

GIS in R: Fundamentals and Economic Applications

Dr. Gaby Perez-Quesada

2026-01-01

Table of contents

Welcome!	4
1 Syllabus	5
1.1 Course Information	5
1.2 Learning Objectives	5
1.3 Prerequisites	5
1.4 Lecture Topics	6
1.5 Course Materials	7
1.6 Grading	7
1.7 Academic Integrity	7
1.8 Generative AI Tools in Coursework	8
1.9 Accommodations for Students with Disabilities	8
1.10 Online@UT	8
2 Getting Started with R	9
2.1 Why use R?	9
2.2 Installation: R and RStudio	9
2.3 Navigating RStudio	10
2.4 Scripts	10
2.5 Quarto	11
2.6 Working directory	11
2.7 Create your R project	12
2.8 Quarto file (.qmd) overview	13
3 hello Quarto	14
3.1 Headings	14
3.1.1 subsection	14
3.2 Bold and italic text	14
3.3 Adding figures and links	14
3.4 Code chunks	15
3.5 Common chunk options	16
3.6 Installing packages (run once) and loading library	16
3.7 Create an object	17
3.8 Using built-in datasets in R	17

4	R Basics	18
4.1	Create an object	18
4.2	Object structure	19
4.2.1	Data types	19
4.2.2	Variables types	20
4.3	Functions	22
4.4	Piping (%>%)	24
5	Data Wrangling	26
5.1	Loading data from a file	26
5.2	Looking at the data	27
5.3	Cleaning variable names	29
5.4	Selecting and filtering data	29
5.5	Creating variables	29
5.6	Aggregating data	30
5.7	Multi-row calculations	30
5.8	Re-code values using logic and conditions	31
5.9	Missing values	33
5.10	Joining data	33
	References	37

Welcome!



This book was created in order to host all of the materials for AREC 493 class.

Spatial data is increasingly essential for understanding economic activity, policy and development. This course introduces students to the use of R as a Geographic Information System (GIS) for economic analysis. Students will learn core GIS concepts alongside the basics of R programming, with a focus on practical applications in economics. The course closely integrates *tidyverse* functions with GIS tools to develop efficient and intuitive coding. Topics include spatial data processing, covering the import, management, and manipulation of raster and vector data, as well as visualization of spatial data through maps to identify spatial patterns. By the end, students will have the skills to integrate spatial analysis into economic research, enabling them to work with powerful and flexible tools for investigating real-world economic questions.

Be aware that the book is currently being written and will be completed by the end of the semester. Let me know if you think any changes are necessary!

1 Syllabus

1.1 Course Information

Course: AREC 493

Instructor: Dr. Gaby Perez-Quesada

Email: gperezqu@utk.edu

Term: Spring 2026

Times: T & R, 2:30-3:45pm

Location: Morgan Hall 212B

Office Hours: T & R, 3:45-5:00pm

1.2 Learning Objectives

- Use R and *tidyverse* tools to import, clean, and manipulate spatial data efficiently and reproducibly
- Explain and apply core GIS concepts, including spatial data types, projections, and coordinate reference systems
- Work with vector and raster datasets, performing basic operations, analysis, and integration between data types
- Visualize spatial data effectively through maps to identify and interpret spatial patterns
- Access and process real-world spatial datasets from public sources for applied economic analysis

1.3 Prerequisites

AREC 270. Prior coursework in basic statistics or econometrics is recommended. Students should be comfortable with fundamental concepts such as descriptive statistics, regression analysis, and interpreting empirical results.

1.4 Lecture Topics

(Subject to change based on progression through the material)

Getting started with R

- Get started with running code
- Install and load libraries
- Load in data from files, the internet, or libraries
- Write good code and responsibly use AI in your coding
- Look at your data and get basic statistical results
- Manipulate your data and get it ready for analysis

GIS fundamentals

- Types of spatial data
- Projections and coordinate reference system

The basics of vector data handling using *sf* package

- Vector data (points, lines, polygons)
- Import and export vector data
- (re)projection of spatial datasets
- Single-layer geometrical operations (e.g., create buffers)

Spatial interactions of vector datasets

- Understand topological relations of multiple *sf* objects
- Spatially subset a layer based on another layer
- Extracting values from one layer to another layer

The basics of raster data handling using *raster* and *terra* packages

- Import and export raster data
- Stack raster data
- Quick plotting

Spatial interactions of vector and raster datasets

- Cropping a raster layer to the geographic extent of a vector layer
- Extracting values from a raster to a vector layer

Creating maps using *ggplot2* package

- Visualizing spatial data with *ggplot2*

Download and process publicly available datasets

- USDA NASS QuickStat (*tidyUSDA*)
- PRISM (*prism*)
- Daymet (*daymetr*)
- Cropland Data Layer (*CropScapeR*)
- Census (*tidycensus*)

1.5 Course Materials

This course will follow a set of high-quality, freely available online resources, alongside DataCamp Classroom. Students are encouraged to consult these materials to complement lectures and exercises.

Introduction to R

- [R for Data Science](#)
- [Introduction to Working with Data: R version](#)

R for GIS

- [R as GIS for Economists](#)
- [Spatial Statistics for Data Science: Theory and Practice with R](#)

1.6 Grading

Assignments: 30%

Midterm Exam: 25%

Participation: 15%

Final Project: 30%

1.7 Academic Integrity

In accordance with the college's academic honesty policy, students are expected to submit original work for all assignments. Any form of academic dishonesty will not be tolerated. Plagiarism includes copying answers, phrases, sentence structures, or ideas from any source without proper attribution.

This also applies to the use of artificial intelligence tools such as ChatGPT. While you may use AI to assist with coding or to explore solutions, you must not simply feed it questions and

submit the generated responses as your own work. All submitted assignments should reflect your understanding and effort.

1.8 Generative AI Tools in Coursework

Open Use Guidelines: Embrace and encourage AI use in assignments, with the requirement that students disclose any AI assistance.

AI policy: permitted in this course with attribution

In this course, students are allowed to use Generative AI Tools like ChatGPT to support their work. To maintain academic integrity, students must disclose any AI-generated material they use and properly attribute it, including in-text citations, quotations, and references.

A student should include the following statement in assignments to indicate use of a Generative AI Tool: “The author(s) would like to acknowledge the use of [Generative AI Tool Name], a language model developed by [Generative AI Tool Provider], in the preparation of this assignment. The [Generative AI Tool Name] was used in the following way(s) in this assignment [e.g., coding, brainstorming, grammatical correction, citation, which portion of the assignment].”

1.9 Accommodations for Students with Disabilities

I am available to discuss appropriate academic accommodations that may be required for student with disabilities. The University of Tennessee, Knoxville, is committed to providing an inclusive learning environment for all students. If you anticipate or experience a barrier in this course due to a chronic health condition, a learning, hearing, neurological, mental health, vision, physical, or other kind of disability, or a temporary injury, you are encouraged to contact Student Disability Services (SDS) at 865-974-6087 or sds@utk.edu. An SDS Coordinator will meet with you to develop a plan to ensure you have equitable access to this course. If you are already registered with SDS, please contact your instructor to discuss implementing accommodations included in your course access letter.

1.10 Online@UT

Students are required to routinely access their UTK email account and the course website located on the Online@UT (Canvas) portal

2 Getting Started with R

2.1 Why use R?

As stated on the [R Project website](#), R is a programming language and environment for statistical computing and graphics. It's flexible, easy to build on, and supported by a large, welcoming community.

Cost

R is free and open-source for everyone.

Reproducibility

Using a programming language for data management and analysis, rather than relying on Excel or other point-and-click tools, makes your work easier to reproduce, helps you catch mistakes more quickly, and can save you a lot of time and effort.

Community

The R community is huge and supportive. New packages and tools to tackle real-world problems are developed all the time, and the community helps test, improve, and share them.

2.2 Installation: R and RStudio

To get started, you'll need to have R running on your computer. There are a few options, but **RStudio** is a popular choice that makes working with R much easier.

Installing RStudio takes two steps. First, you'll need to install the R language itself.

How to install R

Visit this website <https://www.r-project.org> and download the latest version of R suitable for your computer. *Note:* Any mirror will work fine, but perhaps pick one close to you.

Once that's done, you can **install RStudio**

Go to this website <https://posit.co/download/rstudio-desktop/> and download the latest free Desktop version of RStudio suitable for your computer.

2.3 Navigating RStudio

Open RStudio. RStudio is divided into four main *panes*, each showing different information like your code, console, and files.

The source pane

By default, this pane appears in the upper-left corner. It's where you can write, run, and save your scripts (your scripts are basically the sets of commands you want R to execute. You can also use this pane to view your datasets (data frames) while working.

The R console pane

The R Console (usually in the lower-left pane) is where the R “engine” lives. When you run commands here, you'll see results, warnings, or error messages right away. You can type commands directly in the Console, but unlike scripts, these commands won't be saved for later, so it's best for quick tests or checks.

The environment pane

By default, this pane appears in the upper-right. It's mainly used to get a quick look at the objects in your R environment during the current session. These objects can include datasets you've imported, created, or modified, as well as parameters or vectors and lists you've defined. You can click the little arrow next to a data frame to explore its variables.

Plots, Viewer, Packages, and Help pane

The lower-right pane contains several useful tabs. The *Plots* tab displays your charts, graphs, and maps, while the *Viewer* tab shows interactive or HTML outputs. The *Help* tab gives access to R documentation and help files, and the *Files* tab works like a mini file explorer, letting you open, move, or delete files. Finally, the *Packages* tab lets you see which R packages are installed, add new ones, update or remove them, and load or unload them for your session.

2.4 Scripts

Scripts are a fundamental part of programming. They're files that store the commands you want R to run, like creating or modifying datasets and generating visualizations. You can save a script and run it again later, which comes with some big advantages:

Portability: you can easily share your work with others by sending them your script.

Reproducibility: using scripts makes it clear exactly what steps you took, so you, or anyone else, can repeat your analysis with confidence.

Example script

```
3 + 4
```

```
[1] 7
```

```
a <- 3 + 4 # you can annotate around your code!
```

2.5 Quarto

[Quarto](#) is a tool that lets you combine your R code, text, and visualizations in a single document. It's perfect for creating reports, tutorials, or assignments where you want your analysis and explanations to live side by side. With Quarto, you can run your code, include the results, and produce a polished document; all in one place, making your work easy to share and easy to reproduce.

We will use Quarto document (.qmd) to write and edit your code!

Free online resources:

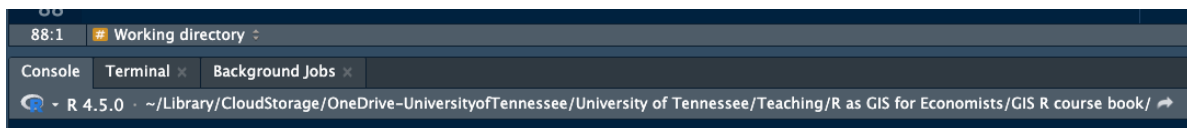
[Tutorial: Hello Quarto](#)

[R for Data Science: Quarto](#)

2.6 Working directory

The working directory (wd) is the root folder location used by R for your work, where R looks for and save files by default. R will save new files and outputs to this location, and will look for files to import (e.g. datasets) here as well.

The wd information appears at the top pf the Console pane. You can also print the current wd by running `getwd()` function.



```
getwd()
```

```
[1] "/Users/gperezqu/Library/CloudStorage/OneDrive-UniversityofTennessee/University of Tennessee/Teaching/R as GIS for Economists/GIS R course book/"
```

A recommended (and very efficient) approach is to use [R projects](#). When you work inside the R project, your `wd` is automatically set to the project's root folder, which contains the `.rproj` file. This happens whenever you open RStudio by double-clicking the “.rproj” file, so you don't need to set the working directory yourself.

2.7 Create your R project

1. Create a folder in your computer (e.g., `Project1`)
2. Open RStudio
3. Click File → New Project
4. Click Browse
5. Select the folder you created (`Project1`)
6. Click Create Project

RStudio will now:

- Open the project
- Create a `.rproj` in the folder
- Set the `wd` automatically

7. Confirm you're in the project

You should see:

- the project name in the top-right corner of RStudio
- a file ending in `.rproj` in the Files pane

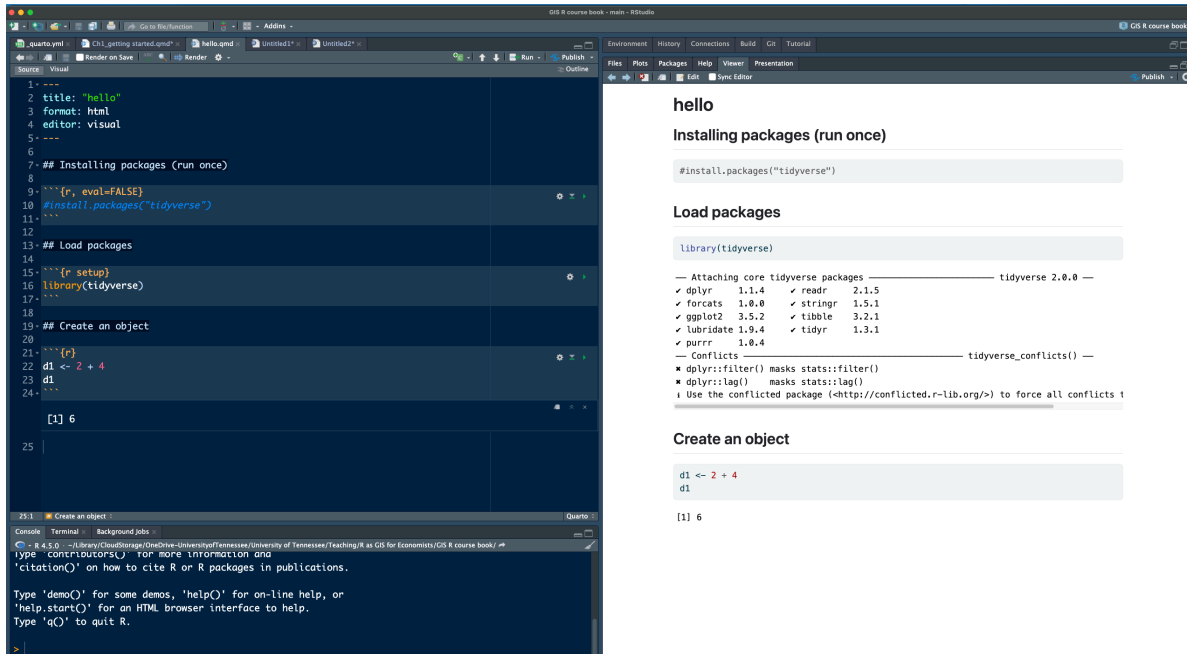
8. Create your first **Quarto** document

- Click File → New File → Quarto Document...
- Choose: Format: HTML; Engine: Knitr
- Click Create
- Save as → `hello`

Always open your project **by double-clicking** the `.rproj` file, not by opening RStudio first and setting the `wd` manually.

2.8 Quarto file (.qmd) overview

Below is a Quarto document (the .qmd file on the left) and its rendered output as an HTML page (on the right). The .qmd file is where you write your content and code, and when you render it, Quarto turns it into a polished document. You're not limited to HTML, you can also render the same file as a PDF, a Word document, and more.



- Select **Preview** in **Viewer Pane** if you want to see the rendered output on the **Viewer Pane**.
- Click on the **Render** button to render the file and preview the output.
- Two modes of the RStudio editor: **visual** (on the left) and **source** (on the right). **visual**: what-you-see-is-what-you-mean (WYSIWYM) experience, so you can format text without worrying about markdown syntax. **source**: plain-text markdown code.

3 hello Quarto

3.1 Headings

In this paper....

3.1.1 subsection

Some more text... This is **bold** text

My name is **Gaby**...

Use # symbols for section titles. Leave a blank line after headings for readability.

Section title

Subsection title

Smaller heading (more # = smaller heading)

3.2 Bold and italic text

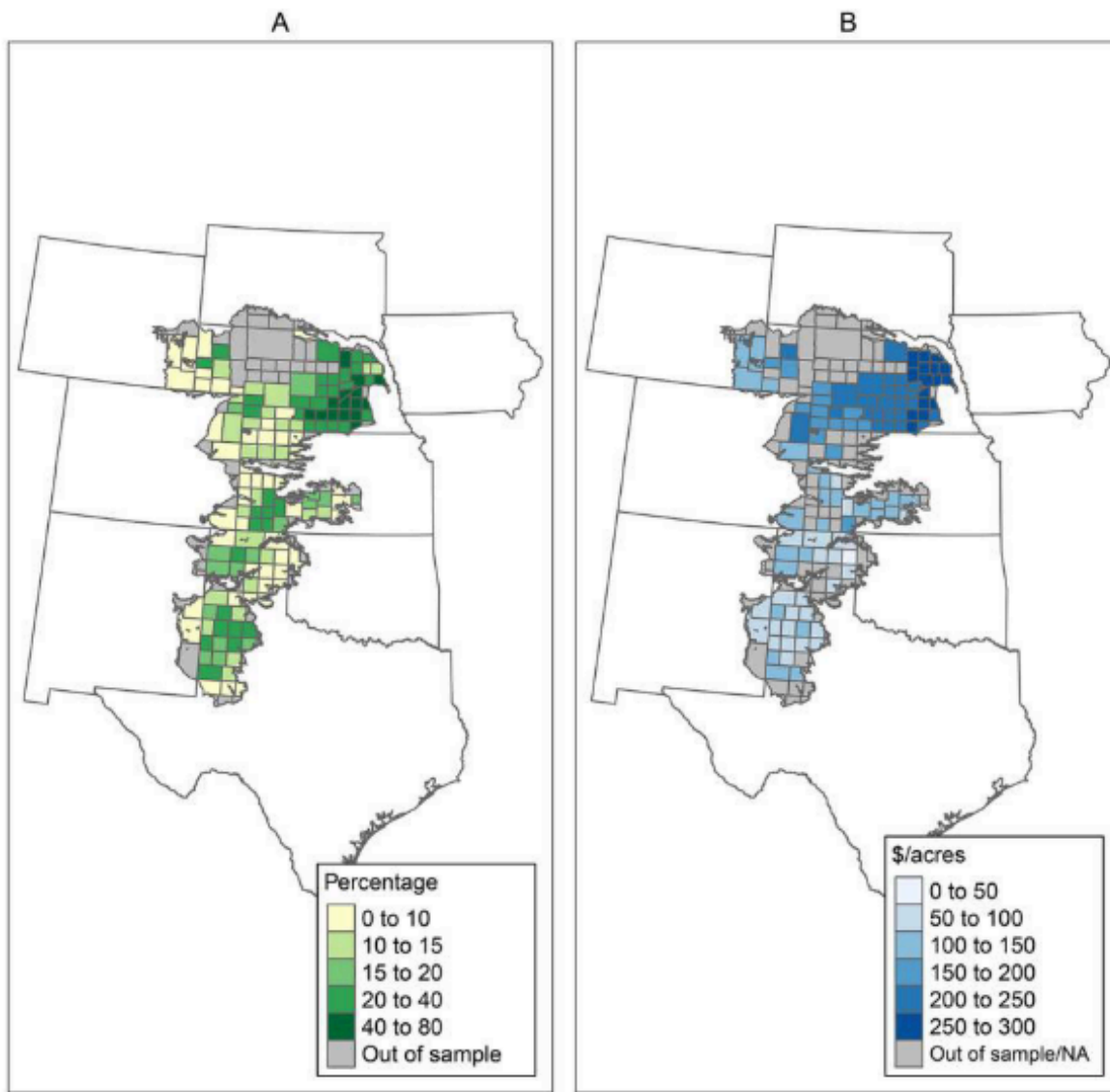
If you are using Source: **text** and *italic*

If you are using Visual: Click Format → Bold

3.3 Adding figures and links

Find any image and save it as .png in your R project.

The figure below shows that...



You can learn more about this paper [here](#).

How to include figures and links when using Visual?

3.4 Code chunks

R code chunks are marked with `{r}`. Type the chunk by hand or using keyboard shortcut.

Mac: Command + Option + I

Windows: Ctrl + Alt + I

```
# your R code goes here  
# comment
```

3.5 Common chunk options

- `eval = FALSE` → show code, don't run it
- `echo = FALSE` → run code, don't show it
- `include = FALSE` → run code, show nothing
- `message = FALSE` → hide package messages
- `warning = FALSE` → hide warnings

Example:

```
d1 <- 4 + 8  
d1
```

```
[1] 12
```

```
[1] 12
```

3.6 Installing packages (run once) and loading library

```
#install.packages("tidyverse")  
library(tidyverse) # Load packages
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --  
v dplyr      1.1.4      v readr      2.1.5  
v forcats   1.0.0      v stringr   1.5.1  
v ggplot2    3.5.2      v tibble    3.2.1  
v lubridate  1.9.4      v tidyr     1.3.1  
v purrr      1.0.4
```

```
-- Conflicts ----- tidyverse_conflicts() --
```

```
x dplyr::filter() masks stats::filter()
```

```
x dplyr::lag()     masks stats::lag()
```

```
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```


3.7 Create an object

Everything in R is an object, and R is an object-oriented language. An object exists once you assign a value to it.

After an object is created, it appears in the Environment pane. From there, you can use the object in your code (inspect it, manipulate it, change it, or redefine it by assigning it a new value).

```
# Defining an object (<-)
d1 <- 2 + 4
d1
```

```
[1] 6
```

3.8 Using built-in datasets in R

```
#data() # list of datasets
data(mtcars) # load mtcars

# create an object
my.data <- mtcars

# Inspect it
head(my.data)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

4 R Basics

4.1 Create an object

Everything in R is an object, and R is an object-oriented language. An object exists once you assign a value to it.

After an object is created, it appears in the Environment pane. From there, you can use the object in your code (inspect it, manipulate it, change it, or redefine it by assigning it a new value).

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr      1.1.4      v readr      2.1.5
v forcats    1.0.0      v stringr    1.5.1
v ggplot2    3.5.2      v tibble     3.2.1
v lubridate  1.9.4      v tidyr      1.3.1
v purrr      1.0.4
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
# Defining an object (<-)
d1 <- 2 + 4
d1
```

```
[1] 6
```

```
#data()
# Using built-in datasets in R
data(mtcars) # load mtcars

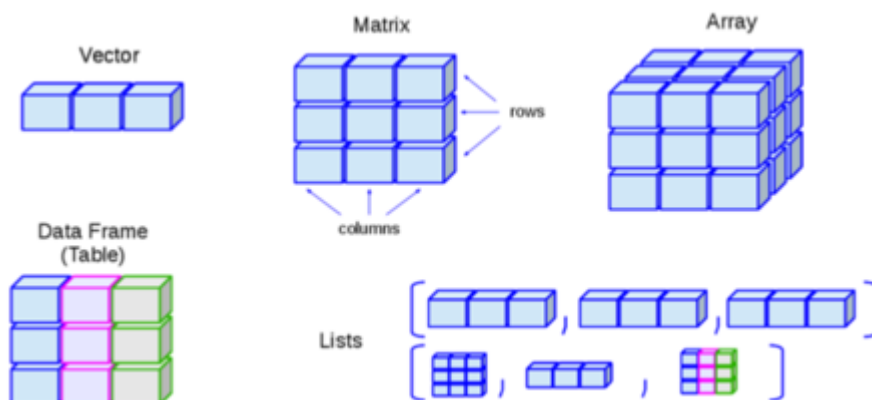
# create an object
```

```
my.data <- mtcars
# Inspect it
head(my.data)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

4.2 Object structure

Objects can be as simple as a single value (e.g., `d1`), or they can hold more structured data, such as vectors, tables, or datasets, like the examples shown in the figure below (borrowed from this online [R tutorial](#)).



4.2.1 Data types

In this course, we will be using mostly vector and data frame.

Vector: a collection of values of the same type (e.g., numbers or text) stored together in one object.

```
# Numeric vector
numbers <- c(1, 2, 3, 4, 5)
numbers
```

```
[1] 1 2 3 4 5
```

```
class(numbers)
```

```
[1] "numeric"
```

```
# Character vector
names <- c("Alice", "Bob", "Charlie")
class(names)
```

```
[1] "character"
```

data.frame: a table of data where each column is a vector, and all columns have the same number of rows.

```
data(mtcars) # load mtcars

# create an object
my.data <- mtcars
class(my.data)
```

```
[1] "data.frame"
```

4.2.2 Variables types

Variable	Description	Example
integer	Whole numbers	5, 160, -30
numeric	Decimals	0.5, -0.38, 234.567
character	Text	"A", "Nice", "welcome"
logical	Booleans	TRUE or FALSE
factor	Categorical	"green", "blue", "red"
empty	—	NULL

Examples

```
class(d1)
```

```
[1] "numeric"
```

```
# Numeric (decimal)
num <- 3.14
num2 <- 56 # R treats all numbers without a decimal as numeric by default

class(num)
```

```
[1] "numeric"
```

```
class(num2)
```

```
[1] "numeric"
```

```
# Integer
int <- as.integer(num)
class(int)
```

```
[1] "integer"
```

```
int <- 56L # L tells R to store this as an integer
class(int)
```

```
[1] "integer"
```

```
x <- 89
x <- 289

# Character (text)
char_var <- "welcome"

# Logical (TRUE/FALSE)
log_var <- TRUE

# Factor (categorical)
fact_var <- factor(c("green", "blue", "red"))

# variables type in my.data
class(my.data$hp)
```

```
[1] "numeric"
```

```
class(my.data$mpg)
```

```
[1] "numeric"
```

4.3 Functions

Functions are at the core of using R. Want to calculate, plot, or transform something? There's probably a function for it. R comes with lots built in, you can add more with packages, and you can even make your own! Learn more [here](#).

A function is like a little machine: you give it some inputs, it does something with them, and then it gives you an output. What comes out depends on the function you're using.

Inputs → [function] → Output

Simple functions

```
# Function: sum()
# Inputs: numbers
# Action: adds them together
# Output: the total
sum(4, 7, 10)
```

```
[1] 21
```

```
result <- sum(4, 7, 10)
result
```

```
[1] 21
```

```
# Function: as.numeric()
# Inputs: char_nums
# Action: convert to numeric
# Output: num_nums

# Convert a character vector to numeric
char_nums <- c("1", "2", "3", "4")
class(char_nums)
```

```
[1] "character"
```

```
# Convert to numeric
num_nums <- as.numeric(char_nums)
class(num_nums)
```

```
[1] "numeric"
```

Custom function

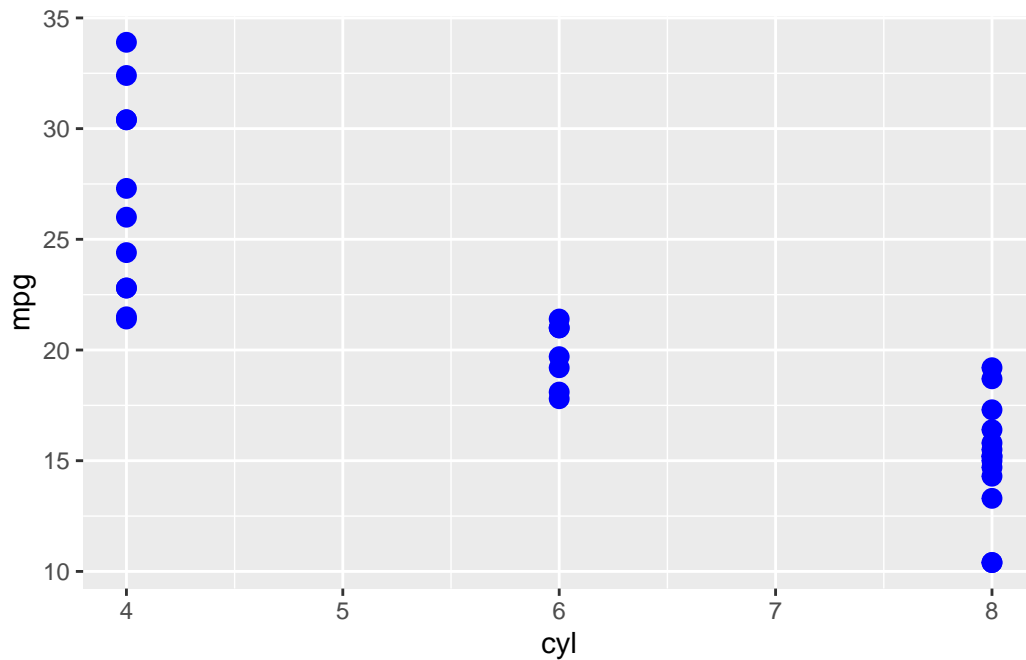
```
# Define a function that doubles a number
double_it <- function(x) {
  x * 2
}
```

```
# Use the function
double_it(4576)
```

```
[1] 9152
```

```
# Functions with multiple arguments
# Load ggplot2 package (install first if you don't have it)
library(ggplot2)

# Use the built-in dataset 'mtcars'
# We'll plot miles per gallon (mpg) vs number of cylinders (cyl)
ggplot(data = mtcars,           # Required argument: dataset
        mapping = aes(x = cyl, y = mpg)) + # Required: aesthetics
  geom_point(color = "blue", size = 3)      # Optional arguments inside geom_point()
```



4.4 Piping (%>%)

Instead of nesting functions inside each other, you can write code step by step, which is often easier to read.

`%>%` (the “then” operator): pipes send an object from function to function.

Using **Tidyverse** + `%>%` you can do complex tasks in easy-to-read, step-by-step code.

[Tidyverse](#) is a collection of R packages designed to work well together.

Example without a pipe

```
# Find the average mpg of cars with 6 cylinders
mean(subset(my.data, cyl == 6)$mpg)
```

```
[1] 19.74286
```

Example with a pipe

```
my.data %>%
  filter(cyl == 6) %>%
  pull(mpg) %>%
  mean()
```



```
[1] 19.74286
```

```
# save the avg mpg as an object
avg.mpg <- mtcars %>%
  filter(cyl == 6) %>%
  pull(mpg) %>%
  mean()

class(avg.mpg)
```

```
[1] "numeric"
```

5 Data Wrangling

This chapter walks through the most common steps involved in taking raw data and turning it into analysis-ready data. You'll learn how to import datasets, clean and transform variables, handle missing values, combine multiple data sources, and filter data to answer specific questions. Along the way, we'll introduce essential R data-management functions—mostly from the **tidyverse**—and show how to chain them together using pipes to create clear, readable workflows.

5.1 Loading data from a file

Often your data will be stored in a file that you need to load into R. Common examples include:

File type	Extension	Common function	Package
Excel spreadsheet	.xlsx	read_excel()	readxl
Comma-separated values	.csv	read_csv()	readr
Stata dataset	.dta	read_dta()	haven

Note: You can use `read_csv()` and `read_excel()` after loading **tidyverse**. You can also use **rio** package, which has an `import()` function that works for just about any data file type you might want.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ----- tidyverse 2.0.0 --
v dplyr      1.1.4      v readr      2.1.5
v forcats    1.0.0      v stringr    1.5.1
v ggplot2    3.5.2      v tibble     3.2.1
v lubridate  1.9.4      v tidyr      1.3.1
v purrr      1.0.4
-- Conflicts ----- tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become
```

```
water <- read_csv("../data/water_data.csv")
```

```
Rows: 76736 Columns: 6
```

```
-- Column specification -----
```

```
Delimiter: ","
```

```
dbl (6): year, field, price_water, q_water, center_pivot, county
```

```
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.
```

```
#water <- read_csv("water_data.csv")
```

```
ls(water)
```

```
[1] "center_pivot" "county"      "field"      "price_water" "q_water"
[6] "year"
```

5.2 Looking at the data

```
class(water)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
head(water)
```

```
# A tibble: 6 x 6
```

	year	field	price_water	q_water	center_pivot	county
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	1992	1	0.585	14.6	0	12
2	1993	1	0.639	10.8	0	12
3	1994	1	0.644	17.0	1	12
4	1995	1	0.570	18.7	1	12
5	1996	1	0.760	21.4	1	12
6	1997	1	0.768	15	1	12

```
water %>% glimpse()
```

```
Rows: 76,736
```

```
Columns: 6
```

```
$ year      <dbl> 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 200~
$ field     <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 4, 4, ~
$ price_water <dbl> 0.5845152, 0.6391169, 0.6440908, 0.5702707, 0.7600425, 0.~
$ q_water   <dbl> 14.58335, 10.79831, 17.02275, 18.71972, 21.36000, 15.0000~
$ center_pivot <dbl> 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ county    <dbl> 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 1~
```

```
summary(water)
```

year	field	price_water	q_water
Min. :1992	Min. : 1	Min. :0.01553	Min. : 0.000005
1st Qu.:1996	1st Qu.: 6015	1st Qu.:0.20199	1st Qu.:10.110240
Median :2000	Median :12099	Median :0.51795	Median :14.066380
Mean :2000	Mean :12058	Mean :0.71312	Mean :14.048993
3rd Qu.:2004	3rd Qu.:18118	3rd Qu.:0.98047	3rd Qu.:17.824100
Max. :2007	Max. :24086	Max. :5.42991	Max. :30.000000

center_pivot	county
Min. :0.0000	Min. : 1.00
1st Qu.:1.0000	1st Qu.: 7.00
Median :1.0000	Median :22.00
Mean :0.9261	Mean :20.83
3rd Qu.:1.0000	3rd Qu.:33.00
Max. :1.0000	Max. :41.00

```
nrow(water)
```

```
[1] 76736
```

```
ncol(water)
```

```
[1] 6
```

```
# type of variables in water
class(water$year)
```

```
[1] "numeric"
```

```
unique(water$year)
```

```
[1] 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006  
[16] 2007
```

5.3 Cleaning variable names

```
water <- water %>%  
  rename(county_code = county) #rename
```

5.4 Selecting and filtering data

```
# Create a subset with only two variables: year and county_code  
sub1 <- water %>%  
  select(year, county_code)  
  
# Get rid of the field variable  
sub2 <- water %>%  
  select(-field)  
  
# Create a subset containing only year=2000  
sub.2000 <- water %>%  
  filter(year == 2000)
```

5.5 Creating variables

```
water <- water %>%  
  mutate(year2 = year + 1,  
         year3 = year2 + 3) # takes our year variable and adds 1 to it  
  
# create a subset center_pivot == 1 and create a new variable = total cost  
cost <- water %>%
```

```

filter(center_pivot == 1) %>%
mutate(tot.cost = price_water*q_water)

options(scipen = 999)
summary(cost$tot.cost)

```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.000001	2.452759	6.182512	11.046254	15.240035	141.681762

5.6 Aggregating data

A dataset's observation level is simply what each row of the data represents. For example, **water** dataset contains one row per field for a given year, then the observation level is field-year.

Each row corresponds to one field in one specific year. Sometimes your data are more detailed than what you need for your analysis, and you may want to zoom out. For instance, you might start with field-year data but want to analyze outcomes at the field level overall. To do that, you need to aggregate the data.

In the tidyverse, this is usually done with `group_by()`, which tells R to perform calculations separately within each group. You'll then typically use `summarize()` to combine multiple rows into a single row per group. The resulting dataset will have an observation level based on whatever variable(s) you used in `group_by()`.

```

# get average total water cost by field
# averaged over all the years
avg.cost <- cost %>%
  group_by(field) %>%
  summarize(avg.cost = mean(tot.cost)) # add sum

```

5.7 Multi-row calculations

It's pretty common to need to use more than one row of data (but not the entire dataset) to calculate something you care about. For example, to calculate statistics by group, we can use `group_by()` to do by-group calculations. However, if we want to calculate by group and create a new column with that calculation rather than change the observation level, we want to follow that `group_by()` with a `mutate()` instead of a `summarize()`.

```
cost <- cost %>%
  group_by(field) %>%
  mutate(avg.cost = mean(tot.cost, na.rm = T),
         tot.use = sum(q_water))

# What if I need the avg cost and total use by field-year?
```

```
# Your time to practice
# 1) Create a subset that includes only years > 1999
# 2) create a new variable: q_water10 = (q_water + 10)
# 3) create a two new variables: min.use and min.price by year
```

```
data1 <- water %>%
  filter(year > 1999) %>%
  mutate(q_water10 = q_water + 10) %>%
  group_by(year) %>%
  mutate(min.use = min(q_water),
         min.price = min(price_water))
```

5.8 Re-code values using logic and conditions

Specific values

To re-code with simple logical criteria, you can use `replace()` within `mutate()`.

```
# avg cost is changed for field = 24

avg.cost <- avg.cost %>%
  mutate(avg.cost = replace(avg.cost, field == 24, 8.564))
```

Another function for simple logic is `ifelse()`. It applies a simple condition to a whole vector at once and return one value if the condition is true and another if it's false. It's especially useful for creating or transforming variables based on rules.

Let's create a new variable, `type`, that takes the value 1 when `q_water > mean(q_water)`, and 0 otherwise.

```
# calculate the mean of q_water
summary(water$q_water)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.000005	10.110240	14.066380	14.048993	17.824100	30.000000

```
water <- water %>%
  mutate(type = ifelse(q_water > 14.05, 1, 0))

class(water$type)
```

```
[1] "numeric"
```

```
# Convert type to factor variable and create labels.
water <- water %>%
  mutate(type = factor(type, levels = c(0, 1), labels = c("low", "high")))

?factor
class(water$type)
```

```
[1] "factor"
```

Complex logic

Use `case_when()` when you need to recode a variable into many groups or apply more complex logical rules. The function checks each row of the data and assigns a new value based on the conditions you specify.

Each `case_when()` rule has a condition on the left and a value on the right, separated by a tilde (~):

- Conditions go on the left-hand side (LHS)
- Assigned values go on the right-hand side (RHS)
- Rules are separated by commas

Now, we want to create `type2` but taking three values, low, medium and high.

```
summary(water$q_water)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.000005	10.110240	14.066380	14.048993	17.824100	30.000000


```

water <- water %>%
  mutate(type2 = case_when(
    q_water < 12 ~ 0,
    q_water >= 12 & q_water < 15 ~ 1,
    q_water >= 15 ~ 2
  ))

# using ifelse()
water <- water %>%
  mutate(type.test = ifelse(q_water < 12, 0, ifelse(q_water >= 12 & q_water < 15, 1, 2)))

# Convert type to factor variable and create labels.
water <- water %>%
  mutate(type_fac = factor(type2, levels = c(0, 1, 2), labels = c("low", "medium", "high")))

# or create a factor variable directly
water <- water %>%
  mutate(type2 = factor(case_when(
    q_water < 12 ~ 0,
    q_water >= 12 & q_water < 15 ~ 1,
    q_water >= 15 ~ 2),
    levels = c(0, 1, 2),
    labels = c("low", "medium", "high")))

```

5.9 Missing values

When we are cleaning our data, we need to handle missing values.

5.10 Joining data

Often, you'll work with multiple datasets that describe the same observations (for example, the same people, places, or time periods). To combine these datasets, you use a merge (or join) based on one or more key variables that appear in both datasets. These key variables act like an ID that tells R which rows belong together. At least one of the datasets should have only one row per key value, so R knows exactly how to match the observations without ambiguity.

```
# let's create a new dataset to merge with water
water.2 <- water %>%
  select(year, field) %>% # IDs variables
  mutate(
    x = rnorm(n()),
    y = rnorm(n())
  )
```

We now have a second dataset that includes variables `x` and `y`, and we want to bring those variables back into our original `water` dataset. To do this, we use `joins` functions, which are provided by the **dplyr** package (included in the **tidyverse**).

Join functions can be used in two ways: as standalone commands that create a new data frame, or inside a pipe (`%>%`) as part of a larger data-cleaning workflow.

In this example, we use `left_join()` as a standalone command to create a new data frame called `joined_data`. The first data frame listed (`water`) is the baseline dataset we want to keep, and the second data frame (`water.2`) is joined to it.

The `by =` argument tells R which columns to use to match rows between the two datasets. These columns must exist in both data frames and should uniquely identify the observations so the rows line up correctly.

```
joined_data <- left_join(water, water.2, by = c("year", "field")) # note we use two IDs vari
```

Left and right joins

```
left_join(), right_join()
```

Left and right joins are commonly used to add new information to an existing dataset.

The order of the datasets matters:

- In a left join, the first dataset is the baseline.
- In a right join, the second dataset is the baseline.

All rows from the baseline dataset are kept. Data from the other dataset is added only when there is a match on the identifier column(s). Rows in the secondary dataset that do not match are dropped.

If one row in the secondary dataset matches multiple baseline rows, the same information is added to each matching row. If a baseline row matches multiple rows in the secondary dataset, new rows will be created, increasing the size of the data.

Should we use a left join or a right join? To answer this question, ask yourself: Which dataset do I want to keep all rows from?

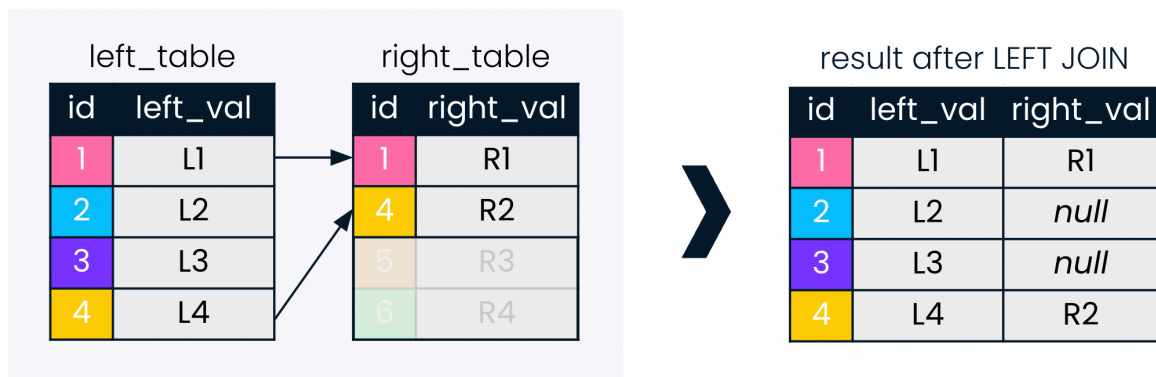


Figure 5.1: Source: Datacamp

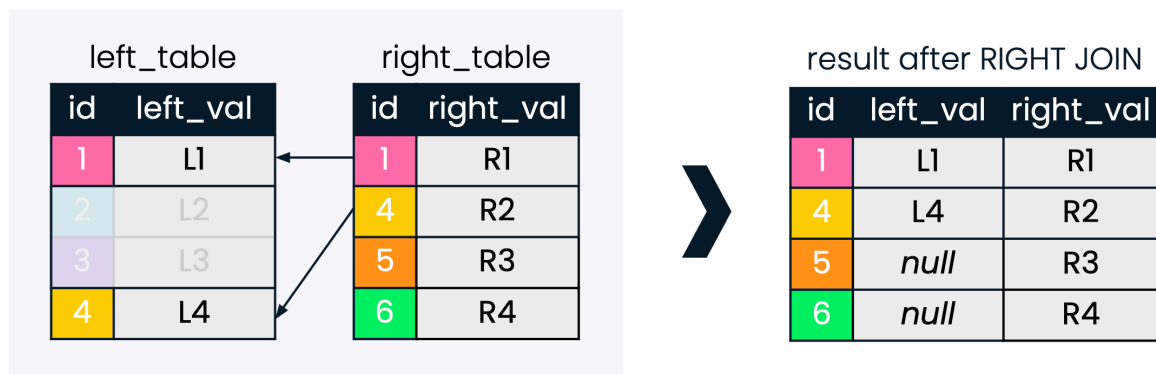


Figure 5.2: Source: Datacamp

Full joins

`full_join()`

A full join is the most inclusive type of join. It keeps all rows from both datasets.

If a row appears in one dataset but not the other, it is still included in the result. Any missing information created by these unmatched rows is filled in with NA. Because full joins can increase both the number of rows and columns, it's a good idea to keep an eye on the output to catch issues like mismatched variable names, case sensitivity, or small differences in character values.

In a full join, the dataset listed first in the command is treated as the baseline. While this does not change which rows are returned, it can affect the order of rows and columns and which identifier columns are kept.

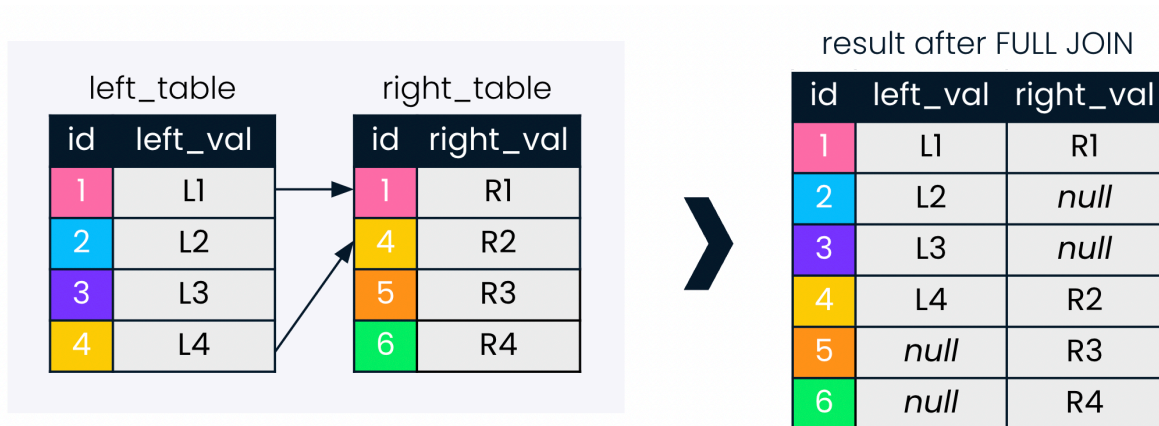


Figure 5.3: Source: Datacamp

Inner join

`inner_join()`

An inner join is the most restrictive type of join—it keeps only rows that have matching values in both datasets.

Because of this, the number of rows in the resulting dataset may be smaller than in the original (baseline) dataset.

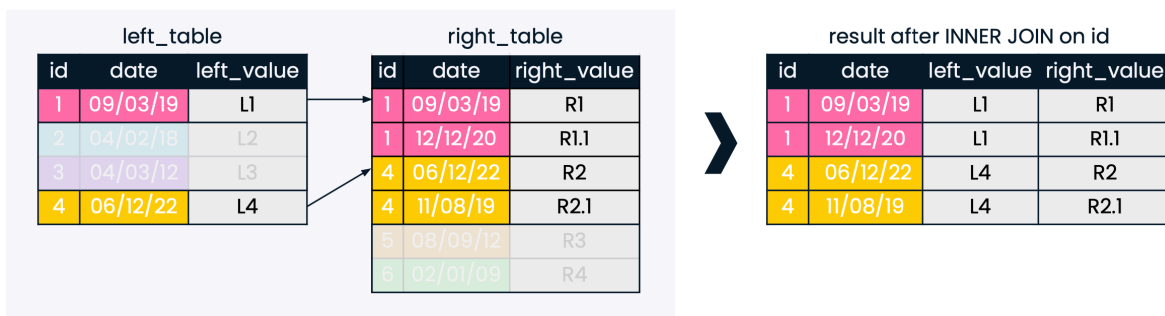


Figure 5.4: Source: Datacamp

References