Introduction to R

ESCWA

2025-07-18

# Chapter 1: Introduction to R

This chapter is designed to introduce beginners to R, ensuring that trainees with no prior exposure to the software can grasp its basic functionalities. It begins with an overview of R, instructions for installing the software, and a description of the RStudio interface. Finally, it provides guidance on creating basic code.

## What is R?

* **R** is a programming language and environment specifically designed for statistical computing and graphics.
* It offers a wide range of statistical and graphical techniques, including linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, and more. These tools are invaluable for systematizing the quantitative analysis of social assistance programmes, enabling evidence-based decision-making.
* By default, R does not have a user-friendly interface for visualizing inputs and outputs. To enhance usability, it is recommended to download **RStudio**, an integrated development environment (IDE) that simplifies interaction with R.

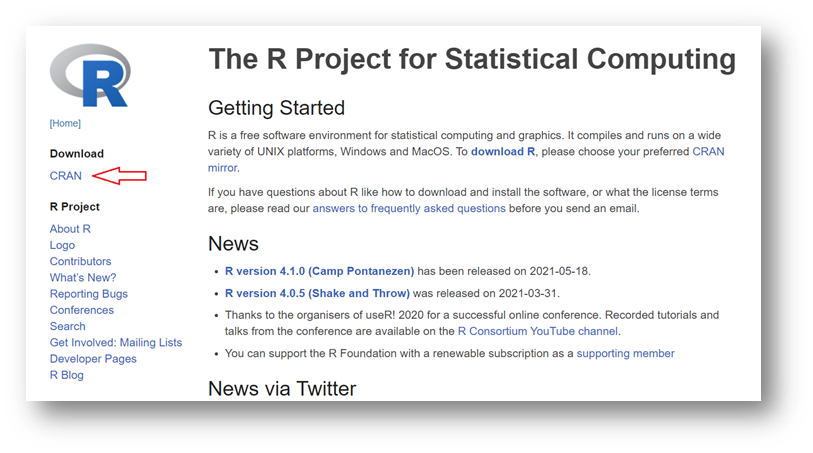
Given the importance of RStudio, the following section outlines the steps to install both R and RStudio. For a more detailed installation guide, refer to the following resources:

* [Posit Cloud Learn Recipes](https://posit.cloud/learn/recipes)
* [R Introduction Manual](https://cran.r-project.org/doc/manuals/R-intro.pdf)
* [Applied Statistics with R](https://daviddalpiaz.github.io/appliedstats/applied_statistics.pdf) (This manual is based on Chapters 1-7 of this book.)

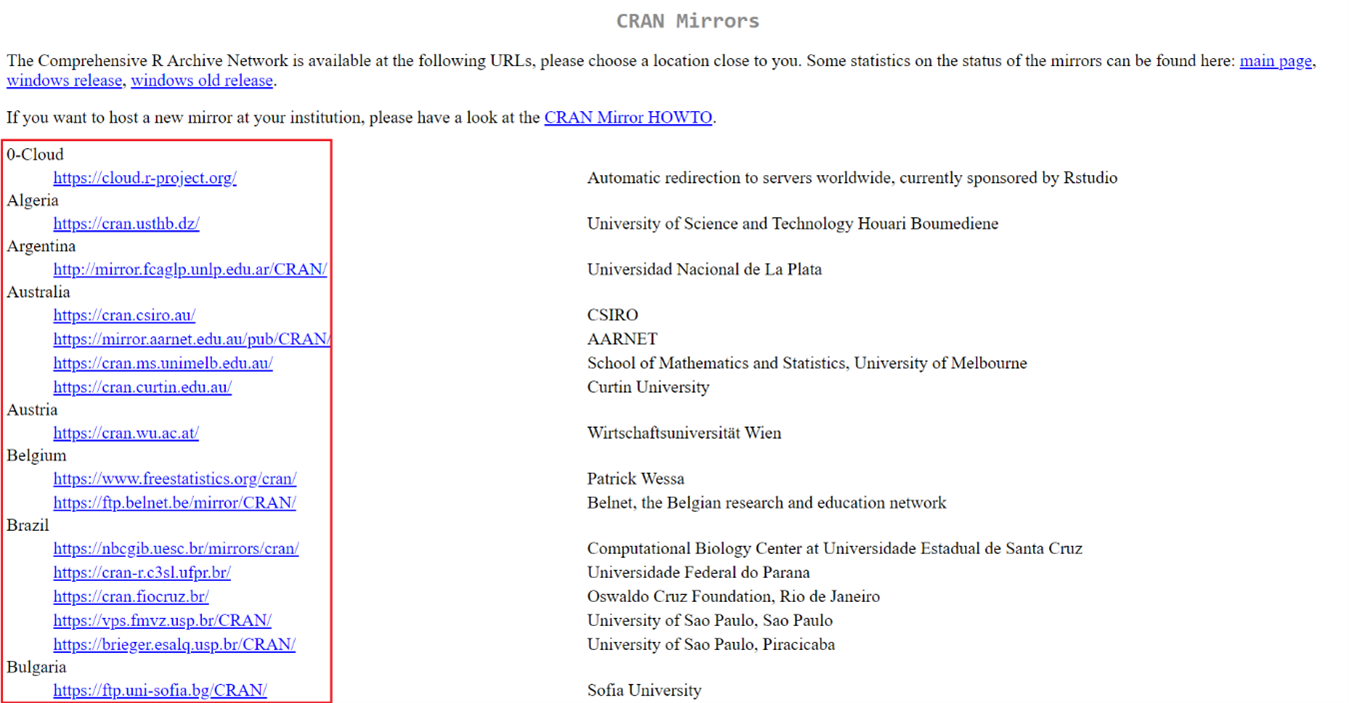
## Installing R and RStudio

### Step 1: Download R

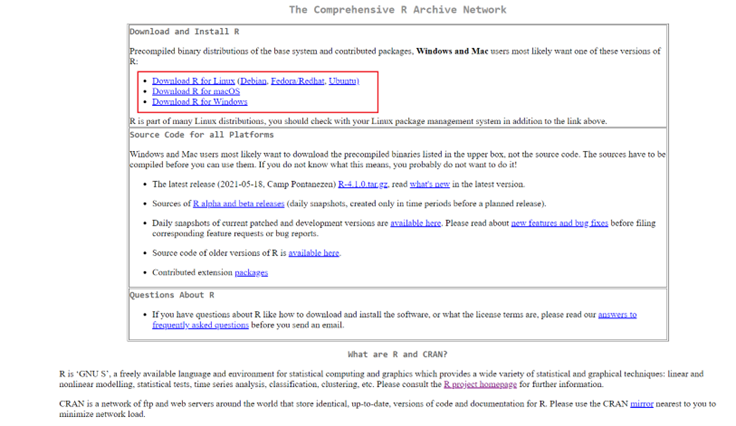
1. Visit the official R website: <https://www.r-project.org/>.
2. Click on the **CRAN** link located on the left side of the webpage.

* 

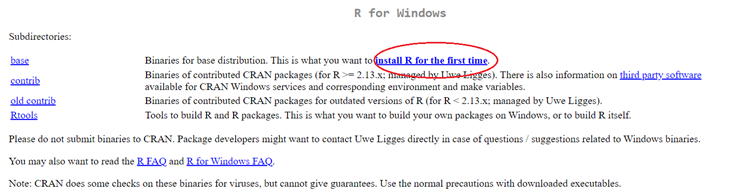
1. Select a mirror from the list (e.g., **0-Cloud**).

* 

1. Choose the appropriate version for your operating system (Windows or Mac) and click **Download R**.

* 

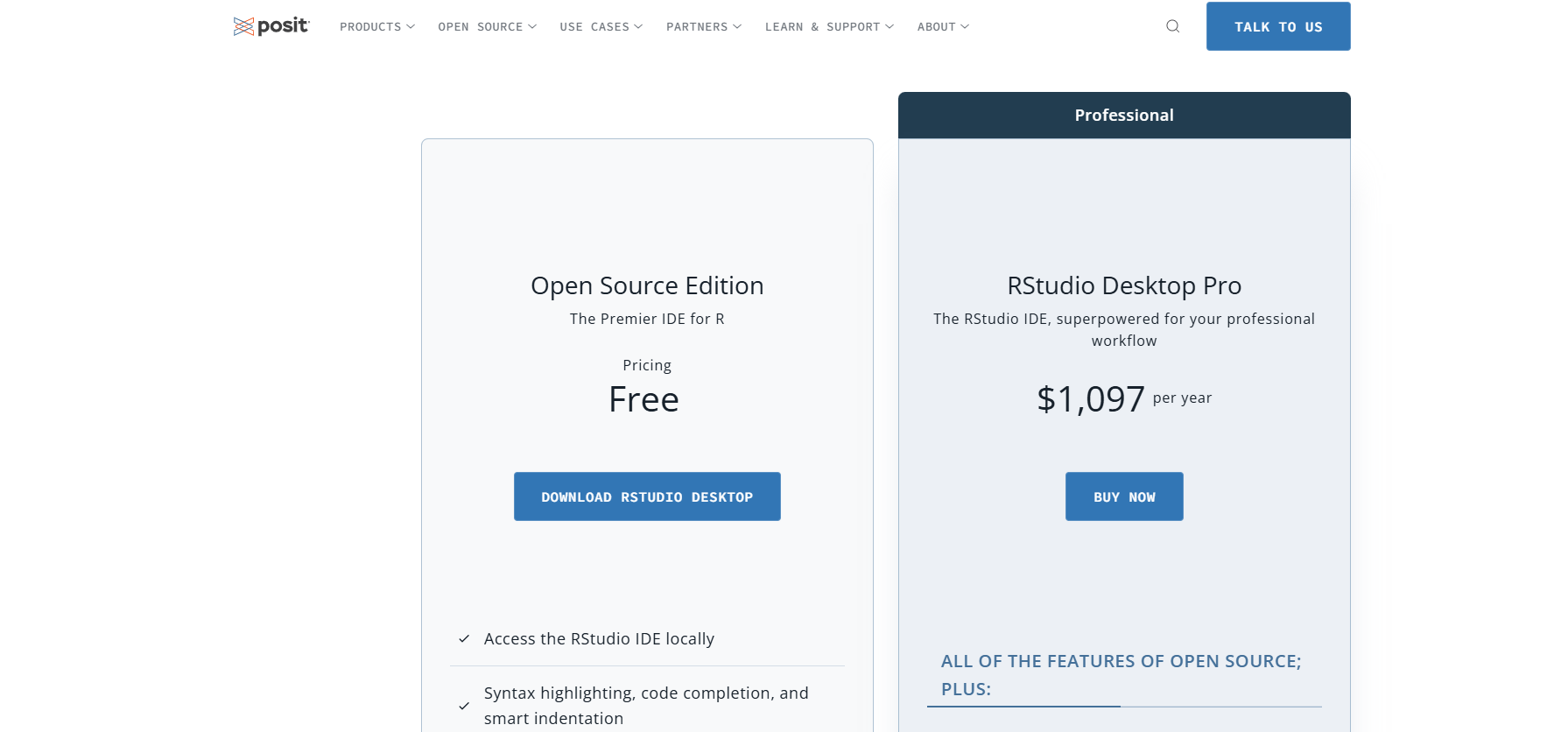
1. Click on **Install R for the first time**.

* 

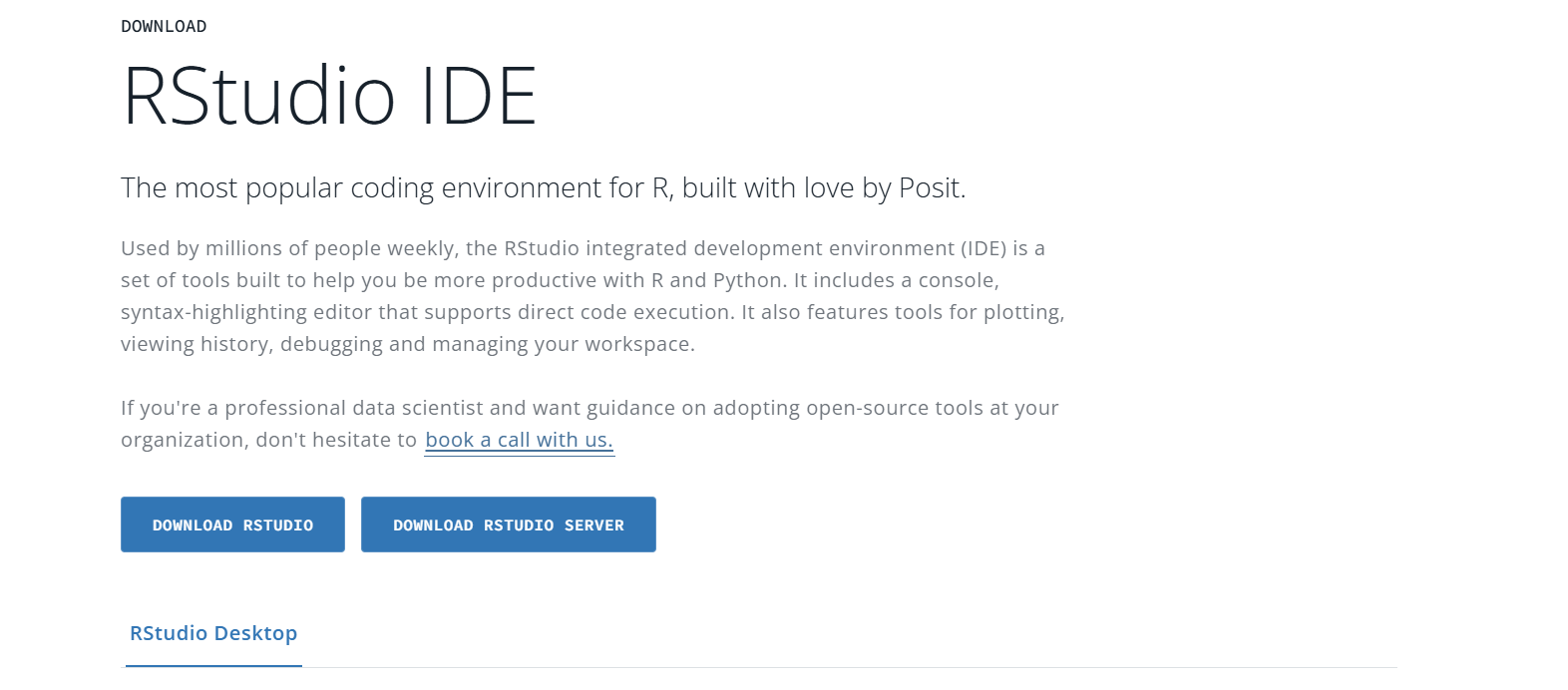
1. Download the installation file for your operating system (e.g., **Download R for Windows**).
2. Run the installation file and follow the on-screen instructions to complete the installation.

### Step 2: Download RStudio

1. Visit the RStudio website: <https://posit.co/products/open-source/rstudio/>.
2. Click on the **Download** button.

* 

1. Select the **Free** version of RStudio.

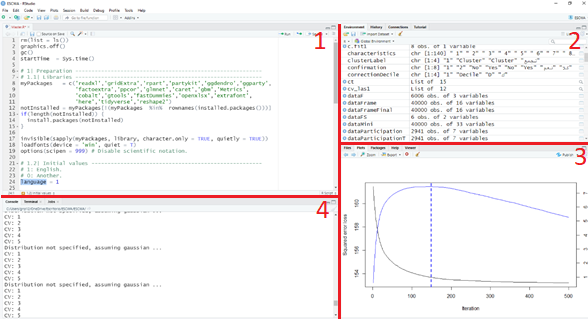
* 

1. Click **Download RStudio** and follow the installation instructions.

## RStudio Interface

When you launch RStudio, you will see an interface divided into four panels:

1. **Source Panel**:
   * Located in the top-left quadrant, this panel is where you write and edit code scripts.
   * You can save scripts, view databases, and execute specific lines of code here.
2. **Environment and History Panel**:
   * Located in the top-right quadrant, this panel displays all variables, datasets, functions, and objects loaded into the system.
   * It also provides access to history, connections, and tutorials, though these features are not covered in this manual.
3. **Multipurpose Panel**:
   * Located in the bottom-right quadrant, this panel is highly versatile.
   * It allows you to view files, preview plots, manage installed packages, and access help manuals.
4. **Console Panel**:
   * Located in the bottom-left quadrant, this panel displays the results of executed code.
   * While you can write code directly in the console, it is recommended to use the Source panel for better traceability.

* 

## Packages and Libraries

Packages are collections of R functions, data, and code that extend R’s capabilities. Some packages are included with the base R installation and are automatically accessible. These provide essential functions for statistical analysis and data visualization.

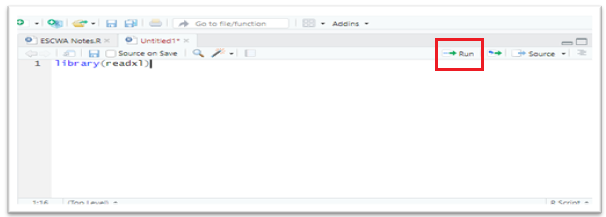
To use additional packages, you must load them into your R session using the library() function. For example:

library(readxl)

Once you write the previous statement in the **Source** panel, the natural question is how to execute this line (i.e., to let R know that you want it to perform the instruction). There are two ways to do this:

1. **First Option**: Select the command line you want to run and press **“Ctrl+Enter”** on the keyboard (recommended option).
2. **Second Option**: Select the command line and click the **“Run”** button at the top of the **Source** panel.

Both methods will execute the selected command in the R console.

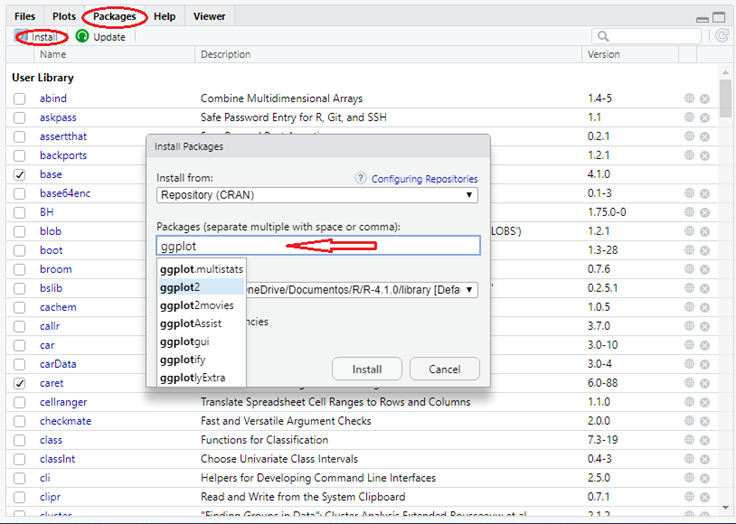


## Installing Packages

R is an open-source software, and new packages created by users are continually being developed to simplify analytical tasks. To use these packages, they must first be downloaded and installed. There are two primary ways to install packages in R.

### Method 1: Using the RStudio Interface

1. Navigate to the **Packages** tab in the Multipurpose panel.
2. Click the **Install** button.
3. Type the name of the package you want to install (e.g., ggplot2). As you type, RStudio will suggest available packages to help avoid spelling errors.



### Method 2: Using the Command Line

You can also install packages directly from the command line using the install.packages() function. This method requires you to know the exact name of the package but ensures your code is self-contained and reproducible. For example:

install.packages("ggplot2")

**Note:** Installing a package is only the first step. To use it in your session, you must load it using the library() function:

library(ggplot2)

## Objects in R

R uses various types of objects to store and manipulate data. Below, we explore the most common ones.

### Variables

Variables in R can store different types of data, such as numeric, character, or logical values. To create a variable, assign a value to it using the = or <- operator.

# R variables  
# Examples of variables  
A = 4.5 # Numeric  
B = "House" # Character  
C = FALSE # Logical  
  
# Display the value of C  
C

## [1] FALSE

### Vectors

A vector is a collection of elements of the same type (e.g., all numeric, all character, or all logical). Use the c() function to create a vector.

# R vectors  
# Create vectors  
A = c(1, 2, 3) # Numeric vector  
B = c("blue", "red", "yellow") # Character vector  
C = c(TRUE, FALSE, FALSE) # Logical vector  
  
# Display the vector B  
B

## [1] "blue" "red" "yellow"

# Access specific elements of B  
B[1:2]

## [1] "blue" "red"

### Matrices

Matrices are two-dimensional objects that store elements of the same type. They can be created by combining vectors using rbind() (row-wise) or cbind() (column-wise).

# Create vectors  
vectorA = c(1, 2, 3)  
vectorB = c(4, 5, 6)  
  
# Combine vectors into matrices  
Matrix1 = rbind(vectorA, vectorB) # Combine by row  
Matrix2 = cbind(vectorA, vectorB) # Combine by column  
  
# Display Matrix1  
Matrix1

## [,1] [,2] [,3]  
## vectorA 1 2 3  
## vectorB 4 5 6

# Display Matrix2  
Matrix2

## vectorA vectorB  
## [1,] 1 4  
## [2,] 2 5  
## [3,] 3 6

# Check dimensions of Matrix2  
dim(Matrix2)

## [1] 3 2

# Access specific elements of Matrix2  
Matrix2[2, 2]

## vectorB   
## 5

### Factors

Factors are used to represent categorical data. They assign labels to numeric values, making it easier to interpret data.

# Create a numeric vector  
vectorA = c(1, 3, 2, 2, 4, 1)  
  
# Convert to a factor with labels  
factorA = factor(vectorA,   
 levels = c(1, 2, 3),   
 labels = c("Primary", "Secondary", "Tertiary"))  
  
# Display factorA  
factorA

## [1] Primary Tertiary Secondary Secondary <NA> Primary   
## Levels: Primary Secondary Tertiary

### Data Frames

Data frames are similar to matrices but can store columns of different data types. They are ideal for working with structured data.[To explore more data types](https://www.geeksforgeeks.org/r-data-types/).

# Create vectors  
edu = c(1, 2, 3)  
edu = factor(edu,   
 levels = c(1, 2, 3),   
 labels = c("Primary", "Secondary", "Tertiary"))  
Name = c("Amira", "Jumana", "Carole")  
Age = c(35, 32, 43)  
Beneficiaries = c(TRUE, TRUE, FALSE)  
  
# Combine vectors into a data frame  
data = data.frame(Name, Age, Education = edu, Beneficiaries)  
  
# Display the data frame  
data

## Name Age Education Beneficiaries  
## 1 Amira 35 Primary TRUE  
## 2 Jumana 32 Secondary TRUE  
## 3 Carole 43 Tertiary FALSE

# Access specific columns  
data$Education

## [1] Primary Secondary Tertiary   
## Levels: Primary Secondary Tertiary

data$Age

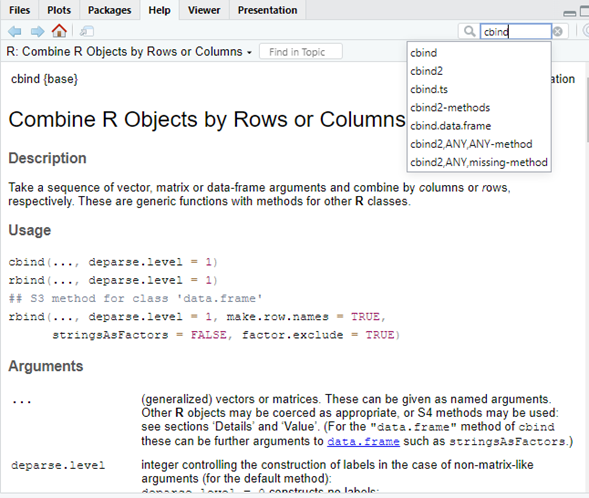
## [1] 35 32 43

## Functions

Functions are reusable blocks of code that perform specific tasks. R includes many built-in functions, and you can also create your own. R comes with many built-in functions, such as c(), rbind(), and factor(), which you have already encountered. However, R also allows users to define their own custom functions. This section delves deeper into how functions work in R. For example, the cbind() function is used to combine objects by columns, while rbind() combines objects by rows.

Both functions take arguments to specify the objects to be combined. To understand the arguments of a function already created in R, you can use the ? operator or the help() function. For example, upon run the command ?cbind R will display documentation for the function, including its arguments and usage as illustrated in the image xxx

dbind(..., deparse.level = 1)  
The ... argument refers to one or more vectors, matrices, or other R objects that can be combined by rows. These can be provided as named or unnamed arguments.  
  
The deparse.level argument is an integer that controls how labels are constructed for non-matrix-like arguments. By default, it is set to 1.



### Creating a Function

A function in R has the following structure:

function\_name = function(arg1, arg2, ...) {  
 # Code to execute  
 return(result)  
}

**Example:**

# Define a function to sum two numbers  
sumNumbers = function(a, b) {  
 y = a + b  
 return(y)  
}  
  
# Use the function with 1 and 3 values as arguments. y= 1 + 3 will return as a result the 4  
sumNumbers(1, 3)

## [1] 4

## Conditional Statements

Conditional statements allow you to execute code based on specific conditions. The if/else statement is commonly used for this purpose.

If (condition)   
{code to be executed if the condition is TRUE }   
else { code to be executed if the condition is FALSE }

# Example of if/else  
x = 3  
if (x < 0) {  
 print("Negative")  
} else if (x > 0) {  
 print("Positive")  
} else {  
 print("Zero")  
}

## [1] "Positive"

## Loops

Loops are used to repeat a block of code multiple times. R supports two main types of loops: while and for.

### While Loop

A while loop repeats code as long as a condition is true.

while (test\_expression) {  
code to be executed  
}

# Display numbers from 1 to 6  
i = 1  
while (i <= 6) {  
 print(i)  
 i = i + 1  
}

## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6

### For Loop

A for loop iterates over a sequence of values.

for (element in a limited sequence)  
code to be executed  
}

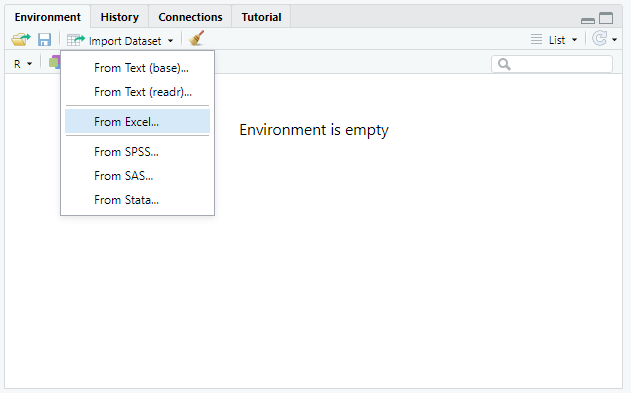
# Sum numbers from 1 to 6  
x = c(1, 2, 3, 4, 5, 6)  
y = 0 # initialize y in 0  
for (i in x) {  
 y = y + i  
}  
y

## [1] 21

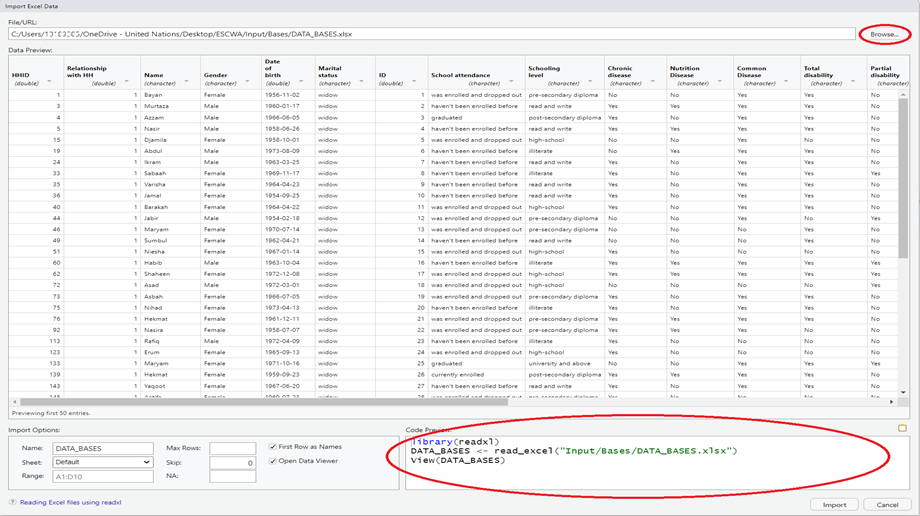
## Chapter 2: Data management

Mots of the times, the user will analyse external datasets, for example, surveys or administrative records. These files are usually saved in folders in your laptop or pc, and frequenly are saved in Excel folder. Alhtough this example uses an Excel file since R Studio interface has integrated a funcionality to make the process simpler for the user, similar process via buttons and/or scripts can be followed to upload data in other formats. Please consider that applied examples of this section could be observed in ⚠️.

To upload the dataset, go to the second quadrant and click on “Import Dataset” there as this example shows in the Figure xx, click on “From Excel…” , you could see other formats available (SPPS, SAS and Stata). Other programs can be uploaded using packages like For example, [“Rio”](https://cran.r-project.org/web/packages/rio/vignettes/rio.html)



A pop-up window will appear. Click the ‘Browse’ button to locate and upload the relevant dataset. Once the file is uploaded and you’ve verified that all parameters are correct, you can click ‘Import’. However, it’s recommended to take note of the generated code. If you copy this code into your main script, you won’t need to repeat the entire process from scratch—you can run it directly from the first cell.



### Data management

To understand better the dataset, you need to work more on its content and for that purpose this section relies on the package “[dplyr”.](https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html) Remember that for any package you want to use, you need to upload the library first. as mentioned in earlier sections, so please make sure to include this first line in any code you create:

The library dplyr is very useful for data management as it allows to organize complex processes into simple segmented instructions. However, to do it in a proper way, it is important to know the variables in the dataset and for that purpose the best starting point is by knowing the variables inside. For applied excercised you can run SPP RAF code and routine.

Note that, in case the Excel sheet of the data uploaded has two separate sheets, for example one for “Individuals” and another for “Household”, the function *merge* helps in combining the two sheets in one output using for example “Household ID”.

For didactic purposes we will create a employee dataset considering we cannot upload locally the information.

# Create fake employee dataset  
set.seed(123) #   
  
# Generate 100 employee records  
employee\_data <- data.frame(  
 Employee\_ID = 1001:1100,  
 Name = paste0("Employee\_", 1001:1100),  
 Age = round(rnorm(100, mean = 35, sd = 8)),  
 Active\_Status = sample(c(TRUE, FALSE), 100, replace = TRUE, prob = c(0.85, 0.15)),  
 Hire\_Date = sample(seq(as.Date('2010-01-01'), as.Date('2023-12-31'), by = "day"), 100),  
 Performance = sample(c("Low", "Medium", "High"), 100, replace = TRUE,   
 prob = c(0.2, 0.6, 0.2)),  
 Department = sample(c("Sales", "Marketing", "IT", "HR", "Finance"), 100, replace = TRUE),  
 Salary = round(rnorm(100, mean = 60000, sd = 15000)),  
 stringsAsFactors = FALSE  
)  
  
# Ensure age is between 22 and 65  
employee\_data$Age <- pmax(22, pmin(65, employee\_data$Age))  
  
# Clean up negative salaries  
employee\_data$Salary <- pmax(30000, employee\_data$Salary)  
  
# Convert Performance to ordered factor  
employee\_data$Performance <- factor(employee\_data$Performance,   
 levels = c("Low", "Medium", "High"),  
 ordered = TRUE)  
  
# Add tenure in years  
employee\_data$Tenure <- as.numeric(difftime(Sys.Date(), employee\_data$Hire\_Date, units = "days"))/365.25  
  
# Show structure  
str(employee\_data)

## 'data.frame': 100 obs. of 9 variables:  
## $ Employee\_ID : int 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 ...  
## $ Name : chr "Employee\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004" ...  
## $ Age : num 31 33 47 36 36 49 39 25 30 31 ...  
## $ Active\_Status: logi TRUE FALSE TRUE TRUE TRUE FALSE ...  
## $ Hire\_Date : Date, format: "2016-03-15" "2011-09-10" ...  
## $ Performance : Ord.factor w/ 3 levels "Low"<"Medium"<..: 3 2 1 1 2 2 3 2 1 2 ...  
## $ Department : chr "IT" "Marketing" "Finance" "HR" ...  
## $ Salary : num 67142 80679 66844 42966 53465 ...  
## $ Tenure : num 9.43 13.94 10.42 14.71 9.96 ...

Please keep in mind that the file path is specific to each computer, so the green text (which represents the file path) may vary depending on your system. A new function introduced here is **colnames()**. This function helps you inspect the names of the variables (columns) in your dataset. Since variable names can sometimes be unclear or inconsistent, it’s always a good idea to review them. If anything is confusing, it’s recommended to consult the data source for clarification.

For the purpose of this exercise, a subset of the original dataset containing 10 entries has been selected and named “subdata”.

subdata <- employee\_data[1:10,]

To continue the inspection of the database, you could use **str()** function which displays the structure of the dataset, for this brief manual, the different columns have different type of information. Also to check the categories or values, when these are limited, the function **unique()** will display the unique categories of a variable

str(subdata)

## 'data.frame': 10 obs. of 9 variables:  
## $ Employee\_ID : int 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010  
## $ Name : chr "Employee\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004" ...  
## $ Age : num 31 33 47 36 36 49 39 25 30 31  
## $ Active\_Status: logi TRUE FALSE TRUE TRUE TRUE FALSE ...  
## $ Hire\_Date : Date, format: "2016-03-15" "2011-09-10" ...  
## $ Performance : Ord.factor w/ 3 levels "Low"<"Medium"<..: 3 2 1 1 2 2 3 2 1 2  
## $ Department : chr "IT" "Marketing" "Finance" "HR" ...  
## $ Salary : num 67142 80679 66844 42966 53465 ...  
## $ Tenure : num 9.43 13.94 10.42 14.71 9.96 ...

unique(subdata$Name)

## [1] "Employee\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004"  
## [5] "Employee\_1005" "Employee\_1006" "Employee\_1007" "Employee\_1008"  
## [9] "Employee\_1009" "Employee\_1010"

To change one of the categories or information of the variable, the user can call the function **mutate()** within the **dplyr** package. This package has a specific logic to understand datasets under the package **tidyverse** The **dplyr** *cheatsheet* with information about other functions and syntax’s is available [here](https://github.com/rstudio/cheatsheets/blob/main/data-transformation.pdf). Cheatsheets are one pagers with the most important functions and tricks to use a package in the R universe.

# Correct using mutate() and pipeline (%>%)  
subdata <- subdata %>%  
 mutate(Name = if\_else(Name %in% c("Employee\_1001"),  
 "Renamed\_1001",  
 Name))  
unique(subdata$Name)

## [1] "Renamed\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004"  
## [5] "Employee\_1005" "Employee\_1006" "Employee\_1007" "Employee\_1008"  
## [9] "Employee\_1009" "Employee\_1010"

The pipeline operator %>% passes the left-side object as the first argument to the right-side function. The mutate() function modifies variables, while if\_else() handles conditional replacement. To replace specific observations, recode() can be used instead of if\_else.

Manage columns with select() in order to select specific variables, for example, to create a new dataset like subdata\_new.  
Note that there is another package with a function named select(), which may cause confusion. It is recommended to call the function explicitly using dplyr::select().

# Keep specific columns  
subdata\_new <- subdata %>%  
 dplyr::select(c("Employee\_ID", "Name", "Age"))  
  
str(subdata\_new)

## 'data.frame': 10 obs. of 3 variables:  
## $ Employee\_ID: int 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010  
## $ Name : chr "Renamed\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004" ...  
## $ Age : num 31 33 47 36 36 49 39 25 30 31

# Remove specific columns  
subdata\_new <- subdata %>%  
 dplyr::select(-c("Active\_Status", "Hire\_Date", "Performance"))  
  
str(subdata\_new)

## 'data.frame': 10 obs. of 6 variables:  
## $ Employee\_ID: int 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010  
## $ Name : chr "Renamed\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004" ...  
## $ Age : num 31 33 47 36 36 49 39 25 30 31  
## $ Department : chr "IT" "Marketing" "Finance" "HR" ...  
## $ Salary : num 67142 80679 66844 42966 53465 ...  
## $ Tenure : num 9.43 13.94 10.42 14.71 9.96 ...

### Renaming Variables for Clarity

Clear and descriptive variable names are essential for effective data analysis. They make your code easier to read, understand, and maintain—especially when collaborating with others or revisiting your work after some time. In the dataset used the columns are already named clearly, but let’s say that we would like to change the column named “Tenure” into “Years\_exp”:

'data.frame': 10 obs. of 9 variables:  
$ Employee\_ID : int 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010  
$ Name : chr "Employee\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004" ...  
$ Age : num 31 33 47 36 36 49 39 25 30 31  
$ Active\_Status: logi TRUE FALSE TRUE TRUE TRUE FALSE ...  
$ Hire\_Date : Date, format: "2016-03-15" "2011-09-10" ...  
$ Performance : Ord.factor w/ 3 levels "Low"<"Medium"<..: 3 2 1 1 2 2 3 2 1 2  
$ Department : chr "IT" "Marketing" "Finance" "HR" ...  
$ Salary : num 67142 80679 66844 42966 53465 ...  
$ Tenure : num 9.4 13.91 10.39 14.68 9.92 ...

To rename these variables, use the rename() function from the dplyr package:

subdata <- subdata %>%  
 rename(  
 Years\_exp = Tenure  
 )  
  
str (subdata)

## 'data.frame': 10 obs. of 9 variables:  
## $ Employee\_ID : int 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010  
## $ Name : chr "Renamed\_1001" "Employee\_1002" "Employee\_1003" "Employee\_1004" ...  
## $ Age : num 31 33 47 36 36 49 39 25 30 31  
## $ Active\_Status: logi TRUE FALSE TRUE TRUE TRUE FALSE ...  
## $ Hire\_Date : Date, format: "2016-03-15" "2011-09-10" ...  
## $ Performance : Ord.factor w/ 3 levels "Low"<"Medium"<..: 3 2 1 1 2 2 3 2 1 2  
## $ Department : chr "IT" "Marketing" "Finance" "HR" ...  
## $ Salary : num 67142 80679 66844 42966 53465 ...  
## $ Years\_exp : num 9.43 13.94 10.42 14.71 9.96 ...

Data analyst frequently need to filter in and out certain variables, for example, to select all registers which contain “IT” in the *Department* and <30 in *Age*. To subset these rows, the function filter() with logical operators:

subdata\_new <- subdata %>%  
 filter(Age < 30 & Department %in% c("IT"))

Consider the following logical operators as follow for further reference:

To aggregate and summarize the respective functions are group\_by() and summarise(). For example to find the average of Age grouping by Name

| Symbol | Definition |
| --- | --- |
| < | less than |
| > | greater than |
| == | equal to |
| <= | less or equal to |
| >= | grater or equal to |
| ! | NOT |
| %in% | value in vector |
| & | AND |
| | | OR |

summary\_Age <- subdata %>%  
 group\_by(Name) %>%  
 summarise(Average\_Age = mean(Age))

For quick counts of observations by categorical variables, and get frequency tables , use table():

table(subdata$Name, subdata$Performance)

##   
## Low Medium High  
## Employee\_1002 0 1 0  
## Employee\_1003 1 0 0  
## Employee\_1004 1 0 0  
## Employee\_1005 0 1 0  
## Employee\_1006 0 1 0  
## Employee\_1007 0 0 1  
## Employee\_1008 0 1 0  
## Employee\_1009 1 0 0  
## Employee\_1010 0 1 0  
## Renamed\_1001 0 0 1

For advanced dplyr usage: - [dplyr vignettes](https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html)

# Chapter 3: Graphs

Easy interpreted graphs make data interpretation easier. That being said, the type of graph used should perfectly match the type of data and the values that need to be highlighted. The “ggplot2” package in R helps you create clear and flexible visualizations. This section introduces the basics of ggplot2 and guides the reader through creating their own graphs. The goal is to make it simple for beginners to learn and apply, without getting lost in complexity.

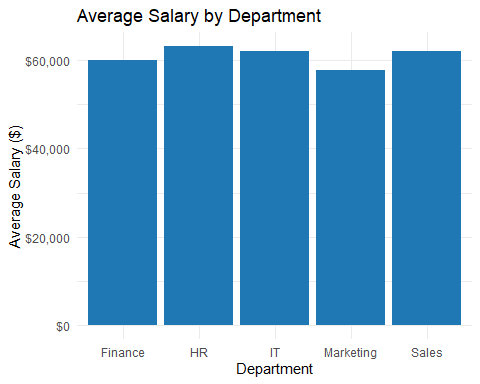
## Basic Graphs

The first graph example is a simple **bar plot**./ In order to create a graph, three steps are needed:/ 1. A pipeline that links the data that will be used for the plot with the ggplot() function./ 2. X and y axes specified in the function called *aes()*./ 3. The choice of the graph type, for example here *geom\_bar()*./

For a more advanced graph, the task is to group the employee dataset uploaded by the **Department** variable and calculate the **Average Salary** for each department. It then generates a bar chart using *ggplot2* to visualize the average salaries, with customized labels, formatting, and styling for presentation. Any characteristic should be added to the *aes()* command.

Note that the “+” sign in the end of the line works as a pipeline to connect the command with the next instruction.

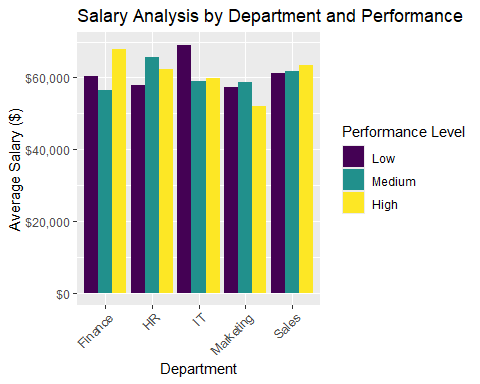
library(readxl)  
library(here)  
library(dplyr)  
library(tidyr)  
library(lubridate)   
library(ggplot2)  
library(scales)  
library(openxlsx) # New library for Excel file manipulation  
  
# Load the necessary libraries  
library(dplyr) # For data manipulation  
library(ggplot2) # For data visualization  
library(scales) # For formatting axis labels (e.g., currency)  
  
# Start with the employee\_data dataset and begin a pipeline  
employee\_data %>%  
 # Group the data by the 'Department' column  
 group\_by(Department) %>%  
   
 # Calculate the average salary for each department  
 summarise(Avg\_Salary = mean(Salary)) %>%  
   
 # Begin creating a plot with ggplot2  
 ggplot(aes(x = Department, y = Avg\_Salary)) +  
   
 # Add a bar chart layer using the actual average salary values  
 geom\_bar(stat = "identity", fill = "#1f78b4") +  
   
 # Add a title and axis labels to the plot  
 labs(title = "Average Salary by Department",  
 x = "Department",  
 y = "Average Salary ($)") +  
   
 # Format the y-axis labels to display as currency  
 scale\_y\_continuous(labels = dollar\_format()) +  
   
 # Apply a minimal theme for a clean look  
 theme\_minimal()



Graphs can always be adjusted to feed the purpose of the data. The next graph groups the employee data by both **Department** and **Performance** levels, then calculates the average salary for each group. It produces a comparative bar chart using *ggplot2*, where average salaries are displayed side-by-side for each performance category within departments, with customized labels, colors, and formatting for enhanced readability.

The role of the position called “position\_dodge()” allows a better interpretation of the graph, showing that governorate D and F tend to have older female population, while C and E have older male population. In a similar way, the gap in governorates B and E is higher for men, but the difference is not as dramatic. After specifying the type of the graph, an essential step would be to format it so that it looks appealing. To read the graph, *clear labels* need to be assigned. For this, ggplot2 provides the function labs() which allows a great control of these parameters.

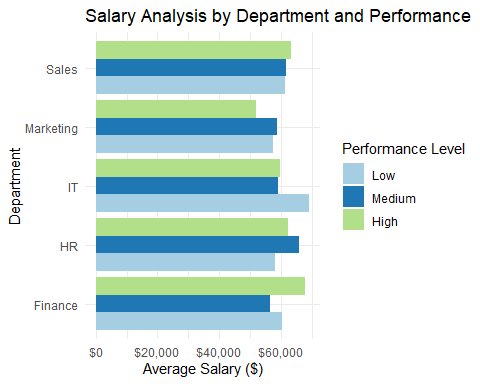
employee\_data %>%  
 group\_by(Department, Performance) %>%  
 summarise(Avg\_Salary = mean(Salary)) %>%  
   
 # Begin plotting: set Department on x-axis, Avg\_Salary on y-axis, and fill bars by Performance  
 ggplot(aes(x = Department, y = Avg\_Salary, fill = Performance)) +  
   
 # Create a bar chart with bars side-by-side for each performance level  
 geom\_bar(stat = "identity", position = position\_dodge()) +  
  
 # Add a title and axis/legend labels to the plot  
 labs(  
 title = "Salary Analysis by Department and Performance",  
 y = "Average Salary ($)",  
 fill = "Performance Level"  
 ) +  
   
 # Format the y-axis labels to display as currency  
 scale\_y\_continuous(labels = dollar\_format()) +  
   
 # Rotate x-axis labels for better readability  
 theme(axis.text.x = element\_text(angle = 45, hjust = 1))



When creating graphs, flipping the coordinates might makes labels easier to read. It’s also helpful to remove internal grid lines and use, for example, simple, consistent color gradients throughout the report.

Two commands were added to the command line: 1. *coord\_flip()*: changes the axis 2. *Scale\_fill\_brewer()*: changes the color palette. The palette used in this example is “Paired” but others can be found, please go to <https://www.r-graph-gallery.com/38-rcolorbrewers-palettes.html>

employee\_data %>%  
 group\_by(Department, Performance) %>%  
 summarise(Avg\_Salary = mean(Salary)) %>%  
 ggplot(aes(x = Department, y = Avg\_Salary, fill = Performance)) +  
 geom\_bar(stat = "identity", position = position\_dodge()) +  
 # Apply a color palette for better visual distinction between performance levels  
 scale\_fill\_brewer(palette = "Paired") +  
 # Flips the coordinates  
 coord\_flip()+  
 labs(  
 title = "Salary Analysis by Department and Performance",  
 y = "Average Salary ($)",  
 fill = "Performance Level"  
 ) +  
   
 scale\_y\_continuous(labels = dollar\_format()) +  
  
# Apply a minimal theme for a clean look  
 theme\_minimal()

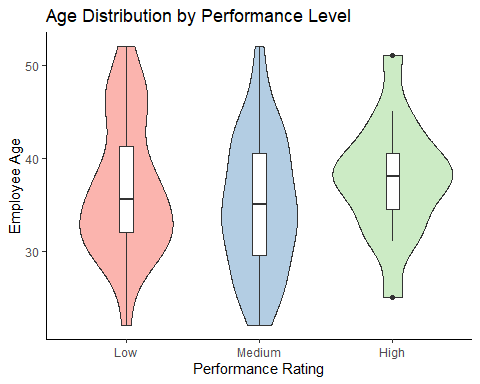


While many other formatting criteria will be added to the illustration codes seen in the next chapters, it is recommended to practice turning them on and off to understand what effect they have over the graphs. Some detailed guidance on these topics can be seen in references such as [ggtheme](https://ggplot2.tidyverse.org/reference/ggtheme.html) and [themes and backgrounds](http://www.sthda.com/english/wiki/ggplot2-themes-and-background-colors-the-3-elements.)

## Graph types

Choosing the type of graph is and art and, depending on the message that we want to convet, it is possible to prefer one graph over others. For example, consider a researcher that is interested in learning not only the mean values, but the whole age distribution, for example by performance level

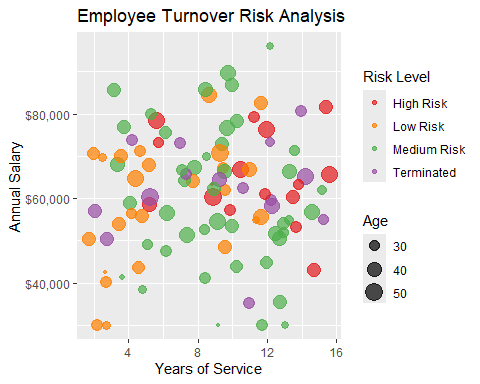
employee\_data %>%  
 ggplot(aes(x=Performance, y=Age, fill=Performance)) +  
 geom\_violin(show.legend=FALSE) + #new function rather than geom\_bar  
 geom\_boxplot(width=0.1, fill="white") +  
 scale\_fill\_brewer(palette="Pastel1") +  
 labs(title="Age Distribution by Performance Level",  
 x="Performance Rating",  
 y="Employee Age") +  
 theme\_classic()



Three changes are spotted here: 1. The type of the graph **violin plot** is specified instead of bar plot. 2. In contrast to the bar plot, the violin plot can be done directly over the full data set and does not require any pre-processing of it. 3. The violin plot helps to notice that older employees have better performance and that younger lower performance, while medium performers are found across all ages

Once the user understands this hypothesis, the type of graph can be chosen to support the claim further. This scatter plot provides a visual analysis of employee turnover risk by mapping tenure against salary, with additional dimensions of age and employment status. Each point represents an employee, where the x-axis shows years of service and the y-axis displays annual salary. The size of each point reflects the employee’s age, while the color indicates their turnover risk level—ranging from low risk to terminated. This visualization helps identify patterns, such as whether long-tenured, low-performing employees are at higher risk, or if certain age groups are more prone to termination or retention challenges.

employee\_data %>%  
 # Create a new column 'Turnover\_Risk' based on employee status, tenure, and performance  
 mutate(Turnover\_Risk = ifelse(!Active\_Status, "Terminated", # If not active, mark as Terminated  
 ifelse(Tenure > 5 & Performance == "Low", "High Risk", # Long tenure + low performance = high risk  
 ifelse(Tenure > 3 & Performance == "Medium", "Medium Risk", "Low Risk")))) %>%  
   
 # Begin plotting: x-axis is Tenure, y-axis is Salary, color by Turnover\_Risk, size by Age  
 ggplot(aes(x = Tenure, y = Salary, color = Turnover\_Risk, size = Age)) +  
   
 # Use scatter plot with semi-transparent points  
 geom\_point(alpha = 0.7) +  
   
 # Manually assign colors to each risk category for clarity  
 scale\_color\_manual(values = c("#e41a1c", "#ff7f00", "#4daf4a", "#984ea3")) +  
   
 # Add title and axis labels  
 labs(title = "Employee Turnover Risk Analysis",  
 x = "Years of Service",  
 y = "Annual Salary",  
 color = "Risk Level",  
 size = "Age") +  
   
 # Format y-axis labels as currency  
 scale\_y\_continuous(labels = dollar\_format())



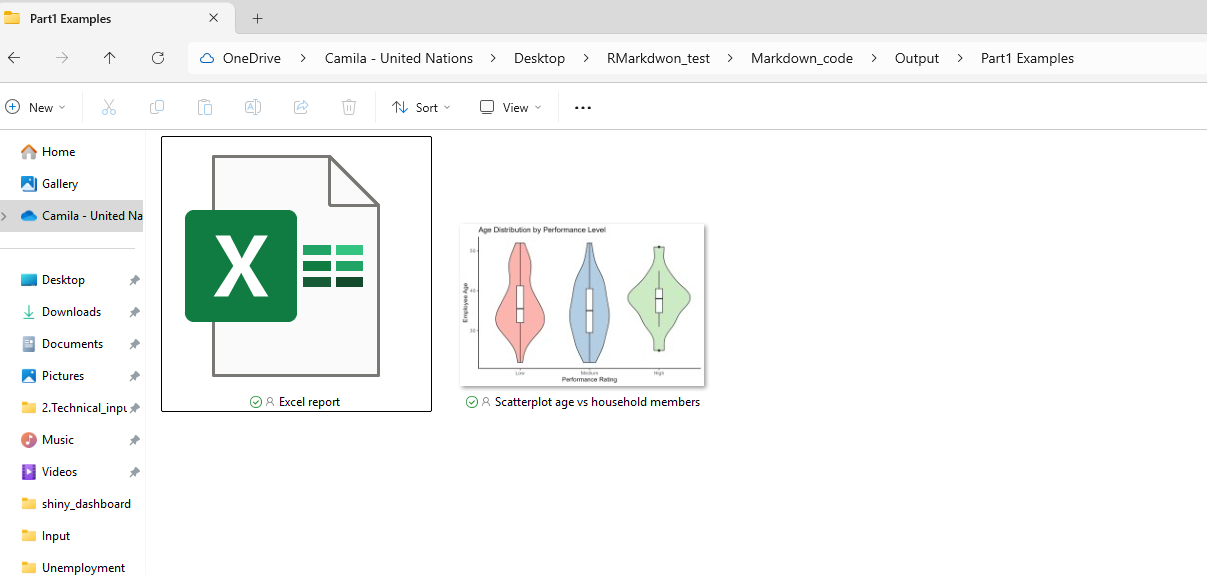
During the previous exercises, the user has been exposed to different types of graphs, where the common patterns among the codes have been highlighted. Currently there are multiple portals with amazing ideas on how to display information. One recommendation for curious minds is the official site of the [R gallery](https://r-graph-gallery.com/) where access to multiple designs is found using these same code patterns.

# Chapter 4: Automated report

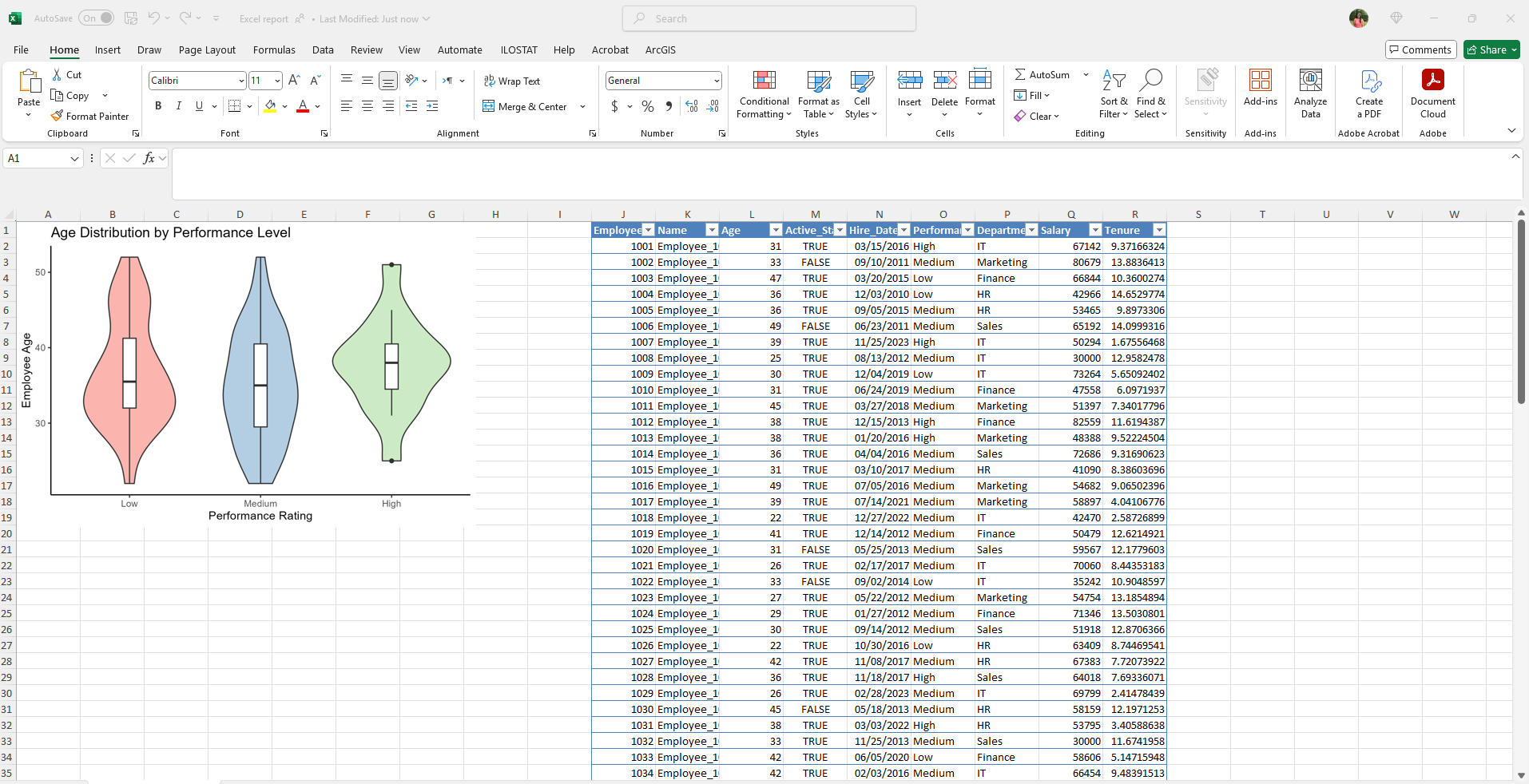
Excel is widely used for data handling, but automating tasks within it can be time-consuming. For instance, adding or removing titles from hundreds of graphs manually wastes valuable time. This chapter and the next show how to automate such tasks by integrating Excel with R, requiring only minor code changes.

Using the openxlsx package, R can create and edit Excel files. The exercise builds on above graphs from Chapter 3, saving it as a PNG and embedding it in an Excel sheet alongside a microdata table. Key functions like createWorkbook(), insertImage(), and writeDataTable() streamline the process. The result: a ready-to-use Excel report with both visual and tabular data.

library(openxlsx) # New library for Excel file manipulation  
  
# Create a new Excel workbook  
wb = openxlsx::createWorkbook(creator = 'ESCWA') # The operator "::" calls a specific function from the `openxlsx` package.  
  
# Add a new worksheet to the workbook  
newSheet = addWorksheet(wb, sheetName = "Age\_Performance Level")   
  
# Specify the location of the file using `here`  
here = here()   
  
# Create the column 'Age' from 'Date of birth'  
newPlot = employee\_data %>%  
 ggplot(aes(x=Performance, y=Age, fill=Performance)) +  
 geom\_violin(show.legend=FALSE) + #new function rather than geom\_bar  
 geom\_boxplot(width=0.1, fill="white") +  
 scale\_fill\_brewer(palette="Pastel1") +  
 labs(title="Age Distribution by Performance Level",  
 x="Performance Rating",  
 y="Employee Age") +  
 theme\_classic()  
  
# Specify the file name and path for the graph  
fileName = "Output/Part1 Examples/Scatterplot age vs household members.png"  
  
  
# Save the graph as a PNG file  
ggsave(fileName, plot= newPlot, width= 6, height = 4, scale = 1)  
  
# Insert the graph into the Excel sheet  
insertImage(wb, file = fileName, sheet= newSheet, startRow = 1, startCol = 1, width = 6, height = 4)  
  
# Conver and Write the table employee\_data into the Excel sheet.  
writeDataTable(wb, sheet=newSheet, x = employee\_data, startRow = 1, startCol = 10)  
  
# Save the Excel workbook  
saveWorkbook(wb,  
 file = "Output/Part1 Examples/Excel report.xlsx",  
 overwrite = TRUE)



The file corresponding to the Excel report once accessed looked like



To highlight:

* Notice that the starting cells of the graph and table matches those that are assigned by the code.
* Notice the name of the spreadsheet matches the code.
* If this data needs to go into a report, now it is easy to copy and paste it from the Excel file.