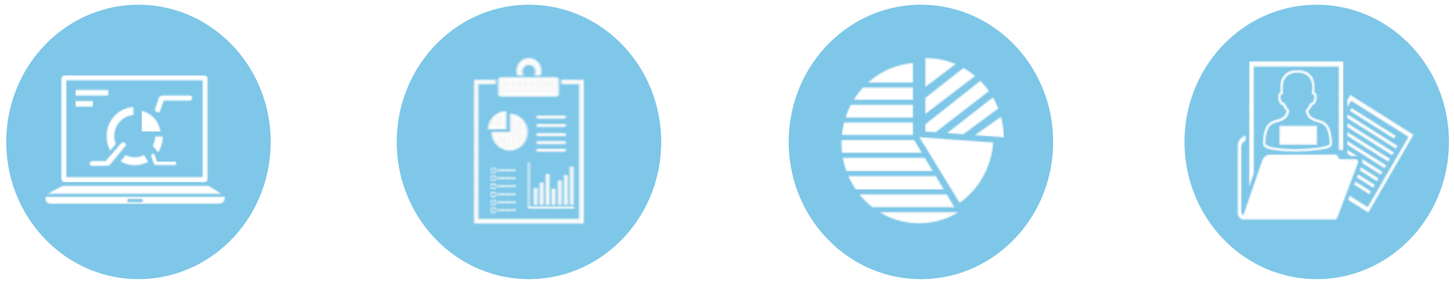
Data Science with R

Lesson 3— Data Structures

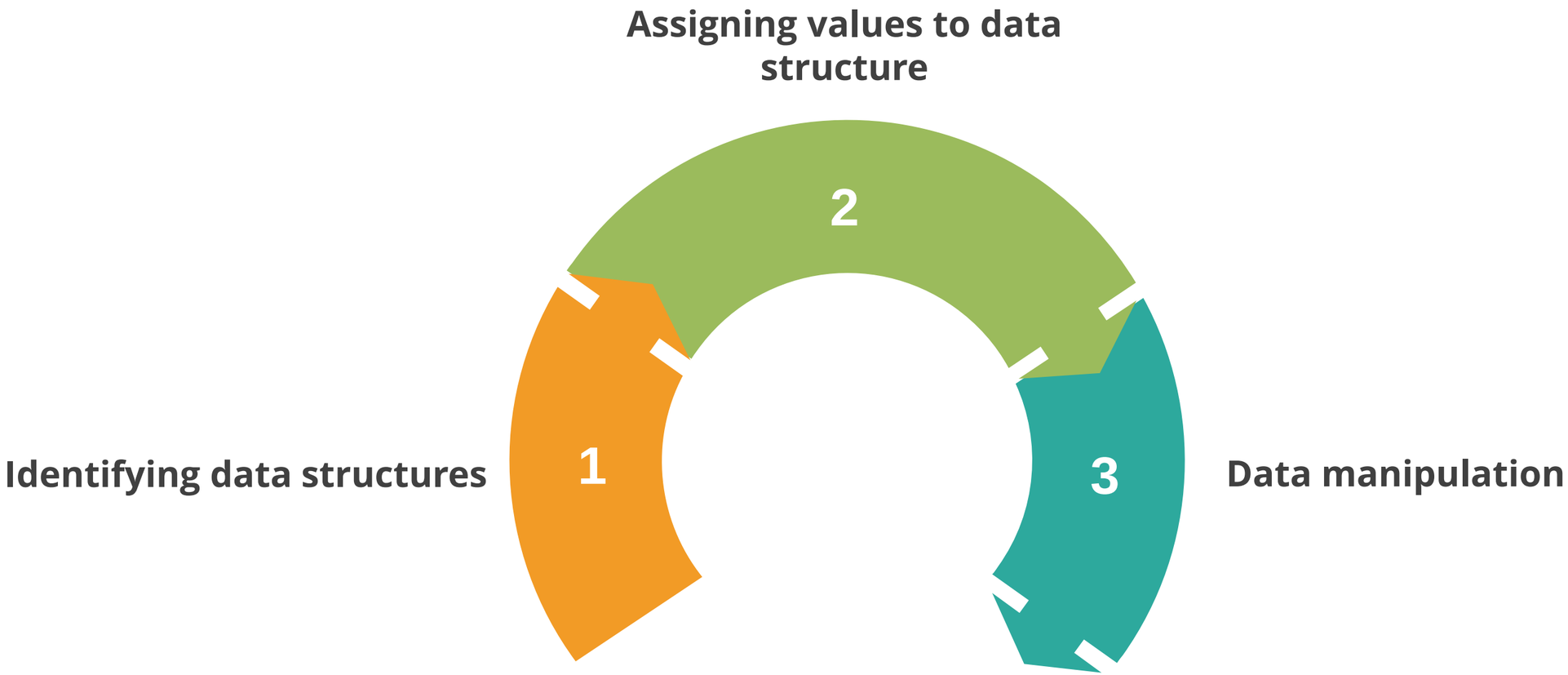


Learning Objectives

- Explain how to identify data structures in R
- Discuss how to assign values to data structures
- Describe how to manipulate data using the dplyr package

Introduction

There are three important steps when working with data:



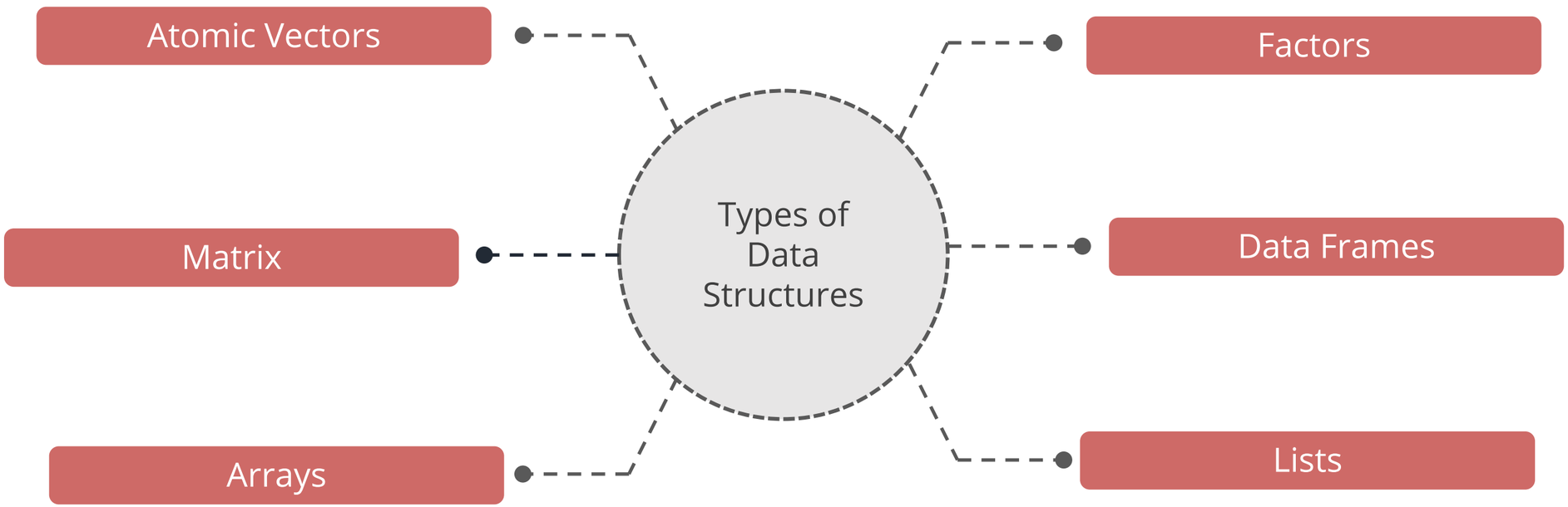
Data Structures

Topic 1— Identifying Data Structures

Why Is It Necessary to Identify Data Structures?

As an R programmer, it is important to know how to identify and represent the data in R before starting to analyze it.

Types of Data Structures



Types of Data Structures

TYPE AND DIMENSIONALITY

Data Structure	Type	Dimensionality
Atomic Vectors	Homogeneous	1
List	Heterogeneous	1
Matrix	Homogeneous	2
Array	Homogeneous	n
Factor	Homogeneous	1
Data Frame	Heterogeneous	2

Types of Data Structures

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- An atomic vector is a one-dimensional object and is the simplest data structure.
- It is called an atomic vector as all elements in it are of the same type.
- The data types in atomic vectors:
  - Numeric Data Type
  - Integer Data Type
  - Character Data Type
  - Logical Data Type

Example

```
a <- c(1, 2, 5, 3, 6, -2, 4)
b <- c("one", "two", "three")
c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
```

Types of Data Structures

CREATING ATOMIC VECTORS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Vectors of consecutive numbers can be created using the ‘ : ‘ operator.

Example

```
> x <- 1:5; x
[1] 1 2 3 4 5

> y <- 1:-1; y
[1] 1  0 -1
```

Types of Data Structures

ACCESSING ELEMENTS OF ATOMIC VECTORS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Vector elements can be accessed by vector indexing. The vector can be numeric, character, or logical.
- An individual element of a vector is accessed by its position, which is indicated within square brackets.

Example

```
vec <- c("a", "b", "c", "d", "e", "f")
vec[1]  # will return the first element in the vector
vec[c(2,4)]     # will return the second and fourth elements in the vector
```

To comment in R software, special character # is placed in the beginning

Types of Data Structures

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Matrix is a two-dimensional data structure.
- It is similar to a vector but has the dimension attribute.

Example

```
vector <- c(1,2,3,4)
foo <- matrix(vector, nrow=2, ncol=2)
```

Elements in a matrix must be of the same type, whether a number, character, or Boolean.

Types of Data Structures

CREATING MATRIX

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

The values for the rows and columns are assigned using nrow and ncol arguments respectively.

Example

```
> matrix(1:9, nrow = 3, ncol = 3, byrow=TRUE)
[,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Byrow:
Matrix is filled column-wise. By assigning TRUE to the argument by row, it can be reversed to row-wise filling

Types of Data Structures

ACCESSING ELEMENTS OF MATRIX

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Elements are accessed using the square bracket '[ ]' indexing method.

Example

```
> x
[,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x[c(1,2),c(2,3)]    # select rows 1 & 2 and columns 2 & 3
```

Types of Data Structures

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Arrays are similar to a matrix but can have more than two dimensions.

Example

```
A <- array(1: 24, dim = c(3, 4, 2))
```

Types of Data Structures

CREATING ARRAYS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Arrays take vectors as input in the matrix.
- Rows and columns are named using the ‘dimnames’ parameter.

Example

```
vector1 <- c(4,2,1)
vector2 <- c(22,34,76,88,98,65)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")
result <- array(c(vector1,vector2),dim = c(4,2,1),
dimnames = list(row.names,column.names,matrix.names))
print(result)
```

Output:

```
, , Matrix1

     COL1 COL2 COL3
ROW1    4   22   88
ROW2    2   34   98
ROW3    1   76   65

, , Matrix2

     COL1 COL2 COL3
ROW1    4   22   88
ROW2    2   34   98
ROW3    1   76   65
```

Types of Data Structures

ACCESSING ELEMENTS OF ARRAYS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Using the index position, one can access or change the individual elements in an array.

Example

```
print(result[2,,1])
# Prints the second row of the first matrix of the
array

Output:
COL1 COL2 COL3
   2   34   98
```

Types of Data Structures

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Factors take only a predefined, finite number of categorical values.

Example

```
> x
[1] male female female male
Levels: female male
```

Types of Data Structures

CREATING FACTORS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Factors are created using the factor() function.
- They are built using two attributes: class and levels.

Example

```
> x <- factor(c("male", "female", "female", "male"));
> x
[1] male female female male
Levels: female male
> x <- factor(c("male", "female", "female", "male"), levels = c("male", "female"));
> x
[1] male female female male
Levels: male female
```

Types of Data Structures

ACCESSING ELEMENTS OF FACTORS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Accessing elements of factors is similar to accessing elements of an atomic vector.

Example

```
x
[1] single married married single
Levels: married single
>x[3]   #access 3rd element
[1] married
Levels: married single
```

Types of Data Structures

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Data frames are the most commonly used data structures in R.
- A data frame is similar to a general matrix, but its columns can contain different modes of data, such as a number and character.

Examples

```
name <- c( “Joe” , “John” , “Nancy” )
sex <- c(“M”, “M”, “F”)
age <- c(27,26,26)
df <- data.frame(name,sex,age)
```

Types of Data Structures

CREATING DATA FRAMES

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Data frames are created using the data.frame() function.
- When the argument StringsAsFactors = FALSE is passed, the data.frame() function will not convert character vector into factor.

Example

```
df <- data.frame(
Name <- c(“Joe”, “John”, “Nancy”)
Sex <- c(“M”, “M”, “F”)
Age <- c(27, 26, 26),
StringsAsFactors = FALSE
)
```

Types of Data Structures

ACCESSING ELEMENTS OF DATA FRAMES

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

The data can be accessed using column names.

Example

```
result <-data.frame(name$age,name$sex)
print(result)

Output:
Name     Age     Sex
Joe      27       M
John     26       M
Nancy    26       F
```

Types of Data Structures

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- Lists are the most complex data structures.
- A list is a vector that has elements of different types.
- A list may contain a combination of vectors, matrices, data frames, and even other lists.

Example

```
vec <- c(1,2,3,4)
mat <- matrix(vec,2,2)
List_data <- list(vec, mat)
```

Types of Data Structures

CREATING LISTS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

Lists are created using list() function.

Example:

```
vec <- c(1,2,3,4)
mat <- matrix(vec,2,2)
list_data <- list(vec, mat)
Print(List_data)
```

Output:

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

[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Types of Data Structures

ACCESSING ELEMENTS OF LISTS

- Atomic Vectors
- Matrix
- Arrays
- Factors
- Data Frames
- Lists

- List elements can be accessed by indexing.
- The vector can be an integer, character, or logical vector.

Example

```
print(list_data[mat])
```

Output:

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

[[2]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
NULL

[[4]]
NULL
```

Data Structures

Topic 2— Assigning Values to Data Structures

Assigning Values to Data Structures

Now that data structures are identified, the next step is to assign values to the data structure. This is achieved by importing and exporting data from files.

Assigning Values to Data Structures

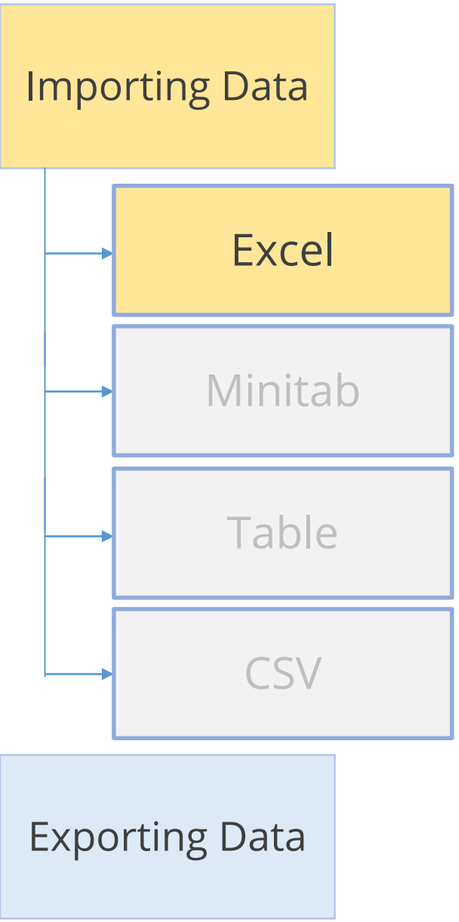
Importing Data

Exporting Data

You can import data from four types of files in R:

- Excel
- Minitab
- Table
- CSV

Assigning Values to Data Structures



Before using the sample data available in an Excel format, you need to import the data into R.

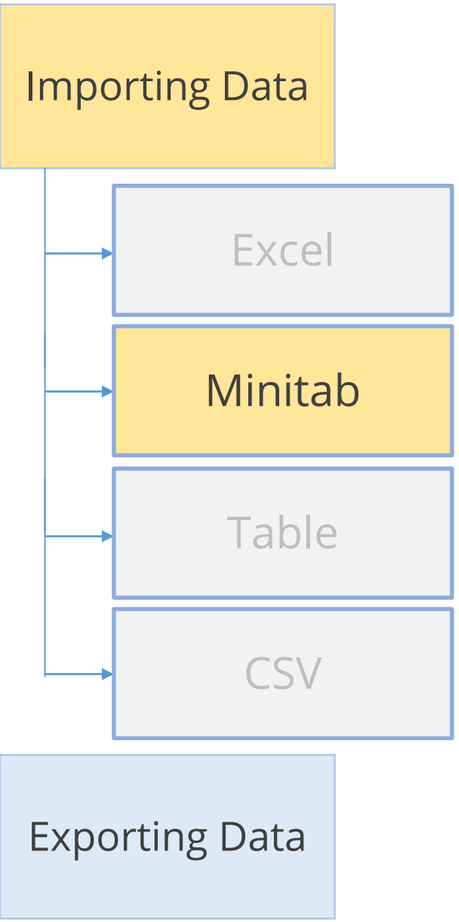
Example 1

```
library(gdata)                          #load gdata package
help(read.xls)                          #documentation
mydata = read.xls("mydata.xls")         #read from first sheet
```

Example 2

```
library(XLConnect)
wk = loadWorkbook("mydata.xls")
df = readWorksheet(wk, sheet="Sheet1")
```

Assigning Values to Data Structures

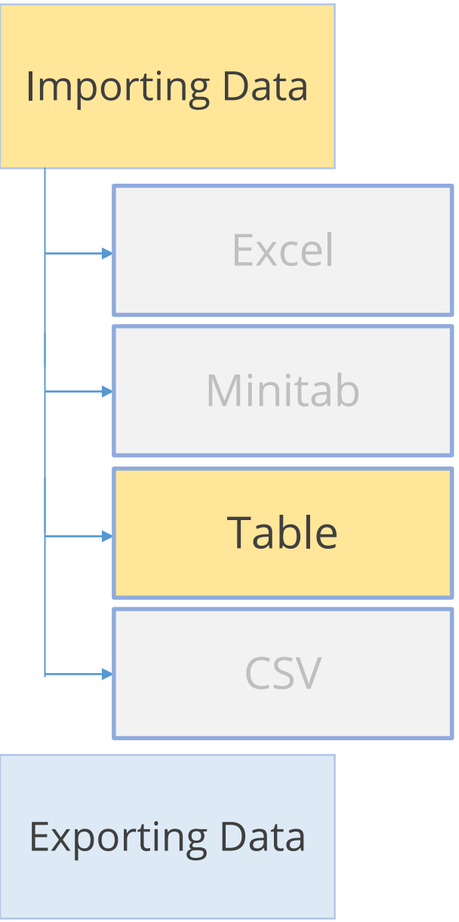


- Use the function read.mtp to import the sample data from a Minitab Portable Worksheet format.
- This function returns a list of components in the Minitab worksheet.

Example

```
library(foreign)
help(read.mtp)
mydata = read.mtp("mydata.mtp")
```

Assigning Values to Data Structures



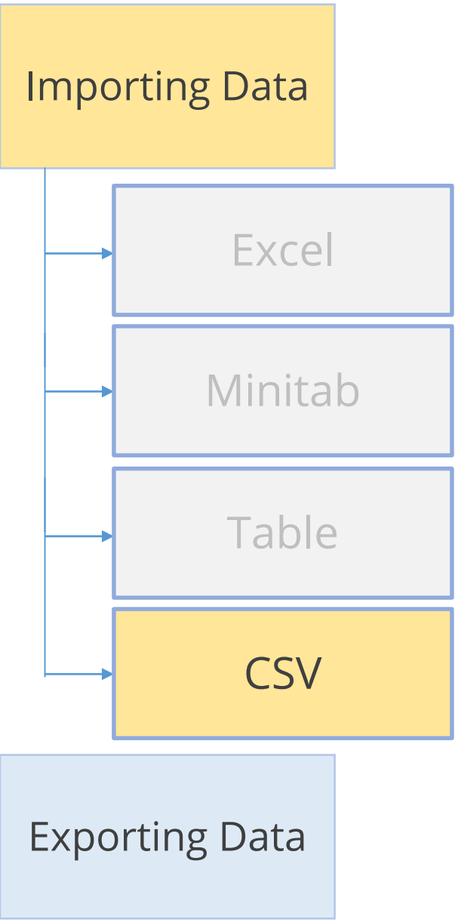
- A text file can have a data table in it. The cells inside the table are separated by blank characters.
- Here's an example of a table with four rows and three columns. Let's see how to import this data.

Example

100	a1	b1
200	a2	b2
300	a3	b3
400	a4	b4

```
help(read.table)
mydata = read.table("mydata.txt")
```

Assigning Values to Data Structures



- R allows data import from a Comma Separated Values (CSV) format as well.
- Each cell inside such a data file is separated by a special character, usually a comma.

Example

```
help(read.csv)
mydata = read.csv("mydata.csv", sep=",")
```

Assigning Values to Data Structures

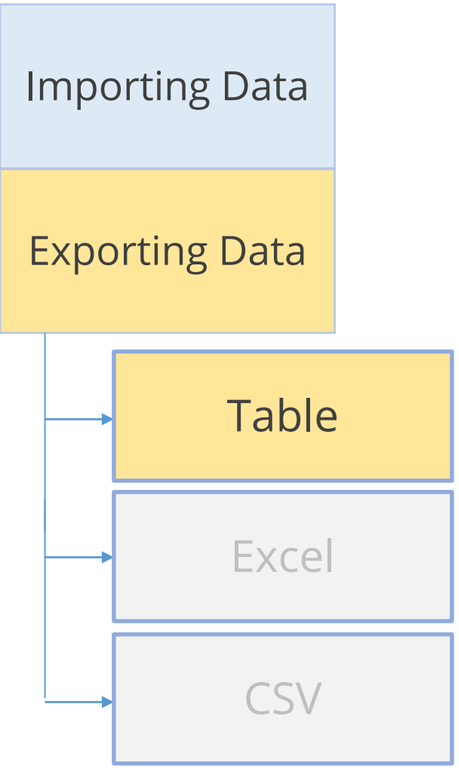
Importing Data

Exporting Data

R supports data export from three types of files:

- Table
- Excel
- CSV

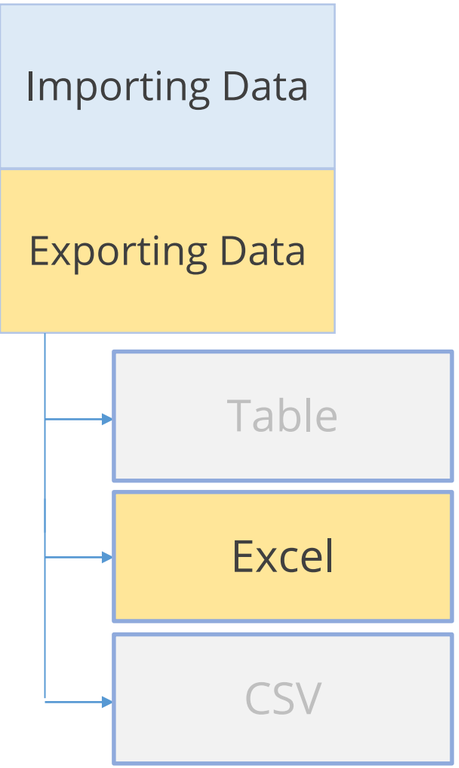
Assigning Values to Data Structures



Example

```
help(write.table)
write.table(mydata, "c:/mydata.txt", sep="\t")
```

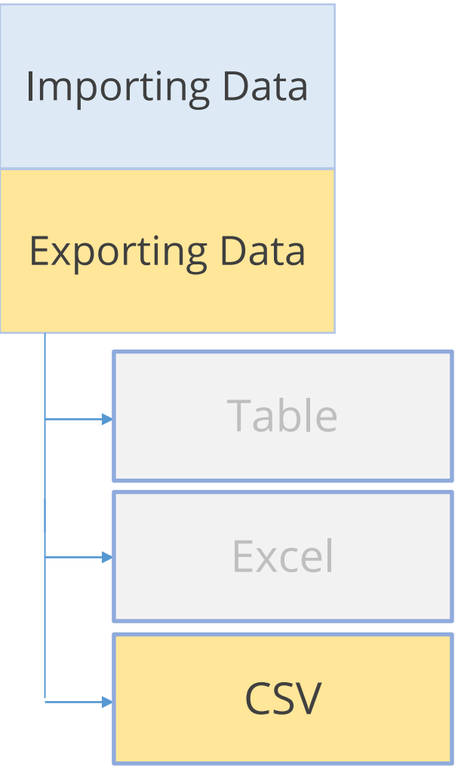
Assigning Values to Data Structures



Example

```
library(xlsx)
help(write.xlsx)
write.xlsx(mydata, "c:/mydata.xlsx")
```

Assigning Values to Data Structures



Example

```
help(write.csv)
write.csv(mydate, file = "mydata.csv")
```

Data Structures

Topic 3 —Data Manipulation

Data Manipulation

- Data manipulation is required to bring accuracy and precision in the data.
- R base package has '**apply**' functions in it, which helps in manipulating the data multiple times, thus avoiding the use of loop constructs.

Apply Functions

- The apply functions are used to perform a specific change to each column or row of R objects.
- Types of apply functions in R:
  - apply()
  - lapply()
  - sapply()
  - tapply()
  - mapply()
  - vapply()
  - rapply()

apply(), lapply(), and sapply() are the most commonly used functions. We will limit our scope to apply(), lapply(), and sapply() in this course.

Types of Apply Functions

- apply() helps apply a function to a matrix row or column and returns a vector, array, or list.

Syntax:

```
apply(x, margin, function)
```

Where,

- margin indicates whether the function is to be applied to a row or column.
  - margin = 1 indicates that the function needs to be applied to a row.
  - margin = 2 indicates that the function needs to be applied to a column.
- function can be any function such as mean, sum, or average

apply()	lapply()	sapply()

Types of Apply Functions

Examples:

```
m <- matrix( c(1,2,3,4),2,2 )
apply(m,1,sum)
apply(m,2,sum)
```

apply()

lapply()

sapply()

Types of Apply Functions

- lapply() takes a list as an argument and works by looping through each element in the list. The output of the function is a list.

Syntax:

```
lapply(list, function)
```

Examples:

```
list <-list(a=c(1,1, b=c(2,2, c=c(3,3))
lapply(list,sum)
lapply(list,mean)
```

apply()	lapply()	sapply()

Types of Apply Functions

- sapply() is similar to lapply(), except that it simplifies the result so that:
  - If the result is a list and every element in the list is of size 1, then a vector is returned.
  - If the result is a list and every element in the list is of the same size (>1), then a matrix is returned.
- Otherwise, the result is returned as a list itself.

apply()	lapply()	sapply()

Types of Apply Functions

Syntax:

```
sapply(list, func)
```

Examples:

```
list <- list(a = c(1,1), b=c(2,2), c=c(3,3))
sapply(list,sum)
list <- list(a = c(1,2), b=c(1,2,3), c=c(1,2,3,4))
sapply(list, range)
```

apply() | lapply() | sapply()

dplyr Package

- There are packages available consisting of many functions which help in data manipulation.
- dplyr is one of the most commonly used functions and is a powerful R package.

We will limit our scope to dplyr package in this course.

Features of dplyr Package

- dplyr package transforms and summarizes tabular data with rows and columns.
- It provides simple verbs— functions that correspond to the most common data manipulation tasks to help you translate your thoughts into code.
  - Select
  - Filter
  - Arrange
  - Mutate
  - Summarize
- The use of efficient data storage backends by dplyr results in quicker processing speed.

Functions of dplyr Package

- The dplyr package has the following functions:
  - Select()
  - Filter()
  - Arrange()
  - Mutate()
  - Summarize()
- To understand the use of these functions, let’s consider the dataset "mtcars“

Functions of dplyr Package

- Select()
- Filter()
- Arrange()
- Mutate()
- Summarize()

This function allows you to select specific columns from large data sets.

Examples

Different ways to select column by name:

```
select(mtcars, mpg, disp)
```

```
select(mtcars, mpg:hp)
```

```
select(iris, starts_with("Petal"))
select(iris, ends_with("Width"))
select(iris, contains("etal"))
select(iris, matches(".t."))
```

Functions of dplyr Package

- Select()
- Filter()
- Arrange()
- Mutate()
- Summarize()

- This function enables easy filtering, zoom in, and zoom out of relevant data.
- The two types of filters are explained below:

Examples

Simple filter

```
filter(mtcars, cyl == 8)
filter(mtcars, cyl < 6)
```

Multiple criteria filter

```
filter(mtcars, cyl < 6 & vs == 1)
filter(mtcars, cyl < 6 | vs == 1)
```

Comma separated arguments are equivalent to the "And" condition.
Example: filter(mtcars, cyl < 6, vs == 1)

Functions of dplyr Package

- Select()
- Filter()
- Arrange()
- Mutate()
- Summarize()

This function helps arrange the data in a specific order.

Examples

```
arrange(mtcars, desc(disp))
arrange(mtcars, cyl, disp)
```

Functions of dplyr Package

- Select()
- Filter()
- Arrange()
- Mutate()
- Summarize()

This function helps add new variables to an existing data set.

Examples

```
mutate(mtcars, my_custom_disp = disp / 1.0237)
```

Functions of dplyr Package

- Select()
- Filter()
- Arrange()
- Mutate()
- Summarize()

This function summarizes multiple values to a single value in a dataset.

Examples

```
summarise(group_by(mtcars, cyl), mean(disp))
summarise(group_by(mtcars, cyl), m = mean(disp), sd = sd(disp))
```

Functions of dplyr Package

- Select()
- Filter()
- Arrange()
- Mutate()
- Summarize()

Here's a list of summary functions that can be used within this function:

- first: Returns the first element of a vector
- last: Returns the last element of a vector
- nth(x,n): Returns the 'n'th element of a vector
- n(): Returns the number of rows in a dataframe
- n_distinct(x): Returns the number of unique values in vector x
- In addition, the following functions are also used:

Examples

mean	median	mode
max	min	sun
var	length	IQR

Key Takeaways

- The types of data structures in R are vectors, matrix, arrays, factors, data frames, and lists.
- R supports Excel, Minitab, Table, and CSV format for importing data and Table, Excel, and CSV for exporting data.
- dplyr is a powerful R package that transforms and summarizes tabular data with rows and columns.
- The five types of dplyr functions are select, filter, arrange, mutate, and summarize.