Introduction to R, Season 1

October, 2023

Table of Contents

# 1 About this Course

## 1.1 Curriculum

The course covers fundamentals of R, a high-level programming language, and use it to wrangle data for analysis and visualization.

## 1.2 Target Audience

The course is intended for researchers who want to learn coding for the first time with a data science application, or have explored programming and want to focus on fundamentals.

# 2 Intro to Computing

## 2.1 Goals of the course

* Fundamental concepts in high-level programming languages (R, Python, Julia, WDL, etc.) that is transferable: *How do programs run, and how do we solve problems using functions and data structures?*
* Beginning of data science fundamentals: *How do you translate your scientific question to a data wrangling problem and answer it?*
* 
* Figure : Data science workflow
* Find a nice balance between the two throughout the course: we will try to reproduce a figure from a scientific publication using new data.

## 2.2 What is a computer program?

* A sequence of instructions to manipulate data for the computer to execute.
* A series of translations: English <-> Programming Code for Interpreter <-> Machine Code for Central Processing Unit (CPU)

We will focus on English <-> Programming Code for R Interpreter in this class.

More importantly: **How we organize ideas <-> Instructing a computer to do something**.

## 2.3 A programming language has following elements:

* Grammar structure (simple building blocks)
* Means of combination to analyze and create content (examples around genomics provided, and your scientific creativity is strongly encouraged!)
* Means of abstraction for modular and reusable content (data structures, functions)
* Culture (emphasis on open-source, collaborative, reproducible code)

Requires a lot of practice to be fluent!

## 2.4 What is R and why should I use it?

It is a:

* Dynamic programming interpreter
* Highly used for data science, visualization, statistics, bioinformatics
* Open-source and free; easy to create and distribute your content; quirky culture

## 2.5 R vs. Python as a first language

In terms of our goals, recall:

* Fundamental concepts in high-level programming languages
* Beginning of data science fundamentals

There are a lot of nuances and debates, but I argue that Python is a better learning environment for the former and R is better for the latter.

Ultimately, either should be okay! Perhaps more importantly, *consider what your research group and collaborator are more comfortable with*.

## 2.6 Posit Cloud Setup

Posit Cloud/RStudio is an Integrated Development Environment (IDE). Think about it as Microsoft Word to a plain text editor. It provides extra bells and whistles to using R that is easier for the user.

Today, we will pay close attention to:

* Script editor: where sequence of instructions are typed and saved as a text document as a R program. To run the program, the console will execute every single line of code in the document.
* Console (interpreter): Instead of giving a entire program in a text file, you could interact with the R Console line by line. You give it one line of instruction, and the console executes that single line. It is what R looks like without RStudio.
* Environment: Often, code will store information *in memory*, and it is shown in the environment. More on this later.

## 2.7 Using Quarto for your work

Why should we use Quarto for data science work?

* Encourages reproducible workflows
* Code, output from code, and prose combined together
* Extendability to Python, Julia, and more.

More options and guides can be found in [Introduction to Quarto](https://quarto.org/docs/get-started/hello/rstudio.html) .

## 2.8 Grammar Structure 1: Evaluation of Expressions

* **Expressions** are be built out of **operations** or **functions**.
* Operations and functions combine **data types** to return another data type.
* We can combine multiple expressions together to form more complex expressions: an expression can have other expressions nested inside it.

For instance, consider the following expressions entered to the R Console:

18 + 21

## [1] 39

max(18, 21)

## [1] 21

max(18 + 21, 65)

## [1] 65

18 + (21 + 65)

## [1] 104

nchar("ATCG")

## [1] 4

Here, our input **data types** to the operation are **numeric** in lines 1-4 and our input data type to the function is **character** in line 5.

Operations are just functions in hiding. We could have written:

sum(18, 21)

## [1] 39

sum(18, sum(21, 65))

## [1] 104

Remember the function machine from algebra class? We will use this schema to think about expressions.



Figure : Function machine from algebra class.

If an expression is made out of multiple, nested operations, what is the proper way of the R Console interpreting it? Being able to read nested operations and nested functions as a programmer is very important.

3 \* 4 + 2

## [1] 14

3 \* (4 + 2)

## [1] 18

Lastly, a note on the use of functions: a programmer should not need to know how the function is implemented in order to use it - this emphasizes [abstraction and modular thinking](#X4289ef2db5109e8791918a3c7b0cce6f2f89fd2), a foundation in any programming language.

### 2.8.1 Data types

Here are some data types that we will be using in this course:

* **Numeric**: 18, 21, 65, 1.25
* **Character**: “ATCG”, “Whatever”, “948-293-0000”
* **Logical**: TRUE, FALSE

## 2.9 Grammar Structure 2: Storing data types in the global environment

To build up a computer program, we need to store our returned data type from our expression somewhere for downstream use. We can assign a variable to it as follows:

x = 18 + 21

If you enter this in the Console, you will see that in the Environment, the variable x has a value of 39.

### 2.9.1 Execution rule for variable assignment

Evaluate the expression to the right of =.

Bind variable to the left of = to the resulting value.

The variable is stored in the environment.

<- is okay too!

The environment is where all the variables are stored, and can be used for an expression anytime once it is defined. Only one unique variable name can be defined.

The variable is stored in the working memory of your computer, Random Access Memory (RAM). This is temporary memory storage on the computer that can be accessed quickly. Typically a personal computer has 8, 16, 32 Gigabytes of RAM. When we work with large datasets, if you assign a variable to a data type larger than the available RAM, it will not work. More on this later.

Look, now x can be reused downstream:

x - 2

## [1] 37

y = x \* 2

## 2.10 Grammar Structure 3: Evaluation of Functions

A function has a **function name**, **arguments**, and **returns** a data type.

### 2.10.1 Execution rule for functions:

Evaluate the function by its arguments, and if the arguments are functions or contains operations, evaluate those functions or operations first.

The output of functions is called the **returned value**.

sqrt(nchar("hello"))

## [1] 2.236068

(nchar("hello") + 4) \* 2

## [1] 18

## 2.11 Tips on Exercises / Debugging

Common errors:

* Syntax error.
* It did something else than I expected!
  + The function or operation does not accept the input data type.
  + Changing a variable without realizing you did so.

Solutions:

* Where is the problem?
* What kind of problem is it?
* Explain your problem to someone!

# 3 Working with data structures

## 3.1 Vectors

In the first exercise, you started to explore **data structures**, which store information about data types. You played around with **vectors**, which is a ordered collection of a data type. Each *element* of a vector contains a data type, and there is no limit on how big a vector can be, as long the memory use of it is within the computer’s memory (RAM).

We can now store a vast amount of information in a vector, and assign it to a single variable. We can now use operations and functions on a vector, modifying many elements within the vector at once! This fits with the *theme of abstraction and modular organization* described in the first lesson!

We often create vectors using the combine function, c() :

staff = c("chris", "shasta", "jeff")  
chrNum = c(2, 3, 1)

If we try to create a vector with mixed data types, R will try to make them be the same data type, or give an error:

staff = c("chris", "shasta", 123)  
staff

## [1] "chris" "shasta" "123"

Our numeric got converted to character so that the entire vector is all characters.

### 3.1.1 Using operations on vectors

Recall from the first class:

* Expressions are be built out of **operations** or **functions**.
* Operations and functions combine **data types** to return another data type.

Now that we are working with data structures, the same principle applies:

* Operations and functions combine **data structures** to return another data structure (or data type!).

What happens if we use some familiar operations we used for numerics on a numerical vector? If we multiply a numerical vector by a numeric, what do we get?

chrNum = chrNum \* 3  
chrNum

## [1] 6 9 3

All of chrNum’s elements tripled! Our multiplication operation, when used on a *numeric vector with a numeric*, has a *new* meaning: it multiplied all the elements by 3. Multiplication is an operation that can be used for multiple data types or data structures: we call this property **operator overloading**. Here’s another example: *numeric vector multiplied by another numeric vector*:

chrNum \* c(2, 2, 0)

## [1] 12 18 0

but there are also limits: a numeric vector added to a character vector creates an error:

#chrNum + staff

When we work with operations and functions, we must be mindful what inputs the operation or function takes in, and what outputs it gives, no matter how “intuitive” the operation or function name is.

### 3.1.2 Subsetting vectors explicitly

In the exercise this past week, you looked at a new operation to subset elements of a vector using brackets.

Inside the bracket is either a single numeric value or an a **numerical indexing vector** containing numerical values. They dictate which elements of the vector to return.

staff[2]

## [1] "shasta"

staff[c(1, 2)]

## [1] "chris" "shasta"

small\_staff = staff[c(1, 2)]

In the last line, we created a new vector small\_staff that is a subset of the staff given the indexing vector c(1, 2). We have three vectors referenced in one line of code. This is tricky and we need to always refer to our rules step-by-step: evaluate the expression right of the =, which contains a vector bracket. Follow the rule of the vector bracket. Then store the returning value to the variable left of =.

Alternatively, instead of using numerical indexing vectors, we can use a **logical indexing vector**. The logical indexing vector must be the *same length* as the vector to be subsetted, with TRUE indicating an element to keep, and FALSE indicating an element to drop. The following block of code gives the same value as before:

staff[c(TRUE, FALSE, FALSE)]

## [1] "chris"

staff[c(TRUE, TRUE, FALSE)]

## [1] "chris" "shasta"

small\_staff = staff[c(TRUE, TRUE, FALSE)]

### 3.1.3 Subsetting vectors implicitly

Here are two applications of subsetting on vectors that need distinction to write the correct code:

1. **Explicit subsetting**: Suppose someone approaches you a 100-length vector of people’s ages, and say that they want to subset to the first 10 elements.
2. **Implicit subsetting**: Suppose someone approaches you a 100-length vector of people’s ages, and say that they want to subset to elements < 18 age.

We already know how to explicitly subset:

set.seed(123) #don't worry about this function  
age = round(runif(100, 1, 100)) #don't worry about these functions  
first\_ten\_age = age[1:10]

For implicit subsetting, we don’t know which elements to select off the top of our head! If we know which elements have less than 18, then we can give the elements for an explicit subset. Therefore, we need to create a logical indexing vector using a **comparison operator**:

indexing\_vector = age < 18  
indexing\_vector

## [1] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE  
## [13] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE  
## [25] FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE FALSE  
## [37] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE TRUE TRUE FALSE FALSE  
## [49] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE  
## [61] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
## [73] FALSE TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE  
## [85] TRUE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE  
## [97] FALSE TRUE FALSE FALSE

The comparison operator < compared the numeric value of age to see which elements of age is less than 18, and then returned a logical vector that has TRUE if age is less than 18 at that element and FALSE otherwise.

Then,

age\_young = age[indexing\_vector]  
age\_young

## [1] 6 11 5 16 3 15 16 15 6 13 14 10 1 12 11 14 10

We could have done this all in one line without storing the indexing vector as a variable in the environment:

age\_young = age[age < 18]

We have the following comparison operators in R:

< less than

<= less or equal than

== equal to

!= not equal to

> greater than

>= greater than or equal to

You can also put these comparison operators together to form more complex statements, which you will explore in this week’s exercise.

Another example:

age\_90 = age[age == 90]  
age\_90

## [1] 90 90 90

age\_not\_90 = age[age != 90]  
age\_not\_90

## [1] 29 79 41 88 94 6 53 89 56 46 96 46 68 58 11 25 5 33 95 89 70 64 99 66 71  
## [26] 55 60 30 16 96 69 80 3 48 76 22 32 24 15 42 42 38 16 15 24 47 27 86 6 45  
## [51] 80 13 57 21 14 76 38 67 10 39 28 82 45 81 81 80 45 76 63 71 1 48 23 39 62  
## [76] 36 12 25 67 42 79 11 44 99 89 89 18 14 66 35 66 33 20 78 10 47 52

For most of our subsetting tasks on vectors (and dataframes below), we will be encouraging implicit subsetting. The power of implicit subsetting is that you don’t need to know what your vector contains to do something with it! This technique is related to *abstraction* in programming mentioned in the first lesson: by using expressions to find the specific value you are interested instead of *hard-coding* the value explicitly, it generalizes your code to handle a wider variety of situations.

## 3.2 Dataframes

Before we dive into dataframes, check that the tidyverse package is properly installed by loading it in your R Console:

library(tidyverse)

## Warning: package 'tidyverse' was built under R version 4.0.3

## Warning: package 'purrr' was built under R version 4.0.5

## Warning: package 'stringr' was built under R version 4.0.3

Here is the data structure you have been waiting for: the **dataframe**. A dataframe is a spreadsheet such that each column must have the same data type. Think of a bunch of vectors organized as columns, and you get a dataframe.

For the most part, we load in dataframes from a file path (although they are sometimes created by combining several vectors of the same length, but we won’t be covering that here):

load(url("https://github.com/fhdsl/S1\_Intro\_to\_R/raw/main/classroom\_data/CCLE.RData"))

### 3.2.1 Using functions and operations on dataframes

We can run some useful functions on dataframes to get some useful properties, similar to how we used length() for vectors:

nrow(metadata)

## [1] 1864

ncol(metadata)

## [1] 30

dim(metadata)

## [1] 1864 30

colnames(metadata)

## [1] "ModelID" "PatientID" "CellLineName"   
## [4] "StrippedCellLineName" "Age" "SourceType"   
## [7] "SangerModelID" "RRID" "DepmapModelType"   
## [10] "AgeCategory" "GrowthPattern" "LegacyMolecularSubtype"  
## [13] "PrimaryOrMetastasis" "SampleCollectionSite" "Sex"   
## [16] "SourceDetail" "LegacySubSubtype" "CatalogNumber"   
## [19] "CCLEName" "COSMICID" "PublicComments"   
## [22] "WTSIMasterCellID" "EngineeredModel" "TreatmentStatus"   
## [25] "OnboardedMedia" "PlateCoating" "OncotreeCode"   
## [28] "OncotreeSubtype" "OncotreePrimaryDisease" "OncotreeLineage"

The last function, colnames() returns a character vector of the column names of the dataframe. This is an important property of dataframes that we will make use of to subset on it.

We introduce an operation for dataframes: the dataframe$column\_name operation selects for a column by its column name and returns the column as a vector. For instance:

metadata$OncotreeLineage[1:5]

## [1] "Ovary/Fallopian Tube" "Myeloid" "Bowel"   
## [4] "Myeloid" "Myeloid"

metadata$Age[1:5]

## [1] 60 36 72 30 30

We treat the resulting value as a vector, so we can perform implicit subsetting:

metadata$OncotreeLineage[metadata$OncotreeLineage == "Myeloid"]

## [1] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [8] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [15] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [22] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [29] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [36] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [43] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [50] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [57] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [64] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"  
## [71] "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid" "Myeloid"

Lastly, try running View(metadata) in RStudio Console…whew, a nice way to examine your dataframe like a spreadsheet program!

### 3.2.2 “What do you want to do with this dataframe”?

Before diving into the technical part of subsetting dataframes, we will use different mindset to think about what we want to do with this dataframe as scientists.

Remember that a major theme of the course is about: **How we organize ideas <-> Instructing a computer to do something.**

Until now, we haven’t focused too much on how we organize our scientific ideas to interact with what we can do with code. Let’s write our code driven by our scientific curiosity.

Here’s a starting prompt:

In the dataframe you have here, which rows would you filter for and columns would you select that relate to a scientific question?

Use the implicit subsetting mindset here: ie. “I want to filter for rows such that the subtype is breast cancer and look at the Age and Sex.” and *not* “I want to filter for rows 20-50 and select columns 2 and 8”.

*Notice that when we filter for rows in an implicitly way, we often formulate criteria about the columns.*

(This is because we are guaranteed to have column names in dataframes. Some dataframes have row names, but because the data types are not guaranteed to have the same data type, it makes describing by row properties difficult.)

Let’s convert this into code!

metadata\_filtered = filter(metadata, OncotreeLineage == "Breast")  
breast\_metadata = select(metadata\_filtered, ModelID, Age, Sex)  
head(breast\_metadata)

## ModelID Age Sex  
## 1 ACH-000017 43 Female  
## 2 ACH-000019 69 Female  
## 3 ACH-000028 69 Female  
## 4 ACH-000044 47 Female  
## 5 ACH-000097 63 Female  
## 6 ACH-000111 41 Female

Here, filter() and select() are functions from the tidyverse package.

### 3.2.3 Filter rows

Let’s carefully a look what how the R Console is interpreting the filter() function:

* We evaluate the expression right of =.
* The first argument of filter() is a dataframe, which we give metadata.
* The second argument is strange: the expression we give it looks like a logical indexing vector built from a comparison operator, but the variable OncotreeLineage does not exist in our environment! Rather, OncotreeLineage is a column from metadata, and we are referring to it as a **data variable** in the context of the dataframe metadata. So, we make a comparison operation on the column OncotreeLineage from metadata and its resulting logical indexing vector is the input to the second argument.
  + How do we know when a variable being used is a variable from the environment, or a data variable from a dataframe? It’s not clear cut, but here’s a rule of thumb: most functions from the tidyverse package allows you to use data variables to refer to columns of a dataframe. We refer to documentation when we are not sure.
  + This encourages more *readable* code at the expense of consistency of referring to variables in the environment. The authors of this package [describes this trade-off](https://dplyr.tidyverse.org/articles/programming.html#data--and-env-variables).
* Putting it together, filter() takes in a dataframe, and an logical indexing vector described by data variables as arguments, and returns a data frame with rows that match condition described by the logical indexing vector.
* Store this in metadata\_filtered variable.

### 3.2.4 Select columns

Let’s carefully a look what how the R Console is interpreting the select() function:

* We evaluate the expression right of =.
* The first argument of filter() is a dataframe, which we give metadata.
* The second and third arguments are data variables referring the columns of metadata.
  + For certain functions like filter(), there is no limit on the number of arguments you provide. You can keep adding data variables to select for more column names.
* Putting it together, select() takes in a dataframe, and as many data variables you like to select columns, and returns a dataframe with the columns you described by data variables.
* Store this in breast\_metadata variable.

# 4 Functions and pipes

Today, we will understand deeply how functions work, and continue learning more functions on dataframes.

## 4.1 Functions deep dive



Figure : Function machine from algebra class.

We write functions for two main, often overlapping, reasons:

1. Following DRY (Don’t Repeat Yourself) principle: If you find yourself repeating similar patterns of code, you should write a function that executes that pattern. This saves time and the risk of mistakes.
2. Create modular structure and abstraction: Having all of your code in one place becomes increasingly complicated as your program grows. Think of the function as a mini-program that can perform without the rest of the program. Organizing your code by functions gives modular structure, as well as abstraction: you only need to know the function name, inputs, and output to use it and don’t have to worry how it works.

Some advice on writing functions:

* Code that has a well-defined set of inputs and outputs make a good function.
* A function should do only one, well-defined task.

### 4.1.1 Anatomy of a function definition

Recall that a function has a function name, input arguments, and a return value.

*Function definition consists of assigning a* ***function name*** *with a “function” statement that has a comma-separated list of named* ***function arguments****, and a* ***return expression****. The function name is stored as a variable in the global environment.*

In order to use the function, one defines or import it, then one calls it.

Example:

addFunction = function(argument1, argument2) {  
 result = argument1 + argument2   
 return(result)  
}  
z = addFunction(3, 4)

With function definitions, not all code runs from top to bottom. The first four lines defines the function, but the function is never run. It is called on line 5, and the lines within the function are executed.

When the function is called in line 5, the variables for the arguments are reassigned to function arguments to be used within the function and helps with the modular form. We need to introduce the concept of local and global environments to distinguish variables used only for a function from variables used for the entire program.

Some syntax equivalents on calling the function:

addFunction(3, 4)  
addFunction(argument1 = 3, argument2 = 4)  
addFunction(argument2 = 4, argument1 = 3)

but this *could* be different:

addFunction(4, 3)

### 4.1.2 Local and global environments

*{ } represents variable scoping: within each { }, if variables are defined, they are stored in a* ***local environment****, and is only accessible within { }. All function arguments are stored in the local environment. The overall environment of the program is called the* ***global environment*** *and can be also accessed within { }.*

The reason of having some of this “privacy” in the local environment is to make functions modular - they are independent little tools that should not interact with the rest of the global environment. Imagine someone writing a tool that they want to give someone else to use, but the tool depends on your environment, vice versa.

### 4.1.3 A step-by-step example

Using the addFunction function, let’s see step-by-step how the R interpreter understands our code:



Figure : We define the function in the global environment.

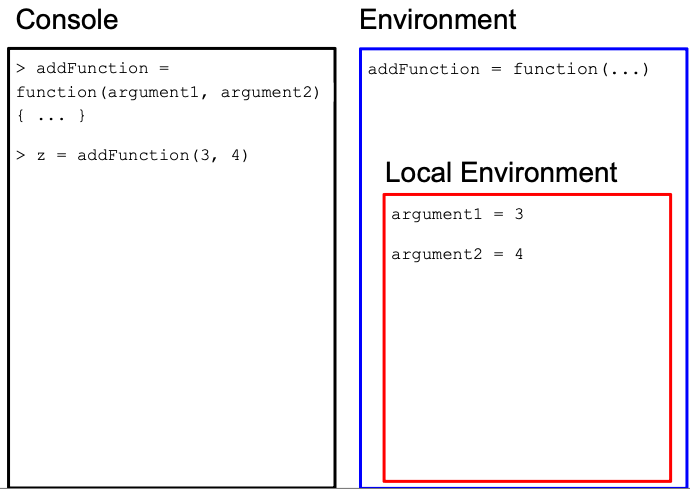


Figure : We call the function, and the function arguments 3, 4 are assigned to argument1 and argument2, respectively in the function’s local environment.



Figure : We run the first line of code in the function body. The new variable “result” is stored in the local environment because it is within { }.

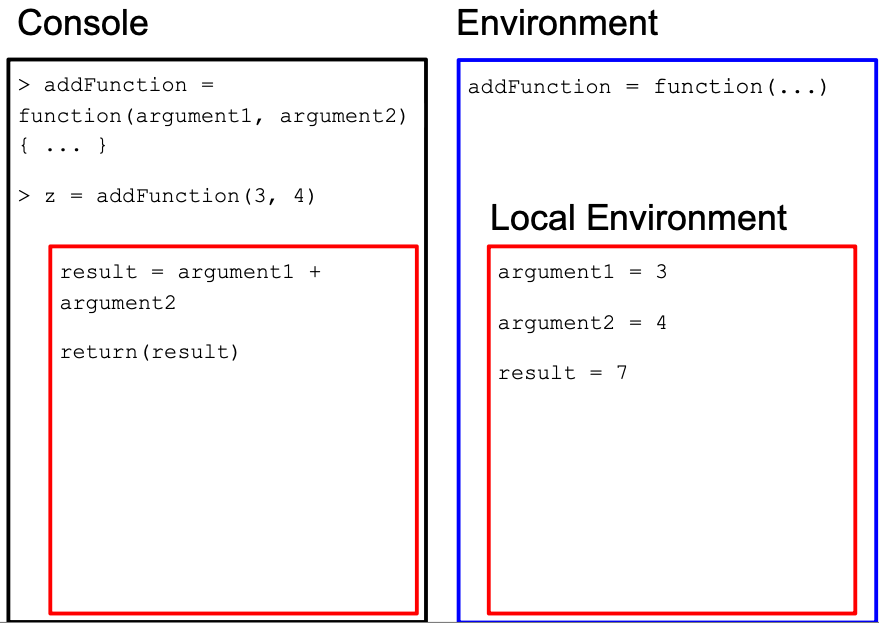


Figure : We run the second line of code in the function body to return a value.

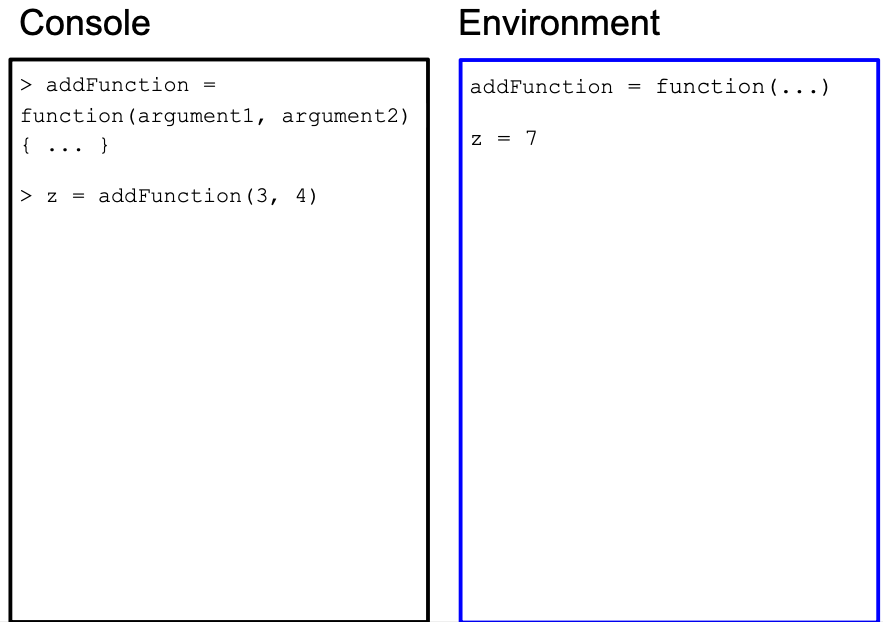


Figure : The return value from the function is assigned to the variable z in the global environment. All local variables for the function are erased now that the function call is over.

### 4.1.4 Function arguments create modularity

First time writers of functions might ask: why are variables we use for the arguments of a function *reassigned* for function arguments in the local environment? Here is an example when that process is skipped - what are the consequences?

x = 3  
y = 4  
addFunction = function(argument1, argument2) {  
 result = x + y   
 return(result)  
}  
z = addFunction(x, y)  
w = addFunction(10, -5)

What do you expect the value of z to be? How about w?

Here is the execution for w:

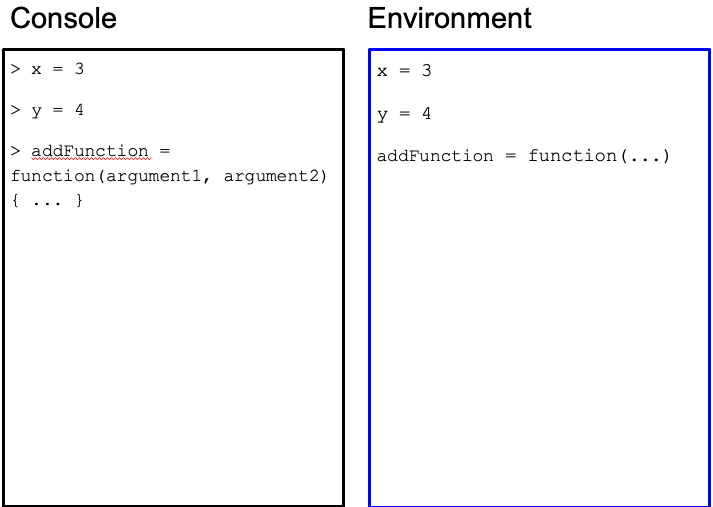
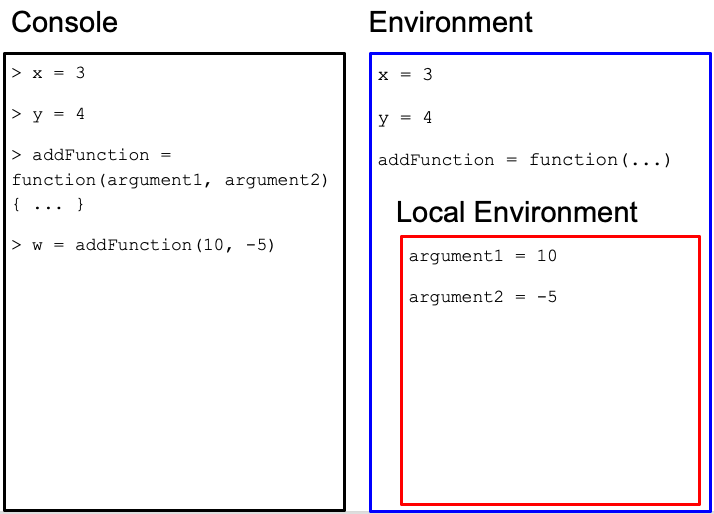
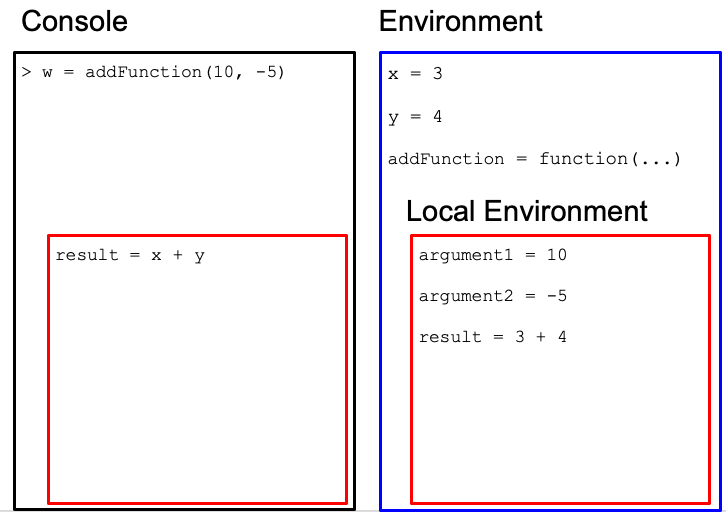


Figure : We define the variables and function in the global environment.

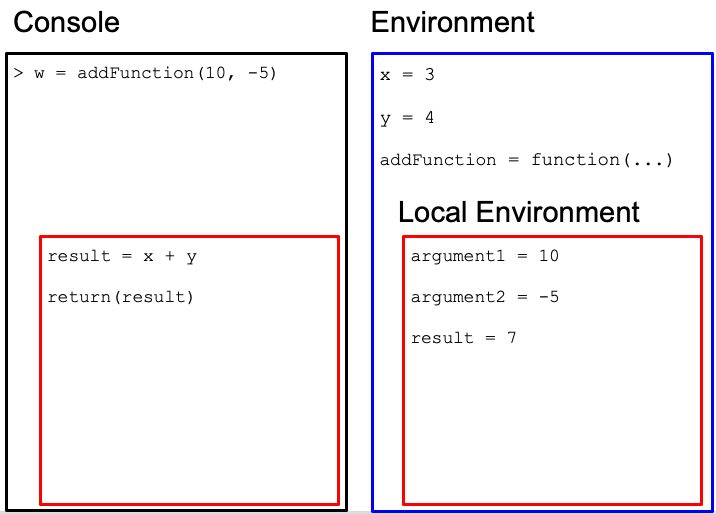


Figure : We run the second line of code in the function body to return a value.

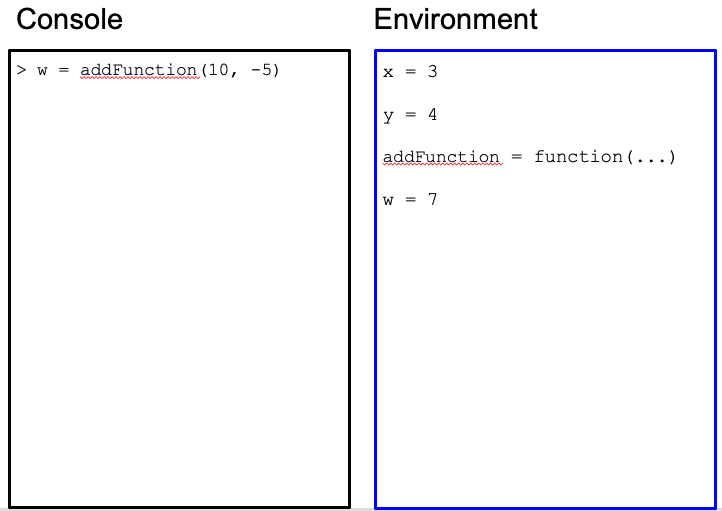


Figure : The return value from the function is assigned to the variable w in the global environment. All local variables for the function are erased now that the function call is over.

The function did not work as expected because we used hard-coded variables from the global environment and not function argument variables unique to the function use!

### 4.1.5 Exercises

* Create a function, called add\_and\_raise\_power in which the function takes in 3 numeric arguments. The function computes the following: the first two arguments are added together and raised to a power determined by the 3rd argument. The function returns the resulting value. Here is a use case: add\_and\_raise\_power(1, 2, 3) = 9 because the function will return this expression: (1 + 2) ^ 3. Another use case: add\_and\_raise\_power(3, 1, 2) = 16 because of the expression (3 + 1) ^ 2. Confirm with that these use cases work.
* Create a function, called my\_dim in which the function takes in one argument: a dataframe. The function returns the following: a length-2 numeric vector in which the first element is the number of rows in the dataframe, and the second element is the number of columns in the dataframe. Your result should be identical as the dim function. How can you leverage existing functions such as nrow and ncol? Use case: my\_dim(metadata) = c(1864, 30)
* Create a function, called medicaid\_eligible in which the function takes in one argument: a numeric vector called age. The function returns a numeric vector with the same length as age, in which elements are 0 for indicies that are less than 65 in age, and 1 for indicies 65 or higher in age. Use cases: medicaid\_eligible(c(30, 70)) = c(0, 1)

## 4.2 Pipes

Sometimes, in data analysis, we want to transform our dataframe in multiple steps via different functions. In your exercise, you started combining filter() and select() using one line of code:

library(tidyverse)  
metadata = read.csv("https://github.com/caalo/Intro\_to\_R/raw/main/classroom\_data/CCLE\_metadata.csv")

breast\_metadata = select(filter(metadata, OncotreeLineage == "Breast"), ModelID, Age, Sex)

This is a bit hard to read. A computer doesn’t care how difficult it is to read this line of code, but there is a lot of instructions going on in one line of code. This multi-step function composition will lead to an unreadable pattern such as:

result = function1(function2(function3(dataframe, df\_col4, df\_col2), arg2), df\_col5, arg1)

To untangle this, you have to look into the middle of this code, and slowly step out of it.

To make this more readable, programmers came up with an alternative syntax for function composition via the **pipe** metaphor. The ideas is that we push data through a chain of connected pipes, in which the output of a pipe becomes the input of the subsequent pipe.

Instead of a syntax like result2 = function1(function2(function3(dataframe))),

we linearize it with the %>% symbol: result2 = dataframe %>% function1 %>% function2 %>% function3.

In the previous example,

result = function1(df\_col5, arg1) %>%  
 function2(arg2) %>%  
 function3(df\_col4, df\_col2)

Looks much easier to read. Notice that we have broken up one expression in to three lines of code for readability. If a line of code is incomplete (the first line of code is piping to somewhere unfinished), the R will treat the next line of code as part of the current line of code.

### 4.2.1 Exercises

* Rewrite the select() and filter() function composition example above using the pipe metaphor and syntax.

## 4.3 Modifying and creating new columns in dataframes

To put together what we have learned today, we will modify and create new columns in dataframes.

The mutate() function takes in the following arguments: the first argument is the dataframe of interest, and the second argument is a new or existing data variable that is defined in terms of other data variables.

We create a new column newAge that is 10 years older than the original Age column.

metadata$Age[1:10]

## [1] 60 36 72 30 30 64 63 56 72 53

metadata2 = mutate(metadata, newAge = Age + 10)  
metadata2$newAge[1:10]

## [1] 70 46 82 40 40 74 73 66 82 63

or medicaid\_eligible():

medicaid\_eligible = function(age) {  
 age[age < 65] = 0  
 age[age >= 65] = 1  
 return(age)  
}  
  
metadata$Age[1:10]

## [1] 60 36 72 30 30 64 63 56 72 53

metadata2 = mutate(metadata, medicaid = medicaid\_eligible(Age))  
metadata2$medicaid[1:10]

## [1] 0 0 1 0 0 0 0 0 1 0

table(metadata2$medicaid)

##   
## 0 1   
## 1148 348

### 4.3.1 All together now

Let’s put all of our analysis together via pipes:

breast\_metadata = metadata %>% filter(OncotreeLineage == "Breast") %>%  
 select(ModelID, Age, Sex) %>%  
 mutate(medicaid = medicaid\_eligible(Age))  
  
head(breast\_metadata)

## ModelID Age Sex medicaid  
## 1 ACH-000017 43 Female 0  
## 2 ACH-000019 69 Female 1  
## 3 ACH-000028 69 Female 1  
## 4 ACH-000044 47 Female 0  
## 5 ACH-000097 63 Female 0  
## 6 ACH-000111 41 Female 0

### 4.3.2 Alternative: Creating and modifying columns via $

Instead of mutate() function, we can also create a new or modify a column via the $ symbol:

metadata$medicaid = medicaid\_eligible(metadata$Age)

# 5 Data wrangling with Tidy data

## 5.1 Datasets within DepMap Project

The [Dependency Map project](https://depmap.org/portal/) is a multi-omics profiling of cancer cell lines combined with functional assays such as CRISPR and drug sensitivity to help identify cancer vulnerabilities and drug targets. Here are some of the data that we have public access.

* Metadata
* Somatic mutations
* Gene expression
* Drug sensitivity
* CRISPR knockout
* and more…

## 5.2 Data Science Workflow



We are now equipped with enough fundamental programming skills to apply it to various steps in the data science workflow. We start with *Transform* and *Visualize* with the assumption that our data is in a nice, “tidy format”. First, we need to understand what it means for a data to be “tidy”.

## 5.3 Describing a tidy dataset

Here, we describe a standard of organizing data. It is important to have standards, as it facilitates a consistent way of thinking about data organization and building tools (functions) that make use of that standard. The principles of **tidy data**, developed by Hadley Wickham:

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

If you want to be technical about what variables and observations are, Hadley Wickham describes:

A **variable** contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An **observation** contains all values measured on the same unit (like a person, or a day, or a race) across attributes.



Figure : A tidy dataframe.

### 5.3.1 Examples of Tidy data

Let’s practice describing the observations, variables, and values of tidy dataframes:

library(tidyverse)

## Warning: package 'tidyverse' was built under R version 4.0.3

## Warning: package 'purrr' was built under R version 4.0.5

## Warning: package 'stringr' was built under R version 4.0.3

library(palmerpenguins)  
load(url("https://github.com/caalo/Intro\_to\_R/raw/main/classroom\_data/CCLE.RData"))

| Dataframe | The observation is | Some variables are | Some values are |
| --- | --- | --- | --- |
| metadata | Cell line | ModelID, Age, OncotreeLineage | “ACH-000001”, 60, “Myeloid” |
| expression |  |  |  |
| mutation |  |  |  |
| penguins |  |  |  |
| table1 |  |  |  |

## 5.4 Transform

When given a tidy dataset, we often still have to do some transformations on it to get it in a form so that we can perforom our analysis and visualization. Here are some common transformation tasks:

### 5.4.1 “The rows and columns I need can be subsetted from the dataframe.”

We use filter() and select() as we have learned already.

breast\_metadata = metadata %>% filter(OncotreeLineage == "Breast") %>%  
 select(ModelID, Age, Sex)

### 5.4.2 “The column I want is an operation or function of other columns I already have in the dataframe.”

We use mutate() or $ operation as we have learned already.

expression = expression %>% mutate(log\_PIK3CA\_Exp = log(PIK3CA\_Exp))  
#or  
expression$log\_PIK3CA\_Exp = log(expression$PIK3CA\_Exp)

### 5.4.3 “The columns that I need are in two different dataframes with a common column. The rows (observations) of both dataframes represent the same thing.”

Suppose we have the following dataframes:

expression

| ModelID | PIK3CA\_Exp | log\_PIK3CA\_Exp |
| --- | --- | --- |
| “ACH-001113” | 5.138733 | 1.636806 |
| “ACH-001289” | 3.184280 | 1.158226 |
| “ACH-001339” | 3.165108 | 1.152187 |

metadata

| ModelID | OncotreeLineage | Age |
| --- | --- | --- |
| “ACH-001113” | “Lung” | 69 |
| “ACH-001289” | “CNS/Brain” | NA |
| “ACH-001339” | “Skin” | 14 |

Suppose that I want to compare the relationship between OncotreeLineage and PIK3CA\_Exp, but they are columns in different dataframes. We want a new dataframe that looks like this:

| ModelID | PIK3CA\_Exp | log\_PIK3CA\_Exp | OncotreeLineage | Age |
| --- | --- | --- | --- | --- |
| “ACH-001113” | 5.138733 | 1.636806 | “Lung” | 69 |
| “ACH-001289” | 3.184280 | 1.158226 | “CNS/Brain” | NA |
| “ACH-001339” | 3.165108 | 1.152187 | “Skin” | 14 |

We see that in both dataframes, the rows (observations) represent cell lines with a common column ModelID, so let’s merge these two dataframes together, using full\_join():

merged = full\_join(metadata, expression, by = "ModelID")

The number of rows and columns of metadata:

dim(metadata)

## [1] 1864 30

The number of rows and columns of expression:

dim(expression)

## [1] 1450 537

The number of rows and columns of merged:

dim(merged)

## [1] 1864 566

We see that the number of *columns* in merged combines the number of columns in metadata and expression, while the number of *rows* in merged is the larger of the number of rows in metadata and expression : full\_join() keeps all observations common to both dataframes based on the common column defined via the by argument.

Therefore, we expect to see NA values in merged, as there are some cell lines that are not in expression dataframe.

There are variations of this function depending on your application:



Given xxx\_join(x, y, by = "common\_col"),

* full\_join() keeps all observations.
* left\_join() keeps all observations in x.
* right\_join() keeps all observations in y.
* inner\_join() keeps observations common to both x and y.

### 5.4.4 “The rows I want is described by a column. The columns I want need to be summarized from other columns.”

In a dataset, there may be multiple levels of observations, and which level of observation we examine depends on our scientific question. For instance, in metadata, the observation is cell lines. However, perhaps we want to understand properties of metadata in which the observation is the cancer type, OncotreeLineage. Suppose we want the mean age of each cancer type, and the number of cell lines that we have for each cancer type. This is a scenario in which the *desired rows are described by a column*, OncotreeLineage, and the columns, such as mean age, need to be summarized from other columns.

As an example, this dataframe is transformed from:

| ModelID | OncotreeLineage | Age |
| --- | --- | --- |
| “ACH-001113” | “Lung” | 69 |
| “ACH-001289” | “Lung” | 23 |
| “ACH-001339” | “Skin” | 14 |
| “ACH-002342” | “Brain” | 23 |
| “ACH-004854” | “Brain” | 56 |
| “ACH-002921” | “Brain” | 67 |

into:

| OncotreeLineage | MeanAge | Count |
| --- | --- | --- |
| “Lung” | 46 | 2 |
| “Skin” | 14 | 1 |
| “Brain” | 48.67 | 3 |

We use the functions group\_by() and summarise() :

metadata\_by\_type = metadata %>%   
 group\_by(OncotreeLineage) %>%   
 summarise(MeanAge = mean(Age),  
 Count = n())

## `summarise()` ungrouping output (override with `.groups` argument)

The group\_by() function returns the identical input dataframe but remembers which variable(s) have been marked as grouped.

The summarise() returns one row for each combination of grouping variables, and one column for each of the summary statistics that you have specified.

## 5.5 Tidy Data Tutor

Guest speaker: [Sean Kross](https://seankross.com/), author of [Tidy Data Tutor](https://tidydatatutor.com/).

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | Chris Lo |
| Lecturer | Chris Lo |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2023-10-20   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2023-03-28 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.7 2023-02-24 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.1 2023-03-23 [1] CRAN (R 4.0.2)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.20 2023-01-17 [1] CRAN (R 4.0.2)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## htmltools 0.5.5 2023-03-23 [1] CRAN (R 4.0.2)   
## knitr 1.33 2023-03-28 [1] Github (yihui/knitr@a1052d1)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 1.1.0 2023-03-14 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2023-03-28 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.3 2022-04-02 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2023-03-28 [1] Github (R-lib/testthat@e99155a)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2023-03-28 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 6 References