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

Sr #	Major and Detailed Coverage Area	Lab#
1	Reading Datasets and Exporting Data	5
2	Manipulating and Processing Data	
3	Exploratory Data Analysis	
4	Debugging	
5	Error Handling & Recovery	

Getting Data

Data can come from many sources. R comes with many datasets built in, and there is more data in many of the add-on packages. R can read data from a wide variety of sources and in a wide variety of formats.

Getting and Setting the Working Directory

```
# Get and print current working directory.
print(getwd())

# Set current working directory.
setwd("/R/Practice")

# Get and print current working directory.
print(getwd())
```

CSV File

The csv file is a text file in which the values in the columns are separated by a comma. Let's consider the following data present in the file named input.csv. The file can be created using windows notepad by copying and pasting the data shown below. Save the file as input.csv using the **Save As** All files(*.*) option in notepad.

```
id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Michelle,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
5,Gary,843.25,2015-03-27,Finance
6,Nina,578,2013-05-21,IT
7,Simon,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance
```

CSV File cont'd

Reading the data

data <- read.csv("input.csv")

print(data)

Analysing the data

print(is.data.frame(data))

print(ncol(data))

print(nrow(data))

Processing the data

# Get the max salary from data frame.

sal <- max(data$salary)

print(sal)

#Get all the people working in IT department

retval <- subset(data, dept == "IT")

print(retval)

# Get the persons in IT department whose salary is greater than 600

info <- subset(data, salary > 600 & dept == "IT")

print(info)

# Get the details of the person with max salary

retval <- subset(data, salary == max(salary))

print(retval)

# Get the people who joined on or after 2014

retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

print(retval)

Lab Exercise

Create airquality.csv file with following content & develop solution for the given queries

```
ID,Ozone,Solar.R,Wind,Temp,Month,Day,Year
1,41,190,7.4,67,5,1,2018
2,36,118,8.0,72,3,2,2018
3,12,149,12.6,74,4,3,2018
4,18,313,11.5,62,5,4,2018
5,NA,NA,14.3,56,5,5,2018
```

Queries

1. Get the max temp
2. Get the minimum temp
3. Get the average temp
4. Get all the air quality data for 5th month
5. Get all the air quality data whose Ozone is not supplied
6. Count the number of rows and columns
7. Get the air quality recorded after 2nd Feb 2018

CSV File cont'd

Writing into a CSV File

R can create csv file form existing data frame. The write.csv() function is used to create the csv file. This file gets created in the working directory.

```
# Create a data frame.
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.
write.csv(retval,"output.csv") # write.csv(retval,"output.csv", row.names = FALSE) to drop column X

# Create a data frame for read operation
newdata <- read.csv("output.csv")
print(newdata)
```

TSV File

Copy and paste the following data in the TSV file “input.tsv”. You should save it in the current working directory of the R workspace.

id	name	salary	start_date	dept
1	Rick	623.3	1/1/2012	IT
2	Dan	515.2	9/23/2013	Operations
3	Michelle	611	11/15/2014	IT
4	Ryan	729	5/11/2014	HR
5	Gary	43.25	3/27/2015	Finance
6	Nina	578	5/21/2013	IT
7	Simon	632.8	7/30/2013	Operations
8	Guru	722.5	6/17/2014	Finance

1. Get the max salary for each dept
2. Get the minimum salary for each dept
3. Get the average salary for each dept

Data Management – Sorting

To sort a data frame in R, use the order( ) function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order.

```
# sorting examples using the mtcars dataset
attach(mtcars)

# sort by mpg
newdata <- mtcars[order(mpg),]

# sort by mpg and cyl
newdata <- mtcars[order(mpg, cyl),]

#sort by mpg (ascending) and cyl (descending)
newdata <- mtcars[order(mpg, -cyl),]

detach(mtcars)
```

Data Management – Missing Data

- Use na.omit() to remove missing data from a dataset
- Use na.fail()to signal an error if a dataset contains NA
- complete.cases() returns a logical vector indicating which rows have no missing data

Example:

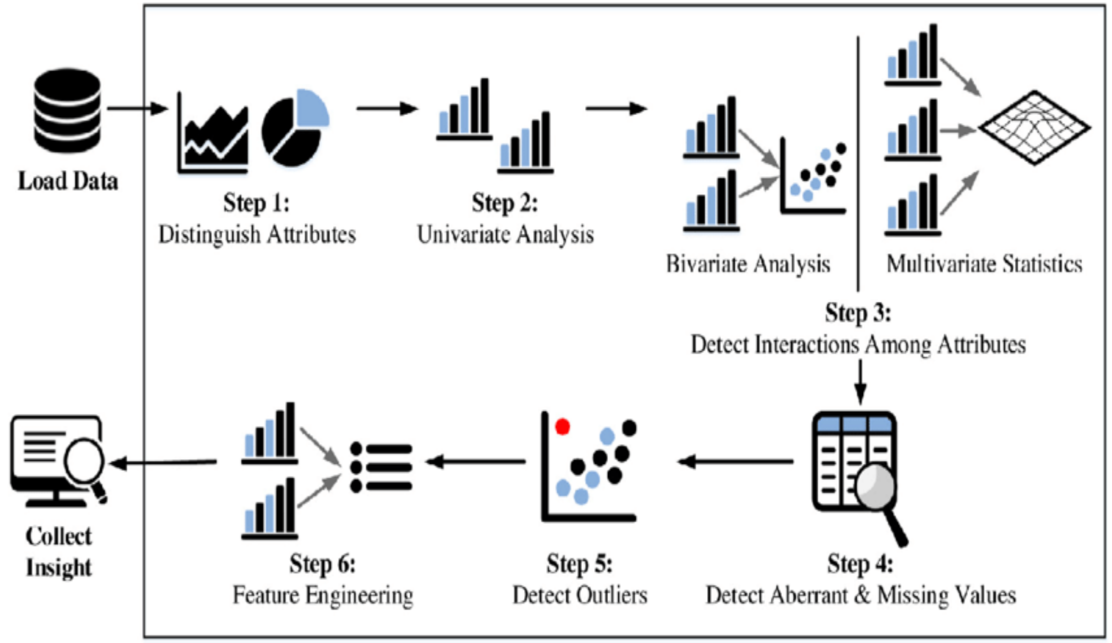
```
data <- read.csv("input.csv")
print(data)
na.omit(data) # Remove all rows with missing data
na.fail(data)  # Return an error message if missing data
sum(complete.cases(data))  # Get the number of complete cases
sum(!complete.cases(data)) # Get the number of incomplete cases
```

Data Management – Duplicates

Self-study on removal of duplicates (rows or columns or both) in a data frame

Exploratory Data Analysis

- Exploratory Data Analysis (EDA) refers to the critical process of performing initial investigations on data so as to discover patterns to spot anomalies, to test hypothesis and to check assumptions with the help of **summary statistics** and **graphical representations**.
- It is a good practice to understand the data first and try to gather as many insights from it. EDA is all about making sense of data in hand before getting them dirty with it.



Introduction

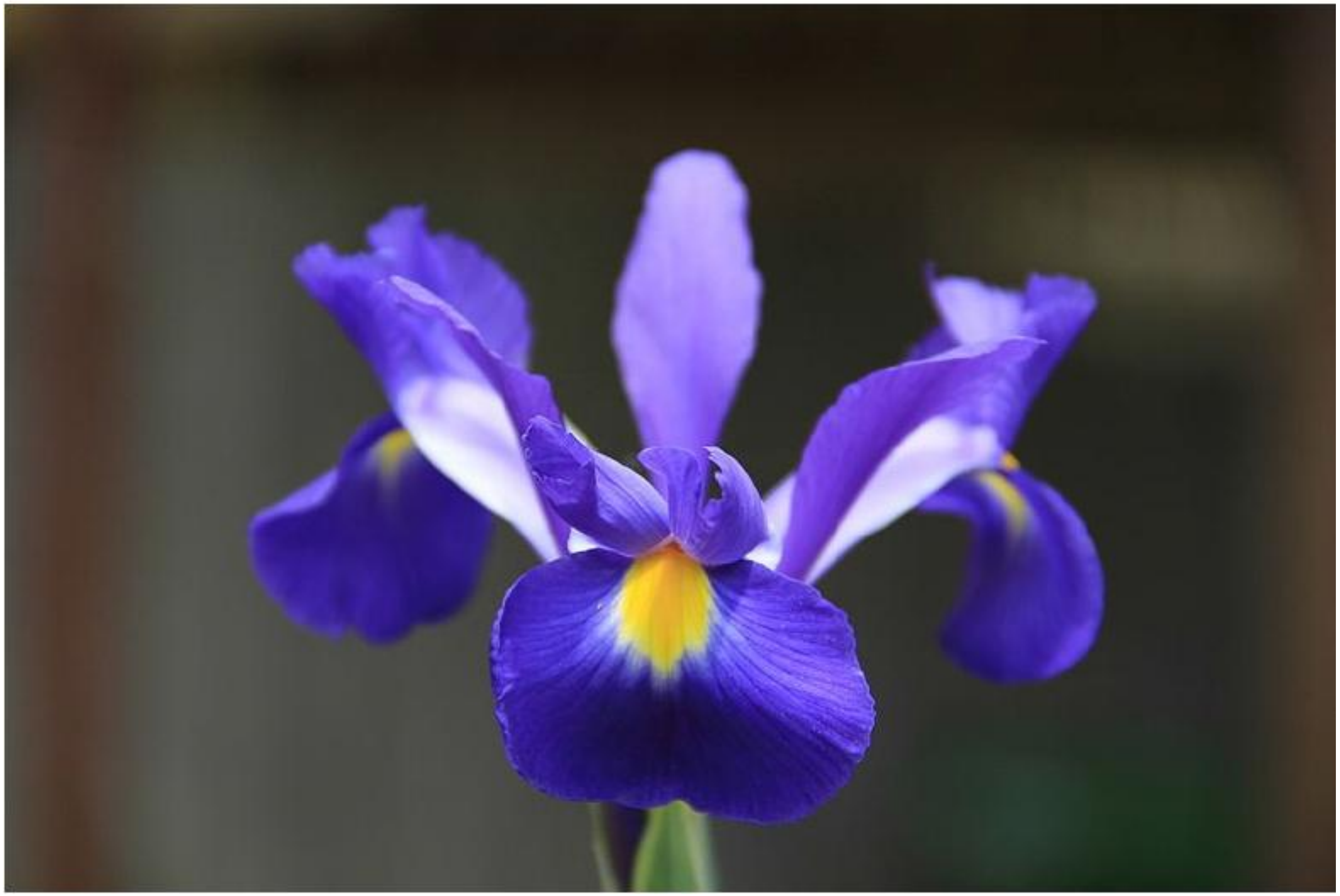
Data exploration is an approach that uses visual exploration to understand what is in a dataset and the characteristics of the data, rather than through traditional data management systems. These characteristics can include size or amount of data, completeness of the data, correctness of the data, possible relationships amongst data elements or files/tables in the data.

Data exploration can also refer to the ad hoc querying and visualization of data to identify potential relationships or insights that may be hidden in the data.

Once this initial understanding of the data is had, the data can be pruned or refined by removing unusable parts of the data (data cleansing), correcting poorly formatted elements and defining relevant relationships across datasets. This process is also known as determining data quality.

Data exploration has become an area of interest in the field of machine learning. By employing machine learning, it is possible to find patterns or relationships in the data that would be difficult or impossible to find via manual inspection, trial and error or traditional exploration techniques.

What's the Name of This Flower?



The Iris Dataset

The iris dataset consists of 50 samples from each of three classes of iris flowers. There are five attributes in the dataset:

- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm
- class: Iris Setosa, Iris Versicolour, and Iris Virginica

Have a look at the data

```
# number of rows
nrow(iris)

# number of columns
ncol(iris)

# dimensionality
dim(iris)

# column names
names(iris)

# structure of the dataset
str(iris)
```

```
# attributes of the data
attributes(iris)

# First 10 rows of dataset
head(iris, 10)

# Last 10 rows of dataset
tail(iris, 10)

# The first 10 values of Sepal.Length
iris[1:10, "Sepal.Length"]
```

OR

```
iris$Sepal.Length[1:10]
```

Explore Individual Variables

Distribution of every numeric variable can be checked with function summary(), which returns the minimum, maximum, mean, median, and the first (25%) and third (75%) quartiles. For factors (or categorical variables), it shows the frequency of every level.

```
summary(iris)
```

The mean, median and range can also be obtained with functions with mean(), median() and range().

```
mean(iris$Sepal.Length)
median(iris$Sepal.Length)
range(iris$Sepal.Length)
```

Quartiles and percentiles are supported by function quantile()

```
quantile(iris$Sepal.Length)
quantile(iris$Sepal.Length, c(.1, .3, .65))
```

Explore Individual Variables cont…

We can check the variance of Sepal.Length with var(), and also check its distribution with histogram and density using functions hist() and density().

```
var(iris$Sepal.Length)
hist(iris$Sepal.Length)
plot(density(iris$Sepal.Length))
```

The frequency of factors can be calculated with function table(), and then plotted as a pie chart with pie() or a bar chart with barplot().

```
table(iris$Species)
pie(table(iris$Species))
barplot(table(iris$Species))
```

Explore Multiple Variables

After checking the distributions of individual variables, we then investigate the relationships between two variables. Below we calculate covariance and correlation between variables with cov() and cor().

```
# covariance
cov(iris$Sepal.Length, iris$Petal.Length)
cov(iris[,1:4])
# correlation
cor(iris$Sepal.Length, iris$Petal.Length)
cor(iris[,1:4])
```

Next, we compute the stats of Sepal.Length of every Species with aggregate().

```
aggregate(Sepal.Length ~ Species, summary, data=iris)
```

Explore Multiple Variables cont…

We then use function boxplot() to plot a box plot, also known as box-and-whisker plot, to show the median, first and third quartile of a distribution (i.e., the 50%, 25% and 75% points in cumulative distribution), and outliers.

The bar in the middle is the median. The box shows the interquartile range (IQR), which is the range between the 75% and 25% observation.

```
boxplot(Sepal.Length~Species, data=iris)
```

A scatter plot can be drawn for two numeric variables with plot() as below. Using function with(), we don't need to add "iris" before variable names. In the code below, the colors (col) and symbols (pch) of points are set to Species.

```
with(iris, plot(Sepal.Length, Sepal.Width, col=Species,
pch=as.numeric(Species)))
```

When there are many points, some of them may overlap. We can use jitter() to add a small amount of noise to the data before plotting.

```
plot(jitter(iris$Sepal.Length), jitter(iris$Sepal.Width))
```

More Explorations

A matrix of scatter plots can be produced with function pairs().

```
pairs(iris)
```

A 3D scatter plot can be produced with package scatterplot3d

```
library(scatterplot3d)
scatterplot3d(iris$Petal.Width, iris$Sepal.Length, iris$Sepal.Width)
```

Package rgl supports interactive 3D scatter plot with plot3d()

```
library(rgl)
plot3d(iris$Petal.Width, iris$Sepal.Length, iris$Sepal.Width)
```

A heat map presents a 2D display of a data matrix, which can be generated with heatmap() in R. With the code below, we calculate the similarity between different owers in the iris data with dist() and then plot it with a heat map.

```
distMatrix <- as.matrix(dist(iris[,1:4]))
heatmap(distMatrix)
```

More Explorations

A level plot can be produced with function levelplot() in package lattice. Function grey.colors() creates a vector of gamma-corrected gray colors. A similar function is rainbow(), which creates a vector of contiguous colors.

```
library(lattice)
levelplot(Petal.Width~Sepal.Length*Sepal.Width, iris, cuts=9, col.regions=grey.colors(10)[10:1])
```

A 3D scatter plot can be produced with package scatterplot3d

```
library(scatterplot3d)
scatterplot3d(iris$Petal.Width, iris$Sepal.Length, iris$Sepal.Width)
```

Package rgl supports interactive 3D scatter plot with plot3d()

```
library(rgl)
plot3d(iris$Petal.Width, iris$Sepal.Length, iris$Sepal.Width)
```

A heat map presents a 2D display of a data matrix, which can be generated with heatmap() in R. With the code below, we calculate the similarity between different flowers in the iris data with dist() and then plot it with a heat map.

```
distMatrix <- as.matrix(dist(iris[,1:4]))
heatmap(distMatrix)
```

More Explorations cont…

In order to visualize multiple dimensions, parallel coordinates can be used.

```
library(MASS)
parcoord(iris[1:4], col = iris$Species)
```

Parallel Coordinates can also be visualize with Package lattice

```
library(lattice)
parallelplot(~iris[1:4] | Species, data = iris)
```

Package ggplot2 supports complex graphics, which are very useful for exploring data.

```
library(ggplot2)
qplot(Sepal.Length, Sepal.Width, data=iris, facets=Species ~.)
```

Debugging

In computer programming, *debugging is a multi-step process which involves identifying a problem, isolating the source of the problem, and then fixing the problem or determining a way to work around it*. *The final step of debugging is to test an improvement or workaround and ensure that it works.*

The grammatically correct program may give us incorrect results due to some logical errors which are known as "**bug**." In case, if such errors occur, then we need to find out why and where they have occurred so that we can fix them. The procedure to identify and fixing bugs are called "**debugging**.“

Like any program, R will occasionally produce errors which are not easily understood. The classical method of debugging is to insert print statements at appropriate locations, which can be time consuming and inefficient.

Figuring Out What's Wrong

The primary task of debugging any R code is correctly diagnosing what the problem is. When diagnosing a problem with the code, it's important first to understand what you were expecting to occur. Then you need to identify what did occur and how did it deviate from your expectations. Some basic questions you need to ask are:

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

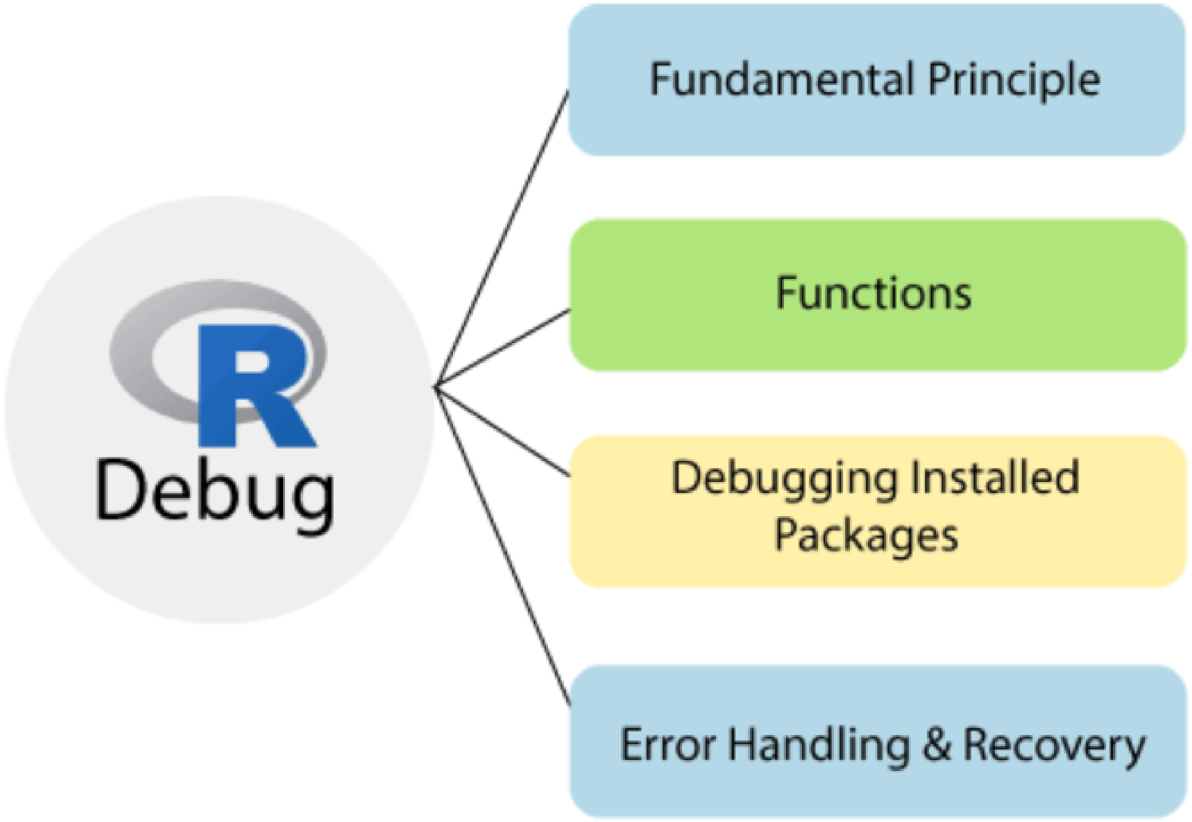
Being able to answer these questions is important not just for your own sake, but in situations where you may need to ask someone else for help with debugging the problem. Seasoned programmers will be asking you these exact questions.

Execution of Function

Executing any function in R may result in the following conditions.

- **message**: A generic notification/diagnostic message. It is produced by the message() function and the execution of the function continues
- **warning:** An indication that something is wrong but not necessarily fatal. Execution of the function continues. Warnings are generated by the warning() function
- **error**: An indication that a fatal problem has occurred and execution of the function stops. Errors are produced by the stop() function.
- **condition**: A generic concept for indicating that something unexpected has occurred. Programmers can create their own custom conditions if they want.

R Debug


R
Debug
Fundamental Principle
Functions
Debugging Installed Packages
Error Handling & Recovery

Fundamental principles of Debugging

R programmers find that they spend more time in debugging of a program than actually writing it or code it. This makes debugging skills less valuable. In R, there are various principles of debugging which help the programmers to spend their time in writing and coding rather than in debugging. These principles are as follows:

- The essence of debugging
- Start Small
- Debug in a Modular
- Antibugging

Fundamental principles of Debugging cont…

The essence of debugging

Fixing a bugging program is a process of confirming, one by one, that many things you believe to be true about code are actually true. When we find one of our assumptions is not true, we have found a clue to the location of a bug.

Start Small

Stick to small simple test cases, at least at the beginning of the R debug process. Working with large data objects may make it harder to think about the problem. Of course, we should eventually test our code in large, complicated cases, but start small.

Debug in Modular

Most professional software developers agree that the code should be written in a modular manner. This makes the code easier to write and helps others understand when the time comes to extend the code. We should debug in a top-down manner.

Fundamental principles of Debugging cont…

Antibugging

If we have a section of a code in which a variable x should be positive, then we can insert this line:

```
stopifnot(x>0)
```

If there is a bug in the code earlier that renders x equals to, say -3, the call to stopifnot() will bring things right there, with an error message like this:

```
Error: x > 0 is not TRUE
```

R Debug Function

- traceback()
- debug()
- browser()
- trace()
- recover()

Debugging Installed Packages

There are possibilities of an error stemming from an installed R package. Some of the various ways to solve this problem are:

- ❑ Setting options(error = recover) and then going line by line through the code using n.
- ❑ When facing complicated problems, you can have a copy of the function code with you. Entering function name in R console will print out function code that can be copied into the text editor. You can then edit this, load it into the global workspace and then perform debugging.
- ❑ If our problems are not solved, then we have to download the source code. We can also use the devtools package and the install(), load_all() functions to make our procedure quicker.

Error Handling & Recovery

Exception or Error handling is a process of responding to anomalous occurrences in the code that disrupt the flow of the code. In general, the scope for the exception handlers begins with try and ends with a catch. R provides try() and trycatch() function for the same.

The try() function is a wrapper function for trycatch() which prints the error and then continues. On the other hand, trycatch() gives you the control of the error function and also optionally, continues the process of the function.

Example 1

```
x <- 3
try(x > 5)
```

Output:

```
[1] FALSE
```

Example 2

```
x <- 3
tryCatch(x>5,error=print("error"))
```

Output:

```
[1] “error”
[1] FALSE
```

Defensive programming

Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs. A key principle of defensive programming is to "**fail fast**" as soon as something wrong is discovered, signal an error. This is more work for the developer (you!), but it makes debugging easier for users because they get errors earlier rather than later, after unexpected input has passed through several functions. In R, the "fail fast" principle is implemented in three ways:

- Be strict about what you accept. For example, if your function is not vectorised in its inputs, but uses functions that are, make sure to check that the inputs are scalars. You can use stopifnot(), or simple if statements and stop().
- Avoid functions that use non-standard evaluation, like subset, transform, and with. These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages
- Avoid functions that return different types of output depending on input.

Thank You
End of Lab 5

Lab Experiment

1. Create a CSV file as Student.csv having 5 columns as rollno, name, branch, percentage and DOA with 10 observations. Now read the Student.csv file to the R- workspace and display that.
2. Retrieve and display the details of that student who has maximum percentage.
3. Retrieve and display the details of those students who are studying in IT or CSE branch.
4. Retrieve and display the details of those CSE students whose percentage is more than or equal to 80.
5. Retrieve and display the details of those students who are admitted on or after 1st May 2021.
6. Create a TSV file Student.tsv with 10 observations, which contains rollno, name, branch, percentage. Read the file and display the duplicate percentage.

Lab Experiment cont…

7. With any in-built 10 dataset , WAP to:
   - Have a look at the data
   - Explore Individual Variables
   - Explore Multiple Variables
   - More Explorations

Catch necessary errors and follow “fail fast” approach.