EpidemiaR Introductory Materials

Justin Millar

Amir Siraj

2024-06-20

Table of contents

# Getting started

This book contains background material for general R basics, as well as some of the packages and functions that we will use during the EpidemiaR training lessons.

|  |
| --- |
| Check your R version! |
| Some of the training material will use a special operator called the “native pipe” (which looks like this: |>). This function was first included in R version 4.1, which was released in May 2021.  If you haven’t updated your R in while then it may be time to update. You can check what your current version is by running R.version in the console. |

## Installing packages

The training lessons will use several R packages. We will discuss these in more detail in the following sections, but it is a good idea to get everything installed and downloaded before we dive in. Note that depending on your internet connection this may take some time.

Use this code in R to install the packages:

# Installing packages  
install.packages(  
 c("dplyr","knitr","lubridate","parallel","readr","readxl","tidyr","tinytex",  
 "tools","tidyverse","janitor","writexl","sf","ISOweek"))

Additional materials and resources:

* WIP!

# 1. Using if statements

Conditional statements are fundamental in programming, allowing for decision-making based on different conditions. In R, the primary conditional statement is the if statement, which can be used alone or combined with else and else if for more complex logic.

## 1.1 Basic If Statement

The basic if statement checks a condition and executes a block of code if the condition is true.

**Syntax:**

if (condition) {  
 # code to execute if the condition is true  
}

**Example:**

x <- 10  
  
if (x > 5) {  
 print("x is greater than 5")  
}

## 1.2 If-Else Statement

To execute a block of code when the condition is false, use an else statement.

**Syntax:**

if (condition) {  
 # code to execute if the condition is true  
} else {  
 # code to execute if the condition is false  
}

**Example:**

x <- 3  
  
if (x > 5) {  
 print("x is greater than 5")  
} else {  
 print("x is not greater than 5")  
}

### 1.2.1 If-Else If-Else Statement

For multiple conditions, use else if to check additional conditions.

**Syntax:**

if (condition1) {  
 # code to execute if condition1 is true  
} else if (condition2) {  
 # code to execute if condition2 is true  
} else {  
 # code to execute if none of the conditions are true  
}

**Example:**

x <- 7  
  
if (x > 10) {  
 print("x is greater than 10")  
} else if (x > 5) {  
 print("x is greater than 5 but less than or equal to 10")  
} else {  
 print("x is 5 or less")  
}

## 1.3 Nested If Statements

You can also nest if statements inside one another to check multiple levels of conditions.

**Example:**

x <- 15  
y <- 20  
  
if (x > 10) {  
 if (y > 15) {  
 print("x is greater than 10 and y is greater than 15")  
 } else {  
 print("x is greater than 10 but y is not greater than 15")  
 }  
} else {  
 print("x is 10 or less")  
}

### 1.3.1 Vectorized If Statements

When working with vectors, ifelse is a more efficient way to apply conditional logic.

**Syntax:**

ifelse(test, yes, no)

* test: A logical condition.
* yes: The value to return if the condition is true.
* no: The value to return if the condition is false.

**Example:**

x <- c(2, 7, 5, 10)  
  
result <- ifelse(x > 5, "Greater than 5", "5 or less")  
print(result)

## 1.4 Conclusion

Using if statements in R allows you to execute code based on conditions, making your programs more dynamic and flexible. Whether you’re using a simple if statement, adding else and else if for more complex logic, or applying conditional logic to vectors with ifelse, mastering these constructs is essential for effective programming in R.

# 2. More base R functions

This section contains additional functions and tools that we will use in later sections during the EpidemiaR Training workshop.

## 2.1 Concatenating strings and printing messages

In R, paste() and paste0() are functions used to concatenate strings, while message() is used to print messages to the console. Understanding these functions helps in creating more informative and readable outputs, as well as in debugging and providing user feedback.

### 2.1.1 paste() Function

The paste() function concatenates strings with a specified separator.

**Syntax:**

paste(..., sep = " ", collapse = NULL)

* ...: One or more R objects to be concatenated.
* sep: A string to separate the terms (default is a space).
* collapse: An optional string to separate the results when concatenating vectors.

**Examples:**

1. Concatenating strings with spaces:

* str1 <- "Hello"  
  str2 <- "World"  
  result <- paste(str1, str2)  
  print(result)  
  # Output: "Hello World"

1. Using a different separator:

* result <- paste("A", "B", "C", sep = "-")  
  print(result)  
  # Output: "A-B-C"

1. Collapsing a vector into a single string:

* words <- c("apple", "banana", "cherry")  
  result <- paste(words, collapse = ", ")  
  print(result)  
  # Output: "apple, banana, cherry"

### 2.1.2 paste0() Function

The paste0() function is a shortcut for paste(..., sep = ""). It concatenates strings without any separator.

**Syntax:**

paste0(...)

* ...: One or more R objects to be concatenated.

**Examples:**

1. Concatenating strings without spaces:

* str1 <- "Hello"  
  str2 <- "World"  
  result <- paste0(str1, str2)  
  print(result)  
  # Output: "HelloWorld"

1. Concatenating multiple strings:

* result <- paste0("A", "B", "C")  
  print(result)  
  # Output: "ABC"

### 2.1.3 message() Function

The message() function is used to print a message to the console. Unlike print(), it sends its output to the standard message stream and is often used for warnings, informational messages, and debugging.

**Syntax:**

message(...)

* ...: One or more R objects to be printed.

**Examples:**

1. Printing a simple message:

* message("This is an informational message.")  
  # Output: This is an informational message.

1. Combining paste() with message():

* name <- "John"  
  age <- 30  
  message(paste("Name:", name, "Age:", age))  
  # Output: Name: John Age: 30

1. Combining paste0() with message():

* prefix <- "ID\_"  
  id <- 1234  
  message(paste0("Generated ID: ", prefix, id))  
  # Output: Generated ID: ID\_1234

## 2.2 Loading objects and data files

### 2.2.1 R objects and data

R provides several functions for saving and loading data, allowing you to preserve your workspace and share data with others. The save() and load() functions save and restore entire R workspaces, while saveRDS() and readRDS() handle individual R objects.

#### 2.2.1.1 save() and load() Functions

The save() function saves R objects to a specified file, which can be loaded back into the R environment using the load() function.

**save() Syntax:**

save(..., file)

* ...: R objects to be saved.
* file: A character string naming the file to save the data to.

**load() Syntax:**

load(file)

* file: A character string naming the file to load the data from.

**Examples:**

1. Saving multiple objects:

* x <- 1:10  
  y <- letters[1:10]  
  save(x, y, file = "data.RData")

1. Loading the saved objects:

* load("data.RData")  
  print(x)  
  print(y)

#### 2.2.1.2 saveRDS() and readRDS() functions

The saveRDS() function saves a single R object to a file, and readRDS() restores it. Unlike save(), saveRDS() does not save the object name, so you can assign it any name when you load it.

**saveRDS() Syntax:**

saveRDS(object, file)

* object: The R object to be saved.
* file: A character string naming the file to save the object to.

**readRDS() Syntax:**

readRDS(file)

* file: A character string naming the file to read the object from.

**Examples:**

1. Saving a single object:

* z <- matrix(1:9, nrow = 3)  
  saveRDS(z, file = "matrix.RDS")

1. Loading the saved object:

* my\_matrix <- readRDS("matrix.RDS")  
  print(my\_matrix)

Using load(), save(), saveRDS(), and readRDS() functions in R enables you to efficiently save and restore your data. Whether you need to save entire workspaces or individual objects, these functions provide flexible options for data persistence and sharing.

### 2.2.2 CSV and Excel files

## 2.3 Loading CSV and Excel Files into R

Reading data from CSV and Excel files is a common task in data analysis. R provides functions to easily load these files into your workspace for analysis.

#### 2.3.0.1 Loading CSV Files

To load CSV files, use the read.csv() or readr::read\_csv() functions. The readr package’s read\_csv() is often preferred for its speed and efficiency.

**read.csv() Syntax:**

data <- read.csv(file, header = TRUE, sep = ",")

* file: Path to the CSV file.
* header: Logical value indicating if the file contains a header row.
* sep: Character separating the values (default is a comma).

**Example:**

data <- read.csv("path/to/yourfile.csv")  
print(head(data))

**readr::read\_csv() Syntax:**

library(readr)  
data <- read\_csv("path/to/yourfile.csv")  
print(head(data))

#### 2.3.0.2 Loading Excel Files

To load Excel files, use the readxl package, which provides the read\_excel() function.

**Syntax:**

library(readxl)  
data <- read\_excel(path, sheet = 1)

* path: Path to the Excel file.
* sheet: Sheet number or name to read from (default is the first sheet).

**Example:**

library(readxl)  
data <- read\_excel("path/to/yourfile.xlsx", sheet = "Sheet1")  
print(head(data))

Using read.csv(), readr::read\_csv(), and read\_excel() functions, you can easily load CSV and Excel files into R for analysis. These functions provide a straightforward way to import your data and begin your analysis quickly.

## 2.4 Source an entire R script

Sometimes you may want to run an entire script all at once. This can be useful when using multiple scripts that must be run in a structured order, or when storing all of your custom functions in a separate script.

We can use the source() function to execute an entire script at once.

**Example:**

script\_file <- file.path("path/to/yourscript.R")  
source(script\_file)

This will run all of the operations in that script in your active R sessions. All libraries, variables, and functions that are loaded or created in your sourced script will then be available in your session.

You can use the exists() function to create to make sure that the path you provide leads to an existing script. This function will return either TRUE or FALSE. This is also useful for checking if a data file is present in an expected location before loading and execute additional functions, especially when combined in an if statement.

A few things to watch out for: 1. Loading libraries may cause name conflicts for functions with shared names. 2. Similarly, sourcing a script will overwrite existing variables with the same name. 3. Sourcing script may take a long time depending on the complexity of the script.

## 2.5 Additional base R functions

Here are some additional functions that you may encounter in during the EpidemiaR Training sections.

* Sys.Date()/Sys.Time() will return the active data or time.
* nrow() will return the number of rows in a dataframe.
* length() will return the number of objects in a vector, or the number of columns in a dataframe.
* unique() will return all of the unique entries in a vector.

# 3. Chaining multiple steps using pipes

|  |
| --- |
| Check your R version! |
| Some of the training material will use a special operator called the “native pipe” (which looks like this: |>). This function was first included in R version 4.1, which was released in May 2021.  If you haven’t updated your R in while then it may be time to update. You can check what your current version is by running R.version in the console. |

Pipes are a powerful tool in R that enable a more readable and efficient way of chaining multiple operations. The %>% pipe operator, introduced by the magrittr package and widely popularized by the dplyr package in the tidyverse, allows you to pass the result of one function directly into the next function.

## 3.1 Pipes in tidyverse

To use pipes, you’ll need to install and load the magrittr or tidyverse package. Remember that you only need to install the package one time, but you will have to load tidyverse if you want to use the pipe in a given script.

**Install tidyverse:**

install.packages("tidyverse")

**Load the tidyverse package:**

library(tidyverse)

### 3.1.1 Basic Pipe Syntax

The pipe operator %>% takes the output from the left-hand side and uses it as the first argument for the function on the right-hand side.

**Syntax:**

data %>%   
 function1() %>%  
 function2() %>%  
 function3()

#### 3.1.1.1 Example without Pipes

Consider the following operations on a data frame without using pipes:

**Example:**

library(dplyr)  
  
data <- mtcars  
filtered\_data <- filter(data, cyl == 6)  
selected\_data <- select(filtered\_data, mpg, hp)  
summarized\_data <- summarize(selected\_data, mean\_mpg = mean(mpg), mean\_hp = mean(hp))  
  
print(summarized\_data)

#### 3.1.1.2 Example with Pipes

The same operations can be performed more concisely using pipes:

**Example:**

library(dplyr)  
  
mtcars %>%  
 filter(cyl == 6) %>%  
 select(mpg, hp) %>%  
 summarize(mean\_mpg = mean(mpg), mean\_hp = mean(hp)) %>%  
 print()

### 3.1.2 Using Pipes with Custom Functions

Pipes can also be used with custom functions. Define your function and use it within a pipe.

**Example:**

custom\_function <- function(df) {  
 df %>%  
 filter(gear == 4) %>%  
 select(mpg, wt)  
}  
  
mtcars %>%  
 custom\_function() %>%  
 head()

### 3.1.3 Using the Dot Placeholder

Sometimes, the data does not automatically fit as the first argument in the next function. In such cases, use the dot (.) as a placeholder.

**Example:**

mtcars %>%  
 filter(cyl == 4) %>%  
 select(mpg, wt) %>%  
 {  
 n <- nrow(.)  
 mean\_wt <- mean(.$wt)  
 data.frame(n = n, mean\_wt = mean\_wt)  
 }

### 3.1.4 Nesting Pipes

Pipes can be nested for more complex operations. This is particularly useful when combining multiple data frames or performing multi-step operations.

**Example:**

data1 <- mtcars %>%  
 filter(cyl == 6) %>%  
 select(mpg, hp)  
  
data2 <- mtcars %>%  
 filter(cyl == 4) %>%  
 select(mpg, wt)  
  
combined\_data <- bind\_rows(data1, data2)  
print(combined\_data)

Using pipes in R enhances code readability and efficiency, allowing you to write more concise and maintainable code. Whether performing simple data manipulations or complex data transformations, pipes streamline your workflow by reducing the need for intermediate variables and nested function calls.

## 3.2 Native Pipe Operator in R (R 4.1+)

Starting with R version 4.1, a native pipe operator (|>) was introduced, providing an alternative to the %>% pipe from the magrittr package. The native pipe is part of the base R language, eliminating the need to load external packages for basic piping operations.

### 3.2.1 Basic Syntax

The native pipe operator works similarly to the %>% operator but uses |> instead.

**Syntax:**

data |>   
 function1() |>  
 function2() |>  
 function3()

Here is an example using traditional function chaining without the native pipe:

**Example:**

data <- mtcars  
filtered\_data <- filter(data, cyl == 6)  
selected\_data <- select(filtered\_data, mpg, hp)  
summarized\_data <- summarize(selected\_data, mean\_mpg = mean(mpg), mean\_hp = mean(hp))  
  
print(summarized\_data)

The same operations can be performed more concisely using the native pipe:

**Example:**

mtcars |>  
 (\(df) filter(df, cyl == 6))() |>  
 (\(df) select(df, mpg, hp))() |>  
 (\(df) summarize(df, mean\_mpg = mean(mpg), mean\_hp = mean(hp)))() |>  
 print()

### 3.2.2 Using Native Pipe with Custom Functions

Native pipes can also be used with custom functions, providing a clean and readable way to chain operations.

**Example:**

custom\_function <- function(df) {  
 df |>  
 filter(gear == 4) |>  
 select(mpg, wt)  
}  
  
mtcars |>  
 custom\_function() |>  
 head()

### 3.2.3 Using the Dot Placeholder

While the native pipe does not use the dot (.) placeholder in the same way as %>%, it can still be used flexibly with anonymous functions.

**Example:**

mtcars |>  
 (\(df) filter(df, cyl == 4))() |>  
 (\(df) select(df, mpg, wt))() |>  
 (\(df) {  
 n <- nrow(df)  
 mean\_wt <- mean(df$wt)  
 data.frame(n = n, mean\_wt = mean\_wt)  
 })()

The introduction of the native pipe operator in R 4.1 provides a built-in and efficient way to chain operations, enhancing code readability without relying on external packages. While it lacks some of the tools of %>%, such as the dot placeholder, it integrates seamlessly with base R functions and custom workflows.

You may encounter both types of pipes, and most often they work interchangeably. However, it is important to note the stuble differences between the tidyverse and native pipes, especially when debugging errors.

# 4. Joining data in tidyverse

Joining dataframes is a common task in data analysis, enabling you to combine datasets based on common keys. The dplyr package in the tidyverse provides several functions for different types of joins. This tutorial will focus on the left\_join() function.

## 4.1 Joins overview

### 4.1.1 Installing and Loading the tidyverse Package

First, ensure that you have the tidyverse package installed and loaded.

**Load the tidyverse package:**

library(tidyverse)

── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──  
✔ dplyr 1.1.4 ✔ readr 2.1.4  
✔ forcats 1.0.0 ✔ stringr 1.5.1  
✔ ggplot2 3.4.3 ✔ tibble 3.2.1  
✔ lubridate 1.9.3 ✔ tidyr 1.3.0  
✔ purrr 1.0.2   
── Conflicts ────────────────────────────────────────── tidyverse\_conflicts() ──  
✖ dplyr::filter() masks stats::filter()  
✖ dplyr::lag() masks stats::lag()  
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors

### 4.1.2 Example Dataframes

Let’s create two example dataframes for illustration.

**Creating Dataframe 1:**

df1 <- tibble(  
 id = c(1, 2, 3, 4),  
 name = c("Alice", "Bob", "Charlie", "David")  
)  
print(df1)

# A tibble: 4 × 2  
 id name   
 <dbl> <chr>   
1 1 Alice   
2 2 Bob   
3 3 Charlie  
4 4 David

**Creating Dataframe 2:**

df2 <- tibble(  
 id = c(1, 2, 4, 5),  
 score = c(85, 90, 88, 92)  
)  
print(df2)

# A tibble: 4 × 2  
 id score  
 <dbl> <dbl>  
1 1 85  
2 2 90  
3 4 88  
4 5 92

### 4.1.3 Using left\_join()

The left\_join() function combines rows from df1 with matching rows from df2. If there is no match, the result will contain NA for the columns from df2.

**Syntax:**

left\_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)

* x: The left dataframe.
* y: The right dataframe.
* by: A character vector of variables to join by. If NULL, defaults to all variables with common names in x and y.

**Example:**

joined\_df <- left\_join(df1, df2, by = "id")  
print(joined\_df)

# A tibble: 4 × 3  
 id name score  
 <dbl> <chr> <dbl>  
1 1 Alice 85  
2 2 Bob 90  
3 3 Charlie NA  
4 4 David 88

### 4.1.4 Handling Different Column Names

If the key columns have different names in the dataframes, use the by argument to specify the columns to join by.

**Example with Different Column Names:**

df3 <- tibble(  
 student\_id = c(1, 2, 4, 5),  
 grade = c("A", "B", "B+", "A-")  
)  
  
joined\_df2 <- left\_join(df1, df3, by = c("id" = "student\_id"))  
print(joined\_df2)

# A tibble: 4 × 3  
 id name grade  
 <dbl> <chr> <chr>  
1 1 Alice A   
2 2 Bob B   
3 3 Charlie <NA>   
4 4 David B+

## 4.2 Example from training data

Comparing cases between different woredas is sometimes desirable when we compare ‘apples with apples’, and make the variable normalized for the population in each woreda. Let us look at the following question.

Question 1: We want to identify woredas that have the highest annual confirmed cases per population in 2020.

To do this task, we need two datasets: one with confirmed cases, and another with population totals, both at the woreda-level. First, let’s load the two datasets we will be using.

confirmed\_cases\_annual<- read\_csv("data/training\_case\_data\_long.csv")

Rows: 94176 Columns: 7  
── Column specification ────────────────────────────────────────────────────────  
Delimiter: ","  
chr (4): region, zone, woreda, data\_type  
dbl (2): year, count  
date (1): period  
  
ℹ Use `spec()` to retrieve the full column specification for this data.  
ℹ Specify the column types or set `show\_col\_types = FALSE` to quiet this message.

population\_annual <- read\_csv("data/training\_population\_data\_long.csv")

Rows: 4905 Columns: 5  
── Column specification ────────────────────────────────────────────────────────  
Delimiter: ","  
chr (3): region, zone, woreda  
dbl (2): year, population  
  
ℹ Use `spec()` to retrieve the full column specification for this data.  
ℹ Specify the column types or set `show\_col\_types = FALSE` to quiet this message.

Before we join two data frames, we need to identify which columns we can use to join them together.

There are two important things we need to do before we can do the join.

* Identify which columns are common in both data frames. Note that the columns we use for joining may not have similar names in both data frames.
* Make sure the common columns (fields) have the same data type. We cannot use columns that have different data types to join two data frames.

The following script does the task of joining the two data frames, where we choose the case data to be the receiving data frame and the population dataset the server data frame.

annual\_incidence <- confirmed\_cases\_annual %>%  
 filter( year == 2020) %>%  
 group\_by (region, zone, woreda, year) %>%  
 summarise(annual\_cases = sum(count, na.rm=TRUE)) %>%  
 left\_join(population\_annual, by = c("region" = "region",  
 "zone" = "zone",  
 "woreda" = "woreda",  
 "year" = "year")) %>%  
 mutate(api = annual\_cases/ population \* 1000) %>%  
 arrange(desc(api))

`summarise()` has grouped output by 'region', 'zone', 'woreda'. You can  
override using the `.groups` argument.

The function left\_join() assumes that the data frame on the left of it is the master (receiver) while the one on its right side is the server data frame. This makes sure all rows on the left side will be included in the output even if there are no corresponding data rows in the data frame on the right side of the function. Any observations from the left side with no match on the right will have an NA value associated with them in the new joined dataset for the joined fields.

## 4.3 Conclusion

Using the left\_join() function from the dplyr package in the tidyverse, you can easily join dataframes based on common keys. This is useful for combining related datasets and performing comprehensive data analysis. By understanding the basics of joins, you can leverage the power of dplyr to handle more complex data manipulations.