Understanding and Applying Data in the Middle East & Africa

Week 02: R Coding Supplement

Introduction

R, a free and open-source software package, is widely recognised for its exceptional graphical capabilities and the comprehensive range of advanced data analysis techniques it supports. However, the foundational step in leveraging these capabilities involves data preparation, often presenting significant challenges. Data seldom comes in a ready-to-use format, necessitating initial efforts to import and format data effectively for analysis in R. This involves two primary hurdles: first, importing data into R in a recognisable format, and second, manipulating the data post-import to facilitate graphical representation and statistical analysis.

Objects, Variables, and Workspaces

Unlike the row-by-column data organisation typical in software like SPSS, SAS, and Stata, R distinguishes itself from other statistical packages through its highly flexible data model. In these packages, rows typically represent observational units (individuals, cases, etc.), while columns represent variables (attributes like height, income, etc.). Conversely, R handles data in various shapes and types, referring to these data blocks as 'objects'. Variables in R are essentially names assigned to these objects, enabling dynamic and complex data manipulation.

R's object-oriented approach is more aligned with programming languages like C, Fortran, Java, or Python, offering flexibility in how variables are associated with data types. A simple example of a variable assignment is a <- 1, where the value 1(one) is assigned to a variable named 'a'. R supports many object types, from simple numeric sequences to complex structures like estimated statistical models, including traditional rectangular data frames for compatibility with other statistical software.

Central to R's functionality is the concept of 'functions'. Functions are special objects that take input values (arguments) and produce output. For instance, calling the mean() function with a set of numbers returns their arithmetic mean. Almost every operation in R involves evaluating functions, even basic tasks like variable assignment or displaying variable contents.

Variables and their associations are organised into 'environments', with the global environment or 'workspace' being the primary focus for users, where user-defined variables reside. Environments are also temporarily created when functions are executed, or extension packages are activated. The workspace's contents can be managed using functions like **ls()** to list variables, **exists()** to check variable presence, and rm() to remove variables. This management ensures an organised and efficient R workflow, accommodating basic and advanced data manipulation tasks.

Key Concepts: Objects, Variables, Workspaces, and Functions

Definitions:

Object: Any standalone data element in R's main memory. It is independent and not nested within another data element.

Variable: A named reference in R's memory holding an object. Variable names can include any printable characters. However, if a name doesn't start with a period (.) or letter, or contains characters other than letters, digits, underscores (_), and periods, it must be enclosed in backticks () in R code (e.g., 12e` <- 1).

Function: A type of object designed to perform operations on inputs (arguments) and produce outputs (return values).

Environment: A structure that maps variable names to objects in R's memory, detailing their associations.

Package: A collection of pre-defined functions and variables that users can activate for use in their work.

Workspace: The default environment containing all user-defined variables.

Workspace Management and Package Activation Functions

print(): Displays an argument in the console. Typing a variable name directly is equivalent to calling **print()** with that variable.

library(): Activates a package, making its functions and variables accessible.

install.packages(): Installs packages for future activation with library().

ls(), objects(): Lists variables defined in the user's workspace or a specified environment.

rm(), remove(): Removes variables from the workspace or a specified environment.

Working with External Files and R Scripts:

Saving and Restoring Data:

Saving and restoring data and objects is crucial, especially for long-running analyses. To save variables a and b to a file, use save(a, b, file="somefile.RData"). Restore them with load("ab.RData").

R sessions can automatically prompt you to save the workspace on exit, storing it in a ".RData" file in the working directory. Unless specified otherwise with command-line options, this file is reloaded in the next session.

R Scripts for Efficient Workflows:

R scripts and text files containing R code streamline repetitive tasks and avoid manual code entry or GUI navigation. They end with .r or .R.

Execute a script with source("filename.R"). For verbose execution, displaying each step, use source("filename.R", echo=TRUE).

GUIs like RStudio enhance script editing and execution. However, for full utility, scripts should be written to run entirely without manual intervention, following good practices like proper order, defining variables before use, syntactical correctness, and clear commenting for complex expressions.

Files, Paths, and Working Directories in R

Understanding File Paths and Locations

When using functions like **load(), save(),** or **source()** in R, the most crucial argument is the filename or the path to a file. These functions depend on correct file paths to locate and interact with files on your computer's hard disk. Here's how R finds and works with these files:

File System Hierarchy: Modern computers organize files in a hierarchical structure. In Unix-like systems (Linux, macOS, etc.), this starts from a root directory ("/") and extends to subdirectories. A file path describes the exact location of a file within this hierarchy, like "/home/tyrion/research/analysis-1.R", indicating a file located in a series of nested directories.

Windows File Paths: Windows systems use a slightly different approach, with file hierarchies beginning with a drive letter (e.g., "C:\"). Backslashes ("") are used in paths, but since R uses backslashes for special characters, they must be doubled (e.g., "C:\Users\tyrion\research\analysis-1.R"). Modern R versions also support forward slashes ("/") for easier cross-platform compatibility.

Tilde Expansion: R simplifies path specification through tilde expansion, where " represents the user's home directory on both Unix-like and Windows systems. This allows for consistent paths across operating systems, like "/research/analysis-1.R".

Relative vs. Absolute Paths

Using relative paths instead of absolute ones is generally recommended for better portability and simplicity. An absolute path begins from the root directory (or a drive letter in Windows), whereas a relative path is interpreted from the current working directory.

Setting and Getting the Working Directory: R allows you to change the current working directory with setwd() and check it with getwd(). This flexibility helps in working with relative paths, making your scripts more adaptable and easier to use across different environments.

Key Functions for File Management

Saving and Loading Variables: save() function saves specified variables to a file, while load() restores them. The save.image() function saves the entire workspace.

Working with RDS Files: saveRDS() and readRDS() are used for saving and loading single R objects, offering a more granular control compared to the broader save() and load() functions.

Executing R Scripts: source() reads and evaluates expressions from an R script file. It can be run silently or verbosely, echoing evaluated expressions and their results to the console for debugging or demonstration purposes.

Best Practices

Use Scripts for Reproducibility: R scripts (.R files) are essential for collecting function definitions, preparing data, and conducting analyses. They ensure your work is reproducible, allowing you to run complex sequences of commands with a single action.

Comment and Organize Your Code: Including comments in your scripts and breaking complex expressions into multiple lines improves readability and maintainability, especially for long scripts or when revisiting code after some time.

This revised explanation aims to make the original concepts more accessible by clarifying how R interacts with the file system, the importance of path specifications, and the utility of key functions for managing files and workspaces in R.

Building Blocks of Data in R

R provides a wide range of data structures tailored for various analytical needs. At the core of these structures are fundamental data types and basic manipulation techniques, essential for creating and processing data effectively.

Basic Data Types in R

Numeric Vectors

R, designed with statistical analysis in mind, primarily operates with numeric vectors rather than single numbers. Numeric vectors are simply sequences of numbers. For example, creating a numeric vector can be done using the concatenate function **c()**, like so: **c(1.2, 3.5, 5.0, 6.7, 1.09e-3)**. This command creates a sequence of numbers, and the notation **[1]** before the sequence indicates the first element of the sequence.

The **length()** function can be used to find out how many elements a vector has. Vectors can vary in length, with the only limitation being the computer's memory.

Another way to create a numeric vector is through the sequence operator **:**, which generates a sequence of numbers, for example, **1:100** for integers from 1 to 100.

Numeric vectors support arithmetic operations like addition, multiplication, subtraction, and division, applied element-wise when vectors of the same length are involved. Operations with vectors of differing lengths lead to the "recycling" of the shorter vector to match the length of the longer one.

Handling missing data is crucial in R, marked with **NA** for missing values. Arithmetic operations involving **NA** result in **NA**, highlighting the importance of dealing with missing information carefully.

Logical Vectors

Logical vectors, representing true or false values, are the outcome of applying comparison operators (**==**, **!=**, **>**, **<**, **>=**, **<=**) to data points. For example, checking which voters are old enough to have voted in the previous election might result in a logical vector of **TRUE** or **FALSE** values.

Logical operations include the 'and' (**&**), 'or' (**|**), and 'not' (**!**) operators, allowing for complex conditions to be evaluated. Logical vectors can also handle missing values (**NA**), with logical operators yielding sensible results even when missing values are involved.

Manipulating Vector Data

Extracting, replacing, and reordering elements within vectors are fundamental skills in data manipulation in R. These operations allow for detailed and precise handling of data, essential for effective analysis and interpretation.

Working with Missing Data and Special Values

R distinguishes itself by how it handles special values like **NA** for missing data, **Inf** for infinity resulting from division by zero, and **NaN** for results undefined in mathematical terms. This nuanced approach to special values enables more precise control and understanding of data, particularly in edge cases.

Summary

Understanding and manipulating the basic data types in R—numeric vectors, logical vectors, and handling special values—are foundational skills for anyone working with data in R. These concepts form the basis for more complex data structures and analytical techniques available in R, catering to a wide range of applications and research needs.

Character Vectors in R

In R, character vectors come into play when we deal with data that cannot be easily or meaningfully converted into numbers, such as country names in a comparative study. Unlike what the term might suggest, character vectors contain strings (sequences of characters) rather than individual characters. These strings, represented within quotation marks in R, can encapsulate anything from letters and digits to symbols, with quotation marks serving purely as syntactical delimiters.

For instance, creating a character vector to represent the Beatles would look like this:

```
Beatles <- c("John", "Paul", "George", "Ringo")
print(Beatles)
```

This code snippet would output the names of the Beatles, each a string within the character vector **Beatles.**

While arithmetic or logical operations don't apply to character vectors, R provides functions tailored for string manipulation. For example, concatenating strings is made easy with the **paste()** function:

```
# Concatenating strings with a space
print(paste("one", "and", "only"))
```

```
# Collapsing a vector into a single string separated by custom delimiters
print(paste(Beatles, collapse=" & "))
```

The collapse= argument in paste() combines elements of a character vector into a single string, while sep= can alter the separator between elements when combining multiple vectors.

Extracting parts of strings is possible with substr(), which allows for the specification of start and end positions within the strings:

```
# Extracting the first two characters of each name
print(substr(Beatles, 1, 2))
```

Moreover, character vectors can be named, making it easier to document and reference data meaningfully. The names() function is used both to retrieve and to assign names to the elements of a vector:

```
# Assigning and printing named elements of a numeric vector
onetofour <- 1:4
names(onetofour) <- c("first", "second", "third", "fourth")
print(onetofour)
```

This assigns descriptive names to each element of the vector onetofour, which R prints instead of the default index numbers, facilitating clearer data presentation and interpretation.

Factors in R

Factors are R's solution for handling categorical data, such as survey responses, political affiliations, or social classes. These data can be nominal (unordered) or ordinal (ordered), with factors and ordered factors in R designed to accommodate these distinctions.

A factor's unique characteristic is its set of levels, which represent the different categories within the data. These levels are internally encoded as integers but are typically associated with more descriptive labels for clarity.

To illustrate, consider creating a factor to represent levels of life satisfaction among survey respondents:

```
# Creating an ordered factor for life satisfaction levels
satisfaction <- sample(1:4, size=20, replace=TRUE)
satisfaction <- ordered(satisfaction, levels=1:4, labels=c("not at all", "low", "medium",
"high"))
print(satisfaction)
```

This creates an ordered factor satisfaction with labels that are more informative than mere numbers, aiding in the interpretation of the data.

For unordered categorical data, such as the origins of a sample of people from the UK, factors can be similarly constructed and manipulated:

```
# Creating an unordered factor for countries
country.orig <- sample(c("England", "Northern Ireland", "Scotland", "Wales"), size=50,
replace=TRUE)
country <- factor(country.orig)
print(table(country))
```

This example showcases how factors efficiently manage categorical data, providing a structured approach to handling such variables in R.

Factors are foundational in statistical modeling within R, offering a way to work with categorical variables that reflect the complexity and nuances of real-world data. Their use spans across various types of analyses, from linear regression to more complex statistical models, underlining the importance of mastering factors for effective data analysis in R.

Key Concepts in Handling Data with R

R supports a variety of data structures, such as vectors and factors, to accommodate different data types and analytical needs. This section outlines the key concepts related to these structures and provides an overview of important functions for managing vectors and factors in R.

Vectors in R

Numeric Vector: Represents sequences of numbers. Numeric vectors are fundamental in R for storing numerical data and can interact with arithmetic operators (e.g., +, *), arithmetic functions (e.g., exp(), log()), and statistical functions (e.g., mean(), var()).

Logical Vector: Contains logical values (TRUE or FALSE). These vectors can result from comparisons (e.g., a == 0) or be used with logical operators (& for "and", | for "or").

Character Vector: A sequence of character strings used to represent textual data. Beyond storing text, character vectors play a crucial role in documenting data by naming elements of vectors and other structures.

Factors

Factor: Designed to handle categorical data, factors have a finite set of "levels" that denote categories. Factors can be ordered or unordered, influencing how they are processed in statistical modeling.

Special Values

Missing and Non-finite Values: R uses NA to denote missing values. For infinite values, Inf and -Inf are used, while NaN represents undefined arithmetic results. These special values can appear in numeric, logical, and character vectors.

Recycling Rule

When applying operations on vectors of different lengths, R employs the recycling rule. This means the shorter vector's elements are repeated to match the length of the longer vector, allowing operations to proceed.

Important Functions for Vectors and Factors

length(): Determines the number of elements in a vector or factor. Changing a vector's length can introduce NA values for new elements.

names(): Manages names attached to vector elements, allowing for clearer documentation and easier data manipulation.

levels(): For factors, this function reveals the names of different categories (levels).

nlevels(): Reports the total number of levels in a factor.

is.finite(): Identifies finite and non-missing elements in a vector.

is.na(): Checks for NA or NaN values in a vector.

Basic Data Manipulations

Extracting and Replacing Elements: Using the bracket operator [], elements of vectors can be selected or replaced based on numeric, logical, or character indices.

Reordering Elements: Functions like sort() and order() help organize vector elements in ascending or descending order.

Creating Sequences and Repetitions: The colon operator :, seq(), and rep() functions generate regular sequences and repeat elements or sequences.

Sampling: The sample() function draws random samples from vectors, supporting both replacement and specifying sample sizes.

Constructing Complex Data Types

Beyond the fundamental types (numeric, logical, character vectors, and factors), R allows for more complex structures through lists and attributes, enhancing data handling capabilities for diverse analytical tasks. These mechanisms support the creation of sophisticated data types, serving as a foundation for advanced data management and analysis in R.

Understanding Lists in R

In R, the three primary data types—numeric, logical, and character vectors—consist of elements of a single type. These data types are considered 'atomic' because their elements cannot exist outside of them in R's memory. In contrast, lists in R can hold objects of various types, including other lists, making them 'recursive' data types. Like vectors, lists have a defined length and order to their elements, and both element extraction and naming are possible.

Creating and Working with Lists

A list is a versatile container that can include different data types:

```
# Creating a list with various data types
AList <- list(1:5, letters[1:6], c(TRUE, FALSE, FALSE, TRUE))
print(AList)
```

This list contains a numeric vector, a character vector, and a logical vector, showcasing the flexibility of lists in handling complex data structures.

Accessing List Elements

Lists differ from atomic vectors in how you access their elements:

Single-bracket operator []: Returns a subset of the list still as a list, even if it's just one element.

Double-bracket operator [[]]: Extracts individual elements, which can be of any data type, including other lists.

Naming and Manipulating Lists

You can assign names to list elements for easier access and readability:

```
# Naming list elements
names(AList) <- c("Numbers", "Letters", "Booleans")
print(AList)
```

Lists can represent complex data objects, including data sets:

```
# Example of a list representing a data set
UK <- list(
  country.name = c("England", "Northern Ireland", "Scotland", "Wales"),
  population = c(54786300, 1851600, 5373000, 3099100),
  area.sq.km = c(130279, 13562, 77933, 20735),
  GVA.cap = c(26159, 18584, 23685, 18002)
)
print(UK)
```

This list combines various types of information into a structured data object, similar to a data frame but with more flexibility in the types of data it can contain.

Attributes: Extending Data Structures

Attributes are additional information attached to R objects, such as vectors or lists, enhancing their functionality. These can include element names or other metadata:

```
# Creating a named vector with attributes
onetofour <- c(first=1, second=2, third=3, fourth=4)
print(attributes(onetofour))
```

Attributes are crucial for factors and for defining the 'class' of an object, affecting how functions interact with it.

Object Classes and Method Dispatch

R functions can behave differently based on the 'class' of their arguments. This behavior is controlled through 'S3' or 'S4' method dispatch systems, allowing for custom function behaviors for different data types:

```
# Using a generic function
print(satisfaction) # Prints based on the 'class' of the satisfaction object
```

Generic functions and class-specific methods enable R to be adaptable and extendible for various data types and analytical methods.

Summary

Lists and attributes in R allow for the construction of complex, flexible data structures, while the system of object classes and method dispatch enables tailored functionalities across different types of data. These features make R a powerful tool for data analysis, accommodating a wide range of data management and analytical needs.