

**Application Development Laboratory (CS 33002)**

# **KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY**

## **School of Computer Engineering**



Strictly for internal circulation (within KIIT) and reference only. Not for outside circulation without permission

***2 Credit***

**Analytics Application Development using R**

# Lab Contents



2

Sr #	Major and Detailed Coverage Area	Lab#
1	Data Frame	3
2	Handling Data in R Workspace	
3	Character	
4	Date and Time	

# Data Frame



3

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- ❑ The column names should be non-empty.
- ❑ The row names should be unique.
- ❑ The data stored in a data frame can be of numeric, factor or character type.
- ❑ Each column should contain same number of data items.

Data frames are used to store spreadsheet-like data.

# Create Data Frame



4

Data frames to be created with the **data.frame** function:

```
# Create the data frame.
```

```
x <- data.frame("SN" = 1:2, "Age" = c(21,15), "Name" = c("John","Dora"))
```

```
# Print the data frame.
```

```
print(x)
```

```
# Print the data type.
```

```
print(class(x))
```

```
# Get the structure of the data frame.
```

```
str(x)
```



```
'data.frame':    2 obs. of  3 variables:
```

```
$ SN : int  1 2
```

```
$ Age : num  21 15
```

```
$ Name: Factor w/ 2 levels "Dora","John": 2 1
```

Notice above that the third column, Name is of type factor, instead of a character vector. By default, data.frame() function converts character vector into factor. To suppress this behavior, we can pass the argument stringsAsFactors=FALSE.

# Factor



5

Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data). For example: a data field such as marital status may contain only values from single, married, separated, divorced, or widowed. Using factors with labels is better than using integers because factors are self-describing. Having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

In such case, we know the possible values beforehand and these predefined, distinct values are called levels.

A factor is created using the function `factor()`. Levels of a factor are inferred from the data if not provided.

```
x <- factor(c("single", "married", "married", "single"));  
print(x)
```

```
x <- factor(c("single", "married", "married", "single"), levels = c("single",  
"married", "divorced"));  
print(x)
```

# Create Data Frame Example



6

```
# Create the data frame of few employees.
```

```
emp.data <- data.frame(  
  emp_id = c(1:5),  
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),  
  salary = c(623.3,515.2,611.0,729.0,843.25),  
  start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-  
11", "2015-03-27")),  
  stringsAsFactors = FALSE  
)
```

```
# Print the data frame.
```

```
print(emp.data)
```

```
# Print the data type.
```

```
print(class(emp.data))
```

```
# Get the structure of the data frame.
```

```
str(emp.data)
```

# Slice Data Frame

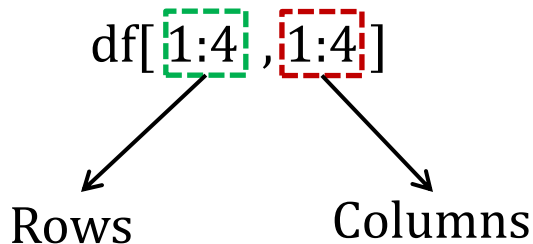


7

It is possible to SLICE values of a Data Frame. We select the rows and columns to return into bracket precede by the name of the data frame.

A data frame is composed of rows and columns, `df[A, B]`. A represents the rows, B the columns, and df represents a collection of variables to join. We can slice either by specifying the rows and/or columns.

The left part represents the **rows**, and the right part is the **columns**. Note that the symbol `:` means to. For instance, `1:3` intends to select values from 1 to 3.

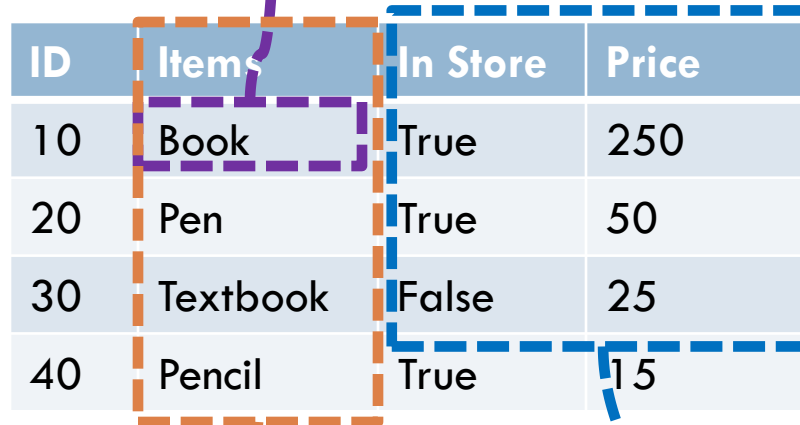


# Slice Data Frame cont'd



8

# Select row 1 in column 2  
`df[1,2]`



ID	Items	In Store	Price
10	Book	True	250
20	Pen	True	50
30	Textbook	False	25
40	Pencil	True	15

# Select column 1  
`df[,1]`

# Select rows 1 to 3  
and columns 3 to 4  
`df[1:3,3:4]`



# Extract Data from Data Frame



9

```
# Extract Specific columns using column name
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

```
# Extract the first two rows and then all columns
result <- emp.data[1:2,]
print(result)
```

```
#Extract 3rd and 5th row with 2nd and 4th column
result <- emp.data[c(3,5),c(2,4)]
print(result)
```

# Lab Work



10

- ☐ Create a data frame with the following variables  
Died.At <- c(22,40,72,41)  
Writer.At <- c(16, 18, 36, 36)  
First.Name <- c("John", "Edgar", "Walt", "Jane")  
Second.Name <- c("Doe", "Poe", "Whitman", "Austen")  
Sex <- c("MALE", "MALE", "MALE", "FEMALE")  
Date.Of.Death <- c("2015-05-10", "1849-10-07", "1892-03-26","1817-07-18")
- ☐ Extract Sex and Name
- ☐ Extract 3rd and 4th row with 2nd and 4th column

# Expand Data Frame



11

A data frame can be expanded by adding columns and rows.

## Add Column

A column can be appended to a Data Frame. The symbol \$ is used to append a new variable.

# Add the "dept" coulumn.

```
emp.data$dept <- c("IT","Operations","IT","HR","Finance")  
print(emp.data)
```

OR

# Add the "dept" coulumn.

```
department <- c("IT","Operations","IT","HR","Finance")  
emp.data$dept <- department  
print(emp.data)
```

**Note:** The number of elements in the vector has to be equal to the no of elements in data frame.

# Expand Data Frame - Add Row



12

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function. **Example:** we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the new data frame
```

```
emp.newdata <- data.frame(  
  emp_id = c(6:8),  
  emp_name = c("Rasmi","Pranab","Tusar"),  
  salary = c(578.0,722.5,632.8),  
  start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),  
  dept = c("IT","Operations","Fianance"),  
  stringsAsFactors = FALSE  
)
```

```
# Bind the two data frames.
```

```
emp.finaldata <- rbind(emp.data, emp.newdata)  
print(emp.finaldata)
```

# Subset a data frame



13

It is possible to subset based on whether or not a certain condition was true. `subset()` function is used to do so and the syntax is **`subset(df, condition)`** arguments:

df: data frame used to perform the subset

condition: define the conditional statement

We want to return only the employees with salary is above 600, we can do:

```
# Select salary above 600
```

```
subset(emp.finaldata, subset = salary > 600)
```

# Data frame Manipulation



14

Like matrices, data frames can be transposed using the `t` function, but in the process all the columns (which become rows) are converted to the same type, and the whole thing becomes a matrix.

Create the Data frame

Attr1	Attr2	Attr3	Attr4
1	A	100	A
2	B	102	B
3	C	104	C

Transpose?



# Data frame Manipulation cont'd



15

Data frames can be joined together using `cbind` function assuming that they have the same rows. It combine data frames as columns in matrix.

`cbind`

df1

df2

Subtype, Gender, Expression

A,m,-0.54

A,f,-0.8

B,f,-1.03

Age, City

32,New York

21,Houston

34,Seattle

`cbind(df1,df2)`

Subtype, Gender, Expression, Age, City

A,m,-0.54, 32,New York

A,f,-0.8, 21,Houston

B,f,-1.03, 34,Seattle

The row number of the two data frame must be equal.

# Data frame Manipulation cont'd



16

Data frames can also be joined together using `rbind`, assuming that they have the same columns. It combine data frames as rows in a matrix.

*`cbind`*

df1

df2

Subtype, Gender, Expression

A,m,-0.54

A,f,-0.8

Subtype, Gender, Expression

D,m,3.22

B,f,-1.03

`rbind(df1,df2)`

Subtype, Gender, Expression

A,m,-0.54

A,f,-0.8

D,m,3.22

B,f,-1.03

The column of the two data frame must be same, otherwise the combination will be meaningless.



# Workspace



17

The environment/workspace is current R working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions). At the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started.

Environments themselves are just another type of variable—we can assign them, manipulate them, and pass them into functions as arguments, just like we would any other variable. when you assign a variable, it will automatically go into an environment called the global environment. When you call a function, an environment is automatically created to store the function-related variables.

Slightly annoyingly, environments aren't created with the environment function (that function returns the environment that contains a particular function). Instead, what we want is `new.env` i.e. `an_environment = new.env()`

# Workspace cont'd



18

Assigning variables into environments can be with either double square brackets or the dollar sign operator.

```
an_environment[["pythag"]] <- c(12, 15, 20, 21)
```

```
an_environment$root <- polyroot(c(6, -5, 1))
```

```
assign("moonday", weekdays(as.Date("1969/07/20")), an_environment)
```

Retrieving the variables works in the following way:

```
an_environment[["pythag"]]
```

```
[1] 12 15 20 21
```

```
an_environment$root
```

```
[1] 2+0i 3-0i
```

```
get("moonday", an_environment)
```

```
[1] "Sunday"
```

# Inspecting Variables in Workspace



19

While we're working, it's often nice to know the names of the variables that we've created and what they contain. To list the names of existing variables, use the function **ls**. This is named after the equivalent Unix command, and follows the same convention: by default, variable names that begin with a `.` are hidden. To see them, pass the argument `all.names = TRUE`

```
ls()
```

```
ls(pattern = "ea")
```

`browseEnv` function provides a similar capability, but displays its output in an HTML page in our web browser.

After working for a while, especially while exploring data, our workspace can become quite cluttered. We can clean it up by using the **rm** function to remove variables.

```
rm(var1, var2, var3)
```

```
rm(list = ls()) #Removes everything. Use with caution!
```

# Inspecting Variables in Workspace cont'd



20

Below are some standard commands for managing workspace.

`getwd()` # print the current working directory – cwd

`setwd("c:/docs/mydir")` # change to a directory

**Note:** R gets confused if you use a path in your code like:

`c:\mydocuments\myfile.txt`

This is because R sees "\" as an escape character. Instead, use:

`c:\\my documents\\myfile.txt`

`c:/mydocuments/myfile.txt`

# String



21

Text data is stored in character vectors (or, less commonly, character arrays). It's important to remember that each element of a character vector is a whole string, rather than just an individual character.

## *Constructing String*

character vectors can be created with the `c` function. We can use single or double quotes around our strings, as long as they match, though double quotes are considered more standard.

```
c(  
  "You should use double quotes most of the time",  
  'Single quotes are better for including " inside the string'  
)
```

The `paste` function combines strings together. Each vector passed to it has its elements recycled to reach the length of the longest input, and then the strings are concatenated, with a space separating them. We can change the separator by passing an argument called **sep**, or use the related function `paste0` to have no separator.

# Constructing String



22

```
paste(c("red", "yellow"), "lorry")
```

```
[1] "red lorry" "yellow lorry"
```

```
paste(c("red", "yellow"), "lorry", sep = "-")
```

```
[1] "red-lorry" "yellow-lorry"
```

```
paste(c("red", "yellow"), "lorry", collapse = ", ")
```

```
[1] "red lorry, yellow lorry"
```

```
paste0(c("red", "yellow"), "lorry")
```

```
[1] "redlorry" "yellowlorry"
```

The function `toString` is a variation of `paste` that is useful for printing vectors. It separates each element with a comma and a space, and can limit how much we print.

# Constructing String cont'd



23

```
x <- (1:15) ^ 2
```

```
toString(x)
```

```
[1] "1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225"
```

```
toString(x, width = 40)
```

```
[1] "1, 4, 9, 16, 25, 36, 49, 64, 81, 100...."
```

cat is a low-level function that works similarly to paste, but with less formatting. You should rarely need to call it directly yourself, but it is worth being aware of, since it is the basis for most of the print functions

```
cat(c("red", "yellow"), "lorry")
```

Usually, when strings are printed to the console they are shown wrapped in double quotes. By wrapping a variable in a call to the **noquote** function, we can suppress those quotes. This can make the text more readable in some instances:

```
print(noquote(c("I", "saw", "a", "saw", "that", "could", "hit", "me")))
```

# Formatting Numbers & String



24

Numbers and string can be formatted to a specific style using **format** function. The basic syntax for format function is –

```
format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))
```

Following is the description of the parameters used –

- ☐ x is the vector input.
- ☐ digits is the total number of digits displayed.
- ☐ nsmall is the minimum number of digits to the right of the decimal point.
- ☐ scientific is set to TRUE to display scientific notation.
- ☐ width indicates the minimum width to be displayed by padding blanks in the beginning.
- ☐ justify is the display of the string to left, right or center.



# Formatting Example



25



```
result <- format(23.123456789, digits = 9) # Total number of digits displayed. Last digit rounded off.
print(result)
result <- format(c(6, 13.14521), scientific = TRUE) # Display numbers in scientific notation.
print(result)
result <- format(23.47, nsmall = 5) # The minimum number of digits to the right of the decimal point.
print(result)
result <- format(6) # Format treats everything as a string.
print(result)
result <- format(13.7, width = 6) # Numbers are padded with blank in the beginning for width.
print(result)
result <- format("Hello", width = 8, justify = "l") # Left justify strings.
print(result)
result <- format("Hello", width = 8, justify = "c") # Justify string with center.
print(result)
```

# String Function





26

Function	Description	Syntax	Example
nchar()	Counts the number of characters including spaces in a string.	nchar(x) where x is the vector input.	<pre>result &lt;- nchar("Count me") print(result)</pre>
toupper() & tolower()	change the case of characters of a string.	<pre>toupper(x) tolower(x)</pre> where x is the vector input.	<pre>print(toupper("Changing To Upper")) print(tolower("Changing To Lower"))</pre>
substring()	extracts parts of a String	The basic syntax for substring() function is substring(x,first,last) where x is the character vector input. first is the position of the first character to be extracted. last is the position of the last character to be extracted.	<pre>print( substring("Extract", 5, 7))</pre>



**Read by yourself,  
find out ways to  
motivate yourself  
and recycle success**



# **Thank You**

# **End of Lab 3**

# Experiment



29

1. Print the class, type, mode, and storage mode of the following values: Inf, NA, NaN, ""
2. Randomly generate 1,000 pets, from the choices “dog,” “cat,” “hamster,” and “goldfish,” with equal probability of each being chosen. Display the first few values of the resultant variable, and count the number of each type of pet.
3. The beaver1 and beaver2 data frames contain body temperatures of two beavers. Add a column named id to the beaver1 dataset, where the value is always 1. Similarly, add an id column to beaver2, with value 2. Vertically concatenate the two data frames and find the subset where body temp is above 10.
4. Create a data frame to specify data on students given below:  
Roll number, Name, Department, Course, Year of joining
  - ☐ Write a function to print names of all students who joined in a particular year.
  - ☐ Write a function to print the data of a student whose roll number is given.

# Experiment cont...



30

5. Write an R-script to check and count the total no. of vowels within the given string.
6. Write an R-script to reverse a string and display that.
7. Write an R-script to extract a sub-string of 5 characters from the given string and replace that sub-string with "V-Day" within the original string.
8. Write an R-script to search a specific sub-string from a string and display its position. Then insert a new string at that position without changing anything
9. Write an R-script that replaces two or more consecutive blanks in a string by a single blank. For example, if the input is "Grim return to the planet of apes!!" the output should be "Grim return to the planet of apes!!"
10. Write an R-script that receives the month and year from the keyboard as integers and prints the calendar in the following format.

September 2004						
Mon	Tue	Wed	Thu	Fri	Sat	Sun
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

# Experiment cont...



31

11. Write an R-script to check the given vector is a factor or not? if not, then convert it into factor and display that with levels.
12. Write an R-script to create a vector having 10 names, where some names are repeated. Then convert it to a factor and display it to check the levels, then change the order of levels in alphabetical order and display it.
13. A factory has 3 division and stocks 4 categories of products. An inventory table is updated for each division and for each product as they are received. There are three independent suppliers of products to the factory:
  - ☐ Design a data format to represent each transaction.
  - ☐ Write a program to take a transaction and update the inventory.
  - ☐ If the cost per item is also given write a program to calculate the total inventory values.