Top 60 R Interview Questions

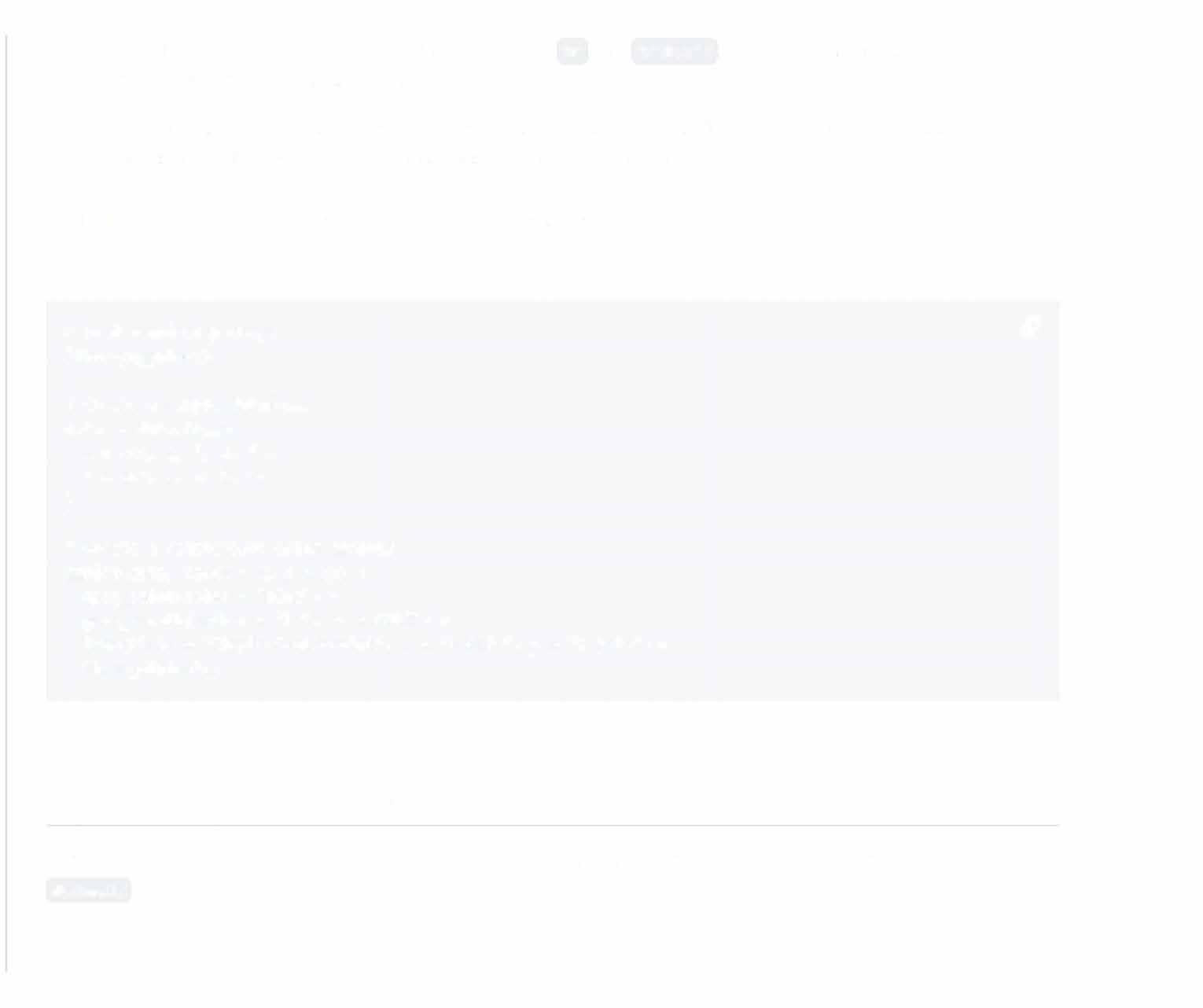
1. What is the significance of ***R*** in ***data analysis*** and ***Machine Learning?***

R is an open source statistical computing and graphics software widely used for data analysis, statistical modeling, and emerging domains such as machine learning. It's popular for its comprehensive library of packages tailored to a wide array of data-related tasks.

# Key Data Analysis Functions in R

* + Exploratory Data Analysis (EDA): R enables data exploration through visual representations, summaries, and tests.
  + Data Visualization: Its diverse libraries, such as ggplot2 , offer flexibility in creating interactive, publication­ standard visualizations.
  + Data Preparation: R provides functions for data cleaning, wrangling, and imputation, often used in both traditional and machine-learning workflows.
  + Descriptive Statistics: It can generate comprehensive statistical summaries, including measures of central tendency, dispersion, and distributions.

# R in Machine Learning

* + Model Building and Validation: R's specialized packages like caret streamline the process of training, testing, and validating models across a variety of algorithms.
  + Performance Evaluation: It provides tools for in-depth model assessment, including ROC curves, confusion matrices, and customized metrics.
  + Predictive Analytics: R is widely used for tasks such as regression, classification, time series forecasting, and clustering.
  +  Text Mining and NLP: With dedicated libraries such as tm and text2vec , R supports natural language processing and text mining applications.
  + Specialized Techniques: From Bayesian networks to ensemble methods like random forests and boosting, R is equipped to handle a range of advanced model-building methodologies.

# Code Example: Visualizing Data with Rand ggplot2

Here is the R code:

# Load required package *c9*

library(ggplot2}

# Create a sample dataframe data<- data.frame(

X = c(1, 2, 3, 4, 5),

y = c(2, 3, 4, 5, 6}

)

# Create a scatterplot using ggplot2 ggplot(data, aes(x = x, y = y)) +

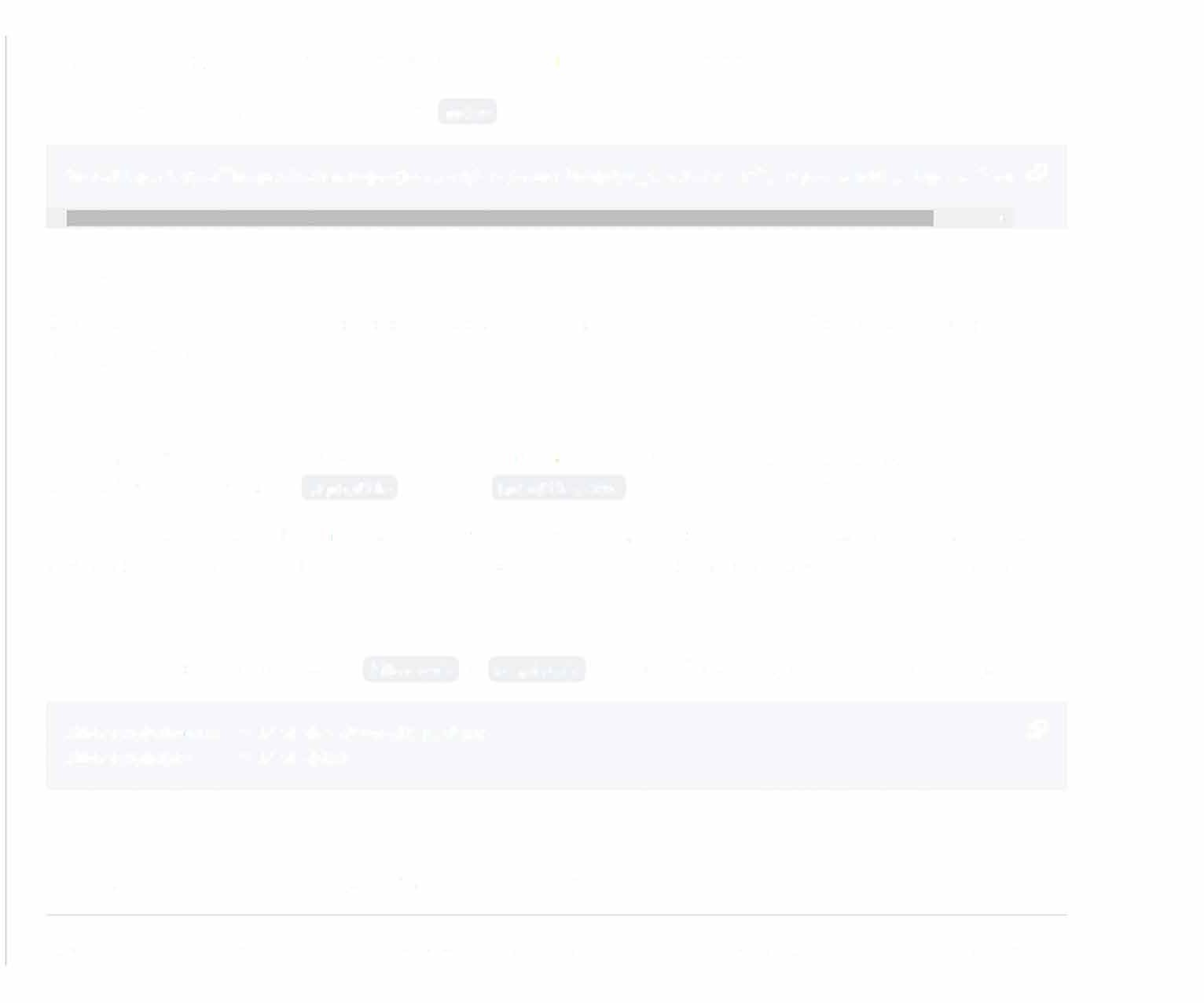
geom\_point(color = "blue") + geom\_smooth(method = "lm", se =FALSE}+

labs(title = "Simple Scatterplot", x = "X Axis", y = "V Axis")+ theme\_minimal()

1. **How do you install *packages* in Rand how do you *load* them?**

In R, you can install and manage packages using CRAN (Comprehensive R Archive Network) or GitHub if you have the devtools package.

# Installing Packages from CRAN

The R console, RStudio, or an R script can all be used to install packages from CRAN. Here, we show the single command to install dplyr from its URL.

install.packages("https://cran.r-project.org/src/contrib/dplyr\_1.0.S.tar.gz", repos = NULL, type = "sou *c9*

' •

# Quick Package Loading

Once a package is installed, it needs to be loaded before using the contained functions. Both automatic and manual loading are options.

Automatic Loading

When automatic loading is enabled, the package is loaded at R startup or when a new R session begins. Automatic loading is achieved using the . Rprofile file or the Rprofile.site file in the R startup directory.

Sometimes, automatic loading can lead to conflicts betwee11 package functions, reducing code clarity and leading to unexpected behavior. To avoid these issues in collaborative work, it's better to load packages manually in your code.

Manual Loading

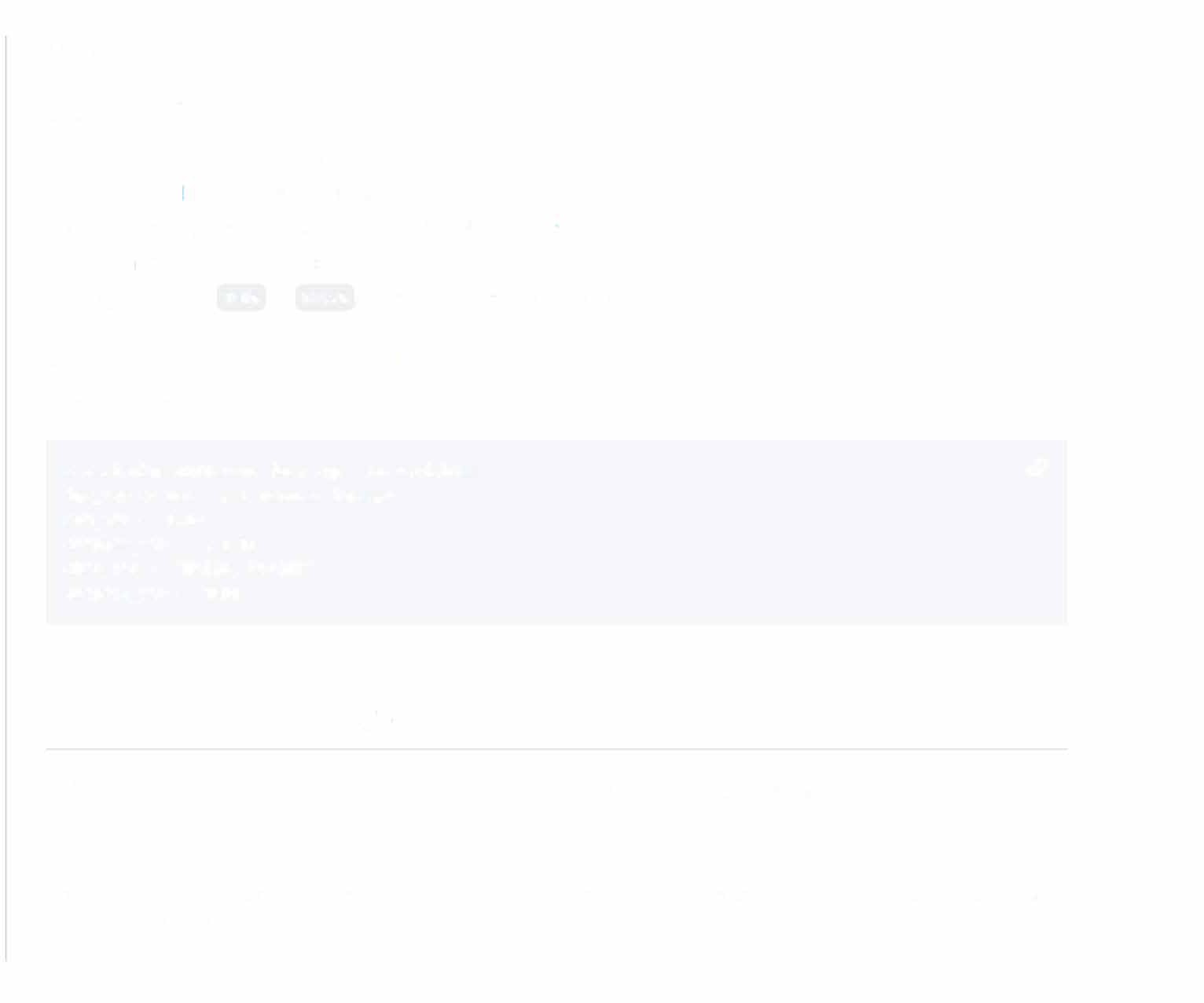
To manually load a package, use the library() or require() functions. These functions load the specified package.

library(devtools) # Load the devtools package *c9*

library(dplyr) # Load dplyr

1. **What are the different *data types* in R?**

R, being a dynamic programming language, performs type conversion based on context. Here are the supported

data types.

# Basic Data Types in R

1. Integer (int): Whole numbers.
2. Numeric (double): Real numbers.
3. Complex: Numbers with both real and imaginary parts..
4. Character (string): Text in quotes.
5. Logical (bool): TRUE or FALSE , used for boolean operations.

# Example: Assigning Data Types in R

Here is the R code:

# Assigning different data types to variables *c9*

int\_var <- 10L # L denotes integer num\_var <- 3.14

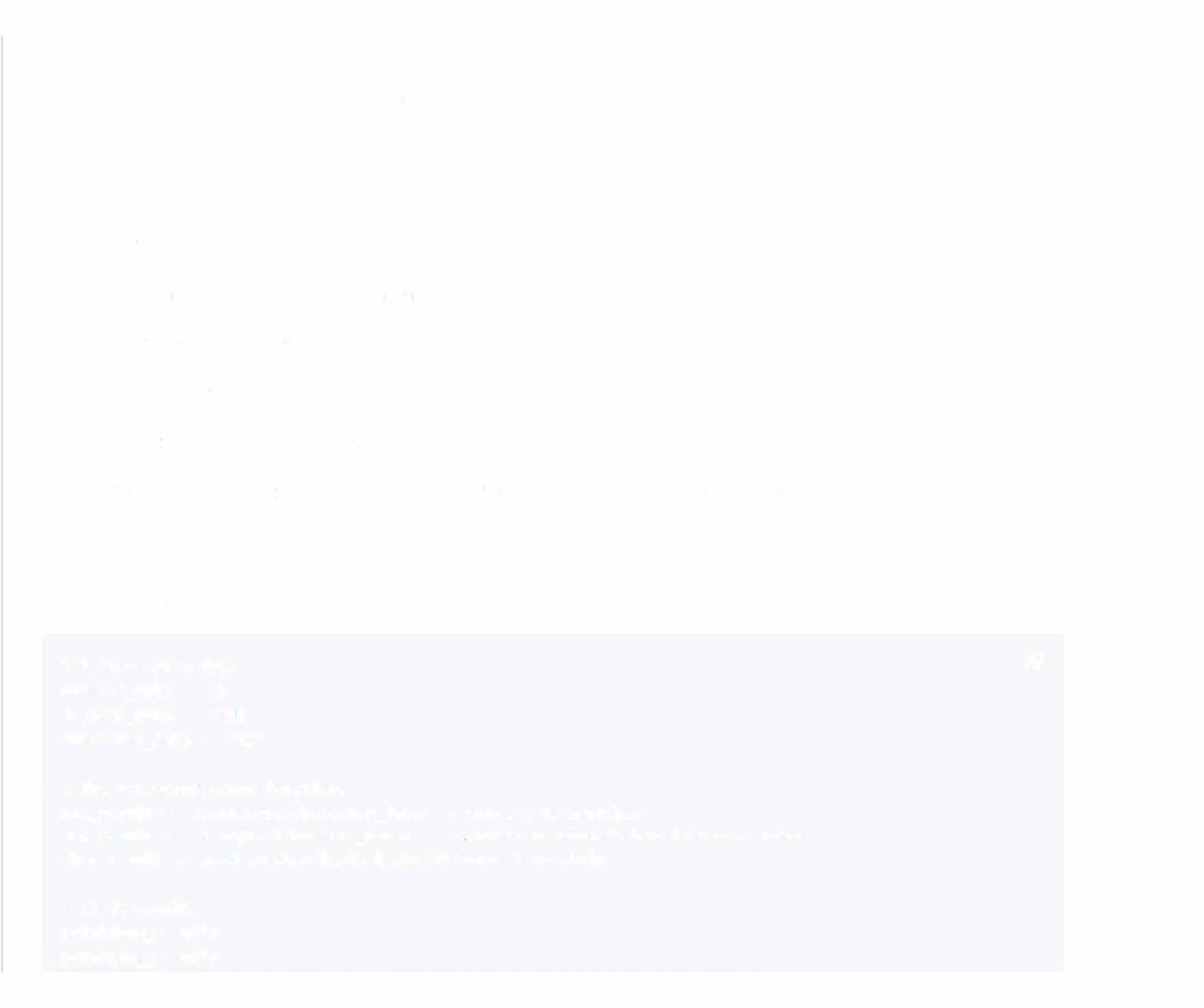
complex\_var <- 2 + 3i char\_var <- "Hello, World!" logical\_var <- TRUE

1. **How do you convert *data types* in R?**

In R, you can use functions to convert between different data types and manage missing values.

# Type Conversion Functions

R offers several conversion functions, making it essential to choose the appropriate one to avoid potential data loss and inconsistencies.

* as.character(x): Convert to string.
* as.numeric(x): Convert to a floating-point number. Missing values are represented as "NA".
* as.integer(x): Convert to a whole number.
* as.logical(x): Convert to a Boolean.
* as.factor(x): Convert to a factor, a categorical variable.
* as.data.frame(x): Convert to a data frame.
* as.list(x): Convert to a list.
* as.vector(x): Convert to a vector.
* as.Date(x): Convert to a date object.
* as.POSIXct(x): Convert to a date-time object, represented in seconds from the epoch.

# Code Example: Type Conversion

Here is the code:

# Declare some data *c9*

numeric\_data <- 23 logical\_data <- TRUE character\_data <- "42"

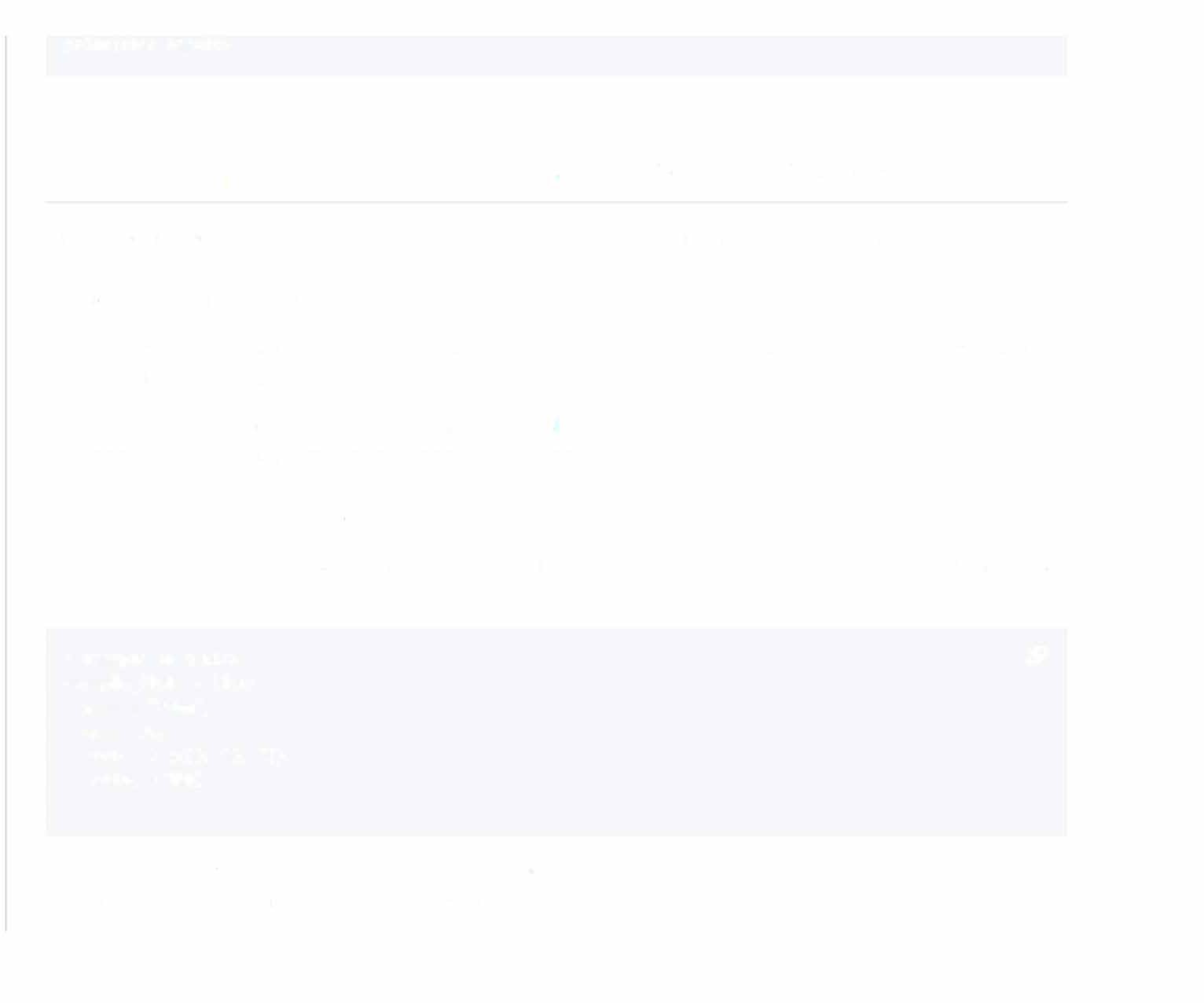
# Use type conversion functions

num\_result <- as.numeric(character\_data) # Convert to a number

**1 ng\_rpc;ult** *<***- �c;. 1 ngi *r.:i* 1 ( numPri *r*\_d,=rt.:i) # C:onvprt to Roo lP.:in h;::ic;p,1 on A or non-A**

char\_result <- as.character(logical\_data)# Convert to string

# Check results print(num\_result) print(log\_result)

print(char\_result)

1. **Can you explain the difference between a *list* and a *dataframe* in R?**

Both lists and dataframes are key structures in R, with the key difference being their tabular nature.

## Introduction to Lists and Dataframes

* + Lists: This structure is highly versatile and can contain mismatched types of data such as numbers, characters, vectors, or even other lists.
  + Dataframes: These are specifically designed to hold tabular, or spreadshee t-like , data where each column is of the same length and typically holds the same type of data.

## Tabular View and Structure

* + Lists: Can be conceptualized as a collection of named elements where each element can be of different data type or length.

# Example of a List *c9*

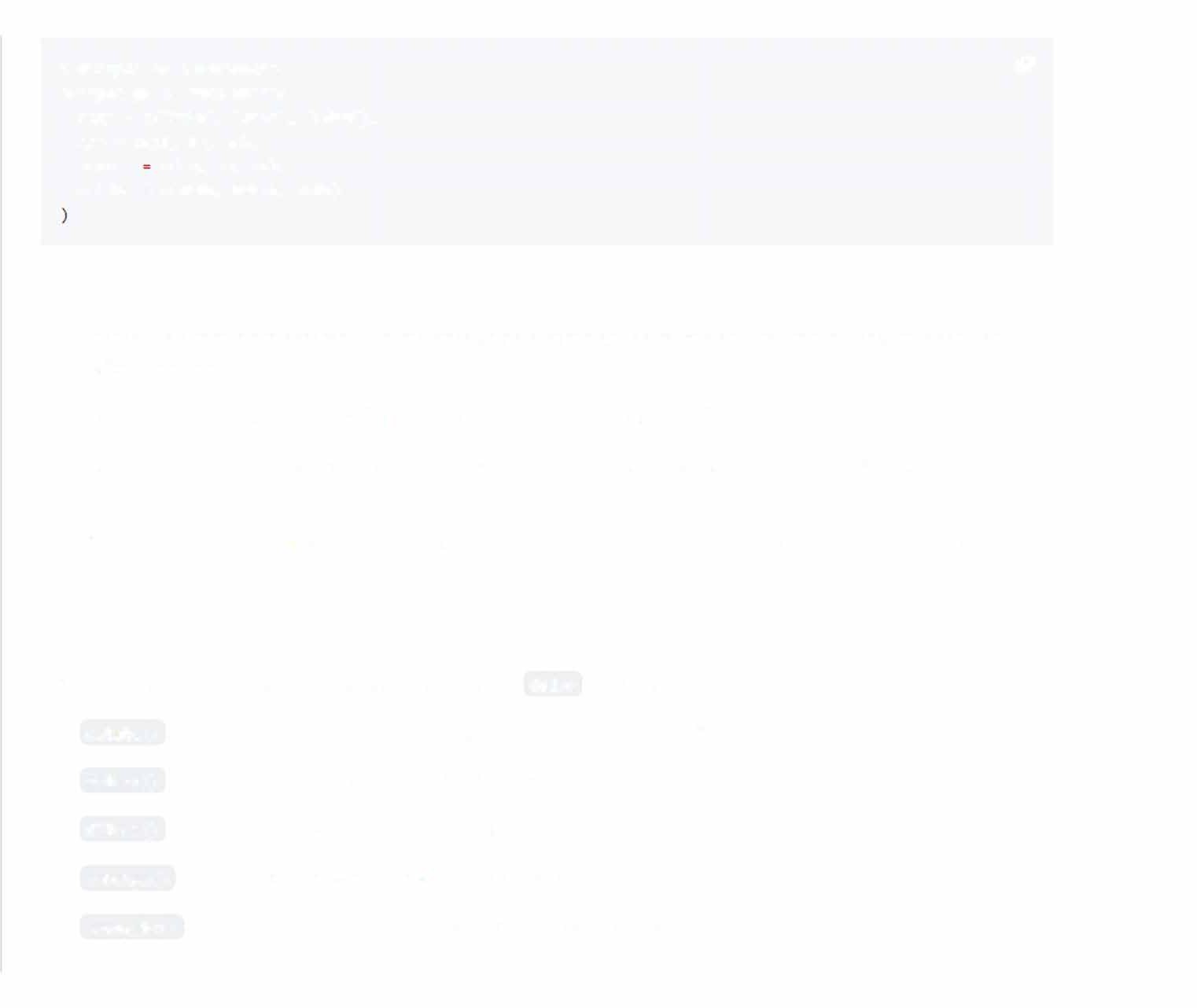
example\_list <- list( name = "'John",

age = 25,

scores = c(80, 90, 75), active = TRUE

**)**

* + Dataframes: Are a two-dimensional data structure, akin to a table or spreadsheet, where data is organized into rows and columns. Each column can be seen as a list.

# Example of a Dataframe *c9*

example\_df <- data.frame(

name= c("John", "Sara", "Adam"), age= c(25, 30, 28),

scores c(80, 90, 85),

active= c(TRUE, FALSE, TRUE)

# Operations and Capabilities

* Access: Dataframes are accessed like matrices using a combination of row and column indices. Lists can use indices or names.
* Homogeneity for Columns: Dataframes enforce data type homogeneity for each column, while lists do not.
* Homogeneity for Rows: All rows in a dataframe must have an equal number of elements. Lists do not have such restrictions.
* Vectorizing Functions: Dataframes have special functions which allow the simplification of operations. Lists do not offer this feature.

# Common Operations

Both lists and dataframes can be manipulated using the dplyr package:

* mutate() : This function can add or modify columns in dataframes and lists.
* select() : Selects a subset of columns or list elements.
* filter() : Applies a selection criterion, retaining only matching rows or list elements.
* arrange() : Sorts the data based on specified columns or list elements.
* group\_by() : Enables grouping of data for subsequent operations such as summarization.

# Best Practices

* Prefer Dataframes: They offer clearer structure, type consistency, and built-in functionality for various data tasks.
* List Usage: Use lists when you need to store an assortment of objects or datasets with irregular structures.
* Consistency and Coherence: Ensure data within a dataframe adheres to consistent data types and standard formatting to avoid potential issues during analysis.
* Package Usage: Familiarize with packages like dplyr for efficient manipulation of dataframe structures.

1. **How do you handle *missing values* in R?**

Handling missing values is a crucial preprocessing step in machine learning. R offers various techniques for identifying and manipulating data with missing values.

# Identifying Missing Values

In R, NA is the default indicator for missing values. You can use the functions is. na() and na. omit() to detect and eliminate them:

# Code example: Identify and Omit Missing Values

Here is the R code:

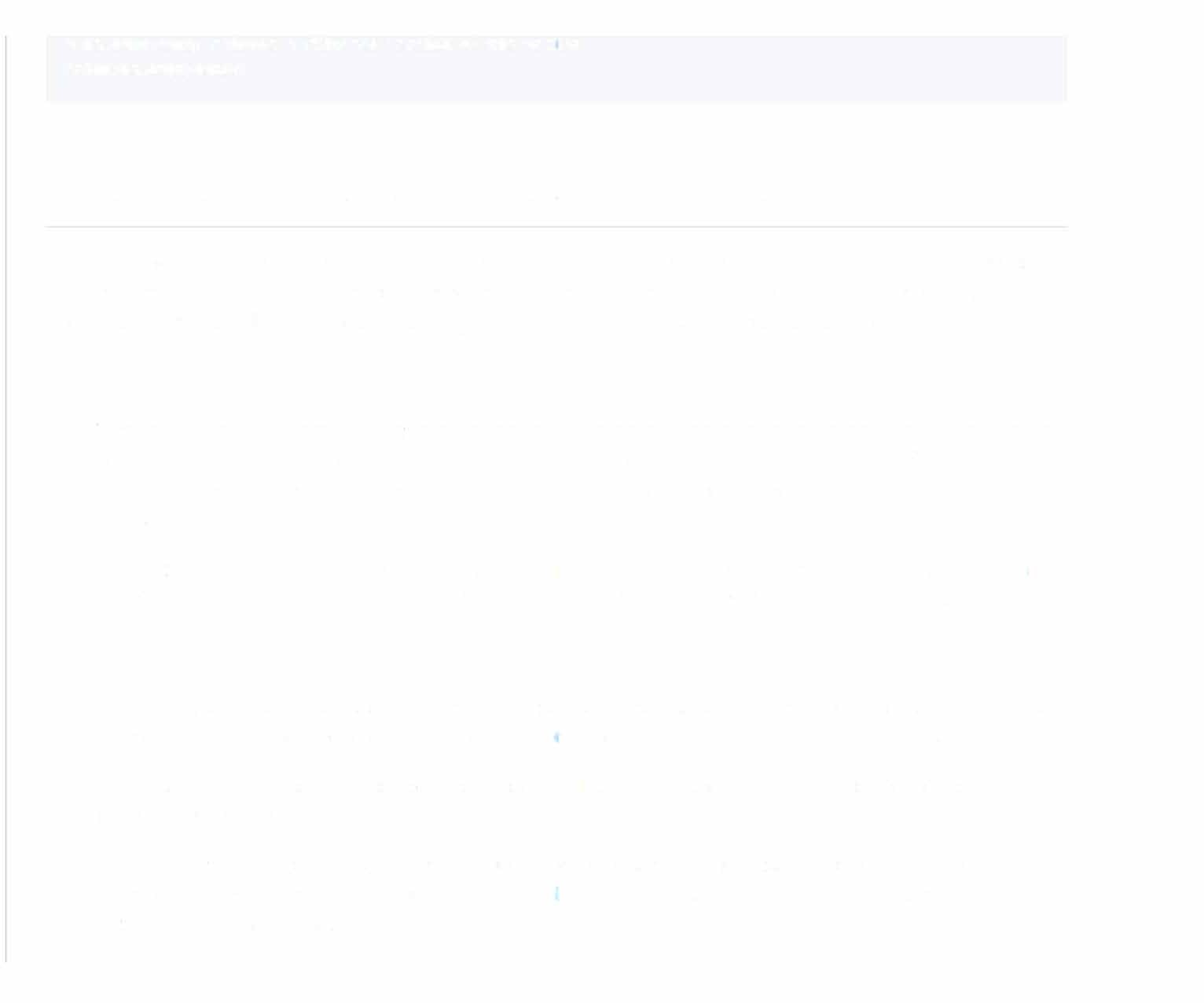
# Sample vector 01ith missing values *c9*

data<- c(1, 2, NA, 4, 5)

# Check for missing values

# is.na(data) ,,ill return a logical vector print(is.na(data))

# Remove missing values

# na.omit(data) returns a filtered version of the vector print(na.omit(data))

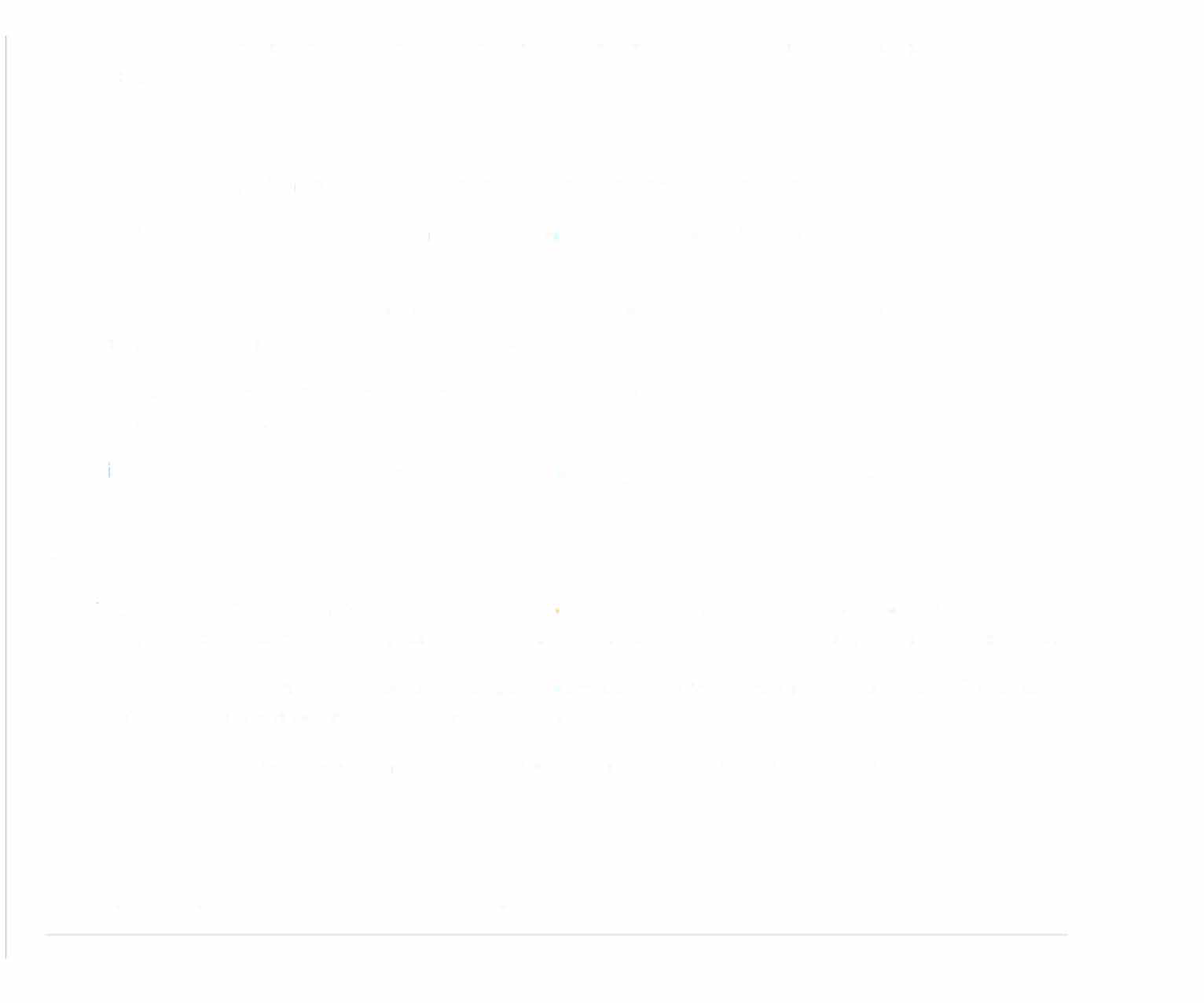
1. **What is the use of the *apply()* family of functions in R?**

The apply() family of functions in R is a powerful tool that allows for efficient vectorized operations. These functions are known for their succinctness and ability to perform complex manipulations without explicit iteration. They are indispensable for tasks like filtering, summarizing, and handling non-rectangular data structures.

# Types of apply Functions

1. Vectorized Functions: These are functions that operate on every element of a vector independently. They do not require any specific order or other elements in the vector for computation. Many of the basic R functions are inherently vectorized. Vectorized functions are generally designed for speed, especially when used on large vectors.
2. Non-vectorized functions: Such functions require explicit iteration over the elements of an object to perform their computation. This can be demanding both in terms of programming and computational efficiency.

# The Four Primaries in the Apply Family

* + apply(): Principal function for applying another function to the rows or columns of a matrix, or to the margins of an array. It is more versatile and can handle diverse datta types, but it's not always the fastest option.
  + Iapply(): Short for "list apply," it's devoted to list objects. lapply() will apply the designated function to every component of the list, then return a list.
  + sapply(): Abbreviated from "simplify apply," it's an extension of lapply() that attempts to collapse the output into a more convenient format (like a vector or matrix). It's highly efficient and one of the most commonly used functions in the apply family.
  + vapply(): An improved, more specific version of sapply0 that enables the user to define the output type of the function.

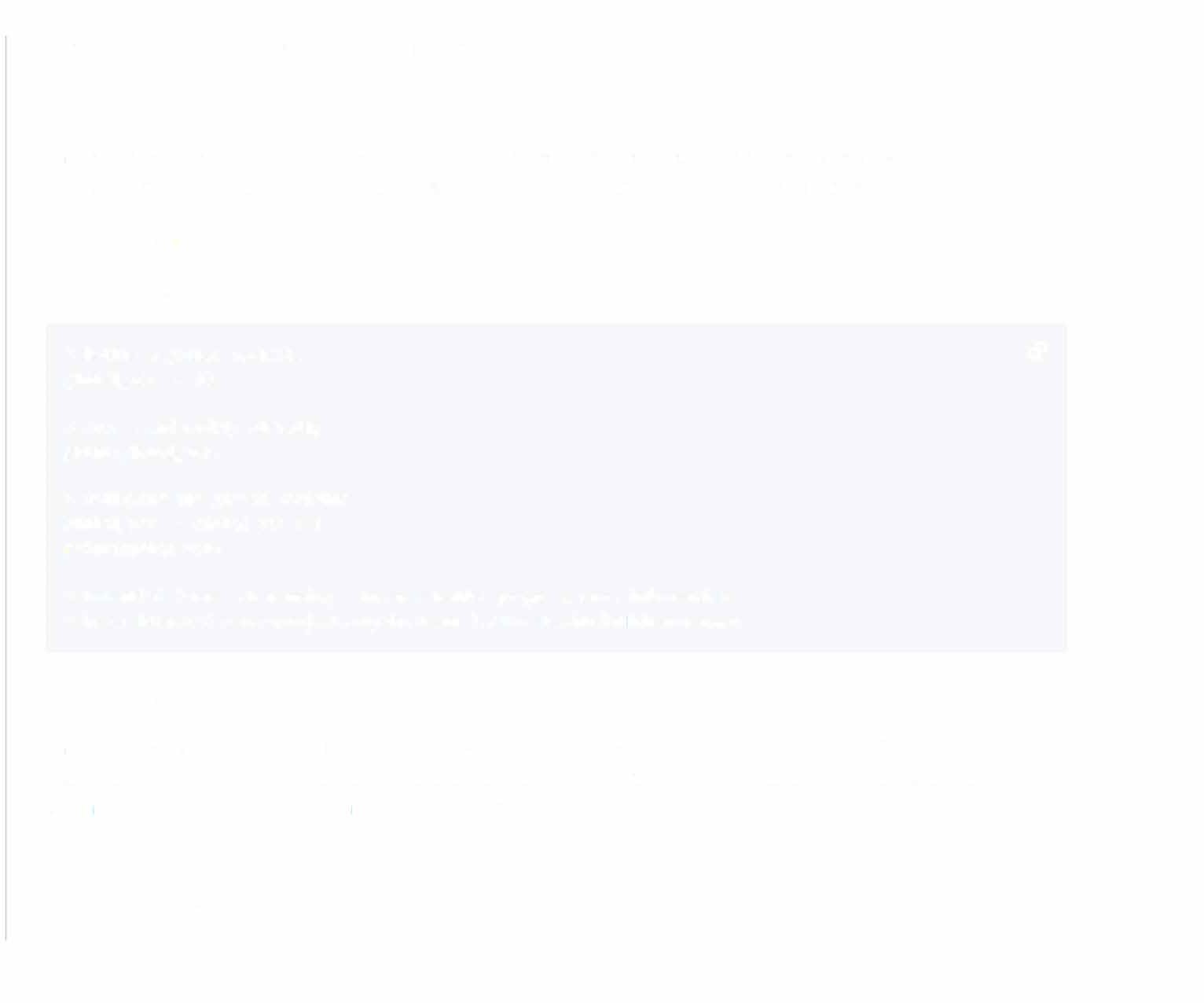
# Use Cases

* + Working with Lists: Ideal for performing the same operation on every element in a list.
  + Data Frame Summaries: Useful for obtaining a summary statistic (or any other function) across multiple columns of a data frame.
  + Data Management: Useful for handling... multi-dimensional data structures like matrices and arrays. This includes applying functions to rows or columns, transforming 2-D matrices, and more.
  + Reshaping: Essential for reshaping data between longer and wider forms, and for splitting data into levels based on group-specific attributes.
  + Grid Searching: Frequently employed in machine learning for grid searching hyperparameters in combination with combn().

# Efficiency Considerations

* + Vectorized Code is Often Faster: In general, R is more efficient when working with vectors. Many of the apply­ related functions involve underlying looping operations, which can be less efficient, especially for larger datasets.
  + Parallel Processing: If you need a performance boost, consider the foreach and doParallel packages. These tools can distribute workloads across multicore or cluster systems.
  + External Libraries: Repp and data.table are two packages known for improving computational efficiency across a range of use cases.

1. **Explain the *scope of variables* in R.**

In R, variables can have either Global or Local scope.

# Global Scope

Variables created outside of a function are globally scoped. Accessible from any part of the program, they maintain their state for as long as the program is running, or until they are explicitly modified or deleted.

Example: Global Scope Here is the R code:

# Define a global variable *c9*

global\_var <· 10

# Access and modify globally print(global\_var)

# Modifying the global variable global\_var <· global\_var + 5 print(global\_var)

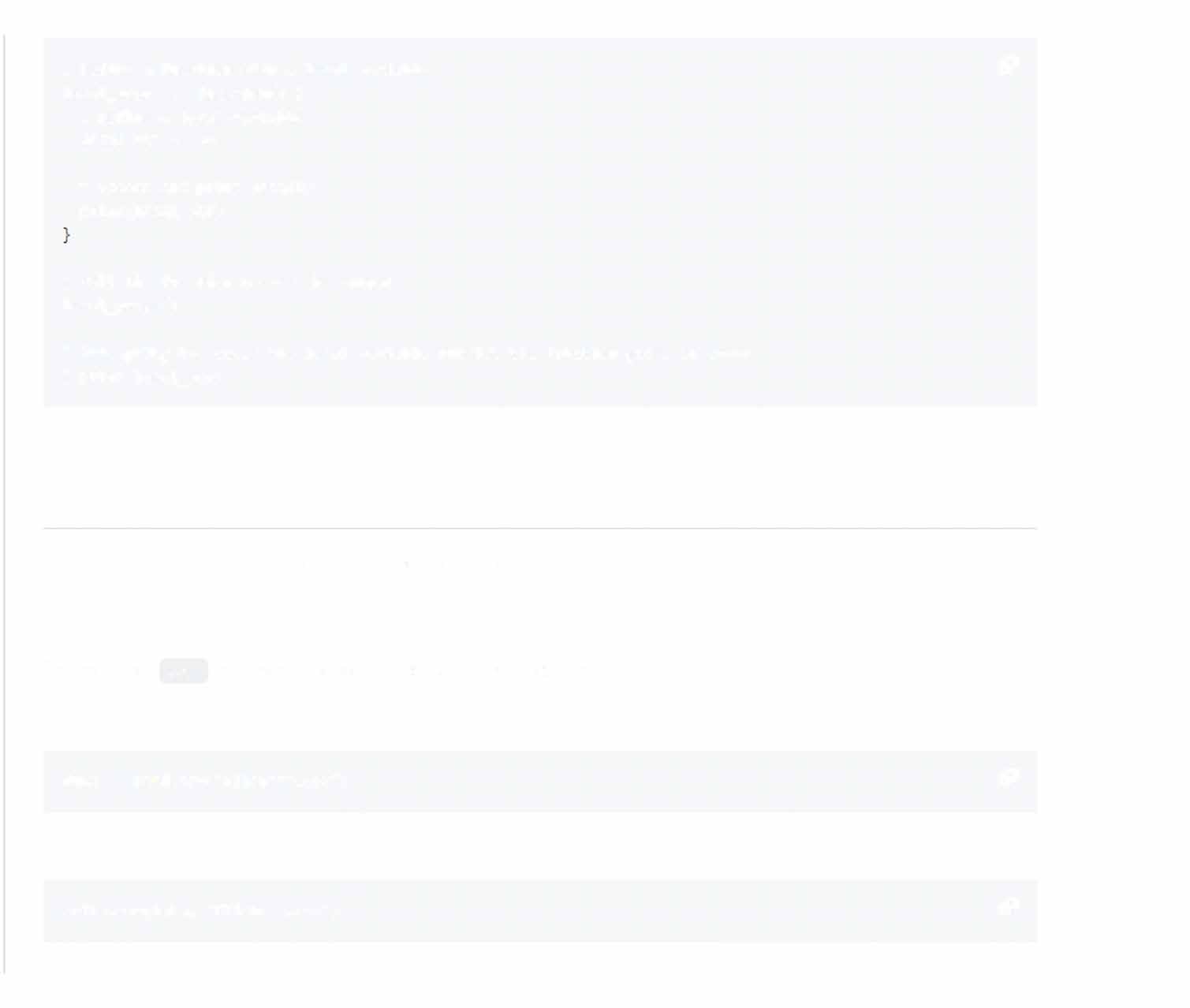
# Potential issues 01hen using R in an embedded program, more information

# here: https://cran.r-project.org/doc/manuals//R-exts.html#Hidden-dangers

# Local Scope

Variables created within a function are locally scoped. These variables are created whenever the function is called and destroyed when the function is exited. Functions get their own environment that acts as a sandbox, separate from the global environment, hence these variables do not overwrite global variables.

Example: Local Scope Here is the R code:

# Define a function with a local variable *c9*

local\_scope <- function() { # Define a local variable local\_var <- 20

# Access and print locally print(local\_var)

# Call the function to see the output local\_scope()

# Attempting to access the local variable outside the function gives an error # print(local\_var)

1. **How do you *read* and *write data* in R?**

R offers a variety of methods to handle data input and output.

**CSV Files**

The common .csv format can be both read from and written to. Read from CSV:

data<- read.csv('filename.csv') *c9*

Write to CSV:

�1rite.csv(data, 'filename.csv') *c9*

# RDS Files

The .rds format is native to Rand preserves both the data and its structure. Read from RDS:

data<- readRDS('filename.rds' ) *c9*

Write to RDS:

saveRDS(data, 'filename.rds·) *c9*

# Excel Files

R can export and import data to Excel, but tool s like readxl and ,,ritexl are often leveraged for smoother interactions.

Read from Excel:

library(readxl) *c9*

data<- read\_excel('filename.xlsx·, sheet = 1)

Write to Excel:

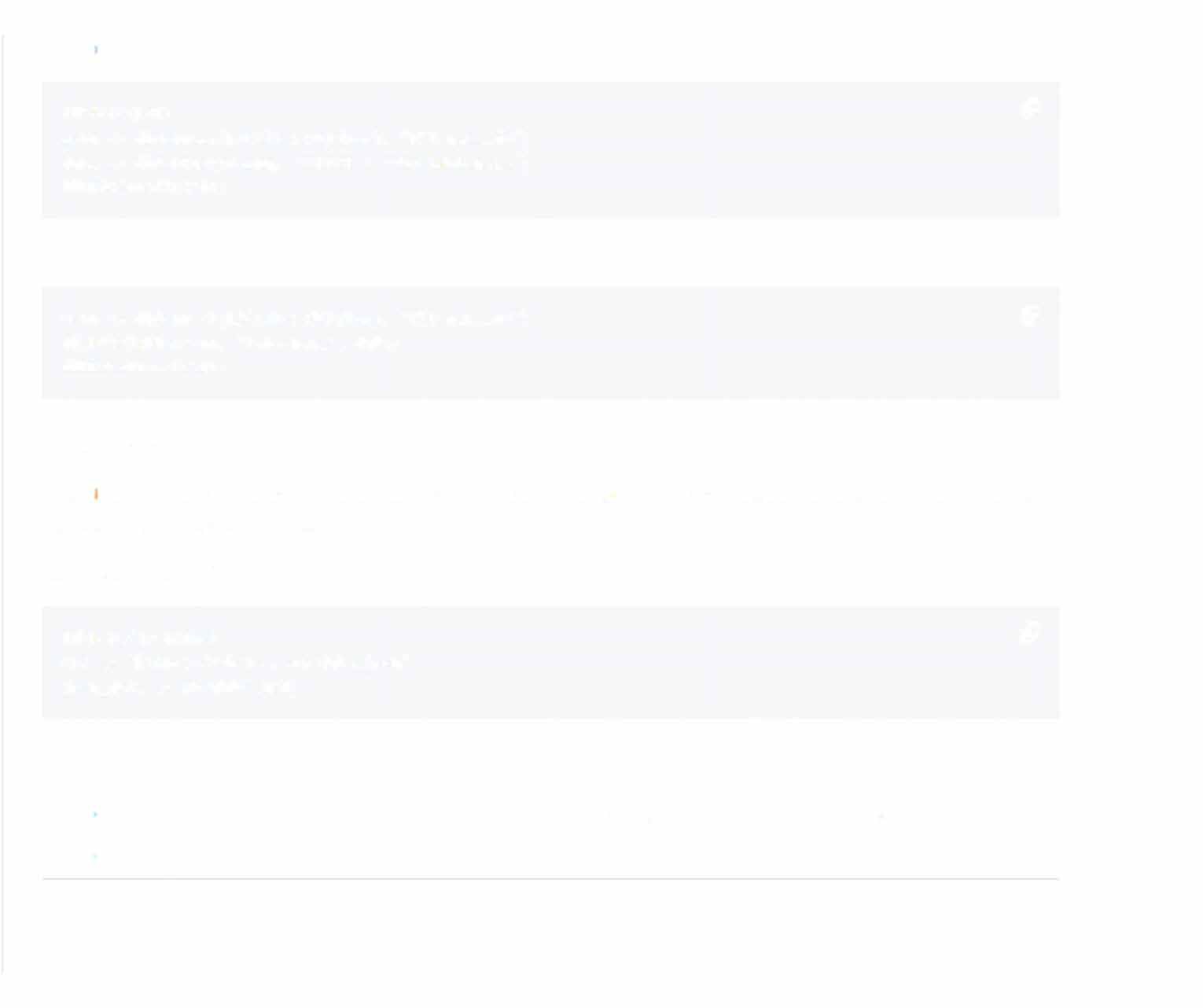
library(e1ritexl) *c9*

* irite\_xlsx(data, 'filename.xlsx·)

# SQL Databases

Using R's interface to SQL databases, users can read and **write** tables.

Read from SQL:



library(DBI} *c9*

conn <· dbConnect(RSQLite:: SQLite(), "filename. db") data<· dbGetQuery(conn, 'SELECT • FROM tablename·) dbDisconnect(conn)

**Write** to SQL:

conn <· dbConnect(RSQLite:: SQLite(), "filename. db") *c9*

dbWriteTable(conn, 'tablename', data) dbDisconnect(conn)

# Web Protocols

R has tools for fetching data from the web, which can be further parsed and processed. For example, JSON data can be directly extracted and processed.

Read JSON from Web:

library(jsonlite) *c9*

url <• • https: //01ebsite. com/data. json' json\_data <· fromJSON(url)

1. **What are the key differences between Rand *Python* for *Machine Learning?***

# Key Distinctions

R: Strengths & Weaknesses

* + Pros: R is a statistical powerhouse, excelling in data visualization, exploration, and statistical analysis. Its libraries, such as ggplot2, are renowned for data visualization.
  + Cons: While the language is beginner-friendly, it might have a steeper learning *curve* for general-purpose programming tasks and building end-to-end ML pipelines.

Python: Strengths & Weaknesses

* + Pros: Python is hailed for its versatility and ease of use.. Its rich ecosystem, especially with libraries like TensorFlow, Keras, and Scikit-Learn, makes it a top choice for many.
  + Cons: Python can be slower than R for certain numeric computations and statistics, depending on the specific libraries being used.

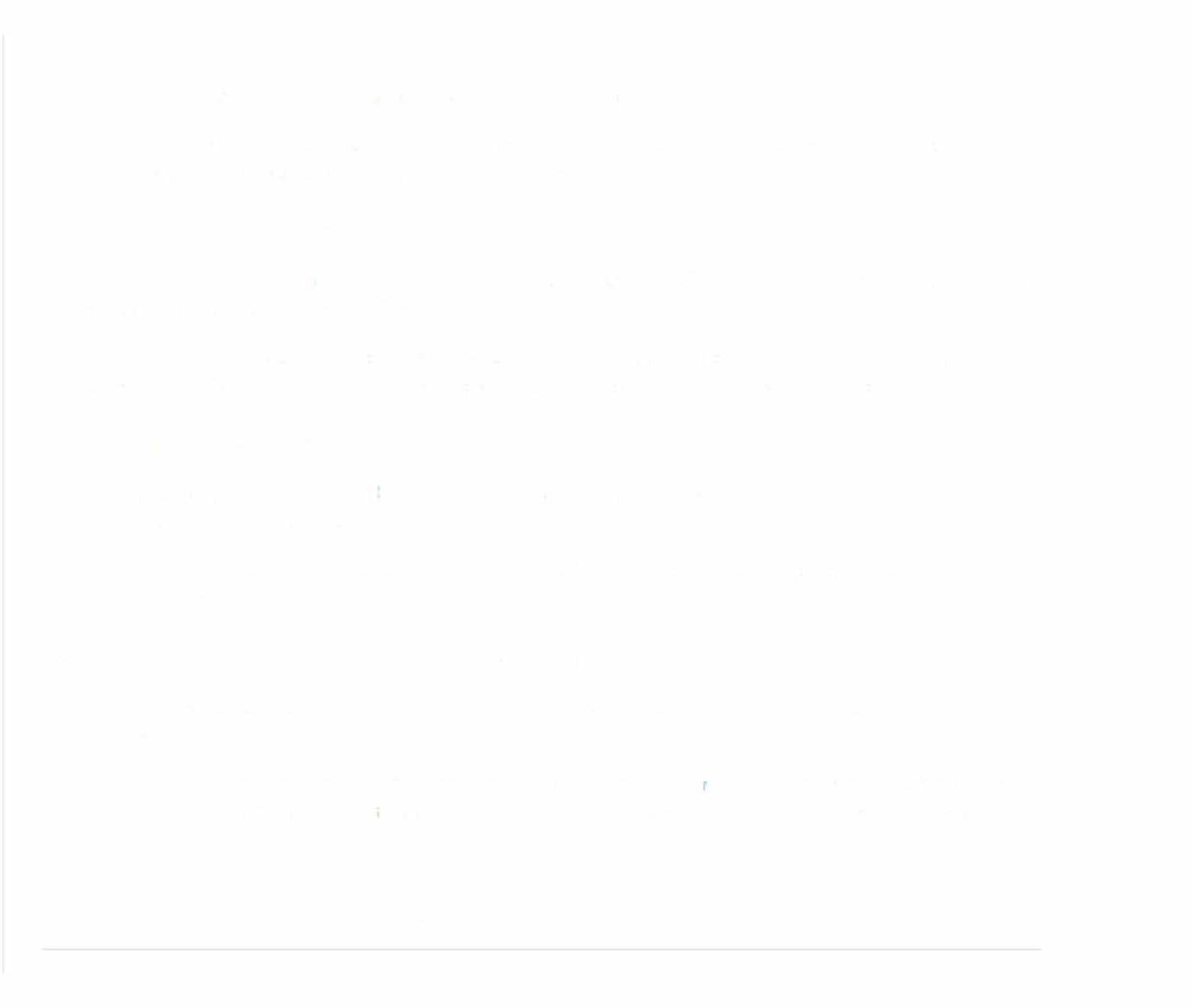
**Speed and Performance**

* + R: Offers excellent built-in tools for data analysis and modeling, benefiting from optimized C and Fortran libraries. However, it might lag behind Python in areas not specialized for such tasks.
  + Python: Typically requires auxiliary libraries, such as NumPy and Pandas, for numeric computing, which can potentially hinder raw computation performance.

Code Quality: R and Python

* + Code Readability: Python, known for its elegant and readable syntax, uses indentation for code structure. R, with its multitude of statistical functions and packages, can sometimes suffer from less readable code, particularly in nested loops.
  + Standard Libraries & Packages: Both R and Python offer vast libraries, each excelling in specific areas. R's dplyr and ggplot2 simplify data analysis and visualization, while Python's scikit-learn is a go-to for machine learning tasks.

# Model Interpretability



* + **R:** *It provides excellent transparency for models with built-in statistical test outputs and visualizations.*
  + *Python: Has evolved with packages like Lime, SHAP, and ELIS, making it more interpretable, but it might still require additional packages for in-depth model interpretability.*

# Graphing and Data Visualization

* + **R:** *Its data visualization libraries, such as ggplot2, are highly regarded, offering a "grammar of graphics" approach for sophisticated and intuitive visualizations.*
  + *Python: Python's Matplotlib is it s base visualization library and is powerful but can require more lines of code for complex plots. To address this, the Seaborn library is often used for statistical data visualization.*

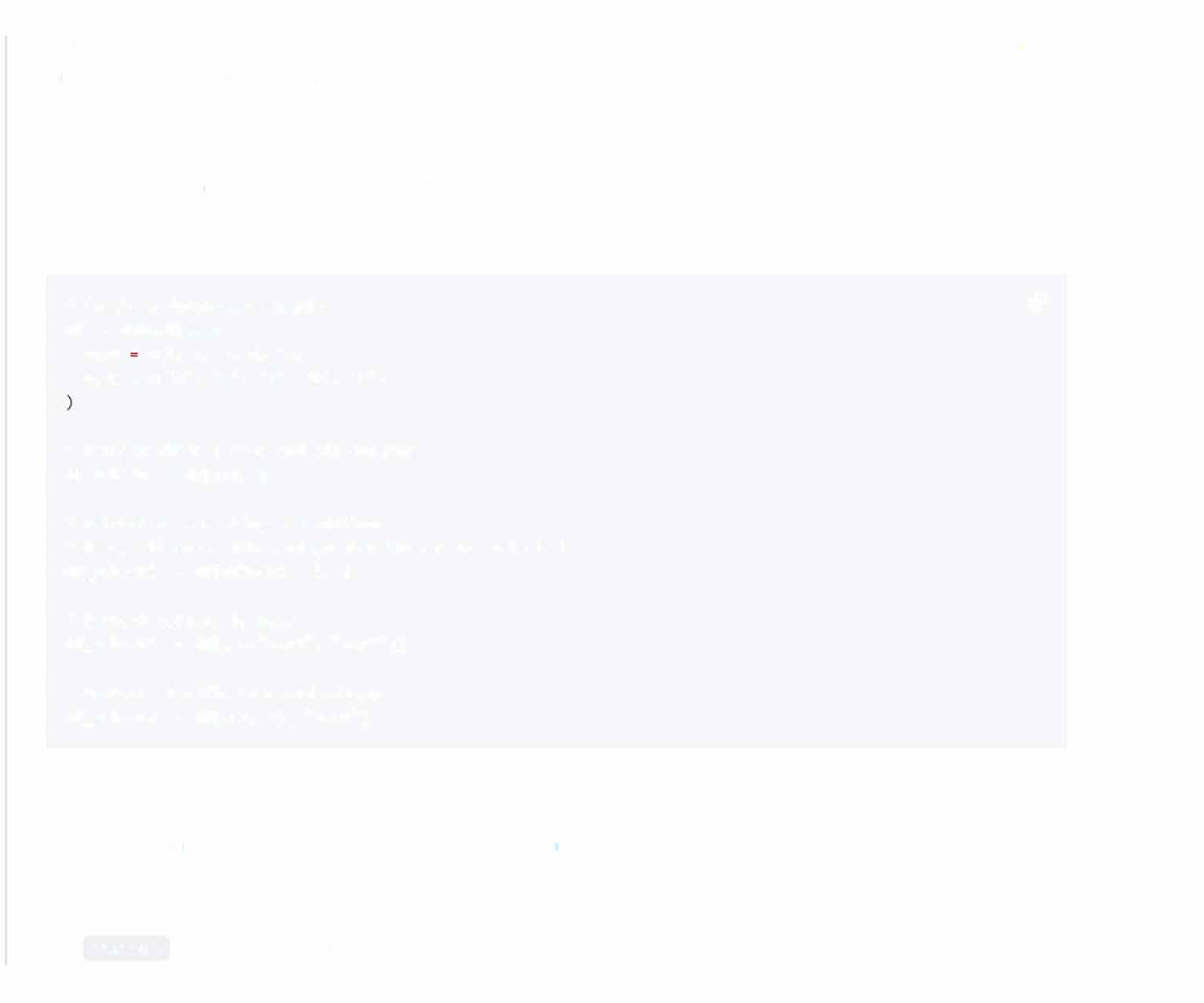
# Data Manipulation and Built-in Data Structures

* + **R:** *Its data frames are centrally usec, based on the DataFrame concept from statistics software S, ensurinq efficient data manipulation and analysis.*
  + *Python: Makes use of lists and dictionaries, offering flexibility at the cost of possible performance trade-offs in certain situations.*

# Text Processing and Natural Language Processing (NLP)

* + **R:** *Employs packages such as "tm" for text mining and analysis. While it is robust, the NLP ecosystem in R might not be as extensive as what Python offers.*
  + *Python: Widely recognized for its NLP capabilities, thanks to libraries like NLTK (Natural Language Toolkit) and spaCy. These are go-to choices for '.asks like tokenization, part-of-speech tagging, and named entity recognition.*

1. **How do you select a *subset* of a *dataframe?***

In R, you can extract a subset from a dataframe in various ways. Some of the most common methods are using base R functions, as well as libraries like dplyr and tidyverse.

# Base R: Using Indices

You can use logical or numeric indices to extract specific rows or columns.

CodeExample

# Create a dataframe example *c9*

df <- data.frame(

var1 c(1, 2, 3, 4, 5),

·c·, ·o·,

·s·,

'E')

var2

=

c('A',

# Extract first 3 rows and all columns df\_subset <- df(1:3, )

# Extract roo,s meeting a condition

# Here, all roo,s 01ith var1 greater than 3 are selected df\_subset2 <- df(df$var1 > 3, )

# Extract columns by name

df\_subset3 <- df(, c("var1", "var2"))

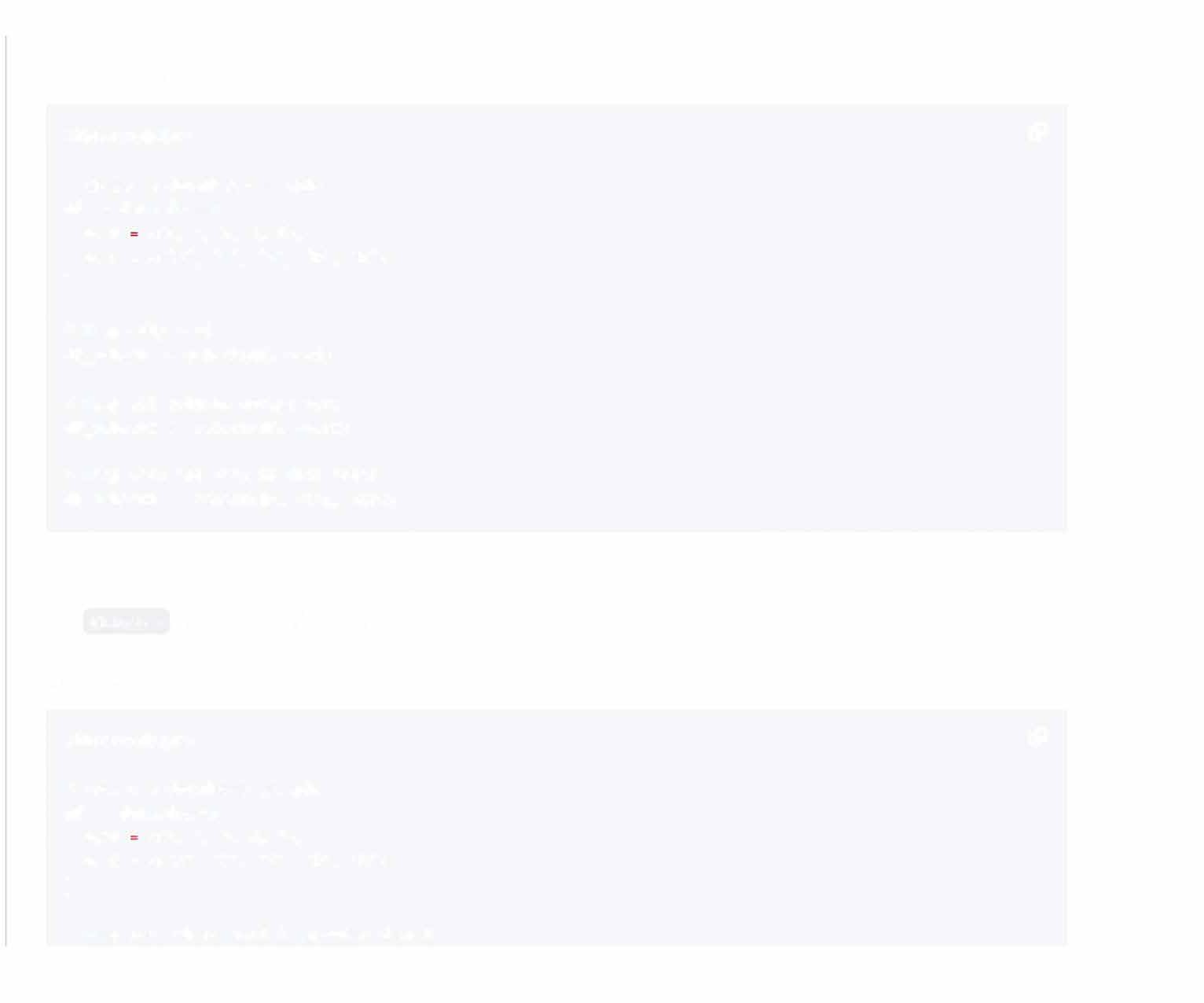
# Extract specific roo1s and columns df\_subset4 <- df(c(1, 3), "var1")

# Dplyr: Concise Data Manipulation

The dplyr package offers a user-friendly grammar for data manipulation.

Selecting in Dplyr: Using select()

Use select() to specify columns to keep or drop.

CodeExample

library(dplyr) *c9*

# Create a dataframe example df <- data.frame(

var1 c(1, 2, 3, 4, 5),

var2 = c('A', ·s·, ·c·, ·o·, 'E'}

**)**

# Keep only var1

df\_subset <- select(df, var1}

# Keep all columns except var2 df\_subset2 <- select(df, -var2}

# Keep var1 and var2 in that order df\_subset3 <- select(df, var1, var2}

Filtering Rows with Dplyr: Using filter()

Use filter() to define conditions for row selection.

CodeExample

library(dplyr) *c9*

# Create a dataframe example df <- data.frame(

var1 c(1, 2, 3, 4, 5),

'E'}

·s·, ·c·,

·o·,

var2

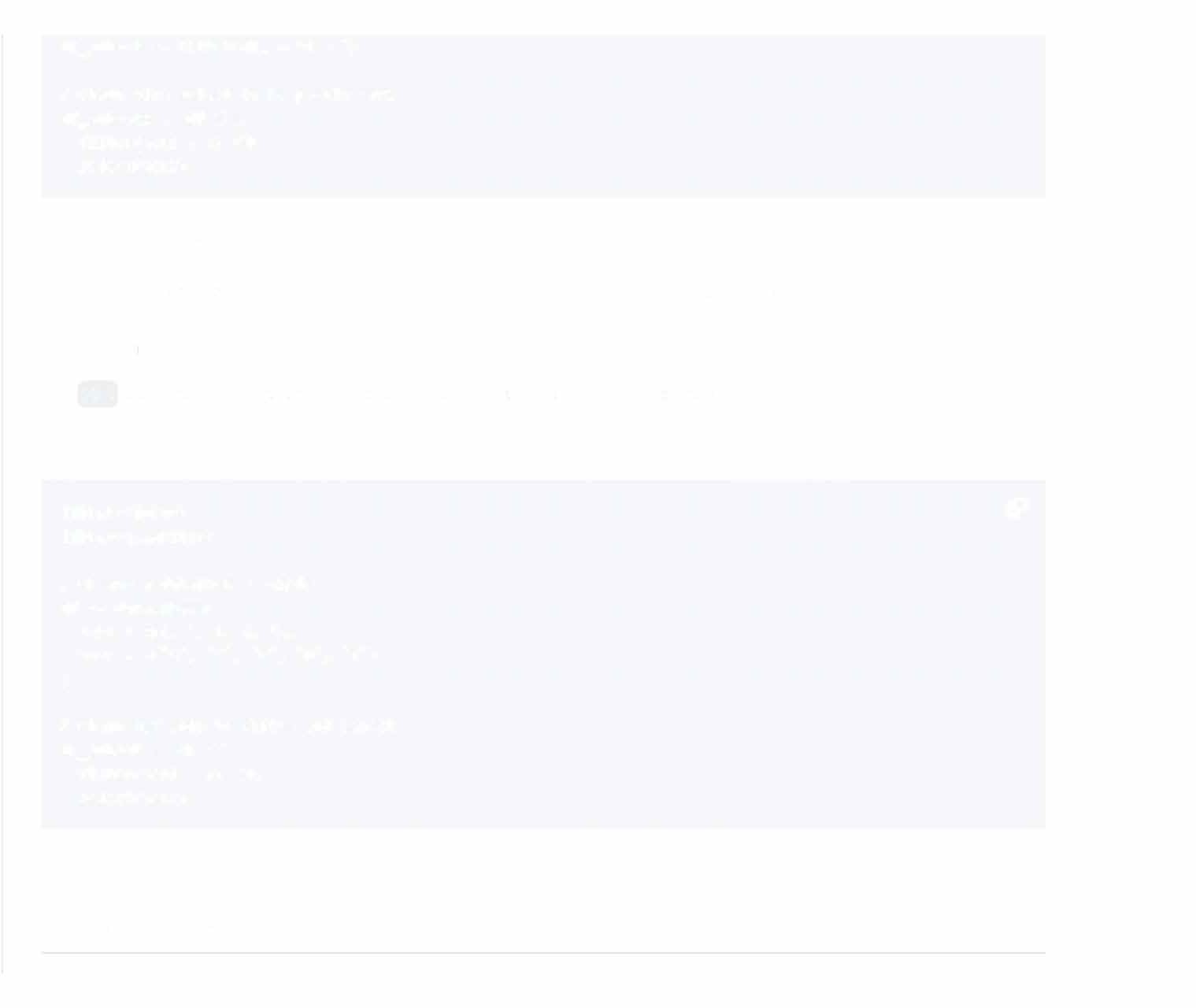
=

c('A',

**)**

# Keep roi.s •1here var1 is greater than 3

df\_subset <- filter(df, var1 > 3)



# Chain 01ith select to keep only var2 df\_subset2 <- df %>%

filter(var1 > 3) %>% select(var2)

# Tidyverse: Combining dplyr and magrittr

The tidyverse promotes a consistent approach to data analysis, integrating packages like dplyr.

Pipe Operator:%>% from magrittr

The %>% operator serves to chain functions, facilitating a more structured workflow.

CodeExample

library(dplyr) *c9*

library(magrittr)

# Create a dataframe example df <- data.frame(

var1 = c(1, 2, 3, 4, 5),

var2 = c('A', ·s·, ·c·, ·o·, 'E')

)

# Chain commands to filter and select df\_subset <- df %>%

filter(var1 > 3) %>% select(var2)

**12\_ Explain the use of the *dplyr package* for *data manipulation.***

dplyr is a versatile data manipulation package in R which is part of the tidyverse ecosystem. It offers a grammar of data manipulation, making common data tasks intuitive with a focus on clarity and consistency.

## Core dplyr Functions

* select(): Choose Variables
* filter(): Pick Observations
* arrange(): Rearrange Rows
* mutate(): Create/Modify Variables
* summarize(): Summarize Variables
* group\_by(): Define Data Groups
* rename(): Change Variable Names
* distinct(): Unique Observations
* top\_n(): Select Top Observations

## Benefits of dplyr

* Intuitive Syntax: Utilizes a dot-pipe operator ( %>%) for natural left-to-right code flow.
* Task-Driven Design: Actions are task-focused, enhancing code clarity.
* Database Compatibility: dplyr works well with databases and other data sources when used in conjunction with dbplyr.
* Optimized Performance: Its back-end, through various: data structures like data frames, provides optimized data processing.
* Consistency and Code Reusability: dplyr commands can be effectively combined, promoting consistent data workflows.

# Code Example: dplyr in Action

Here is the R code:

# Load Libraries and Dataset *c9*

library(dplyr) data<- mtcars

# Filter & Arrange Data filtered\_data <- data%%>

filter(mpg > 20} %%>

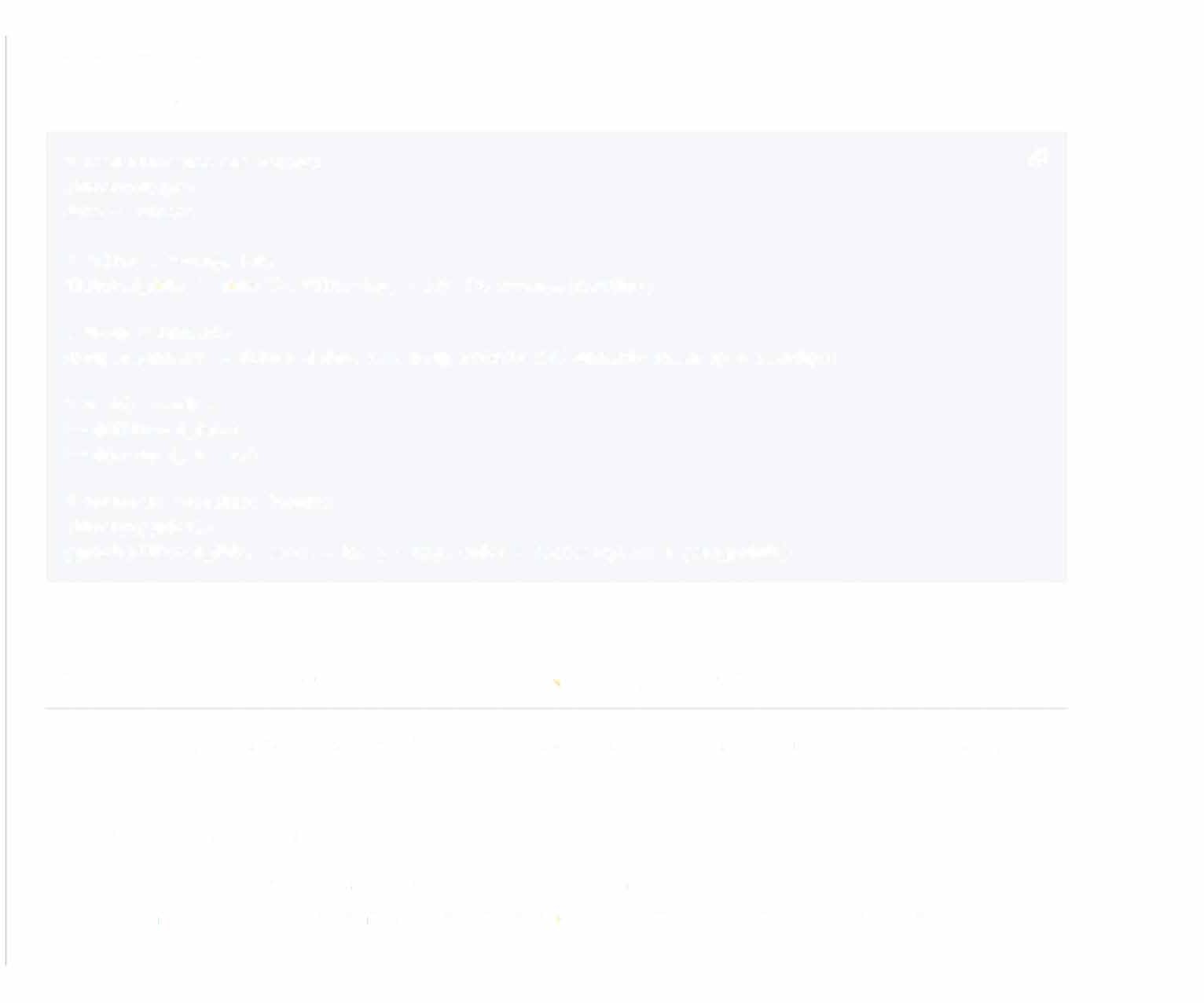
arrange(desc(hp})

# Group & Summarize

grouped\_summary <- filtered\_data%%>

group\_by(cyl)%%>

summarise(mean\_hp = mean(hp})

# Verify Results head(filtered\_data) head(grouped\_summary)

# Optional: Visualize Results library(ggplot2}

ggplot(filtered\_data, aes(x = hp, y = mpg, color = factor(cyl))) + geom\_point()

**13\_ How can you *reshape data* using *tidyr package?***

In R, the tidyr package provides tools for tidying data, allowing for more straightforward visualizations and model building.

# Data Reshaping Principles

* Tidy Data: Each variable has its own column, and each observation has its own row.
* Data Frame: Represents the dataset, where each variable is a column, and each row is an observation.

# Key Functions

* gather() : Converts wide data to long format by stacking columns.
* spread() : Opposite of gather , it spreads unique values into columns.
* separate() : Splits a single column into multiple columns based on a deli miter.
* unite() : Merges multiple columns into one.
* complete() : Ensures that every combination of the specified variables is present.

# Code Example: Data Formatting

Here is the R code:

library(tidyr) *c9*

# Sample Data grades<- data.frame(

student c("Jack", "Jill", "Tom", "Jerry"}, class\_1 c(90, 85, 95, 88},

class\_2 c(80, 92, 89, 91}

)

# Convert 01ide data to long format long\_grades <- gather(

data = grades, key= "class", **value = "score",**

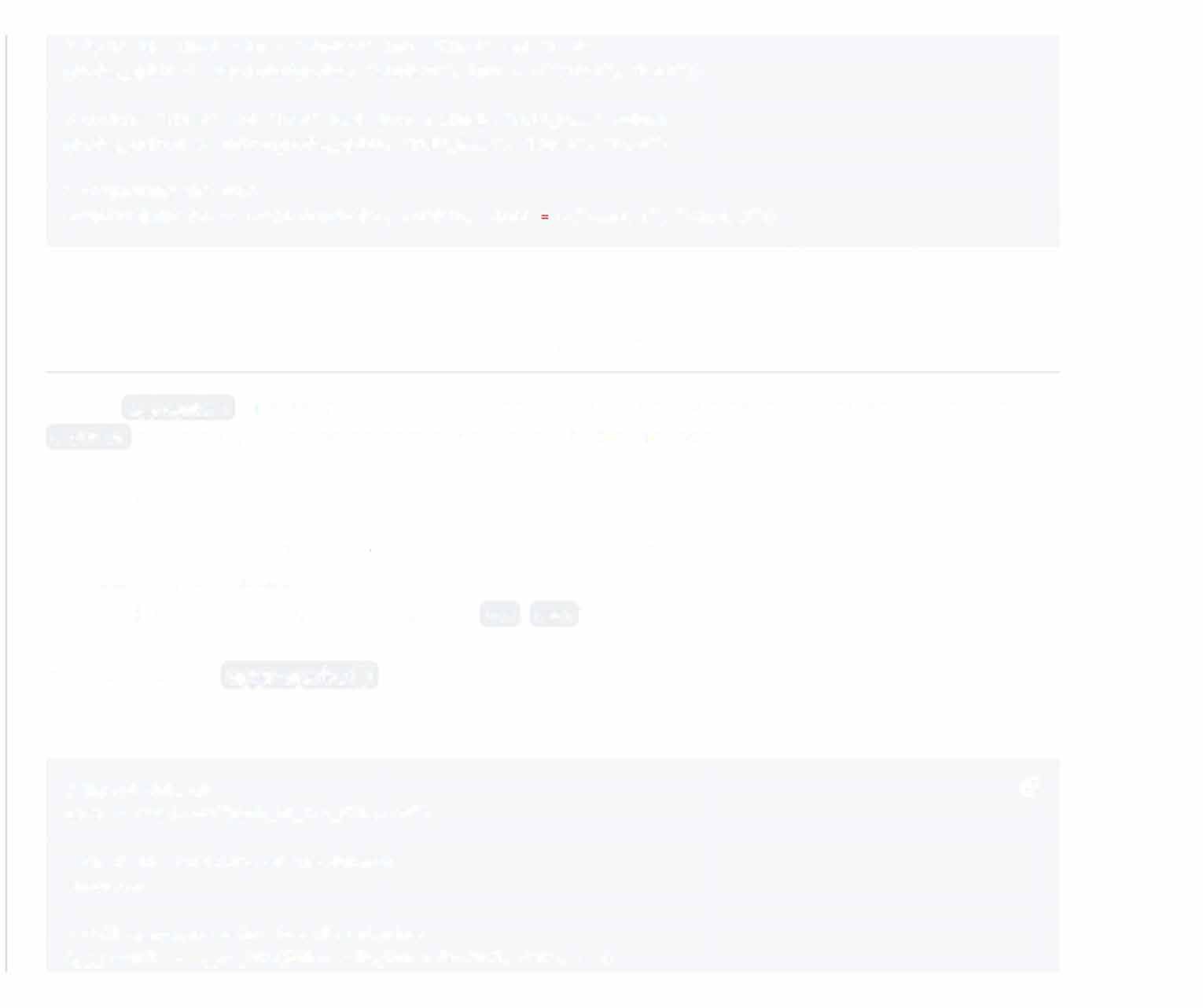
-student

)

# Convert long data back to wide format 01ide\_grades < - spread(

data = long\_grades, key= "class", value= "score"

)

# Split the first column 'student' into 'First' and 'Last' grades\_split<· separate(grades, "student", into = c("First", "Last"))

# Combine 'First' and 'Last' back into a single 'Full\_Name' column grades\_united<· unite(grades\_split, "Full\_Name", "First", "Last")

# Completing the data

completed\_grades<· complete(grades, student, class c("class\_1", "class\_2"})

1. **What is the function of the *aggregateO* function in R?**

In R, the aggregate() function serves to group data based on one or multiple factors. Its functionality is akin to the GROUP BY clause in SQL. This function is accessible through the base package.

# Key Arguments

* + Formula: Utilizes the formula interface to determine which columns to include.
  + Data: The source dataset.
  + FUN: Specifies the aggregation function (e.g., sum, mean).

# Example: Using aggregate ()

Here is the R code:

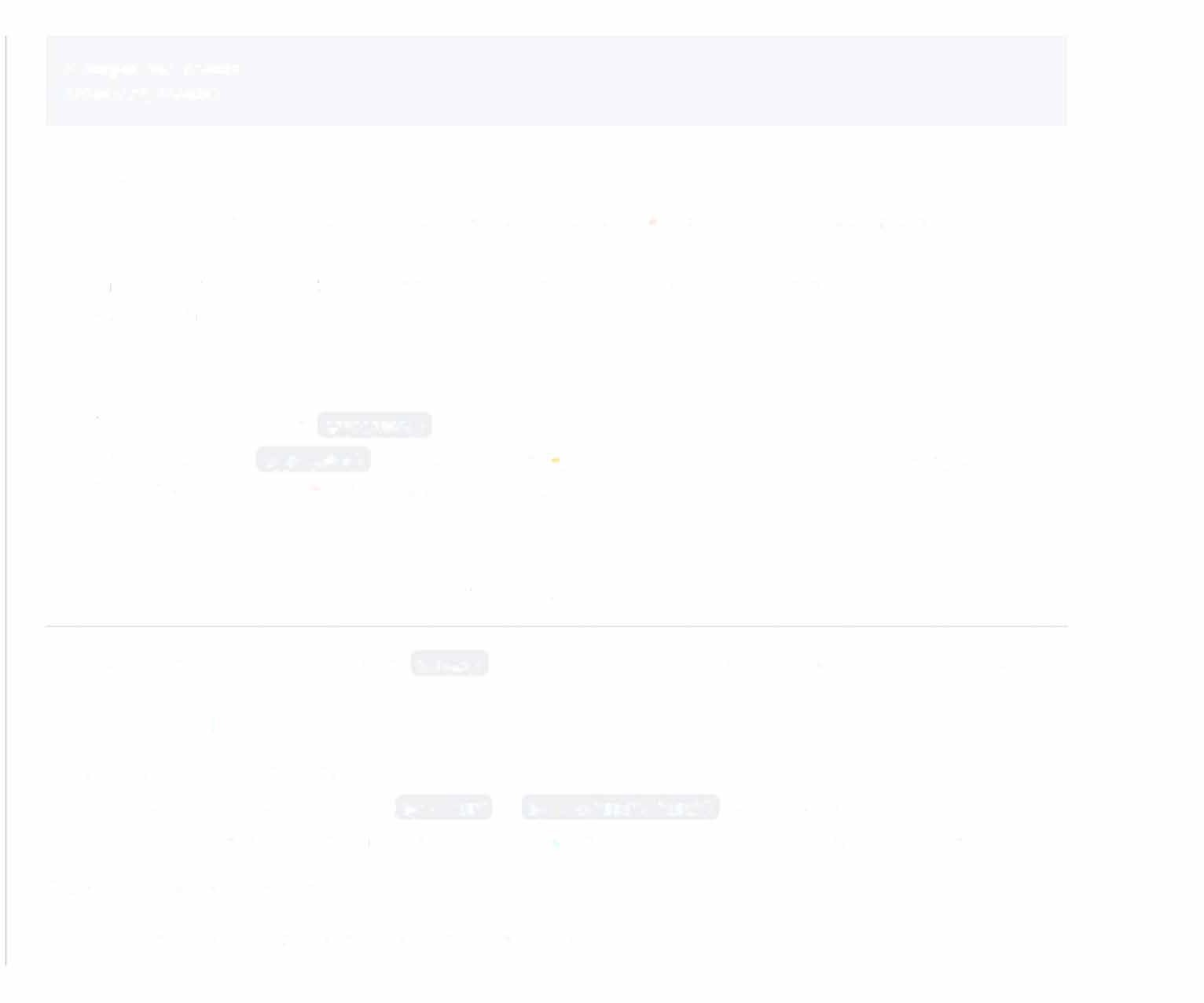
# Import dataset *c9*

data<· read.csv("path\_to\_csv\_file.csv")

# Check the structure of the dataset str(data)

# Call aggregate using formula notation

agg\_result <· aggregate(Sales ~Region + Product, data, sum)

# Output the result print(agg\_result)

# Advantages

* + Easy Grouping: Particularly beneficial for data types or sources that don't naturally support grouping mechanisms.
  + Transparency: The well-determined group-by structure makes it evident which records contribute to each aggregate value.

# Limitations

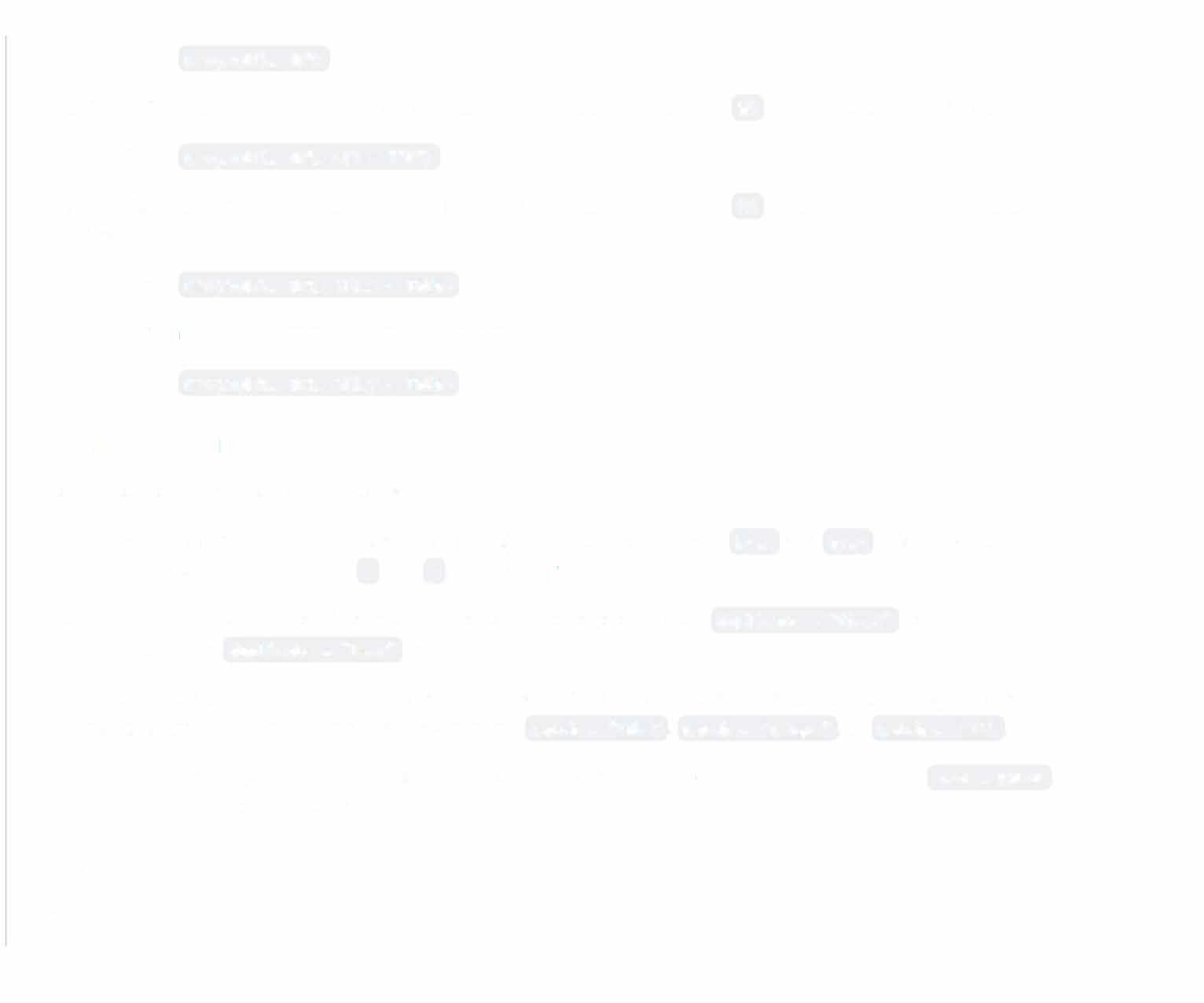
* + Verbosity: Sometimes, the aggregate() function demands more code than its modern equivalents.
  + Single Result Type: aggregate() can return only the a.ggregation result, making it more challenging to merge the aggregated statistics back into the original dataset.

1. **Explain how to *merge dataframes* in R.**

In R, you can merge dataframes using the merge() function, which combines datasets based on common columns.

# Common Merge Parameters

* + x, y: The dataframes to merge.
  + by: Specifies which columns to link ( by = "ID" or by = c("ID1", "ID2") for multiple columns).
  + all.x and all.y: Optional logical values to include all rows from the first and the second dataframe, respectively. The primary types of merge are:

1. Inner: Retains only matching rows from both dataframes.
   * Code: merge(df1, df2)
2. Outer: Retains all rows from both dataframes, filling in missing values with NA when data is not available.
   * Code: merge(df1, df2, all = TRUE)
3. Left: Retains all rows from the first dataframe, fi lling in missing values with NA when not available in the second dataframe.
   * Code: merge(df1, df2, all.x = TRUE)
4. Right: Retains all rows from the second dataframe.
   * Code: merge(df1, df2, all.y = TRUE)

# Advanced Merging

You can perform more complex merges as well:

* Using: You can rename columns before or during the merge. For instance, by.x and by.y allows you to rename columns in dataframes x and y , respectively:
* Duplicate Handling: Specify if duplicates should be included or not. Set duplicate = "first" to keep the first occurrence and duplicate = "last" to keep the last.
* Match: When conducting a merge, the default behavior is to look for exact matches. You can change this to partial or exact with regular expressions by setting match = "like", match = "unique" , or match = "all" .
* Sorting: Dataframes generally need to be sorted in merge columns. You can turn off sorting with sort = FALSE ,

but it might affect the merge results.

# Code Example: Merging Dataframes

Here is the R code:

# Create sample data frames

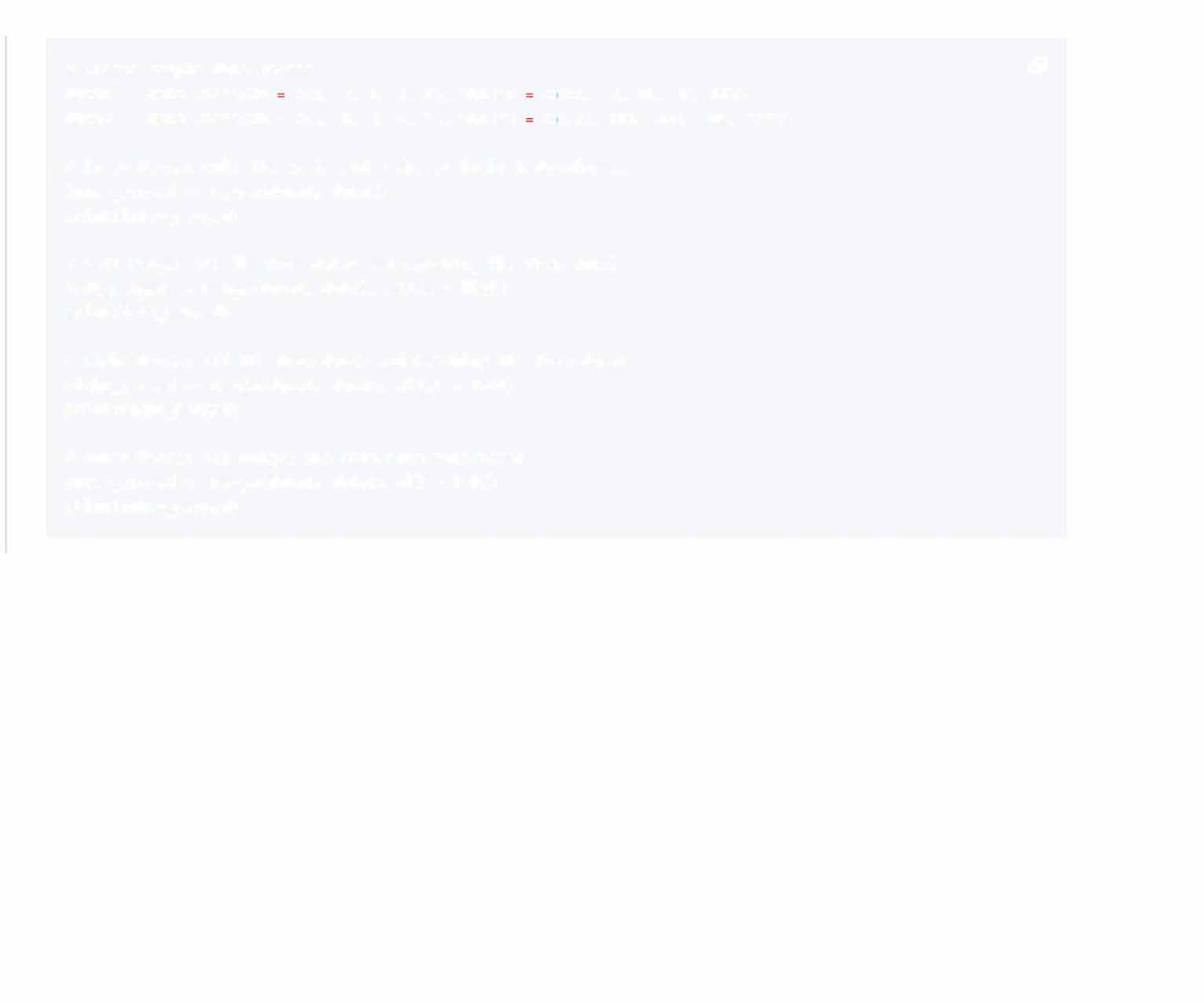
data1 <- data.frame(ID c(1, 2, 3, 4, 5), Value1

data2 <- data.frame(ID= c(2, 3, 4, 6, 7), Value2

*c9*

c{11, 22, 33, 44, 55))

c{222, 333, 444, 666, 777))



# Inner Merge: Only IDs 2, 3, and 4 appear in both dataframes inner\_merged <- merge(data1, data2}

print(inner\_merged}

# Left Merge: All IDs from data1 and matching IDs from data2 left\_merged <- merge(data1, data2, all.x = TRUE} print(left\_merged}

# Right Merge: All IDs from data2 and matching IDs from data1 right\_merged <- merge(data1, data2, all.y = TRUE) print(right\_merged}

# Outer Merge: All unique IDs from both dataframes outer\_merged <- merge(data1, data2, all= TRUE) print(outer\_merged}