinking about *how* to solve your problem
- Follow a **consistent naming** scheme for your functions
- Use explicit and descriptive names so that you can still follow your code if you come back to it later
- Let objects of a family of objects be identified by a common prefix not suffix (helps with autocomplete when writing code)

Style Guide

The Tidyverse Style Guide

- This course mostly follows the coding style proposed by Wickham (2023)
- Note that this style guide reflects *opinions* on how good code should look like
- Many of these style decisions, however, do make your code easier to write, read and understand
- Most importantly, your code should follow a consistent style!
- If you want to learn how to ensure following the tidyverse style guide programmatically, check out the `styler` package

Files

Names

- Name your files meaningfully and in lower case
- Use - or _ to separate words
- If files are run in sequential order, use numbers as prefixes (that way files are properly ordered in the explorer)

Organisation

- Organize files in folders
- Give concise and descriptive names to folders
- Store script where only functions are defined in a folder called "R"
- Complex functions should be stored in separate files

```
fence_migration
|-- data
    |-- processed
        |-- migration_data.csv
    |-- raw
        |-- enoe_survey.csv
        |-- fence_construction.csv
|-- analysis
    |-- 00_data_processing.R
    |-- 01_exploration.R
    |-- 02_transforming.R
    |-- 03_model.R
    |-- 04_results.R
|-- output
    |-- coef_plot.png
    |-- summary_table.tex
|-- R
    |-- standardize.R
|-- fence_migration.Rproj
```

Internal File Structure

- Load packages at the top of your R scripts
- Indicate sections with `#`
  - If your sections have subsections, indicate them with one more `##`
  - First level section `#`, second level section `##`, ...
  - Write `####` at the end of the line to let R know that this is a code section
  - This way, when you open the file outline (`Ctrl + Shift + O`), you can see a table of contents for your script
- Write `#` followed by underscores up until `####` before a new section to create separation lines between sections (makes the ToC more readable)

```
#_____________________________________####
#   Data Preparation                    ####

library(tidyverse)

#_____________________________________####
#   Import                              ####

in_path <- "data/raw/"

##  Survey Data                         ####

df_survey <- in_path %>%
  paste0("enoe_survey.csv") %>%
  read_csv()
```

Syntax

Object Names

- Use **snake case** for variable and function names: Separate words within a name with an underscore (_)
- Variable names should be nouns and function names should be verbs
- Do not re-use names of other functions
- Give objects of the same family a common prefix
- Be descriptive with the names you choose (even if they may be longer)

```r
x <- rnorm(50, 0, 1)

standardize <- function(x, na.rm = TRUE) {
  if(!is.numeric(x))
    stop("x has to be numeric")

  x_mean <- mean(x, na.rm = na.rm)

  x_sd <- sd(x, na.rm = na.rm)

  (x - x_mean) / x_sd
}

x_std <- standardize(x, na.rm = TRUE)
```

Syntax

Spacing

- Always put an empty space after a comma, not before!
- Do not put spaces inside/outside parentheses of functions
- Place a space after closing parentheses
- Surround infix operators (=, +, <−, ...) by spaces

Do not surround `::`, `:::`, `$`, `@`, `?`, `^` and the subsetting operators by spaces!

```r
x <- rnorm(50, 0, 1)

standardize <- function(x, na.rm = TRUE) {
  if(!is.numeric(x))
    stop("x has to be numeric")

  x_mean <- mean(x, na.rm = na.rm)

  x_sd <- sd(x, na.rm = na.rm)

  (x - x_mean) / x_sd
}

x_std <- standardize(x, na.rm = TRUE)
```

Syntax

Function Calls

- Provide names for **detail** arguments when you call a function
- Names for **data** arguments may be omitted when they are used frequently
- Re-use existing data and detail argument names where appropriate
- Do not use `<-` inside function arguments
- Place the function body in the next line after `{` and indent its code
- Only use `return()` for early function exits

```r
x <- rnorm(50, 0, 1)

standardize <- function(x, na.rm = TRUE) {
  if(!is.numeric(x))
    stop("x has to be numeric")

  x_mean <- mean(x, na.rm = na.rm)

  x_sd <- sd(x, na.rm = na.rm)

  (x - x_mean) / x_sd
}

x_std <- standardize(x, na.rm = TRUE)
```

Syntax

Control Flow

- Separate `if` and its statement with a space
- Place the body in a new line after `{` and indent it (two spaces/`Tab`)
- Short control flow content may be placed in the same line (or the following without curly brackets)
- If `else` is used, it should be on the same line as the closing `}` of the `if` statement
- Very short `if-else` controls may be placed in the same line

```
if (x > 0) {
  x_ln <- log(x)
} else {
  message("x is 0 or smaller")
}

if (x <= 0) "not positive" else "positive"
```

Syntax

Limit Long Lines

- If a code line does not fit your script width (indicated by the vertical line on the right), place function arguments in new lines
- Each argument should be in a separate line
- Omit argument names, when they are used commonly
- When omitting argument names, unnamed arguments may be place in the first line of the function even if the named arguments are in separate lines

```
some_long_function(x,
                   argument_1 = c(1, 2),
                   argument_2 = TRUE)

some_longer_almost_too_long_function(
  x,
  argument_1 = c(1, 2),
  argument_2 = TRUE
)
```

Syntax

Assignment

Use <- **not** = to assign variables and functions!

```
# Do this
x <- rnorm(50, 5, 10)

# Not this
x = rnorm(50, 5, 10)
```

Character

- Start characters with "
- Use ' for quotations inside strings

```
chr <- "Experiment 'A'"
```

Syntax

Comments

- Comments should start with # and single space
- Be concise and descriptive with comments
- Clean and readable code should reduce your need for writing comments
- Explain *why* you do something not *what* and *how*

References

Bache, S. M. and H. Wickham (2022). *magrittr: A Forward-Pipe Operator for R*. https://magrittr.tidyverse.org, https://github.com/tidyverse/magrittr.

Wickham, H. (2019). *Advanced R*. 2nd. Chapman & Hall/CRC. URL: http://adv-r.had.co.nz/.

Wickham, H. (2023). *The tidyverse style guide*. URL: https://style.tidyverse.org/index.html (visited on Jan. 04, 2023).

Wickham, H., M. Averick, J. Bryan, et al. (2019). "Welcome to the tidyverse". In: *Journal of Open Source Software* 4.43, p. 1686. DOI: 10.21105/joss.01686.

Wickham, H. and G. Grolemund (2016). *R for data science. import, tidy, transform, visualize, and model data*. O'Reilly. URL: https://r4ds.had.co.nz/.

Wickham, H. and L. Henry (2022). *purrr: Functional Programming Tools*. https://purrr.tidyverse.org/, https://github.com/tidyverse/purrr.