Syllabus :: Module -1 (Introduction to R)

The R Environment - Command Line Interface and Batch processing, R Packages, Variables, Data Types, Vectors- vector operations and factor vectors, List- operations, Data Frames, Matrices and arrays, Control Statements- Branching and looping - For loops, While loops, Controlling loops. Functions- Function as arguments, Named arguments

---

Overview of Textbook Contents Aligned with the Syllabus [Module 1]

Based on the syllabus the first module, titled "Introduction to R", aims to introduce learners to the R programming environment and fundamental concepts. This module covers the following key areas:

- **The R Environment**: This includes understanding the **Command Line Interface** and **Batch processing** in R. It also introduces **R Packages**, which are collections of functions and data sets that extend the capabilities of base R. According to "R in a Nutshell 2nd edition", R is a software environment for statistical computing and graphics, encompassing a language, an interpreter, a graphics system, and applications for different operating systems. Chapter 1 of "R in a Nutshell 2nd edition" discusses getting and installing R, while Chapter 2 provides an overview of the R user interface. The R console is an interactive environment where expressions are entered and evaluated.

- **Variables**: Module 1 covers the concept of **variables** in R. In R, a variable is a symbol to which a value or an object is assigned. According to "R in a Nutshell 2nd edition", you can assign values to variables using the assignment operator <-. Variable names represent variable names.

- **Data Types**: This module introduces fundamental **data types** in R. While the syllabus doesn't explicitly list them, "R in a Nutshell 2nd edition" details various data types such as numeric vectors, character vectors, logical values (TRUE, FALSE), NA (Not Available for missing values), Inf and -Inf (infinity), NaN (Not a Number), and NULL (representing an empty object).

- **Vectors**: A significant part of the first module focuses on **vectors**, including **vector operations** and **factor vectors**. In R, a vector is a one-dimensional array that can hold numeric, character, or logical values. "R in a Nutshell 2nd edition" explains how to create and manipulate vectors, including performing operations on them. Factor vectors are used

to represent categorical data.

- **List**: The module also covers **lists** and their **operations**. A list in R is a versatile data structure that can contain elements of different types, such as vectors, other lists, data frames, and functions.

- **Data Frames**: **Data frames** are another key data structure introduced in this module. A data frame is a tabular data structure with rows and columns, where each column can have a different data type. They are fundamental for data analysis in R.

- **Matrices and arrays**: Module 1 also introduces **matrices** (two-dimensional arrays) and multi-dimensional **arrays** in R. "R in a Nutshell 2nd edition" provides examples of creating and accessing elements in arrays.

- **Control Statements**: This module covers **control statements**, specifically **branching** (like if and else statements) and **looping** constructs such as **For loops**, **While loops**, and how to control the execution of loops. "R in a Nutshell 2nd edition" details conditional statements and loops as control structures in R syntax.

- **Functions**: Finally, the first module introduces **functions**, including how to use **functions as arguments** to other functions and the concept of **named arguments**. In R, a function is an object that performs a specific task. You can define your own functions and use built-in functions. Functions can take other functions as arguments, and you can specify arguments by their names when calling a function.

R Environment

Command Line Interface and Batch processing

R Packages

Module 1 of the "AIT 362 Programming in R" course, as outlined in the syllabus, focuses on introducing the "R Environment". According to "R in a Nutshell 2nd edition", **R is a software environment for statistical computing and graphics**. This environment encompasses a computer language, an interpreter that executes the code, a graphics system, and applications that run on various operating systems.

The syllabus specifically mentions the following aspects of the R Environment that are covered in Module 1:

- **Command Line Interface and Batch processing**:
    - The **R Console** is described as the most crucial tool for interacting with R. It allows users to type commands (expressions) and see the R system's responses. The R interpreter reads these expressions and provides a result or an error message. When R is ready for a command, it displays a prompt, typically ">".
    - "R in a Nutshell 2nd edition" details how to get started with the R console on different operating systems like Windows, Mac OS X, and Linux. It also mentions that on Linux, R can start directly on the command line.
    - The text also briefly touches upon **Batch Mode**, indicating it as another way to run R. In batch mode, R executes commands from a script without interactive input. You can typically invoke batch mode from the command line using a command like $ R CMD BATCH my_script.R (this specific command isn't in the source, but it's a common way to run R in batch mode, and you may want to independently verify this information).
- **R Packages**:
    - The syllabus highlights "R Packages" as a key component of the R environment. "R in a Nutshell 2nd edition" explains that **a package is a related collection of functions, help files, and data files bundled together**. They are similar to modules in Perl or libraries in C/C++.
    - To utilize a package, it first needs to be installed into a local library. By default, R uses one system-level library, but you can add others. After installation, the package must be **loaded** into the current R session to use its functions. The library() function is used for this purpose. On Windows and Linux, you can also load packages via the GUI menu. On Mac OS X, the "Package Manager" provides a user interface for managing packages, including loading them and browsing their help files.
    - R has a vast number of available packages for various tasks, including graphics, statistical tests, and machine learning. Some packages are included with the base R installation and are loaded by default or can be loaded when needed. Examples of base packages include datasets, utils, grDevices, graphics, stats, and methods. The base package, which implements fundamental features of the R language, is always loaded.
    - Other packages are available from public repositories like CRAN (Comprehensive R Archive Network). Inside R, you can use functions like install.packages() to find and install packages from these repositories. When

installing for the first time in a session, R might ask you to choose a CRAN mirror.
    - It's also possible to create **custom packages** by organizing R code, data, and documentation in a specific directory structure. The package.skeleton() function can help create the basic structure for a new package. Building and checking packages can be done using command-line tools like R CMD check and R CMD build.

In summary, Module 1 introduces the learner to the R environment as a powerful platform for statistical analysis and visualization, emphasizing how to interact with R through the command line interface (R console and batch processing) and how to extend R's capabilities using R packages.

Variables in R

Variables are a fundamental concept in R programming. In R, a **variable is a symbol to which a value or an object is assigned**. You use variables to store and refer to data within your R programs.

Here's a detailed breakdown of R variables based on the sources:

- **Assignment:**
    - Values are assigned to variables using the **assignment operator <-**. For example, x <- 1 assigns the numeric value 1 to the variable named x.
    - "R in a Nutshell 2nd edition" notes that the assignment operator is usually pronounced as "gets".
    - You can also use = for assignment in many contexts [You may want to independently verify this as it is common in R but not explicitly stated in the provided sources].
    - It's mentioned that in some programming contexts, an alternative assignment using -> might help write clearer code, where the value is placed on the left and the variable on the right (e.g., 4 -> y).
- **Variable Names as Symbols:**
    - In R, variable names are referred to as **symbols**.

    - Symbols can consist of letters, numbers, and the dot (.) or underscore (_) characters [You may want to independently verify this as it's a common R rule not explicitly stated in the provided sources].
    - **Case matters** in R; x1 and X1 are different variables.
    - Some symbols contain special syntax and to refer to these objects, you enclose them in **backquotes (\)**. For example, to get help on the assignment operator, you would use ?<-. You could even define a symbol with special characters using backquotes, like 1+2=3 <- "hello" ``.
    - However, **not all words are valid as symbols**; some words are reserved in R, such as if, else, function, for, TRUE, FALSE, NULL, Inf, NaN, NA, etc..
- **Variables as Objects:**
    - Like most other languages, R allows you to assign values to variables and refer to them by name.
    - Crucially, in R, **a variable points to an object**. Everything in R is an object. This means that a variable can hold not just simple data values (like numbers or characters) but also more complex data structures such as vectors, lists, data frames, and even functions.
    - For example, b <- c(1, 2) creates a vector object c(1, 2) and assigns it to the variable b.
    - Similarly, f <- function(x,y) {c(x+1, y+1)} defines a function object and assigns it to the variable f.
- **Scope and Environments:**
    - The sources briefly mention that there are subtleties to how variables are evaluated and refer to **variable scope and environments**, which are discussed in more detail in Chapter 8 of "R in a Nutshell 2nd edition". Environments define the context in which variables exist and can be accessed. The global environment is the default workspace where variables you define directly are stored.
- **Variables in Functions and Formulas:**
    - When you define a function, the arguments listed within the function() definition are symbol names that will become variables within the body of the function.
    - In **formulas**, which are a special type of object in R used to describe relationships between variables (often in statistical modeling and graphics), variable names are used to specify the dependent and independent variables. For example, y ~ x1 + x2 indicates that the variable y is modeled as a function of the variables x1 and x2.

In summary, R variables are symbolic names that hold references to objects. They are created through assignment, are case-sensitive, and adhere to certain naming rules. The fact that

variables can hold various types of objects is a key characteristic of R and contributes to its flexibility in data analysis and programming.

Data Types in R [Note : Data Structures also included]

In R, all code manipulates **objects**, which are represented as "things" by the computer. Every object in R has a **type** that defines how it is stored, and each object is also a member of a **class** that defines what information it contains and how it can be used.

Here's an overview of the main aspects of data types in R, according to the sources:

- **Primitive Object Types (Basic Vectors)**: These are vectors that contain a single type of value. The sources identify the following basic vector types:
    - **integer**: Naturally produced from sequences and can be coerced using the integer() function. For example, typeof(1:1) returns "integer".
    - **numeric** (often double-precision floating-point numbers): This is the default type for numbers in R. You can combine arbitrary sets of numbers into a numeric vector using the c() function. typeof(1) returns "double" by default.
    - **complex**: Used for complex numbers.
    - **character**: Vectors containing text. They are created using quotes, e.g., "Hello world.". This is called a character vector in R.
    - **logical**: Represents Boolean values, either TRUE or FALSE.
    - **raw**: A vector containing raw bytes, useful for encoding objects from outside the R environment.
- The c() function can combine elements into a vector. If you combine elements of different types, R will attempt to **coerce** them to a single type. The ordering of coercion is roughly: logical < integer < numeric < complex < character < list.

- You can check the type of an object using the typeof() function and its class using the class() function.

- **Compound Objects**: These objects are containers for basic vectors and other objects.
    - **list**: A (possibly heterogeneous) collection of other objects. Elements of a list can be named, and lists can even contain other lists. Data frames are implemented as lists.
    - **pairlist**: A data structure used to represent a set of name-value pairs, primarily used internally. Standard list objects are generally more efficient and flexible for user-level programs.
    - **S4**: An R object supporting modern object-oriented paradigms like inheritance and methods. You define a class with setClass() and a method with setMethod().
    - **environment**: Describes the set of symbols available in a specific context. It contains symbol-value pairs and a pointer to an enclosing environment.
- **Special Objects**: These objects serve specific purposes in R programming.
    - **any**: Used in specific contexts.
    - **NULL**: Represents a null object, often used as an argument in functions to mean no value was assigned or as a return value from some functions. NULL is not the same as NA.
    - **...**: Used to represent a variable number of arguments in a function.
- **R Language Objects**: These objects represent R code.
    - **expressions**: Represent unevaluated R code.
    - **calls**: Represent a function call. Every expression in R can be rewritten as a function call.
    - **functions**: Objects in R that take input objects (arguments) and return an output object. All work in R is done by functions. You can define your own functions in R.
- **Other Important Data Structures**: While the above are considered primitive and compound object types, several other structures are crucial for data handling in R.
    - **matrices**: Extensions of vectors to two dimensions, used to represent two-dimensional data of a single type. They can be created with the matrix() function. The underlying storage for a matrix is a vector.
    - **arrays**: Extensions of vectors to more than two dimensions, used for multidimensional data of a single type, created with the array() function. The underlying storage for an array is also a vector.

- **factors**: Used to compactly represent vectors of categorical values. They are ordered collections of items, and the distinct values are called levels. You can create a factor using the factor() function.
- **data frames**: Tabular data structures with columns that can be of different types but must have the same length (number of rows). They are implemented as lists with the class "data.frame". You can create them using data.frame().
- **formulas**: A special notation in R to describe relationships between variables, often used in statistical modeling and graphics. For example, y ~ x1 + x2 is a formula object.
- **time series**: Sequences of measurements of a quantity over time, typically taken at equally spaced intervals. R has a specific ts class for this.
- **dates and times**: R includes classes like Date (for dates only) and POSIXct/POSIXlt (for dates and times) to represent temporal data.
- **connections**: Objects used for receiving data from or sending data to applications or files outside the R environment, like files or URLs.

Understanding these data types is fundamental to working with R effectively. The str() function can be used to compactly display the internal structure of an R object, giving information about its type and structure.

Vectors- Vector Operations and Factor Vectors

Vectors in R

In R, a **vector** is a fundamental data structure that contains an ordered sequence of elements of the **same basic type**. According to "R in a Nutshell," these basic vector types are also referred to as primitive object types. The syllabus also lists "Vectors- vector operations and factor vectors" as a key topic in Module 1.

The sources identify the following basic vector types:

- **integer**: Vectors containing whole numbers. They are naturally produced from sequences (e.g., 1:5) and can be explicitly created or coerced using the integer() function.
- **numeric** (often double-precision floating-point numbers): The default type for numbers in R. You can create numeric vectors using functions like c(). For example, c(0, 1, 1, 2, 3, 5, 8) creates a numeric vector.
- **complex**: Vectors for representing complex numbers.
- **character**: Vectors containing text, also known as character arrays. They are created using quotes, e.g., "hello".

- **logical**: Vectors holding Boolean values, TRUE or FALSE.
- **raw**: Vectors containing raw bytes, useful for low-level operations.

You can create vectors using the c() function, which combines its arguments into a vector. For example, v <- c(.295, .300, .250, .287, .215) creates a numeric vector. The c() function will **coerce** all its arguments to a single type if they are different. The typical order of coercion is logical < integer < numeric < complex < character < list [mentioned in the more detailed explanation of data types].

Other ways to create vectors include the sequence operator : (e.g., 1:10 creates a vector of integers from 1 to 10) and the seq() function, which provides more flexibility in specifying the sequence (e.g., seq(from=5, to=25, by=5)). You can also manipulate the length of a vector using the length() attribute.

Vector Operations

R supports various operations on vectors, including:

- **Arithmetic Operators**: Standard arithmetic operators like + (addition), - (subtraction), * (multiplication), / (division), %% (modulus), %/% (integer division), and ^ (exponentiation) work element-wise on vectors. If you apply these operators to two vectors of the same length, the operation is performed on corresponding elements. If the vectors have different lengths, R applies the shorter vector by recycling its elements (with a warning if the length of the longer vector is not a multiple of the shorter vector's length).
- **Logical Operators**: Logical operators like < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to), == (equal to), != (not equal to), ! (negation), & (element-wise AND), && (logical AND), | (element-wise OR), and || (logical OR) also works element-wise on vectors, returning a logical vector of the same length (or the length of the longer vector if recycling occurs).
- **Indexing**: You can access individual elements or subsets of a vector using square brackets [].
    - **Integer indexing**: You can specify the index (position) of the elements you want to retrieve using an integer vector. For example, if b <- c(1, 2, 3, 4, 5), b will return 3, and b[c(1, 3)] will return c(1, 3). You can also use negative integer indices to exclude elements (e.g., b[-2] will return all elements except the second). You can fetch items in any order and even repeat indices.
    - **Logical indexing**: You can use a logical vector of the same length as the original vector inside the square brackets. R will return the elements at the positions where the logical vector has TRUE. For example, b[b %% 2 == 0] would return the even numbers in the vector b.

Factor Vectors

A **factor** is a special type of vector used to represent **categorical data**. According to "R in a Nutshell," a factor is an ordered collection of items, and the distinct values that the factor can take are called **levels**. The syllabus also mentions "factor vectors".

Factors are particularly useful for representing data with a fixed set of possible values, such as eye color (brown, blue, green) or treatment groups (control, treatment A, treatment B). While you could represent these as character vectors, factors can be more efficient in terms of storage and are often required by statistical modeling functions.

You can create a factor from another vector (typically a character vector) using the factor() function. For example:

```
eye.colors <- c("brown", "blue", "blue", "green", "brown", "brown", "brown")

eye.colors.factor <- factor(eye.colors)

eye.colors.factor
```

The output would show the levels of the factor: "blue" "blue" "green" "brown" (the order might vary depending on the input). The underlying representation of the factor is a vector of integers corresponding to the levels, along with a separate attribute storing the actual levels.

Key aspects of factor vectors:

- **Levels**: The unique categories present in the data. You can access the levels of a factor using the levels() function.
- **Order**: Factors can be either unordered (nominal categories) or ordered (ordinal categories, where there is a meaningful order among the levels). You can specify ordered factors using the ordered = TRUE argument in the factor() function and defining the order of levels.
- **Efficiency**: For categorical data with many repetitions of the same values, factors can be more memory-efficient than character vectors.
- **Statistical Modeling**: Many statistical functions in R automatically treat factor variables appropriately for analysis (e.g., creating dummy variables for regression models).

In a data frame, columns containing categorical data are often stored as factors.

In summary, vectors are the basic building blocks for storing sequences of data of the same type in R, supporting element-wise operations and flexible indexing. Factor vectors are a specialized

type of vector designed for representing categorical data efficiently and are crucial for statistical analysis in R.

List- operations in R

What are Lists in R?

In R, a **list** is a versatile data structure that holds an ordered collection of objects. Unlike vectors, which must contain elements of the same basic type, **lists can contain a heterogeneous selection of objects**. This means a single list can include numbers, character strings, vectors, other lists, data frames, functions, and more. According to "R in a Nutshell," lists are a built-in data type for mixing objects of different types. The syllabus for "Programming in R" also identifies "List- operations" as a key data type.

Key characteristics of R lists:

- **Ordered Collection**: The elements in a list have a specific order.
- **Heterogeneous Data**: Lists can store objects of different data types within the same list.
- **Named Components**: Each element in a list can be given a name.

Creating Lists

You can create a list using the list() function. You simply provide the objects you want to include in the list as arguments to the function. You can also assign names to the elements during creation using the name = object syntax.

```
# Creating a list with different data types

my_list <- list(name = "Alice", age = 30, grades = c(85, 92, 78), is_student = TRUE)

my_list

# Creating a list without names

another_list <- list(1, "hello", 1:5)

another_list
```

List Operations

R provides several ways to manipulate and access elements within a list. The "Programming in R" syllabus explicitly mentions "List- operations", and a question in the model paper asks about general list operations.

1. Accessing List Elements

You can access elements in a list using several methods:

- **By Index (Position)**: Use single square brackets [] or double square brackets [[]] along with the index of the element.
    - list[n]: Returns a **sublist** containing the nth element.

my_list # Returns a list containing "Alice"

my_list[2:3] # Returns a list containing "age" and "grades"

-
    - list[[n]]: Returns the **content** of the nth element. This is often what you want when working with individual elements.

my_list[] # Returns "Alice" (a character string)

my_list[] # Returns the vector c(85, 92, 78)

By Name: If the elements in the list have names, you can access them using the $ operator followed by the name of the element.
my_list$name # Returns "Alice"

my_list$grades # Returns the vector c(85, 92, 78)

- You can also use double square brackets with the name as a character string: list[["name"]].

Partial Matching by Name: With [[ ]], you can use the exact = FALSE option to allow partial matching of names.
dairy <- list(milk = "1 gallon", butter = "1 pound")

dairy[["mil", exact = FALSE]] # Returns "1 gallon"

2. Modifying List Elements

You can modify elements in a list by assigning new values using the same access methods:

my_list$age <- 31 # Change the "age" element

```
my_list[] <- FALSE    # Change the fourth element

my_list[["grades"]] <- 90 # Modify an element within a list component

my_list
```

3. Adding and Removing List Elements

Adding: You can add new elements to a list by assigning a value to a new name or index:

```
 my_list$city <- "New York"  # Add a new element with name "city"

my_list[] <- "another item" # Add a new element at the end

my_list
```

Removing: You can remove elements by assigning NULL to them:

```
 my_list$is_student <- NULL

my_list[] <- NULL

my_list
```

4. Combining Lists

You can combine multiple lists into a single list using the c() function.

```
list1 <- list(a = 1, b = 2)

list2 <- list(c = "three", d = "four")

combined_list <- c(list1, list2)

combined_list
```

5. Applying Functions to Lists

R provides functions to apply operations to each element of a list.

lapply(): This function takes a list and a function as arguments and returns a **list** where each element is the result of applying the function to the corresponding element of the input list.

```
 numbers <- list(1, 2, 3, 4)

squared_numbers <- lapply(numbers, function(x) x^2)

squared_numbers
```

sapply(): This function works similarly to lapply() but attempts to simplify the output to a vector or matrix if possible.

```
numbers <- list(1, 2, 3, 4)

squared_numbers_vector <- sapply(numbers, function(x) x^2)

squared_numbers_vector
```

- **rapply()**: This is a recursive version of lapply(), useful for applying a function to elements in nested lists.
- **mapply()**: This is a multivariate version of sapply(), applying a function to the corresponding elements of multiple lists.

6. Other List Operations

- **length()**: Returns the number of elements in a list.
- **names()**: Retrieves or sets the names of the elements in a list.
- **unlist()**: Simplifies a list structure to produce a vector containing all the atomic components of the list. Be cautious as this can only be done if the elements are of compatible data types for coercion.
- **str()**: Compactly displays the internal structure of a list, which is very useful for understanding its components and their types.

Lists and Data Frames

It's important to note that **data frames in R are implemented as lists** with the class attribute set to "data.frame". This means that many of the operations applicable to lists, especially accessing elements using $ and [[ ]], also work for data frames.

In summary, lists are a fundamental and flexible data structure in R for storing collections of diverse objects. Understanding how to create and manipulate lists using these various operations is crucial for effective data handling and programming in R.

Data Frames in R

Definition and Purpose

In R, a **data frame** is a fundamental data structure used to represent **tabular data**. It's designed to resemble a spreadsheet or a database table. According to "R in a Nutshell," a data frame is a list that contains multiple named vectors of the same length. The "Programming in R Syllabus" also lists "Data Frames" as one of the key data types in R. Data frames are particularly well-suited for representing experimental data where you have individual observations, each with several different measurements that might have different dimensions or be qualitative rather than quantitative. They are also commonly used to represent business data kept in database tables.

Structure

- A data frame is organized into **rows** and **columns**.
- Each **column** in a data frame represents a **variable**. The columns can hold different types of data (e.g., numeric, character, factor, logical) within the same data frame. For example, one column might contain names (character strings), another might contain ages (numeric), and another might indicate a category (factor).
- Each **row** in a data frame represents an **observation** or a record, and all columns in a given row must have the same length. This means that each variable must have a value for each observation.
- Usually, each column has a **name**, and sometimes rows can also have names (row names). You can retrieve or set the row or column names using functions like colnames() and rownames().

Relationship to Lists

Crucially, **data frames are implemented as lists** with the class attribute set to "data.frame". This means that many of the ways you access elements in lists also work for data frames. A data frame is a list of equal-length vectors. Each vector in the list represents a column of the data frame.

Creating Data Frames

You can create a data frame using the data.frame() function. You provide vectors as arguments to this function, and these vectors become the columns of the data frame. The vectors must be of the same length, or R will return an error. You can also assign names to the columns during creation using the name = vector syntax.

```
teams <- c("PHI","NYM","FLA","ATL","WSN")

w <- c(92, 89, 94, 72, 59)
```

```
l <- c(70, 73, 77, 90, 102)

nleast <- data.frame(teams, w, l)

nleast
```

This example from "R in a Nutshell" demonstrates the creation of a data frame named nleast with three columns: teams, w, and l.

Another example shows creating a data frame from individual vectors:

```
name.last <- c("Manning", "Brady", "Pepper", "Palmer", "Manning")

name.first <- c("Peyton", "Tom", "Julius", "Carson", "Eli")

team <- c("Colts", "Patriots", "Panthers", "Bengals", "Giants")

position <- c("QB", "QB", "DE", "QB", "QB")

salary <- c(18700000, 14626720, 14137500, 13980000, 12916666)

top.5.salaries <- data.frame(name.last, name.first, team, position, salary)

top.5.salaries
```

Accessing Elements

Because data frames are lists, you can access their components (columns) using list-like syntax:

By Name using $: You can access a specific column by using the $ operator followed by the column name.

```
 nleast$teams # Access the 'teams' column

top.5.salaries$salary # Access the 'salary' column
```

By Name using [[""]]: You can also access a column using double square brackets [[]] and the column name as a character string.

```
 nleast[["w"]]

top.5.salaries[["position"]]
```

By Index using []: You can access columns by their numerical index within single square brackets. This will return a data frame containing the selected column(s).

```
 nleast      # Returns a data frame with the first column ('teams')

nleast[2:3]    # Returns a data frame with the second and third columns ('w' and 'l')
```

By Index using [[]]: Using double square brackets with a single numerical index will return the column as a vector.

```
 nleast[]    # Returns the 'w' column as a numeric vector
```

Accessing Rows and Specific Cells using []: You can access specific rows or individual cells using single square brackets with row and column indices (e.g., data.frame[row_index, column_index]). Omitting an index will select all rows or all columns.

```
 nleast     # Returns the first row

nleast[, 2]    # Returns the second column (similar to nleast[])

nleast   # Returns the element in the third row and first column
```

Key Properties

- **Columns must have the same number of rows**.
- **Different columns can have different data types**.

Usage Examples

- The syllabus mentions "List- operations" and "Data Frames" as distinct data types, implying that while related, they are used differently.
- The baseball data example shows a typical use case for data frames to store structured, mixed-type data. You can then access specific information, such as the number of losses for the Florida team using nleast$l[nleast$teams=="FLA"].
- The top.5.salaries example further illustrates the creation of a data frame with different data types (character and numeric).

Data Import and Export

Data frames are frequently created by importing data from external files such as text files (using functions like read.table, read.csv, read.delim) or from databases (using packages like RODBC or DBI). Conversely, you can export data frames to text files using functions like write.table or write.csv.

Relationship to Other Data Structures

While data frames can hold different data types, **matrices** in R are restricted to holding elements of a single basic type (numeric, character, or logical). If you try to combine different data types in a matrix, R will coerce them to the same type.

Attributes of Data Frames

Data frames have attributes, including:

- names: This attribute stores the column names. You can access and modify it using the names() function.
- row.names: This attribute stores the row names.
- class: This attribute indicates that the object belongs to the "data.frame" class.
- dim: This attribute stores the dimensions of the data frame (number of rows and columns).
- dimnames: This attribute is a list containing the column names and row names.

Operations on Data Frames

Many operations in R are designed to work with data frames, including:

- **Subsetting**: You can extract subsets of rows and columns based on conditions using the [] operator or the subset() function.
- **Merging**: You can combine two data frames based on common columns using the merge() function.
- **Binding**: You can combine data frames row-wise using rbind() or column-wise using cbind().
- **Transformations**: You can create new columns or modify existing ones using various functions.
- **Summarization**: Functions like summary(), aggregate(), and tapply() can be used to calculate summary statistics for data frame columns.
- **Sorting**: You can sort the rows of a data frame based on the values in one or more columns using the order() or sort() function.
- **Reshaping**: Functions like reshape() and packages like reshape2 (using melt() and cast()) allow you to change the structure of your data frame between "wide" and "long" formats.
- **Data Cleaning**: R provides tools for finding and removing duplicate rows using unique() or duplicated(), and for handling missing values (NA).

Data Editor

R provides a built-in **data editor** that allows you to view and edit data frames (and matrices) in a graphical user interface. You can open the data editor using the edit() function. While convenient

for inspection or minor edits, it's generally recommended to use scripting for more serious data entry or manipulation.

In summary, data frames are a cornerstone of data analysis in R, providing a flexible and structured way to store and work with tabular data of mixed types. Their list-based nature allows for powerful and intuitive ways to access and manipulate the data.

Matrices and Arrays in R

Matrices in R

- A **matrix** in R is a two-dimensional array. According to "R in a Nutshell," a matrix is an extension of a vector to two dimensions and is used to represent two-dimensional data of a single type. The "Programming in R Syllabus" also lists "Matrices and arrays" as a key data type in R.
- A matrix has **rows** and **columns**. All elements within a matrix must be of the **same data type**. If you attempt to combine different data types, R will coerce them to a single type.
- You can create a matrix using the matrix() function. Key arguments to the matrix() function include:
    - data: A vector of values to fill the matrix.
    - nrow: The desired number of rows.
    - ncol: The desired number of columns.
    - dimnames: An optional list of character vectors giving names to the dimensions (rows and columns).
- Elements in a matrix can be accessed using **row and column indices** within single square brackets [row_index, column_index]. You can also access entire rows by omitting the column index (e.g., m) or entire columns by omitting the row index (e.g., m[,2]).
- It's possible to transform another data structure into a matrix using the as.matrix() function.
- Matrices are implemented as **vectors**, not as a vector of vectors or a list of vectors. Array subscripts are used for referencing elements but don't reflect the way the data is stored.
- Unlike most other classes, matrices do not have an explicit class attribute. You can check if an object is a matrix using is.matrix().
- You can set or retrieve row and column names using rownames() and colnames() functions, respectively. These are convenience functions for accessing the dimnames attribute.

- Mathematical operations can be performed on matrices. The operator %*% is used for matrix multiplication. The function t() returns the transpose of a matrix. solve() can be used to solve systems of linear equations involving matrices.
- The crossprod() function can compute matrix cross-products.

Arrays in R

- An **array** in R is an extension of a vector to **more than two dimensions**. According to "R in a Nutshell," arrays are used to represent multidimensional data of a single type. The "Programming in R Syllabus" also includes "Matrices and arrays" as a fundamental data type.
- Like matrices, all elements within an array must be of the **same data type**.
- You can create an array using the array() function. Key arguments to the array() function include:
    - data: A vector of values to fill the array.
    - dim: A numeric vector giving the dimensions of the array (e.g., c(rows, columns, layers)).
    - dimnames: An optional list of character vectors giving names to the dimensions.
- Elements in an array are accessed using **indices for each dimension**, separated by commas within single square brackets [dim1_index, dim2_index, dim3_index, ...]. Omitting an index selects all elements along that dimension. You can also refer to a range or a non-contiguous set of indices.
- Like matrices, the underlying storage mechanism for an array is a **vector**.
- Similar to matrices, arrays do not have an explicit class attribute. You can test if an object is an array using is.array().
- The dim() function retrieves or sets the dimension of an array. The dimnames() function accesses the dimension names. The aperm() function transposes an array by permuting its dimensions.

Similarities and Differences

- Both matrices and arrays are used to store collections of data elements of a **single data type**.
- Both are built upon an underlying **vector**.
- Elements in both are accessed using **indices** within square brackets.
- A **matrix is a special case of an array** with exactly two dimensions.
- Arrays can have **more than two dimensions**, while matrices are strictly two-dimensional.

Accessing Elements

- For a matrix m, m[i, j] accesses the element at the i-th row and j-th column. m[i,] accesses the i-th row, and m[,j] accesses the j-th column.
- For an array a with dimensions c(d1, d2, d3), a[i, j, k] accesses the element at the i-th position of the first dimension, j-th of the second, and k-th of the third. a[i,,] would access the i-th "slice" across the first dimension, containing all combinations of the other two dimensions.

Relevant Functions and Operations

- matrix(): Creates a matrix.
- array(): Creates an array.
- as.matrix(): Coerces an object to a matrix.
- is.matrix(): Tests if an object is a matrix.
- is.array(): Tests if an object is an array.
- dim(): Gets or sets the dimensions of a matrix or array.
- dimnames(): Gets or sets the dimension names of a matrix or array.
- rownames(): Gets or sets the row names of a matrix.
- colnames(): Gets or sets the column names of a matrix.
- t(): Transposes a matrix.
- aperm(): Permutes the dimensions of an array.
- %*%: Matrix multiplication.
- crossprod(): Computes matrix cross-product.
- row(): Returns a matrix of row indices.
- col(): Returns a matrix of column indices.
- rowMeans(), colMeans(): Calculate row and column means.
- rowSums(), colSums(): Calculate row and column sums.
- apply(): Applies a function to the margins of an array.

These data structures, matrices and arrays, are fundamental for various mathematical and statistical computations in R.

Control Statements - Branching and looping - For loops, While loops, Controlling loops in R

Control statements in R allow you to manage the flow of execution in your programs, enabling you to make decisions and repeat code blocks. The "AIT 362 Programming in R Syllabus" mentions "**Branching and looping** - For loops, While loops, Controlling loops" as a key topic. "R in a Nutshell 2nd edition" provides detailed explanations of these control structures.

Branching (Conditional Statements)

Conditional statements in R allow you to execute specific blocks of code based on whether certain conditions are true or false.

- **if statement**: This is a fundamental conditional statement with the following forms:
    - if (condition) true_expression
    - if (condition) true_expression else false_expression The condition is a logical expression. If it evaluates to TRUE, the true_expression is executed. If it evaluates to FALSE in the second form, the false_expression is executed. It's crucial to understand that the if statement in R is **not a vector operation**. If the condition is a vector of logical values with more than one element, only the **first element** of the vector will be evaluated to determine which expression to execute. This can lead to unexpected behavior if you intend to perform element-wise conditional operations on vectors. A warning message is typically issued in such cases.
- **ifelse() function**: For element-wise conditional operations on vectors, you should use the ifelse() function. The syntax is: ifelse(condition, value_if_true, value_if_false) Here, condition is a logical vector. The function returns a vector of the same length where each element is taken from value_if_true if the corresponding element in condition is TRUE, and from value_if_false if it is FALSE. For example, ifelse(c(TRUE, FALSE, TRUE, FALSE, TRUE), a, b) will create a new vector by picking elements from vector a where the condition is TRUE and from vector b where it is FALSE.

- **switch statement**: While not explicitly detailed in the provided snippets, switch is another conditional control structure in R (mentioned in the teaching plan as "switch, if else"). It allows you to execute different code blocks based on the value of an expression. Typically, it takes an expression as input and matches its value against several possible cases, executing the code associated with the first matching case.

Looping

Loops in R provide a way to repeatedly execute a block of code. R offers three main types of loops:

repeat loop: This loop repeatedly executes an expression **indefinitely**. repeat expression To exit a repeat loop, you **must** use the break keyword within the loop. The break statement immediately terminates the loop and transfers control to the statement following the loop. You can also use the next command within a repeat loop to skip the current iteration and proceed to the next one. An example of using repeat with break to print multiples of 5 up to 25 is provided:

```
i <- 5

repeat {

  if (i > 25) break else {print(i); i <- i + 5;}

}
```

- If you don't include a break statement, a repeat loop will become an **infinite loop**.

while loop: This loop executes an expression **as long as a specified condition is TRUE**. while (condition) expression The condition is evaluated before each iteration. If it is TRUE, the expression (which can be a block of code enclosed in curly braces {}) is executed. The loop continues until the condition becomes FALSE. Similar to repeat loops, you can use break to exit a while loop prematurely and next to skip to the next iteration. The multiples of 5 example can be rewritten using a while loop:

```
i <- 5

while (i <= 25) {print(i); i <- i + 5}
```

for loop: This loop iterates over each item in a vector (or a list) and executes an expression for each item. for (var in sequence) expression The var takes on the value of each element in the sequence (which is typically a vector or a list) during each iteration of the loop. The expression is executed for each value of var. You can again use break and next within for loops. The multiples of 5 example can also be implemented with a for loop:

```
for (i in seq(from=5, to=25, by=5)) print(i)
```

- It's important to note two key properties of for loops:

    1. Results are **not automatically printed** inside a loop unless you explicitly use the print() function.
    2. The variable var that is assigned in the for loop **is changed in the calling environment** after the loop finishes.

It is mentioned that the looping functions (repeat, while, and for) have a special type because the expression within them is not necessarily evaluated.

Controlling Loops

The break and next statements are the primary mechanisms for controlling the flow within loops in R.

- **break**: When encountered inside a loop (repeat, while, or for), the break statement immediately terminates the innermost loop and execution continues with the statement immediately following the loop.

- **next**: When encountered inside a loop, the next statement stops the execution of the current iteration and immediately jumps to the beginning of the next iteration of the loop. For for loops, it moves to the next element in the sequence; for while loops, it re-evaluates the condition; and for repeat loops, it simply starts the next cycle.

Looping Extensions

The sources also briefly mention that while R doesn't have built-in iterators or foreach loops like some other languages, these functionalities are available through add-on packages, such as those from Revolution Computing (now part of Microsoft) and available on CRAN. Packages like iterators and foreach provide more advanced and often more elegant ways to handle iterations in R.

In summary, R provides a set of control statements (if, ifelse, switch for branching and repeat, while, for for looping) along with control keywords (break, next) to manage the execution flow of your R programs. Understanding how these statements work, especially the vector nature of ifelse versus the scalar nature of if, and the use of break and next in loops, is crucial for writing effective and correct R code.

Functions- Function as arguments, Named arguments

Functions as Arguments

In R, **functions are treated as objects**, just like any other data type. This fundamental property allows you to do some powerful things, including passing functions as arguments to other functions.

- Many functions in R are designed to **take other functions as arguments**. For example, many modeling functions might have an argument that specifies a function for handling

missing values.

The sapply function is a clear example of a function that takes another function as an argument. sapply iterates through each element of a vector and applies the function provided as an argument to each element, returning the results.

```
a <- 1:7

sapply(a, sqrt)
```

- In this example, sqrt is passed as an argument to sapply, and sapply applies the sqrt function to each element of the vector a.

- This capability is particularly useful when you have a function that doesn't work element-wise on a vector. sapply provides a way to **extend the functionality of such functions** to work on vectors.

Anonymous functions are often used when passing functions as arguments. Because functions are objects, you can create functions without giving them a name, and these anonymous functions can be directly passed as arguments to other functions.

```
apply.to.three <- function(f) {f(3)}

apply.to.three(function(x) {x * 7}) # An anonymous function multiplying by 7
```

- Here, function(x) {x * 7} is an anonymous function that is passed as the argument f to apply.to.three.

You can also define an anonymous function and apply it directly to an argument by enclosing the function definition in parentheses:

```
(function(x) {x+1})(1)
```

- The parentheses are necessary because function calls (f(arguments)) have very high precedence in R.

- The use of functions as arguments, especially with functions like sapply and anonymous functions, can be a **good alternative to control structures** like loops for performing operations on collections of data. For instance, calculating the square of each element in a

vector can be done using sapply with an anonymous function instead of a for loop.

Named Arguments

When defining a function in R, you assign names to its arguments. When calling a function, you can specify the values for these arguments in a few different ways, including by using their names.

When calling a function, you can provide arguments **by their exact names**. This explicitly tells R which value corresponds to which argument, regardless of the order in which they are listed in the function call.

Consider a function defined as:

```
addTheLog <- function(first, second) {first + log(second)}
```

You can call this function using exact names:

```
addTheLog(second=exp(4), first=1) # Here, the order is different from the definition
```

- In this case, first is assigned the value 1, and second is assigned the result of exp(4).

If you provide the arguments in the **default order** defined in the function, you can **omit the names**.

Using the same addTheLog function:

```
addTheLog(1, exp(4)) # Order matches the definition (first, second)
```

- Here, 1 is assigned to first, and exp(4) is assigned to second because of their position in the function call.

R also supports **partially matching names** for arguments. This means you can use an abbreviated version of the argument name as long as it uniquely identifies the argument. However, the source mentions that partial name matching is a **deprecated feature** because it can lead to confusion.

Using addTheLog again:

addTheLog(s=exp(4), f=1) # 's' partially matches 'second', 'f' partially matches 'first'

- In this call, s is matched to second, and f is matched to first.

- Using **exact argument names** is generally considered **good practice**. While it might require more typing, it makes your code easier to read and removes potential ambiguity, especially in functions with many arguments or arguments with similar names.

- If you call a function with an ambiguous partial name (a partial name that could match more than one argument), R will typically throw an error.

  For example, if you have a function f <- function(arg1=10, arg2=20) {...} and you call it with f(arg=1), you will get an error because arg could partially match both arg1 and arg2.

Understanding how to use functions as arguments and how to call functions with named arguments provides flexibility and clarity in your R programming. Passing functions as arguments enables higher-order functions and functional programming paradigms, while named arguments improve code readability and reduce the reliance on remembering the exact order of arguments.
