Semester – IV

18CAU404A **R PROGRAMMING** **3H – 3C**

Instruction Hours / week: L:3 T: 0 P: 0 **Marks:** Int : **40** Ext : **60** Total: **100**

End Semester Exam: 3 Hours

Course Objectives :

Upon successful completion of this course, students will be able to:

- To acquire the computing tasks such as using conditional processing statements, loops, and writing one's own functions.
- Perform basic and advanced graphing of data.
- Use statistical distribution functions in R
- Perform basic statistical modeling of data

Course Outcome:

- Learn how to install and configure software necessary for a statistical programming environment.
- Discuss generic programming language concepts as they are implemented in a high-level statistical language.
- The course covers practical issues in statistical computing which includes programming in R, reading data into R, accessing R packages, writing R functions, debugging, and organizing and commenting R code.

UNIT-I

History and Overview of R : The S Philosophy - Back to R -Basic Features of R - Free Software -Design of the R System - Limitations of R- R Resources .Getting Started with R :Installation - Getting started with the R interface -.R Nuts and Bolts :Entering Input - Evaluation -R Objects - Numbers - Attributes - Creating Vectors - Mixing Objects - Explicit Coercion - Matrices - Lists -Factors - Missing Values - Data Frames - Names .

UNIT-II

Getting Data In and Out of R :Reading and Writing Data - Reading Data Files with read.table() - Reading in Larger Datasets with read.table - Calculating Memory Requirements for R Objects . Using the readr Package .Using Textual and Binary Formats for Storing Data :Using dput() and dump() – Binary Formats - Interfaces to the Outside World : File Connections - Reading Lines of a Text File - Reading From a URL Connection - Subsetting R Objects :Subsetting a Vector - Subsetting a Matrix - Subsetting Lists - Subsetting Nested Elements of a List - Extracting Multiple Elements of a List - Partial Matching -Removing NA Values .

UNIT-III

Vectorized Operations :Vectorized Matrix Operations .Dates and Times :Dates in R - Times in R - Operations on Dates and Times .Managing Data Frames with the dplyr package :Data Frames -The dplyr Package - dplyr Grammar - Installing the dplyr package
- select() - filter() -arrange() - rename() - mutate() - group_by()-%>%.Control Structures :if-else - for Loops - Nested for loops - while Loops - repeat Loops - next, break .

UNIT-IV

Functions: Functions in R - Your First Function - Argument Matching - Lazy Evaluation – The Argument - Arguments Coming After the Argument .Scoping Rules of R : A Diversion on Binding Values to Symbol - Scoping Rules - Lexical Scoping: Why Does It Matter? -Lexical vs. Dynamic Scoping -- Application: Optimization - Plotting the Likelihood. Coding Standards for R .Loop Functions : Looping on the Command Line - lapply() - sapply() - split() - Splitting a Data Frame - tapply - apply() - Col/Row Sums and Means -Other Ways to Apply - mapply()- Vectorizing a Function .

UNIT-V

Debugging -:Something's Wrong! - Figuring Out What's Wrong - Debugging Tools in R . Using traceback() - Using debug() - Using recover().Profiling R Code: Using system.time() . Timing Longer Expressions - The R Profiler - Using summaryRprof().Simulation :Generating Random Numbers - Setting the random number seed -Simulating a Linear Model - Random Sampling .

Suggested Readings

1. Daniel Navarro, (2013). *Learning Statistics with R*. University of Adelaide Publications.
2. Hadley Wickham, (2014). *Advanced R Programming*, (1st ed.)
3. Jeffrey Stanton, (2013). *Introduction to Data Science, with Introduction to R*, Version 3 ,
4. Roger.D.Peng, (2015). *R Programming for Data Science*

UNIT-I

SYLLABUS

History and Overview of R: The S Philosophy - Back to R -Basic Features of R – Free Software -Design of the R System - Limitations of R- R Resources. **Getting Started with R:** Installation - Getting started with the R interface -.**R Nuts and Bolts:** Entering Input -Evaluation -R Objects - Numbers - Attributes - Creating Vectors - Mixing Objects -Explicit Coercion - Matrices -Lists -Factors - Missing Values - Data Frames - Names.

HISTORY AND OVERVIEW OF R

- **What is R?**

  - R is a dialect of S.
  - It is a sophisticated computer language and environment for statistical analysis and graphics.

- **What is S?**

  - S is a language that was developed by John Chambers and others at the old Bell Telephone Laboratories, originally part of AT&T Corp.
  - S was initiated in 1976 as an internal statistical analysis environment—originally implemented as FORTRAN libraries.
  - Early versions of the language did not even contain functions for statistical modeling.
  - In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language).
  - The book Statistical Models *in S* by Chambers and Hastie (the white book) documents the statistical analysis functionality. Version 4 of the S language was released in 1998 and is the version we use today.
  - The book Programming with Data by John Chambers (the green book) documents this version of the language.
  - Since the early 90's the life of the S language has gone down a rather winding path. In 1993 Bell Labs gave StatSci (later Insightful Corp.) an exclusive license to develop and sell the S language. In 2004 Insightful purchased the S language from Lucent for $2 million. In 2006, Alcatel purchased Lucent Technologies and is now called Alcatel-Lucent.
  - Insightful sold its implementation of the S language under the product name S-PLUS and built a number of fancy features (GUIs, mostly) on top of it—hence the ‑PLUS‖. In 2008 Insightful was acquired by TIBCO for $25 million. As of this writing TIBCO is the current owner of the S language and is its exclusive developer.
  - The fundamental of the S language itself has not changed dramatically since the publication of the Green Book by John Chambers in 1998. In 1998, S won the

Association for Computing Machinery's Software System Award, a highly prestigious award in the computer science field.

THE S PHILOSOPHY

- The general S philosophy is important to understand for users of S and R because it sets the stage for the design of the language itself, which many programming veterans find a bit odd and confusing.
- In particular, it's important to realize that the S language had its roots in data analysis, and did not come from a traditional programming language background.
- Its inventors were focused on figuring out how to make data analysis easier, first for themselves, and then eventually for others.
- The key part here was the transition from *user* to *developer*. They wanted to build a language that could easily service both ―people‖.
- More technically, they needed to build language that would be suitable for interactive data analysis (more command-line based) as well as for writing longer programs (more traditional programming language-like).

BACK TO R

- The R language came to use quite a bit after S had been developed. One key limitation of the S language was that it was only available in a commercial package, S-PLUS.
- In 1991, R was created by Ross Ihaka and Robert Gentleman in the Department of Statistics at the University of Auckland. In 1993 the first announcement of R was made to the public.
- In 1995, Martin Mächler made an important contribution by convincing Ross and Robert to use the GNU General Public License to make R free software. This was critical because it allowed for the source code for the entire R system to be accessible to anyone who wanted to tinker with it (more on free software later).
- In 1996, a public mailing list was created (the R-help and R-devel lists) and in 1997 the R Core Group was formed, containing some people associated with S and S-PLUS. Currently, the core group controls the source code for R and is solely able to check in changes to the main R source tree. Finally, in 2000 R version 1.0.0 was released to the public.

BASIC FEATURES OF R

- In the early days, a key feature of R was that its syntax is very similar to S, making it easy for S-PLUS users to switch over. While the R's syntax is nearly identical to that of S's, R's semantics, while superficially similar to S, are quite different.
- In fact, R is technically much closer to the Scheme language than it is to the original S language when it comes to how R works under the hood.
- Today R runs on almost any standard computing platform and operating system. Its open source nature means that anyone is free to adapt the software to whatever platform they choose. Indeed, R has been reported to be running on modern tablets, phones, PDAs, and game consoles.
- One nice feature that R shares with many popular open source projects is frequent releases. These days there is a major annual release, typically in October, where major new features are incorporated and released to the public.
- Throughout the year, smaller-scale bugfix releases will be made as needed. The frequent releases and regular release cycle indicates active development of the software and ensures that bugs will be addressed in a timely manner.
- Of course, while the core developers control the primary source tree for R, many people around the world make contributions in the form of new feature, bug fixes, or both.
- Another key advantage that R has over many other statistical packages (even today) is its sophisticated graphics capabilities.
- R's ability to create ‑publication quality‖ graphics has existed since the very beginning and has generally been better than competing packages. Today, with many more visualization packages available than before, that trend continues. R's base graphics system allows for very fine control over essentially every aspect of a plot or graph.
- Other newer graphics systems, like lattice and ggplot2 allow for complex and sophisticated visualizations of high-dimensional data.
- R has maintained the original S philosophy, which is that it provides a language that is both useful for interactive work, but contains a powerful programming language for developing new tools. This allows the user, who takes existing tools and applies them to data, to slowly but surely become a developer who is creating new tools.
- Finally, one of the joys of using R has nothing to do with the language itself, but rather with the active and vibrant user community. In many ways, a language is successful inasmuch as it creates a platform with which many people can create new things. R is that platform and thousands of people around the world have come together to make contributions to R, to develop packages, and help each other use R for all kinds of

applications. The R-help and R-devel mailing lists have been highly active for over a decade now and there is considerable activity on web sites like Stack Overflow.

FREE SOFTWARE

- A major advantage that R has over many other statistical packages and is that it's free in the sense of frees software (it's also free in the sense of free beer). The copyright for the primary source code for R is held by the R Foundation and is published under the GNU General Public License version.
- According to the Free Software Foundation, with free software, you are granted the following four freedoms
  - The freedom to run the program, for any purpose (freedom 0).
  - The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
  - The freedom to redistribute copies so you can help your neighbor (freedom 2).
  - The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

DESIGN OF THE R SYSTEM

- The primary R system is available from the Comprehensive R Archive Network, also known as CRAN. CRAN also hosts many add-on packages that can be used to extend the functionality of R.
- The R system is divided into 2 conceptual parts:
  1. The –base‖ R system that you download from CRAN: Linux, Windows, Mac Source Code
  2. Everything else.
- R functionality is divided into a number of packages.
  - The –base‖ R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.
  - The other packages contained in the –base‖ system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.There are also –Recommended‖ packages: boot, class, cluster, codetools,

foreign, KernS-mooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix.

- When you download a fresh installation of R from CRAN, you get all of the above, which represents a substantial amount of functionality. However, there are many other packages available:
  - There are over 4000 packages on CRAN that have been developed by users and programmers around the world.
  - There are also many packages associated with the Bioconductor project.
  - People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.
  - There are a number of packages being developed on repositories like GitHub and BitBucket but there is no reliable listing of all these packages.

LIMITATIONS OF R

- No programming language or statistical analysis system is perfect. R certainly has a number of drawbacks. For starters, R is essentially based on almost 50 year old technology, going back to the original S system developed at Bell Labs.
- There was originally little built in support for dynamic or 3-D graphics (but things have improved greatly since the ‑old days‖).
- Another commonly cited limitation of R is that objects must generally be stored in physical memory. This is in part due to the scoping rules of the language, but R generally is more of a memory hog than other statistical packages.
- However, there have been a number of advancements to deal with this, both in the R core and also in a number of packages developed by contributors.
- Also, computing power and capacity has continued to grow over time and amount of physical memory that can be installed on even a consumer-level laptop is substantial. While we will likely never have enough physical memory on a computer to handle the increasingly large datasets that are being generated, the situation has gotten quite a bit easier over time.
- At a higher level one ‑limitation‖ of R is that its functionality is based on consumer demand and (voluntary) user contributions. If no one feels like implementing your favorite method, then it’s your job to implement it (or you need to pay someone to do it). The capabilities of the R system generally reflect the interests of the R user community.

As the community has ballooned in size over the past 10 years, the capabilities have similarly increased.

- When I first started using R, there was very little in the way of functionality for the physical sciences (physics, astronomy, etc.). However, now some of those communities have adopted R and we are seeing more code being written for those kinds of applications.

GETTING STARTED WITH R

INSTALLATION

- The first thing you need to do to get started with R is to install it on your computer. R works on pretty much every platform available, including the widely available Windows, Mac OS X, and Linux systems.

  - Installing R on Windows
  - Installing R on the Mac

- There is also an integrated development environment available for R that is built by RStudio. I really like this IDE—it has a nice editor with syntax highlighting, there is an R object viewer, and there are a number of other nice features that are integrated. You can see how to install RStudio here

  • Installing RStudio

- The RStudio IDE is available from RStudio's web site.

GETTING STARTED WITH THE R INTERFACE

- After you install R you will need to launch it and start writing R code. Before we get to exactly how to write R code, it's useful to get a sense of how the system is organized. In these two videos I talk about where to write code and how set your working directory, which let's R know where to find all of your files.
  - Writing code and setting your working directory on the Mac
  - Writing code and setting your working directory on Windows

R NUTS AND BOLTS

ENTERING INPUT

- At the R prompt we type expressions. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

- The grammar of the language determines whether an expression is complete or not.

```
x <- ## Incomplete expression
```

- The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

EVALUATION

- When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5    ## nothing printed
> x## auto-printing occurs
[1] 5
> print(x) ## explicit printing
[1] 5
```

- The [1] shown in the output indicates that x is a vector and 5 is its first element.
- Typically with interactive work, we do not explicitly print objects with the print function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.
- When an R vector is printed you will notice that an index for the vector is printed in square brackets [] on the side. For example, see this integer sequence of length 20.

```
> x <- 10:30
> x
 [1] 10 11 12 13 14 15 16 17 18 19 20
21 [13] 22 23 24 25 26 27 28 29 30
```

- The numbers in the square brackets are not part of the vector itself; they are merely part of the printed output.
- With R, it's important that one understand that there is a difference between the actual R object and the manner in which that R object is printed to the console.
- Often, the printed output may have additional bells and whistles to make the output friendlier to the users. However, these bells and whistles are not inherently part of the object.
- Note that the **":"** operator is used to create integer sequences.

R OBJECTS

- R has five basic or –atomic‖ classes of objects:
  - character
  - numeric (real numbers)
  - integer
  - complex
  - logical (True/False)
- The most basic type of R object is a vector. Empty vectors can be created with the vector() function. There is really only one rule about vectors in R, which is that a vector can only contain objects of the same class.
- But of course, like any good rule, there is an exception, which is a list, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that's usually why we use them.
- There is also a class for –raw‖ objects, but they are not commonly used directly in data analysis and I won't cover them here.

NUMBERS

- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like –1‖ or –2‖ in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like –1.00‖ or –2.00‖). This isn't important most of the time…except when it is.
- If you explicitly want an integer, you need to specify the L suffix. So entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object.

- There is also a special number Inf which represents infinity. This allows us to represent entities like 1 / 0. This way, Inf can be used in ordinary calculations; e.g. 1 / Inf is 0.
- The value NaN represents an undefined value (—not a number‖); e.g. 0 / 0; NaN can also be thought of as a missing value (more on that later)

ATTRIBUTES

- R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are
  - names, dimnames
  - dimensions (e.g. matrices, arrays)
  - class (e.g. integer, numeric)
  - length
  - other user-defined attributes/metadata
- Attributes of an object (if any) can be accessed using the attributes() function. Not all R objects contain attributes, in which case the attributes() function returns NULL.

CREATING VECTORS

- The c() function can be used to create vectors of objects by concatenating things together.

```
x <- c(0.5, 0.6)                ## numeric
> x   <- c(TRUE, FALSE)         ## logical
> x   <- c(T, F)                ## logical
> x   <- c("a", "b", "c")       ## character
> x   <- 9:29                   ## integer
> x   <- c(1+0i, 2+4i)          ## Comple x
```

- Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values.
- You can also use the vector() function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

MIXING OBJECTS

- There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

```
> y <- c(1.7, "a")        ## character
> y  <- c( TRUE, 2)       ## numeric
> y  <- c("a", TRUE)      ## character
```

- In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.
- In the example above, we see the effect of implicit coercion. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

```
> x <- 0:6
> class(x)
  [1] "integer"
 > as.numeric(x) [1] 0 1 2 3 4 5 6
> as.logical(x)
  [1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
  > as.character(x)
  [1] "0" "1" "2" "3" "4" "5" "6"
```

EXPLICIT COERCION

- Objects can be explicitly coerced from one class to another using the as.* functions, if available.
- Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

```
>  x <- c("a", "b", "c")
> as.numeric(x)
```

Warning: NAs introduced by coercion [1] **NA NA NA**

```
>    as.logical(x)
```

```
[1] NA NA NA
> as.complex(x)
Warning: NAs introduced by coercion
[1] NA NA NA
```

- When nonsensical coercion takes place, you will usually get a warning from R.

MATRICES

- Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m) [1] 2 3
> attributes(m) $dim
[1] 2 3
```

- Matrices are constructed column-wise, so entries can be thought of starting in the –upper left‖ corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6 7 8 9 10
> dim(m) <- c(2, 5)
> m
     [, 1] [, 2] [, 3] [, 4] [, 5]
[1,]     1     3     5     7     9
[2,]     2     4     6     8    10
```

- Matrices can be created by column-binding or row-binding with the cbind() and rbind() functions.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
   x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
 > rbind(x, y)
    [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

LISTS

- Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various –apply‖ functions discussed later, make for a powerful combination.
- Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
  [[1]]
  [1] 1
  [[2]]
  [1]
  "a"

  [[3]]
  [1]
```

```
TRU
E
[[4]]
[1]
1+4i
```

- We can also create an empty list of a prespecified length with the vector() function

```
> x <- vector("list", length = 5)
> x
[[1]]
NULL
[[2]]
NULL
[[3]]
NULL
[[4]]
NULL
[[5]]
NULL
```

FACTORS

- Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a label. Factors are important in statistical modeling and are treated specially by modelling functions like lm() and glm().
- 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.
- Factor objects can be created with the factor() function.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
```

```
no yes
2 3
> ## See the underlying representation of factor
> unclass(x)
[1] 2 2 1 2
1
attr(,"levels
") [1] "no"
"yes"
```

- Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often default to creating factors when they encounter data that look like characters or strings.
- The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x ## Levels are put in alphabetical order
[1] yes yes no yes no Levels: no yes
> x <- factor(c("yes", "yes", "no",
"yes", "no"), + levels = c("yes", "no"))
> x
[1] yes yes no yes no Levels: yes no
```

MISSING VALUES

- Missing values are denoted by NA or NaN for q undefined mathematical operations.
  - is.na() is used to test objects if they are NA
  - is.nan() is used to test for NaN
  - NA values have a class also, so there are integer NA, character NA, etc.
  - A NaN value is also NA but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA
> is.na(x)
```

```
[1] FALSE FALSE TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE

> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
 [1]
FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
 [1]
FALSE FALSE TRUE FALSE FALSE
```

DATA FRAMES

- Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr has an optimized set of functions designed to work efficiently with data frames.
- Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.
- Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).
- In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame.
- Data frames are usually created by reading in a dataset using the read.table() or read.csv(). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists.
- Data frames can be converted to a matrix by calling data.matrix(). While it might seem that the as.matrix() function should be used to coerce a data frame to a matrix, almost always, what you want is the result of data.matrix().

```
>       x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
>x
```

```
foo   bar
1     1 TRUE
2     2 TRUE
3     3 FALSE
4     4 FALSE
>     nrow(x)
[1] 4
> ncol(x)
[1] 2
```

NAMES

- R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x
    New York      Seattle  Los Angeles
           1            2            3
> names(x)
[1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
$`Los
Angeles` [1]
1
$Boston
[1] 2
$London
[1] 3
> names(x)
[1] "Los Angeles" "Boston"    "London"
```

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

- Column names and row names can be set separately using the colnames() and rownames() functions.

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
  h f
x 1 3
z 2 4
```

- Note that for data frames, there is a separate function for setting the row names, the row.names() function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the names() function.

Object	Set column names	Set row names
data frame	names()	row.names()
matrix	colnames()	rownames(

- Matrices can have both column and row names.

POSSIBLE QUESTIONS

UNIT – I

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is R?
2. What is S?
3. Define Vector
4. List the types of Data objects.
5. Define Matrices
6. Define List
7. Define Factor
8. List the data types in R
9. Define Data frame
10. How to create names in R programming?

PART – C (6 MARKS)

1. Explain the history and overview of R
2. Explain the Basic Features of R programming
3. Explain about Design of the R System
4. Write in detail: (i) Limitations of R (ii) R Resources
5. Explain the steps involved in R installation
6. Explain the data types or R objects
7. Explain the types of Data objects in R
8. Explain how to create vectors with suitable example
9. Write in detail (i) matrices (ii) Data Frames
10. Explain how to create number and Attributes in R programming
11. Write a R program to demonstrate Operators

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2018 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - I : (Objective Type Multiple choice Questions each Question carries one Mark)

R PROGRAMMING [18CAU404A]

PART - A (Online Examination)

Questions	Opt1	Opt2	Opt3	Opt4	Key
_____programming language is a dialect of S.	B	C	R	K	R
Lucent for $2 million	Insightful	Amazon	IBM	Google	Insightful
In 1991, R was created by Ross Ihaka and Robert Gentleman in the Department of Statistics at the University of _____	John Hopkins	California	Harvard	Auckland	Auckland
Finally, in_____R version 1.0.0 was released to the public.	2000	2005	2010	2012	2000
R is technically much closer to the Scheme language than it is to the original_____language.	B	C	C++	S	S
The R-help and_____mailing lists have been highly active for over a decade now	R-mail	R-devel	R-dev	Rcell	R-devel
Which of the following describes R language ?	Free	Paid	Available for free trial only	Trail	Free
The copyright for the primary source code for R is held by the_____Foundation.	A	S	C++	R	R
They primary R system is available from the	CRAN	CRWO	GNU	RAN	CRAN
R functionality is divided into a number of _____	Packages	Functions	Domains	Library	Packages

The_____R system contains, among other things, the base package which is required to run R and	root	child	base	private	base
Which of the following is a base package for R language ?	util	lang	tools	stats	tools
Which of the following is “Recommended” package in R ?	util	lang	stats	spatial	spatial
How many packages exist in R language for statistics ?	2000	3000	4000	5000	4000
Advanced users can write____code to manipulate R objects directly.	C	C++	Java	PHP	C
Which of the following is used for Statistical analysis in R language ?	RStudio	Studio	Heck	Rstat	RStudio
R has how many atomic classes of objects ?	1	3	5	2	5
Numbers in R are generally treated as _____ precision real numbers.	single	double	real	integer	double
If you explicitly want an integer, you need to specify the_____suffix.	D	R	L	T	L
R objects can have attributes, which are like _____for the object.	metadata	features	expression	data	metadata
What would be the result of following code ? > x<- 2 class(a)	"integer"	"numeric"	"logical"	"real"	"numeric"
Which of the following statement would print “0” “1” “2” “3” “4” “5” “6” for the following code ?	as.character(x)	as.logical(x)	as.numeric(x)	as.integer(x)	as.character(x)
Point out the wrong statement :	The grammar of the language determines whether an expression is complete or not	The <- symbol is the assignment operator in R	The ## character indicates a comment	The = symbol is also the assignment operator in R	The ## character indicates a comment

Files containing R scripts ends with extension :	.S	.R	.Rp	.RR	.R
Point out the wrong statement :	: operator is used to create integer sequences	The numbers in the square brackets are part of the vector itself	The numbers in the paranthesis are part of the vector itself	There is a difference between the actual R object and the manner in which that R object is printed to the console	The numbers in the square brackets are part of the vector itself
The entities that R creates and manipulates are known as _____	objects	task	container	function	objects
Which of the following can be used to display the names of (most of) the objects which are currently stored within R ?	object()	objects()	list()	data.frame()	objects()
Collection of objects currently stored in R is called as :	package	workspace	list	objects	workspace
R objects can have attributes, which are like _____for the object	data	metadata	list	package	metadata
Matrices can be created by column-binding or row-binding with the_____and _____ functions.	rowbind() and columnbind()	r_bind() and c_bind()	rbind() and cbind()	rowbind() and colbind()	rbind() and cbind()
_____are a special type of vector that can contain elements of different classes	factors	matrices	data frames	list	list
_____are used to represent categorical data and can be unordered or ordered	factors	matrices	data frames	list	factors
_____is used to test objects if they are NA	is.nan()	is.na()	na()	as.na()	is.na()
_____is used to test objects if they are NAN	is.nan()	is.na()	na()	as.na()	is.nan()
R objects can have_____, which is very useful for writing readable code and self-describing objects.	list	matrices	attributes	names	names

Column names and row names can be set separately using the_____and _____ functions.	colnames() and rownames()	cnames() and rnames()	col_names() and row_names()	columnnames() and rownames()	colnames() and rownames()
A_____can only contain objects of the same class.	list	vector	data frames	factor	vector
Point out the wrong statement :	Key feature of R was that its syntax is very similar to S	R runs only on Windows computing platform and operating system	R has been reported to be running on modern tablets, phones, PDAs, and game consoles		R runs only on Windows computing platform and operating system
Who developed S?	Dennis Ritchie	Bjarne Stroustrup	James Gosling	John Chambers	John Chambers
R is an Interpreted Language so it can access through _____	Disk Operating System	User Interface Operating System	Operating System	Command Line Interpreter	Command Line Interpreter
R supports_____arithmetic	logical	basic	matrix	vector	matrix
The sequence and number of observations in the vectors must be the same for each vector in the Data Frame to represent a _____	Record	Data object	Data	Data Sets	Data Sets
Matrices must have every element be the _____ class	same	different	literal	unique	same
Data frames can be converted to a matrix by calling _____	data.frame()	data()	data.matrix()	frame()	data.matrix()
Matrices are vectors with a_____attribute	type	nrow	dimension	ncol	dimension
The <- Symbol is the_____operator	Comparison Operator	Assignment Operator	Logical Operator	Boolean Operator	Assignment Operator
_____can store different classes of objects in each column	data frames	matrices	lists	factors	data frames

Factor objects can be created with the _____ function.	data()	factors()	fact()	factor()	factor()
Missing values are denoted by___or____for q undefined mathematical operations.	NA or NaN	NA or AS	Naan or No	N or Naa	NA or NaN
Objects can be explicitly coerced from one class to another using the_____functions	.(datatype)	as.*	.(datatype)as	as()	as.*
R does not support_____comments or comment blocks.	single line	*	multi line	//	multi line
Attributes of an object (if any) can be accessed using the_____function	attrib()	att()	attr()	attributes()	attributes()
Numbers in R are generally treated as _____ objects	integer	real	numeric	number	numeric
>m <- matrix(nrow = 2, ncol = 3) >m > attributes(m)	2 3	3 2	dim	NA	dim
_____function to find the data type of the variable	datatype()	class()	type()	cls()	class()
The_____Function get the current working directory	get()	getwd()	getw()	wd()	getwd()
To change current working directory use _____ function	set()	setw()	swd()	setwd()	setwd()
A_____is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length	data frames	list	factor	vector	factor
_____replicates the value	repl	rep	replicate	rep_c	rep
Which function is used to transpose data frame?	t()	ti()	transpose()	trans()	t()

UNIT-II

SYLLABUS

Getting Data In and Out of R: Reading and Writing Data - Reading Data Files with read.table() - Reading in Larger Datasets with read.table - Calculating Memory Requirements for R Objects . Using the readr Package .**Using Textual and Binary Formats for Storing Data:** Using dput() and dump() – Binary Formats - **Interfaces to the Outside World :** File Connections - Reading Lines of a Text File - Reading From a URL Connection - **Subsetting R Objects :**Subsetting a Vector - Subsetting a Matrix – Subsetting Lists - Subsetting Nested Elements of a List - Extracting Multiple Elements of a List - Partial Matching -Removing NA Values .

GETTING DATA IN AND OUT OF R

READING AND WRITING DATA

- There are a few principal functions reading data into R.
  - read.table, read.csv, for reading tabular data
  - readLines, for reading lines of a text file
  - source, for reading in R code files (inverse of dump)
  - dget, for reading in R code files (inverse of dput)
  - load, for reading in saved workspaces
  - unserialize, for reading single R objects in binary form
- There are of course, many R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.
- There are analogous functions for writing data to files
  - write.table, for writing tabular data to text files (i.e. CSV) or connections
  - writeLines, for writing character data line-by-line to a file or connection
  - dump, for dumping a textual representation of multiple R objects
  - dput, for outputting a textual representation of an R object
  - save, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
  - serialize, for converting an R object into a binary format for outputting to a connection (or file).

READING DATA FILES WITH read.table()

- The read.table() function is one of the most commonly used functions for reading data. The help file for read.table() is worth reading in its entirety if only because the function gets used a lot (run read.table in R).
- The read.table() function has a few important arguments:
  - file, the name of a file, or a connection
  - header, logical indicating if the file has a header line
  - sep, a string indicating how the columns are separated
  - colClasses, a character vector indicating the class of each column in the dataset
  - nrows, the number of rows in the dataset. By default read.table() reads an entire file.
  - comment.char, a character string indicating the comment character. This defalts to "#". If there are no commented lines in your file, it's worth setting this to be the empty string "".
  - skip, the number of lines to skip from the beginning.
  - stringsAsFactors, should character variables be coded as factors? This defaults to TRUE because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be FALSE in those cases. If you always want this to be FALSE, you can set a global option via options(stringsAsFactors = FALSE). I've never seen so much heat generated on discussion forums about an R function argument than the stringsAsFactors argument. Seriously.
  - For small to moderately sized datasets, you can usually call read.table without specifying any other argument.

```
> data <- read.table("foo.txt")
```

- In this case, R will automatically
  - skip lines that begin with a #
  - figure out how many rows there are (and how much memory needs to be allocated)
  - figure what type of variable is in each column of the table.

- Telling R all these things directly makes R run faster and more efficiently. The read.csv() function is identical to read.table except that some of the defaults are set differently (like the sep argument).

READING IN LARGER DATASETS WITH read.table

- Read the help page for read.table, which contains many hints

- Make a rough calculation of the memory required to store your dataset (see the next section for an example of how to do this). If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set comment.char = "" if there are no commented lines in your file.
- Use the colClasses argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set colClasses = "numeric". A quick an dirty way to figure out the classes of each column is the following:

```
> initial <- read.table("datatable.txt", nrows = 100)
> classes <- sapply(initial, class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

- In general, when using R with larger datasets, it's also useful to know a few things about your system.
  - How much memory is available on your system?
  - What other applications are in use? Can you close any of them?
  - Are there other users logged into the same system?
  - What operating system are you using? Some operating systems can limit the amount of memory a single process can access

CALCULATING MEMORY REQUIREMENTS FOR R OBJECTS

- Because R stores all of its objects physical memory, it is important to be cognizant of how much memory is being used up by all of the data objects residing in your workspace. One situation where it's particularly important to understand memory requirements is when you are reading in a new dataset into R. Fortunately, it's easy to make a back of the envelope calculation of how much memory will be required by a new dataset.
- For example, suppose I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame? Well, on most modern computers double precision floating point numbers[38] are stored using 64 bits of memory, or 8 bytes. Given that information, you can do the following calculation

$$1{,}500{,}000 \times 120 \times 8 \text{ bytes/numeric} = 1{,}440{,}000{,}000 \text{ bytes}$$
$$= 1{,}440{,}000{,}000 / 2^{20} \text{ bytes/MB}$$
$$= 1{,}373.29 \text{ MB}$$

= 1.34 GB

- So the dataset would require about 1.34 GB of RAM. Most computers these days have at least that much RAM. However, you need to be aware of

  - what other programs might be running on your computer, using up RAM
  - what other R objects might already be taking up RAM in your workspace
  - Reading in a large dataset for which you do not have enough RAM is one easy way to freeze up your computer (or at least your R session).

- This is usually an unpleasant experience that usually requires you to kill the R process, in the best case scenario, or reboot your computer, in the worst case. So make sure to do a rough calculation of memory requirements before reading in a large dataset

USING THE readr PACKAGE

- The readr package is recently developed by Hadley Wickham to deal with reading in large flat files quickly. The package provides replacements for functions like read.table() and read.csv(). The analogous functions in readr are read_table() and read_csv(). These functions are oven much faster than their base R analogues and provide a few other nice features such as progress meters.
- For the most part, you can read use read_table() and read_csv() pretty much anywhere you might use read.table() and read.csv(). In addition, if there are non-fatal problems that occur while reading in the data, you will get a warning and the returned data frame will have some information about which rows/observations triggered the warning. This can be very helpful for "debugging" problems with your data before you get neck deep in data analysis.

USING TEXTUAL AND BINARY FORMATS FOR STORING DATA

- There are a variety of ways that data can be stored, including structured text files like CSV or tab-delimited or more complex binary formats. However, there is an intermediate format that is textual, but not as simple as something like CSV. The format is native to R and is somewhat readable because of its textual nature.

- One can create a more descriptive representation of an R object by using the dput() or dump() functions. The dump() and dput() functions are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or CSV file, dump() and dput() preserve the metadata (sacrificing some readability), so that another user doesn't have to specify it all over again. For example, we can preserve the class of each column of a table or the levels of a factor variable.
- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files.
- There are a few downsides to using these intermediate textual formats. The format is not very space-efficient, because all of the metadata is specified. Also, it is really only partially readable. In some instances it might be preferable to have data stored in a CSV file and then have a separate code file that specifies the metadata.

USING dput() AND dump()

- One way to pass data around is by deparsing the R object with dput() and reading it back in (parsing it) using dget().

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
    structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")),
      .Names\ = c("a",
    "b"), row.names = c(NA, -1L), class = "data.frame")
```

- Notice that the dput() output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names.
- The output of dput() can also be saved directly to a file.

```
> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file
> new.y <- dget("y.R")
> new.y
```

```
  a b
1 1 a
```

- Multiple objects can be deparsed at once using the dump function and read back in using source.

```
> x <- "foo"
> y <- data.frame(a = 1L, b = "a")
```

- We can dump() R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
```

- The inverse of dump() is source().

```
> source("data.R")
> str(y)
'data.frame': 1 obs. of 2 variables:
 $ a: int 1
 $ b: Factor w/ 1
level "a": 1 > x
[1] "foo"
```

BINARY FORMATS

- The complement to the textual format is the binary format, which is sometimes necessary to use for efficiency purposes, or because there's just no useful way to represent data in a textual manner. Also, with numeric data, one can often lose precision when converting to and from a textual format, so it's better to stick with a binary format.
- The key functions for converting R objects into a binary format are save(), save.image(), and serialize(). Individual R objects can be saved to a file using the save() function.

```
> a <- data.frame(x = rnorm(100), y = runif(100))
> b <- c(3, 4.4, 1 / 3)

> ## Save 'a' and 'b' to a file
> save(a, b, file = "mydata.rda")
>
> ## Load 'a' and 'b' into your workspace
> load("mydata.rda")
```

- If you have a lot of objects that you want to save to a file, you can save all objects in your workspace using the save.image() function.

```
> ## Save everything to a file
> save.image(file = "mydata.RData")
>
> ## load all objects in this file
> load("mydata.RData")
```

- Notice that I've used the .rda extension when using save() and the .RData extension when using save.image(). This is just my personal preference; you can use whatever file extension you want. The save() and save.image() functions do not care. However, .rda and .RData are fairly common extensions and you may want to use them because they are recognized by other software.
- The serialize() function is used to convert individual R objects into a binary format that can be communicated across an arbitrary connection. This may get sent to a file, but it could get sent over a network or other connection.
- When you call serialize() on an R object, the output will be a raw vector coded in hexadecimal format.

```
> x <- list(1, 2, 3)
> serialize(x, NULL)
 [1] 58 0a 00 00 00 02 00 03 02 01 00 02 03 00 00 00 00 13 00 00  00
03 00 [24] 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00 0e
00 00 00 01 [47] 40 00 00 00 00 00 00 00 00 00 00 0e 00 00 00 01  40
08 00 00 00 00 00 [70] 00
```

- If you want, this can be sent to a file, but in that case you are better off using something like save().
- The benefit of the serialize() function is that it is the only way to perfectly represent an R object in an exportable format, without losing precision or any metadata. If that is what you need, then serialize() is the function for you.

INTERFACES TO THE OUTSIDE WORLD

- Data are read in using connection interfaces. Connections can be made to files (most common) or to other more exotic things.

- file, opens a connection to a file
- gzfile, opens a connection to a file compressed with gzip
- bzfile, opens a connection to a file compressed with bzip2
- url, opens a connection to a webpage

- In general, connections are powerful tools that let you navigate files or other external objects. Connections can be thought of as a translator that lets you talk to objects that are outside of R. Those outside objects could be anything from a data base, a simple text file, or a a web service API. Connections allow R functions to talk to all these different external objects without you having to write custom code for each object.

FILE CONNECTIONS

- Connections to text files can be created with the file() function.

```
> str(file)
function (description = "", open = "", blocking = TRUE, encoding
= getOption("en\ coding"), raw = FALSE)
```

- The file() function has a number of arguments that are common to many other connection functions so it's worth going into a little detail here.
  - description is the name of the file
  - open is a code indicating what mode the file should be opened in
- The open argument allows for the following options:
  - "r" open file in read only mode
  - "w" open a file for writing (and initializing a new file)
  - "a" open a file for appending
  - "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

- For example, if one were to explicitly use connections to read a CSV file in to R, it might look like this,

```
> ## Create a connection to 'foo.txt'
> con <- file("foo.txt")
>
> ## Open connection to 'foo.txt' in read-only mode
> open(con, "r")
>
> ## Read from the connection
> data <- read.csv(con)
>
```

```
> ## Close the connection
> close(con)
```

which is the same as

```
> data <- read.csv("foo.txt")
```

- In the background, read.csv() opens a connection to the file foo.txt, reads from it, and closes the connection when it's done.
- The above example shows the basic approach to using connections. Connections must be opened, then they are read from or written to, and then they are closed.

- **READING LINES OF A TEXT FILE**

- Text files can be read line by line using the readLines() function. This function is useful for reading text files that may be unstructured or contain non-standard data.

```
> ## Open connection to gz-compressed text file
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"       "10-point"   "10th"       "11-point" "12-point" "16-point"
 [7] "18-point"   "1st"        "2"          "20-point"
```

- For more structured text data like CSV files or tab-delimited files, there are other functions like read.csv() or read.table().
- The above example used the gzfile() function which is used to create a connection to files compressed using the gzip algorithm. This approach is useful because it allows you to read from a file without having to uncompress the file first, which would be a waste of space and time.
- There is a complementary function writeLines() that takes a character vector and writes each element of the vector one line at a time to a text file.

READING FROM A URL CONNECTION

- The readLines() function can be useful for reading in lines of webpages. Since web pages are basically text files that are stored on a remote server, there is conceptually not much difference between a web page and a local text file. However, we need R to

negotiate the communication between your computer and the web server. This is what the url() function can do for you, by creating a url connection to a web server.

- This code might take time depending on your connection speed.

```
> ## Open a URL connection for reading
> con <- url("http://www.jhsph.edu", "r")
>
> ## Read the web page
> x <- readLines(con)
>
> ## Print out the first few lines
> head(x)
[1]
"<!DOCTYPE
html>"
 [2] "<html
lang=\"en\">"
[3] ""
[4] "<head>"
[5] "<meta charset=\"utf-8\" />"
[6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

- While reading in a simple web page is sometimes useful, particularly if data are embedded in the web page somewhere. However, more commonly we can use URL connection to read in specific data files that are stored on web servers.
- Using URL connections can be useful for producing a reproducible analysis, because the code essentially documents where the data came from and how they were obtained.
- This is approach is preferable to opening a web browser and downloading a dataset by hand. Of course, the code you write with connections may not be executable at a later date if things on the server side are changed or reorganized.

SUBSETTING R OBJECTS

- There are three operators that can be used to extract subsets of R objects.
  - The [operator always returns an object of the same class as the original. It can be used to select multiple elements of an object

- The [[operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.

- The $ operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of [[.

SUBSETTING A VECTOR

- Vectors are basic objects in R and they can be subsetted using the [operator.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]## Extract the first element
[1] "a"
> x[2]## Extract the second element
[1] "b"
```

- The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
> x[1:4]
[1] "a" "b" "c" "c"
```

- The sequence does not have to be in order; you can specify any arbitrary integer vector.

```
> x[c(1, 3, 4)]
[1] "a" "c" "c"
```

- We can also pass a logical sequence to the [operator to extract elements of a vector that satisfy a given condition. For example, here we want the elements of x that come lexicographically after the letter “a”.

```
> u <- x > "a"
> u
[1] FALSE TRUE  TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

- Another, more compact, way to do this would be to skip the creation of a logical vector and just subset the vector directly with the logical expression.

```
> x[x > "a"]
[1] "b" "c" "c" "d"
```

SUBSETTING A MATRIX

- Matrices can be subsetted in the usual way with (i,j) type indices. Here, we create simple 2×3 matrix with the matrix function.

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- We can access the $(1, 2)$ or the $(2, 1)$ element of this matrix using the appropriate indices.

```
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

- Indices can also be missing. This behavior is used to access entire rows or columns of a matrix.

```
> x[1, ] ## Extract the first row
[1] 1 3 5
> x[, 2] ## Extract the second column
[1] 3 4
```

- **Dropping matrix dimensions**

- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. Often, this is exactly what we want, but this behavior can be turned off by setting drop = FALSE.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop =FALSE] [,1] [1,] 3
```

- Similarly, when we extract a single row or column of a matrix, R by default drops the dimension of length 1, so instead of getting a 1×3 matrix after extracting the first

row, we get a vector of length 3. This behavior can similarly be turned off with the drop = FALSE option.

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
        [1] 1 3 5
        > x[1, , drop = FALSE]
             [,1] [,2] [,3]
        [1,] 1    3    5
```

- Be careful of R's automatic dropping of dimensions. This is a feature that is often quite useful during interactive work, but can later come back to bite you when you are writing longer programs or functions.

SUBSETTING LISTS

- Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
> x
        $foo
        [1] 1 2 3 4
        $bar
        [1] 0.6
```

- The [[operator can be used to extract single elements from a list. Here we extract the first element of the list.

```
> x[[1]]
[1] 1 2 3 4
```

- The [[operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the $ operator to extract elements by name.

```
>   x[["bar"]]
    [1] 0.6
> x$bar
        [1] 0.6
```

- Notice you don't need the quotes when you use the $ operator.
- One thing that differentiates the [[operator from the $ is that the [[operator can be used with computed indices. The $ operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
                    >
> ## computed index for "foo"
> x[[name]]
                    [1] 1 2 3 4
                    >
> ## element "name" doesnâ€™t exist! (but no error here)
> x$name
                    NULL
                    >
> ## element "foo" does exist
> x$foo
                    [1] 1 2 3 4
```

SUBSETTING NESTED ELEMENTS OF A LIST

- The [[operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> ## Get the 3rd element of the 1st element
> x[[c(1, 3)]]
                    [1] 14
                    >
> ## Same as above
> x[[1]][[3]]
                    [1] 14
                    >
> ## 1st element of the 2nd element
> x[[c(2, 1)]]
                    [1] 3.14
```

EXTRACTING MULTIPLE ELEMENTS OF A LIST

- The [operator can be used to extract multiple elements from a list. For example, if you wanted to extract the first and third elements of a list, you would do the following

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$foo
[1] 1 2 3 4
$baz
[1] "hello"
```

- Note that x[c(1, 3)] is NOT the same as x[[c(1, 3)]].
- Remember that the [operator always returns an object of the same class as the original. Since the original object was a list, the [operator returns a list. In the above code, we returned a list with two elements (the first and the third).

PARTIAL MATCHING

- Partial matching of names is allowed with [[and $. This is often very useful during interactive work if the object you're working with has very long element names.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4
5
```

REMOVING NA VALUES

- A common task in data analysis is removing missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> print(bad)
[1] FALSE FALSE TRUE FALSE TRUE FALSE
```

```
>
x[!bad]
[1] 1
2 4 5
```

- What if there is multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
```

[1] TRUE TRUE FALSE TRUE FALSE TRUE

```
>x[good]
[1]
1 2
4 5
> y[good]
[1] "a" "b" "d" "f"
> head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	**NA**	**NA**	14.3	56	5	5
6	28	**NA**	14.9	66	5	6

```
> good <- complete.cases(airquality)
> head(airquality[good, ])
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
7	23	299	8.6	65	5	7
8	19	99	13.8	59	5	8

POSSIBLE QUESTIONS

UNIT – II

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. How to read a data using read.csv function?
2. How to read a data using read.table function?
3. List the functions for reading the data in R
4. List the functions for writing the data in R
5. What is meant by Subsetting?
6. Define dump()
7. Define dput()
8. Define source
9. How to create a vector using subset?
10. What is readr package?

PART – C (6 MARKS)

1. Explain the functions of reading and writing Data in R
2. Explain how to read large datasets using read.table function
3. Write in detail (i) dput() (ii)dump()
4. Explain how to describe the interfaces to the Outside world in R
5. Explain about Vector Subsetting
6. Explain about Matrix Subsetting
7. Explain how to create a list using Nested Elements in Subsetting
8. Write in detail (i) readr Package (ii) Removing NA Values
9. Explain about partial matching
10. Explain about List Subsetting

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2018 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - II : (Objective Type Multiple choice Questions each Question carries one Mark)

R PROGRAMMING [18CAU404A]

PART - A (Online Examination)

Questions	Opt1	Opt2	Opt3	Opt4	Key
data	read.csv	dget	readLines	get	read.csv
saved workspaces ?	unserialize	load	get	read	load
Which of the following statement would read file “foo.txt”	data <- read.table(“foo.txt”)	read.data <- read.table(“foo.txt”)	data <- read.data(“foo.txt”)	data.read <- read(“foo.txt”)	data <- read.table(“foo.txt”)
Which of the following function is identical to read.table	read.csv	read.data	read.tab	read.table	read.csv
Which of the following code would read 100 rows	initial <- read.table(“datatable.txt”, nrows = 100)	tabAll <- read.table(“datatable.txt”, colClasses = classes)	initial <- read.table(“datatable.txt”, nrows = 99)	initial <- read.table(“datatable.txt”, ncols= 99)	initial <- read.table(“datatable.txt”, nrows = 100)
Which of the following code opens a connection to the file foo.txt, reads from it, and closes the connection when its done ?	data <- read.csvo(“foo.txt”)	data <- read.csv(“foo.txt”)	data <- readonly.csv(“foo.txt”)	data <- readcsv(“foo.txt”)	data <- read.csv(“foo.txt”)
Which of the following extracts first element from the following vector ? > x <- c("a", "b", "c", "c", "d", "a")	x[10].	x[1].	x[0].	x[11].	x[1].

Point out the correct statement :	There are three operators that can be used to extract subsets of R objects	The [operator is used to extract elements of a list or data frame by literal name	The [[operator is used to extract elements of a list or data frame by string name	The ((operator is used to extract elements of a list or data frame by string name	There are three operators that can be used to extract subsets of R objects
Which of the following extracts first four element from the following vector ? > x <- c("a", "b", "c", "c", "d", "a")	x[0:4].	x[0:3].	x[1:4].	x[1:3].	x[1:4].
What would be the output of the following code ? x <- c("a", "b", "c", "c", "d", "a") > x[c(1, 3, 4)]	“a” “b” “c”	“a” “c” “c”	“a” “c” “b”	“a” “b” “b”	“a” “c” “c”
Point out the wrong statement :	$ operator semantics are similar to that of [[	The [operator always returns an object of the same class as the original	The $ operator is used to extract elements of a list or a data frame	The [[operator is used to extract elements of a list or a data frame	The $ operator is used to extract elements of a list or a data frame
What would be the output of the following code ? > x <- matrix(1:6, 2, 3) > x[1, 2]	3	2	1	0	3
What would be the output of the following code ? > x <- matrix(1:6, 2, 3) > x[1,]	1 3 5	2 3 5	3 3 5	file	1 3 5
Which of the following code extracts the second column for the following matrix ? > x <- matrix(1:6, 2, 3)	x[2,].	x[1, 2].	x[, 2].	x[2, 2].	x[, 2].

Point out the wrong statement :	$ operator semantics are similar to that of [[	The [[operator can take an integer sequence if you want to extract a nested element of a list	The $ operator can be used to extract multiple elements from a list	There are three operators that can be used to extract subsets of R objects	The $ operator can be used to extract multiple elements from a list
Which of the following code extracts 1st element of the 2nd element ? > x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))	x[[c(2, 1)]].	x[[c(1, 2)]].	x[[c(2, 1,1)]].	x[[c(2, 0,1)]].	x[[c(2, 1)]].
_____ , for dumping a textual representation of multiple R objects	dput	save	dump	serialize	dump
_____, for outputting a textual representation of an R object	dput	save	dump	serialize	dput
_____, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.	dput	save	dump	serialize	save
_____, for converting an R object into a binary format for outputting to a connection (or file).	dput	save	dump	serialize	serialize
_____string indicating how the columns are separated	sep	colClasses	nrows	file	sep
_____ character vector indicating the class of each column in the dataset	sep	colClasses	nrows	file	colClasses
_____the number of rows in the dataset. By default read.table() reads an entire file	sep	colClasses	nrows	file	nrows
_____logical indicating if the file has a header line	sep	colClasses	nrows	header	header
_____character string indicating the comment character	sep	colClasses	comment.char	header	comment.char

Partial matching of names is allowed with _____ and ___	[and $	[[and [	[[and [$	[[and $	[[and $
The_____operator can take an integer sequence if you want to extract a nested element of a list.	$	[[	[	((	[[
The_____operator can be used to extract single elements from a list	$	[	[[	((	[[
The_____operator to extract elements by name	$	[	[[	((	$
The_____function can be useful for reading in lines of webpages	Load()	readLines()	read()	readpage()	readLines()
Text files can be read line by line using the _____function.	Load()	readpage()	read()	readLines()	readLines()
The_____package is recently developed by Hadley Wickham to deal with reading in large flat files quickly.	readr	dplyr	read	dr	readr
The_____and_____functions are useful because the resulting textual format is editable, and in the case of corruption, potentially recoverable.	dump() and dget()	dump() and dput()	dget() and dput()	dump() and dp()	dump() and dput()
_____opens a connection to a file	file	gzfile	bzfile	url	file
_____opens a connection to a file compressed with gzip	file	gzfile	bzfile	url	gzfile
_____opens a connection to a file compressed with bzip2	file	gzfile	bzfile	url	bzfile
_____opens a connection to a webpage	file	gzfile	bzfile	url	url
The_____function has a number of arguments that are common to many other connection	f()	close()	file()	open()	file()
_____open file in read only mode	“r”	“a”	“w”	"ab"	“r”
_____open a file for writing (and initializing a new file)	“r”	“a”	“w”	"ab"	“w”
_____open a file for appending	“r”	“a”	“w”	"ab"	“a”

The_____operator can be used to extract multiple elements of a vector by passing the operator an integer sequence	$	[	[[	((	[
What would be the output of the following code ? > x <- list(foo = 1:4, bar = 0.6, baz = "hello") > name <- "foo" > x[[name]]	1 2 3 4	0 1 2 3	1 2 3 4 5	1 2 3 5	1 2 3 4
What would be the output of the following code ? > x <- list(aardvark = 1:5) > x$a	2 3 5	1 3 3 5	1 2 3	1 2 3 4 5	1 2 3 4 5
What would be the output of the following code ? > x <- list(foo = 1:4, bar = 0.6, baz = "hello") > name <- "foo" > x$name	1	3	2	4	2
What would be the output of the following code ? > x <- list(a = list(10, 12, 14), b = c(3.14, 2.81)) > x[[c(1, 3)]]	13	14	15	16	14
The_____function is used to convert individual R objects into a binary format that can be communicated across an arbitrary connection.	dput()	save()	serialize()	dump()	serialize()
Matrices can be subsetted in the usual way with (i,j) type _____	subset	subsetting	indices	sets	indices
The main functions for converting R objects into a binary format are	save(), save.image(), and unserialize()	save(), save.image(), and serialize()	save(), unserialize, and serialize()	unserialize(), save.image(), and serialize()	save(), save.image(), and serialize()
The_____function is one of the most commonly used functions for reading data in R	read.csv()	read.table()	read.data()	read()	read.table()
_____, a character vector indicating the class of each column in the dataset	sep	header	file	colClasses	colClasses
The inverse of dump() is_____function	file()	dput()	source()	dum()	source()
Vectors are basic objects in R and they can be subsetted using the_____operator	((	[	[]	[[	[

The_____function is identical to read.table except that some of the defaults are set differently	read.csv()	read.table()	read()	read.data()	read.csv()
Factors are important in statistical modeling and are treated specially by modelling functions like ___and_____.	l() and gl()	lm() and glm().	lme() and glme()	m() and gm()	lm() and glm().
We can also create an empty list of a prespecified length with the_____function	create()	file()	vector()	list()	vector()
The sequence does not have to be in order; you can specify any_____integer vector.	specified	legel	unarbitrary	arbitrary	arbitrary
The [[operator can be used to extract _____ elements from a list.	no	all	single	double	single
The $ operator can only be used with _____ names.	different	literal	same	unique	literal
A common task in data analysis is removing _____	missing values	segments	changing values	names	missing values

UNIT-III

SYLLABUS

Vectorized Operations : Vectorized Matrix Operations . **Dates and Times :**Dates in R - Times in R - Operations on Dates and Times .**Managing Data Frames with the dplyr package :**Data Frames -The dplyr Package - dplyr Grammar - Installing the dplyr package - select() - filter() - arrange() - rename() - mutate() - group_by()-%>%.**Control Structures :**if-else - for Loops - Nested for loops - while Loops - repeat Loops - next, break.

VECTORIZED OPERATIONS

- Many operations in R are vectorized, meaning that operations occur in parallel in certain R objects.
- This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.
- The simplest example is when adding two vectors together.

```
> x <- 1:4
> y <- 6:9
> z <- x + y
> z
[1] 7 9 11 13
```

Without vectorization ,

```
z <- numeric(length(x))
for(i in seq_along(x)) {
z <- x[i] + y[i]
}
z
[1] 13
```

- Another operation can do in a vectorized manner is logical comparisons. So suppose we wanted to know which elements of a vector were greater than 2. we could do the following.

```
> x
```

```
[1] 1 2 3 4
> x > 2
[1] FALSE FALSE TRUE TRUE
```

- Here are other vectorized logical operations.

```
> x >= 2
[1] FALSE TRUE TRUE TRUE
> x < 3
[1] TRUE TRUE FALSE FALSE
> y == 8
[1] FALSE FALSE TRUE FALSE
```

- Notice that these logical operations return a logical vector of TRUE and FALSE.
- Of course, subtraction, multiplication and division are also vectorized.

```
> x - y
[1] -5 -5 -5 -5
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

VECTORIZED MATRIX OPERATIONS

- Matrix operations are also vectorized, making for nicly compact notation. This way, we can do element-by-element operations on matrices without having to loop over every element.

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
> ## element-wise multiplication
> x * y
[,1] [,2]
[1,] 10 30
[2,] 20 40
> ## element-wise division
```

```
> x / y
[,1] [,2]
[1,] 0.1 0.3
[2,] 0.2 0.4
> ## true matrix multiplication
> x %*% y
[,1] [,2]
[1,] 40 40
[2,] 60 60
```

DATES AND TIMES

- R has developed a special representation for dates and times. Dates are represented by the Date class and times are represented by the POSIXct or the POSIXlt class. Dates are stored internally as the number of days since 1970-01-01 while times are stored internally as the number of seconds since 1970-01-01.
- It‟s not important to know the internal representation of dates and times in order to use them in R

DATES IN R

- Dates are represented by the Date class and can be coerced from a character string using the as.Date() function. This is a common way to end up with a Date object in R.

```
> ## Coerce a 'Date' object from character
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
```

- We can see the internal representation of a Date object by using the unclass() function.

```
> unclass(x)
```

```
[1] 0
> unclass(as.Date("1970-01-02"))
[1] 1
```

TIMES IN R

- Times are represented by the POSIXct or the POSIXlt class. POSIXct is just a very large integer under the hood. It uses a useful class when you want to store times in something like a data frame. POSIXlt is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month. This is useful when you need that kind of information.
- There are a number of generic functions that work on dates and times to help you extract pieces of dates and/or times.
  - weekdays: give the day of the week
  - months: give the month name
  - quarters: give the quarter number ("Q1", "Q2", "Q3", or "Q4")
- Times can be coerced from a character string using the as.POSIXlt or as.POSIXct function.

```
> x <- Sys.time()
> x
[1] "2015-04-13 10:09:17 EDT"
> class(x) ## 'POSIXct' object
[1] "POSIXct" "POSIXt"
```

The POSIXlt object contains some useful metadata.

```
> p <- as.POSIXlt(x)
> names(unclass(p))
[1] "sec" "min" "hour" "mday" "mon" "year" "wday"
[8] "yday" "isdst" "zone" "gmtoff"
> p$wday ## day of the week
[1] 1
```

- We can also use the POSIXct format.

```
> x <- Sys.time()
> x ## Already in „POSIXct‟ format
[1] "2015-04-13 10:09:17 EDT"
> unclass(x) ## Internal representation
[1] 1428934157
> x$sec ## Can't do this with 'POSIXct'!
Error in x$sec: $ operator is invalid for atomic vectors
> p <- as.POSIXlt(x)
> p$sec ## That's better
[1] 17.16238
```

- Finally, there is the strptime() function in case your dates are written in a different format.

  strptime() takes a character vector that has dates and times and converts them into to a POSIXlt object.

```
> datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
> x <- strptime(datestring, "%B %d, %Y %H:%M")
> x
[1] "2012-01-10 10:40:00 EST" "2011-12-09 09:10:00 EST"
> class(x)
[1] "POSIXlt" "POSIXt"
```

- The weird-looking symbols that start with the % symbol are the formatting strings for dates and times

OPERATIONS ON DATES AND TIMES

- We can use mathematical operations on dates and times. Well, really just + and -. We can do comparisons too (i.e. ==, <=)

```
> x <- as.Date("2012-01-01")
> y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
> x-y
Warning: Incompatible methods ("-.Date", "-.POSIXt") for "-"
```

```
Error in x - y: non-numeric argument to binary operator
> x <- as.POSIXlt(x)
> x-y
Time difference of 356.3095 days
```

- The nice thing about the date/time classes is that they keep track of all the annoying things about dates and times, like leap years, leap seconds, daylight savings, and time zones.
- Here‟s an example where a leap year gets involved.

```
> x <- as.Date("2012-03-01")
> y <- as.Date("2012-02-28")
> x-y
Time difference of 2 days
> ## My local time zone
> x <- as.POSIXct("2012-10-25 01:00:00")
> y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
> y-x
Time difference of 1 hours
```

MANAGING DATA FRAMES WITH THE DPLYR PACKAGE

DATA FRAMES

- The data frame is a key data structure in statistics and in R. The basic structure of a data frame is that there is one observation per row and each column represents a variable, a measure, feature, or characteristic of that observation. R has an internal implementation of data frames that is likely the one you will use most often.
- However, there are packages on CRAN that implement data frames via things like relational databases that allow you to operate on very large data frames. Given the importance of managing data frames, it‟s important that we have good tools for dealing with them.
- In previous chapters we have already discussed some tools like the subset() function and the use of [and $ operators to extract subsets of data frames. However, other operations,

like filtering, re-ordering, and collapsing, can often be tedious operations in R whose syntax is not very intuitive.

- The dplyr package is designed to mitigate a lot of these problems and to provide a highly optimized set of routines specifically for dealing with data frames.

THE dplyr PACKAGE

- The dplyr package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package. The dplyr package does not provide any "new" functionality to R per se, in the sense that everything dplyr does could already be done with base R, but it greatly simplifies existing functionality in R.
- One important contribution of the dplyr package is that it provides a "grammar" (in particular, verbs) for data manipulation and for operating on data frames. With this grammar, you can sensibly communicate what it is that you are doing to a data frame that other people can understand (assuming they also know the grammar).
- This is useful because it provides an abstraction for data manipulation that previously did not exist. Another useful contribution is that the dplyr functions are very fast, as many key operations are coded in C++

dplyr GRAMMAR

- Some of the key "verbs" provided by the dplyr package are
  - select: return a subset of the columns of a data frame, using a flexible notation
  - filter: extract a subset of rows from a data frame based on logical conditions
  - arrange: reorder rows of a data frame
  - rename: rename variables in a data frame
  - mutate: add new variables/columns or transform existing variables
  - summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata
  - %>%: the "pipe" operator is used to connect multiple verb actions together into a pipeline

- The dplyr package as a number of its own data types that it takes advantage of. For example, there is a handy print method that prevents you from printing a lot of data to the console. Most of the time, these additional data types are transparent to the user.

- **COMMON dplyr FUNCTION PROPERTIES**

- All of the functions that we will discuss in this Chapter will have a few common characteristics. In particular,

  1. The first argument is a data frame.
  2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
  3. The return result of a function is a new data frame
  4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

INSTALLING THE dplyr PACKAGE

- The dplyr package can be installed from CRAN or from GitHub using the devtools package and the install_github() function. The GitHub repository will usually contain the latest updates to the package and the development version.
- To install from CRAN, just run

```
> install.packages("dplyr")
```

- To install from GitHub you can run

```
> install_github("hadley/dplyr")
```

- After installing the package it is important that you load it into your R session with the library() function.

```
> library(dplyr)
```

- Attaching package: 'dplyr'

- The following object is masked from 'package:stats':

  filter
- The following objects are masked from 'package:base':

  intersect, setdiff, setequal, union
- We may get some warnings when the package is loaded because there are functions in the dplyr package that have the same name as functions in other packages.

select()

- For the examples in this chapter we will be using a dataset containing air pollution and temperature data for the city of Chicago in the U.S. The dataset is available from my web site.
- After unzipping the archive, you can load the data into R using the readRDS() function.

  > chicago <- readRDS("chicago.rds")
- We can see some basic characteristics of the dataset with the dim() and str() functions.

  > dim(chicago)

  [1] 6940 8

  > str(chicago)

  'data.frame': 6940 obs. of 8 variables:

  $ city : chr "chic" "chic" "chic" "chic" ...

  $ tmpd : num 31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...

  $ dptp : num 31.5 29.9 27.4 28.6 28.9 ...

  $ date : Date, format: "1987-01-01" "1987-01-02" ...

  $ pm25tmean2: num NA NA NA NA NA NA NA NA NA NA ...

  $ pm10tmean2: num 34 NA 34.2 47 NA ...

  $ o3tmean2 : num 4.25 3.3 3.33 4.38 4.75 ...

  $ no2tmean2 : num 20 23.2 23.8 30.4 30.3 ...
- The select() function can be used to select columns of a data frame that you want to focus on.
- Often you‟ll have a large data frame containing “all” of the data, but any given analysis

might only use a subset of variables or observations.

- The select() function allows you to get the few columns you might need. Suppose we wanted to take the first 3 columns only. There are a few ways to do this. We could for example use numerical indices. But we can also use the names directly.

```
> names(chicago)[1:3]
[1] "city" "tmpd" "dptp"
> subset <- select(chicago, city:dptp)
> head(subset)
city tmpd dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
4 chic 29.0 28.625
5 chic 32.0 28.875
6 chic 40.0 35.125
```

- Note that the: normally cannot be used with names or strings, but inside the select() function you can use it to specify a range of variable names. You can also omit variables using the select() function by using the negative sign.
- With select() you can do

```
> select(chicago, -(city:dptp))
```

- This indicates that we should include every variable except the variables city through dptp. The equivalent code in base R would be

```
> i <- match("city", names(chicago))
> j <- match("dptp", names(chicago))
> head(chicago[, -(i:j)])
```

- Not super intuitive, right?
- The select() function also allows a special syntax that allows you to specify variable names based on patterns. So, for example, if you wanted to keep every variable that ends with a “2”, we could do

```
> subset <- select(chicago, ends_with("2"))
> str(subset)
'data.frame': 6940 obs. of 4 variables:
$ pm25tmean2: num NA NA NA NA NA NA NA NA NA NA ...
$ pm10tmean2: num 34 NA 34.2 47 NA ...
$ o3tmean2 : num 4.25 3.3 3.33 4.38 4.75 ...
$ no2tmean2 : num 20 23.2 23.8 30.4 30.3 ...
```

- Or if we wanted to keep every variable that starts with a “d”, we could do

```
> subset <- select(chicago, starts_with("d"))
> str(subset)
'data.frame': 6940 obs. of 2 variables:
$ dptp: num 31.5 29.9 27.4 28.6 28.9 ...
$ date: Date, format: "1987-01-01" "1987-01-02" ...
```

filter()

- The filter() function is used to extract subsets of rows from a data frame. This function is similar to the existing subset() function in R but is quite a bit faster in my experience.
- Suppose we wanted to extract the rows of the chicago data frame where the levels of PM2.5 are greater than 30 (which is a reasonably high level), we could do

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> str(chic.f)
'data.frame': 194 obs. of 8 variables:
$ city : chr "chic" "chic" "chic" "chic" ...
$ tmpd : num 23 28 55 59 57 57 75 61 73 78 ...
$ dptp : num 21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
$ date : Date, format: "1998-01-17" "1998-01-23" ...
$ pm25tmean2: num 38.1 34 39.4 35.4 33.3 ...
$ pm10tmean2: num 32.5 38.7 34 28.5 35 ...
$ o3tmean2 : num 3.18 1.75 10.79 14.3 20.66 ...
```

$ no2tmean2 : num 25.3 29.4 25.3 31.4 26.8 ...

- You can see that there are now only 194 rows in the data frame and the distribution of the pm25tmean2 values is.

```
> summary(chic.f$pm25tmean2)
    Min. 1st Qu. Median Mean 3rd Qu. Max.
    30.05 32.12 35.04 36.63 39.53 61.50
```

- We can place an arbitrarily complex logical sequence inside of filter(), so we could for example extract the rows where PM2.5 is greater than 30 and temperature is greater than 80 degrees Fahrenheit.

```
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
> select(chic.f, date, tmpd, pm25tmean2)
date tmpd pm25tmean2
1 1998-08-23 81 39.60000
2 1998-09-06 81 31.50000
3 2001-07-20 82 32.30000
4 2001-08-01 84 43.70000
5 2001-08-08 85 38.83750
6 2001-08-09 84 38.20000
7 2002-06-20 82 33.00000
8 2002-06-23 82 42.50000
9 2002-07-08 81 33.10000
10 2002-07-18 82 38.85000
11 2003-06-25 82 33.90000
12 2003-07-04 84 32.90000
13 2005-06-24 86 31.85714
14 2005-06-27 82 51.53750
15 2005-06-28 85 31.20000
16 2005-07-17 84 32.70000
17 2005-08-03 84 37.90000
```

- Now there are only 17 observations where both of those conditions are met.

arrange()

- The arrange() function is used to reorder rows of a data frame according to one of the variables/- columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R.
- The arrange() function simplifies the process quite a bit.
- Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

  > chicago <- arrange(chicago, date)
- We can now check the first few rows

  > head(select(chicago, date, pm25tmean2), 3)

  date pm25tmean2

  1 1987-01-01 NA

  2 1987-01-02 NA

  3 1987-01-03 NA

  and the last few rows.

  > tail(select(chicago, date, pm25tmean2), 3)

  date pm25tmean2

  6938 2005-12-29 7.45000

  6939 2005-12-30 15.05714

  6940 2005-12-31 15.00000
- Columns can be arranged in descending order too by useing the special desc() operator.

  > chicago <- arrange(chicago, desc(date))
- Looking at the first three and last three rows shows the dates in descending order

  > head(select(chicago, date, pm25tmean2), 3)

  date pm25tmean2

  1 2005-12-31 15.00000

  2 2005-12-30 15.05714

```
3 2005-12-29 7.45000
> tail(select(chicago, date, pm25tmean2), 3)
date pm25tmean2
6938 1987-01-03 NA
6939 1987-01-02 NA
6940 1987-01-01 NA
```

rename()

- Renaming a variable in a data frame in R is surprisingly hard to do! The rename() function is designed to make this process easier.
- Here you can see the names of the first five variables in the chicago data frame.

```
> head(chicago[, 1:5], 3)
city tmpd dptp date pm25tmean2
1 chic 35 30.1 2005-12-31 15.00000
2 chic 36 31.0 2005-12-30 15.05714
3 chic 35 29.4 2005-12-29 7.45000
```

- The dptp column is supposed to represent the dew point temperature adn the pm25tmean2 column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible.

```
> chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
> head(chicago[, 1:5], 3)
city tmpd dewpoint date pm25
1 chic 35 30.1 2005-12-31 15.00000
2 chic 36 31.0 2005-12-30 15.05714
3 chic 35 29.4 2005-12-29 7.45000
```

- The syntax inside the rename() function is to have the new name on the left-hand side of the = sign and the old name on the right-hand side.

mutate()

- The mutate() function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing variables and mutate() provides a clean interface for doing that.
- For example, with air pollution data, we often want to detrend the data by subtracting the mean from the data. That way we can look at whether a given day‟s air pollution level is higher than or less than average (as opposed to looking at its absolute level). Here we create a pm25detrend variable that subtracts the mean from the pm25 variable.

```
> chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
> head(chicago)
      city tmpd dewpoint date pm25 pm10tmean2 o3tmean2 no2tmean2
1 chic 35 30.1 2005-12-31 15.00000 23.5 2.531250 13.25000
2 chic 36 31.0 2005-12-30 15.05714 19.2 3.034420 22.80556
3 chic 35 29.4 2005-12-29 7.45000 23.5 6.794837 19.97222
4 chic 37 34.5 2005-12-28 17.75000 27.5 3.260417 19.28563
5 chic 40 33.6 2005-12-27 23.56000 27.0 4.468750 23.50000
6 chic 35 29.6 2005-12-26 8.40000 8.5 14.041667 16.81944
pm25detrend
1 -1.230958
2 -1.173815
3 -8.780958
4 1.519042
5 7.329042
6 -7.830958
```

- There is also the related transmute() function, which does the same thing as mutate() but then drops all non-transformed variables.
- Here we detrend the PM10 and ozone (O3) variables.

```
> head(transmute(chicago,
```

```
+ pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
+ o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
      pm10detrend o3detrend
1 -10.395206 -16.904263
2 -14.695206 -16.401093
3 -10.395206 -12.640676
4 -6.395206 -16.175096
5 -6.895206 -14.966763
6 -25.395206 -5.393846
```

- Note that there are only two columns in the transmuted data frame.

group_by()

- The group_by() function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the date variable.
- In conjunction with the group_by() function we often use the summarize() function (or summarise() for some parts of the world). The general operation here is a combination of splitting a data frame into separate pieces defined by a variable or group of variables (group_by()), and then applying a summary function across those subsets (summarize()).
- First, we can create a year varible using as.POSIXlt().

```
> chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

- Now we can create a separate data frame that splits the original data frame by year.

```
> years <- group_by(chicago, year)
```

- Finally, we compute summary statistics for each year in the data frame with the summarize() function.

```
> summarize(years, pm25 = mean(pm25, na.rm = TRUE),
+ o3 = max(o3tmean2, na.rm = TRUE),
+ no2 = median(no2tmean2, na.rm = TRUE))
```

Source: local data frame [19 x 4]

year pm25 o3 no2

1 1987 NaN 62.96966 23.49369
2 1988 NaN 61.67708 24.52296
3 1989 NaN 59.72727 26.14062
4 1990 NaN 52.22917 22.59583
5 1991 NaN 63.10417 21.38194
6 1992 NaN 50.82870 24.78921
7 1993 NaN 44.30093 25.76993
8 1994 NaN 52.17844 28.47500
9 1995 NaN 66.58750 27.26042
10 1996 NaN 58.39583 26.38715
11 1997 NaN 56.54167 25.48143
12 1998 18.26467 50.66250 24.58649
13 1999 18.49646 57.48864 24.66667
14 2000 16.93806 55.76103 23.46082
15 2001 16.92632 51.81984 25.06522
16 2002 15.27335 54.88043 22.73750
17 2003 15.23183 56.16608 24.62500
18 2004 14.62864 44.48240 23.39130
19 2005 16.18556 58.84126 22.62387

- summarize() returns a data frame with year as the first column, and then the annual averages of pm25, o3, and no2.
- In a slightly more complicated example, we might want to know what the average levels of ozone (o3) are and nitrogen dioxide (no2) within quintiles of pm25. A slicker way to do this would be through a regression model, but we can actually do this quickly with group_by() and summarize().
- First, we can create a categorical variable of pm25 divided into quintiles.

```
> qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
```

> chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))

- Now we can group the data frame by the pm25.quint variable.

  > quint <- group_by(chicago, pm25.quint)

- Finally, we can compute the mean of o3 and no2 within quintiles of pm25.

  > summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),

  + no2 = mean(no2tmean2, na.rm = TRUE))

  Source: local data frame [6 x 3]

  pm25.quint o3 no2

  1 (1.7,8.7] 21.66401 17.99129

  2 (8.7,12.4] 20.38248 22.13004

  3 (12.4,16.7] 20.66160 24.35708

  4 (16.7,22.6] 19.88122 27.27132

  5 (22.6,61.5] 20.31775 29.64427

  6 NA 18.79044 25.77585

- From the table, it seems there isn‟t a strong relationship between pm25 and o3, but there appears to be a positive correlation between pm25 and no2. More sophisticated statistical modeling can help to provide precise answers to these questions, but a simple application of dplyr functions can often get you most of the way there.

%>%

- The pipeline operator %> % is very handy for stringing together multiple dplyr functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

  > third(second(first(x)))

- This nesting is not a natural way to think about a sequence of operations. The %>% operator allows you to string operations in a left-to-right fashion, i.e.

  > first(x) %>% second %>% third

- Take the example that we just did in the last section where we computed the mean of o3

and no2 within quintiles of pm25. There we had to

1. create a new variable pm25.quint
2. split the data frame by that new variable
3. compute the mean of o3 and no2 in the sub-groups defined by pm25.quint

- That can be done with the following sequence in a single R expression.

```
> mutate(chicago, pm25.quint = cut(pm25, qq)) %>%
+ group_by(pm25.quint) %>%
+ summarize(o3 = mean(o3tmean2, na.rm = TRUE),
+ no2 = mean(no2tmean2, na.rm = TRUE))
Source: local data frame [6 x 3]
pm25.quint o3 no2
        1 (1.7,8.7] 21.66401 17.99129
        2 (8.7,12.4] 20.38248 22.13004
        3 (12.4,16.7] 20.66160 24.35708
        4 (16.7,22.6] 19.88122 27.27132
        5 (22.6,61.5] 20.31775 29.64427
        6 NA 18.79044 25.77585
```

- This way we don‟t have to create a set of temporary variables along the way or create a massive nested sequence of function calls. Notice in the above code that I pass the chicago data frame to the first call to mutate(), but then afterwards I do not have to pass the first argument to group_by() or summarize(). Once you travel down the pipeline with %>%, the first argument is taken to be the output of the previous element in the pipeline.
- Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data.

```
> mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
+ group_by(month) %>%
+ summarize(pm25 = mean(pm25, na.rm = TRUE),
+ o3 = max(o3tmean2, na.rm = TRUE),
+ no2 = median(no2tmean2, na.rm = TRUE))
```

Source: local data frame [12 x 4]

month pm25 o3 no2

1 1 17.76996 28.22222 25.35417

2 2 20.37513 37.37500 26.78034

3 3 17.40818 39.05000 26.76984

4 4 13.85879 47.94907 25.03125

5 5 14.07420 52.75000 24.22222

6 6 15.86461 66.58750 25.01140

7 7 16.57087 59.54167 22.38442

8 8 16.93380 53.96701 22.98333

9 9 15.91279 57.48864 24.47917

10 10 14.23557 47.09275 24.15217

11 11 15.15794 29.45833 23.56537

12 12 17.52221 27.70833 24.45773

- Here we can see that o3 tends to be low in the winter months and high in the summer while no2 is higher in the winter and lower in the summer.

CONTROL STRUCTURES

- Control structures in R allow you to control the flow of execution of a series of R expressions.
- Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.
- Commonly used control structures are
  - **if and else:** testing a condition and acting on it
  - **for:** execute a loop a fixed number of times
  - **while:** execute a loop while a condition is true
  - **repeat:** execute an infinite loop (must break out of it to stop)

- **break:** break the execution of a loop
- **next:** skip an iteration of a loop

➢ Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions. However, these constructs do not have to be used in functions and it‟s a good idea to become familiar with them before we delve into functions.

if-else

➢ The if-else combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it‟s true or false.

➢ For starters, you can just use the if statement.

```
if(<condition>) {
## do something
}
## Continue with rest of code
```

➢ The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an else clause.

```
if(<condition>) {
## do something
}
else {
## do something else
}
```

➢ You can have a series of tests by following the initial if with any number of else ifs.

```
if(<condition1>) {
## do something
} else if(<condition2>) {
## do something different
} else {
```

```
## do something different
}
```

- Here is an example of a valid if/else structure.

```
## Generate a uniform random number
x <- runif(1, 0, 10)
if(x > 3) {
y <- 10
} else {
y <- 0
}
```

- The value of y is set depending on whether x > 3 or not. This expression can also be written a different, but equivalent, way in R.

```
y <- if(x > 3) {
10
} else {
0
}
```

- Neither way of writing this expression is more correct than the other. Which one you use will depend on your preference and perhaps those of the team you may be working with.
- Of course, the else clause is not necessary. You could have a series of if clauses that always get executed if their respective conditions are true.

```
if(<condition1>) {
}
if(<condition2>) {
}
```

for Loops

- For loops are pretty much the only looping construct that you will need in R.
- In R, for loops take an interator variable and assign it successive values from a sequence or vector.

- For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10) {
+ print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
> for(letter in x) {
+ print(letter)
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

- For one line loops, the curly braces are not strictly necessary.

```
> for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

- However, I like to use curly braces even for one-line loops, because that way if you decide to expand the loop to multiple lines, you won‟t be burned because you forgot to add curly braces.

Nested for loops

- for loops can be nested inside of each other.

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
for(j in seq_len(ncol(x))) {
print(x[i, j])
}
}
```

- Nested loops are commonly needed for multidimensional or hierarchical data structures (e.g.matrices, lists). Be careful with nesting though. Nesting beyond 2 to 3 levels often makes it difficult to read/understand the code. If you find yourself in need of a large number of nested loops, you may want to break up the loops by using functions

while Loops

- While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
> while(count < 10) {
+ print(count)
+ count <- count + 1
+ }
      [1] 0
      [1] 1
      [1] 2
      [1] 3
      [1] 4
      [1] 5
      [1] 6
      [1] 7
```

[1] 8

[1] 9

- While loops can potentially result in infinite loops if not written properly. > z <- 5

```
> set.seed(1)
> while(z >= 3 && z <= 10) {
+ coin <- rbinom(1, 1, 0.5)
+
+ if(coin == 1) { ## random walk
+ z <- z + 1
+ } else {
+ z <- z - 1
+ }
+ }
> print(z)
[1] 2
```

repeat Loops

- repeat initiates an infinite loop right from the start. These are not commonly used in statistical or data analysis applications but they do have their uses. The only way to exit a repeat loop is to call break.
- One possible paradigm might be in an iterative algorithm where you may be searching for a solution and you don‟t want to stop until you‟re close enough to the solution. In this kind of situation, you often don‟t know in advance how many iterations it‟s going to take to get “close enough” to the solution.

```
x0 <- 1
tol <- 1e-8
repeat {
x1 <- computeEstimate()
```

```
if(abs(x1 - x0) < tol) { ## Close enough?
break
} else {
x0 <- x1
}
}
```

- Note that the above code will not run if the computeEstimate() function is not defined (I just made it up for the purposes of this demonstration). The loop agove is a bit dangerous because there‟s no guarantee it will stop. You could get in a situation where the values of x0 and x1 oscillate back and forth and never converge. Better to set a hard limit on the number of iterations by using a for loop and then report whether convergence was achieved or not.

next, break.

- **next** is used to skip an iteration of a loop

```
for(i in 1:100) {
if(i <= 20) {
## Skip the first 20 iterations
next
}
## Do something here
}
```

- **break** is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
for(i in 1:100) {
print(i)
if(i > 20) {
## Stop loop after 20 iterations
break
}}
```

POSSIBLE QUESTIONS

UNIT – III

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is Vectorized Operations?
2. How to create dates and times in R?
3. List the looping statements in R
4. Write the syntax of if else statement with suitable example
5. What is the use of dplyr package?
6. Write the syntax of for loop with suitable example
7. Write the syntax of if else statement with suitable example
8. Write the syntax of while loop with suitable example
9. Write the syntax of repeat loop with suitable example
10. Define select()

PART – C (6 MARKS)

1. Explain the Vectorized matrix Operations
2. Explain the Operations on Dates and Times
3. Explain how to manage the Data frames with dplyr package
4. Explain the process of select () function
5. Write in detail (i) mutate() (ii) group_by()
6. Write in detail (i) filter() (ii) arrange()
7. Write in detail (i) rename (ii) %>%
8. Discuss about Control Structures in R programming
9. Explain about dplyr Grammar
10. Explain about dplyr Package

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2018 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - III : (Objective Type Multiple choice Questions each Question carries one Mark)

R PROGRAMMING [18CAU404A]

PART - A (Online Examination)

Questions	Opt1	Opt2	Opt3	Opt4	Key
operation as far as subtraction is concerned ? > x	x+y	x-y	x*y	x/y	x-y
Point out the wrong statement :	operations in R	allows you to	means that	R are vectorized,	are vectorized
What would be the output of the following code ? > x <- 1:4 > y <- 6:9 > z <- x + y > z	7 9 11 13	7 9 11 13 14	9 7 11 13	7 9 11 14	7 9 11 13
Which of the followin code represents internal representation of a Date object ?	class(as.Date("1970-01-02"))	unclass(as.Date("1970-01-02"))	unclassint(as.Date("1970-01-02"))	classint(as.Date("1970-01-02"))	unclass(as.Date("1970-01-02"))
What would be the output of the following code ? > x <- Sys.time() > class(x)	"POSIXct" "POSIXt"	"POSIXXt" "POSIXt"	"POSIXct" "POSIct"	"POSIXct" "POSIXXct"	"POSIXct" "POSIXt"
Which of the following function gives the day of the week ?	weekdays	months	quarters	years	weekdays
What would be the output of the following code ? > p <- as.POSIXlt(x) > names(unclass(p)) > p$wday	1	2	3	0	1

What would be the output of the following code ? > x <- as.Date("2012-03-01") > y <- as.Date("2012-02-28") > x-y	Time difference of 3 days	Time difference of 2 days	Time difference of 1 day	Time difference of 4 days	Time difference of 2 days
Which of the following return a subset of the columns of a data frame ?	select	retrieve	get	hold	select
_____extract a subset of rows from a data frame based on logical conditions.	rename	filter	set	subset	rename
_____generate summary statistics of different variables in the data frame, possibly within strata	rename	summarize	set	subset	summarize
Point out the wrong statement :	The dplyr package was developed by Hadley Wickham of RStudio	The dplyr packageis an optimized and distilled version of his plyr package	The dplyr package provideS any "new" functionality to R	The dplyr package does not provide any "new" functionality to R	The dplyr package provideS any "new" functionality to R
_____add new variables/columns or transform existing variables	mutate	add	apped	arrange	mutate
The_____operator is used to connect multiple verb actions together into a pipeline	pipe	piper	start	end	pipe
The dplyr package can be installed from GitHub using the_____package	dev	devtools	devtool	dtool	devtools
The dplyr package can be installed from CRAN using :	installall.packages("dplyr")	install.packages("dplyr")	installed.packages("dplyr")	installed.package("dplyr")	install.packages("dplyr")
Which of the following object is masked from 'package:stats' ?	difference	setdifference	union	filter	filter
The_____function can be used to select columns of a data frame that you want to focus on.	filter	get	rename	select	select

Point out the correct statement :	You can also omit variables using the select() function by using the negative sign	The arrange() function also allows a special syntax that allows you to specify variable names based on patterns	Reordering rows of a data frame is normally easier to do in R	The rename() function is designed to make this process difficult.	You can also omit variables using the select() function by using the negative sign
_____function is similar to the existing subset() function in R but is quite a bit faster.	rename	filter	set	subset	filter
Columns can be arranged in descending order too by using the special_____operator.	asc()	desc()	descending()	subset	desc()
Point out the wrong statement :	Renaming a variable in a data frame in R is surprisingly hard to do	The mutate() function exists to compute transformations of variables in a data frame	mute() function, which does the same thing as mutate() but then drops all non-transformed variables	The rename() function is designed to make this process easier.	mute() function, which does the same thing as mutate() but then drops all non-transformed variables
The_____function is used to generate summary statistics from the data frame within strata defined by a variable.	groupby()	group()	group_by()	arrange	group_by()
The_____operator allows you to string operations in a left-to-right fashion.	%>%>	%>%	>%>%	>%>%>	%>%
There is an SQL interface for relational databases via the_____package.	DIB	DB2	DBI	DB	DBI
dplyr can be integrated with the _____ package for large fast tables.	data.table	read.table	data.data	read.data	data.table

Which of the following function is similar to summarize ?	arrange_by()	group()	group_by()	arrange	group_by()
Which of the following is valid syntax for if else statement in R ?	if(<condition>) { ## do something } else { ## do something else }	if(<condition>) { ## do something } elseif { ## do something else }	if(<condition>) { ## do something } else if { ## do something else }	if(<condition>) { ## do something } elsif{ ## do something else }	if(<condition>) { ## do something } else { ## do something else }
Point out the correct statement :	Blocks are evaluated until a new line is entered after the closing brace	Single statements are evaluated when a new line is typed at the start of the syntactically complete statement	The if/else statement conditionally evaluates two statements	The jump statement conditionally evaluates two statements	The if/else statement conditionally evaluates two statements
Which of the following syntax is correct for while loop ?	while (statement1) statement2	while (statement1) else statement2	while (statement1) do statement2	while (statement1) else if statement2	while (statement1) statement2
_____is used to break the execution of a loop.	next	skip	break	if	break
Which of the following statement can be used to explicitly control looping ?	if	while	break	next	break
Which of the following should be preferred for evaluation from list of alternatives ?	subsett	eval	switch	if	eval
_____initiates an infinite loop right from the start.	never	repeat	break	set	repeat
Which of the following code snippet stops loop after 20 iterations ?	for(i in 1:100) { print(i) if(i>20){ break }}	for(i in 1:100) { print(i) if(i>19){ break }}	for(i in 1:100) { print(i) if(i<19){ break }}	for(i in 1:100) { print(i) if(i<20){ break }}	for(i in 1:100) { print(i) if(i>20){ break }}

Point out the wrong statement :	Statements cannot be grouped together using braces '{' and '}'	Computation in R consists of sequentially evaluating statements	Computation in R consists of sequentially evaluating statements	Control structures in R allow you to control the flow of execution of a series of R expressions.	Statements cannot be grouped together using braces '{' and '}'
_____is used to skip an iteration of a loop.	group by	group	skip	next	next
R has_____statements that provide explicit looping.	two	three	four	five	three
The syntax of the repeat loop is :	rep statement	repeat statement	repeat else	else statement	repeat statement
What will be the output of the following code ? > x <- 3 > switch(2, 2+2, mean(1:10), rnorm(5))	5	5.5	0	5.3	5.5
Point out the correct statement :	The next statement causes an exit from the innermost loop that is currently being executed	There are two statements that can be used to explicitly control looping	The break statement immediately causes control to return to the start of the loop	There are two statements that can be used to implicitly control looping	There are two statements that can be used to explicitly control looping
What will be the output of the following code ? > y <- "fruit" > switch(y, fruit = "banana", vegetable = "broccoli", "Neither")	"banana"	"Neither"	"broccoli"	"fruit"	"banana"
R has_____basic indexing operators.	two	three	four	five	three
The syntax of the for loop is :	for ($name in vector) statement1	for loop(name in vector) statement1	for (name in vector) statement1	for loop ($name in vector) statement1	for (name in vector) statement1

What would be the output of the following code ? > x <- matrix(1:4, 2, 2) > y <- matrix(rep(10, 4), 2, 2) > x * y	[,1] [,2] [1,] 10 30 [2,] 20 40	[,1] [,2] [1,] 10 30 [2,] 30 40	[,1] [,2] [1,] 20 30 [2,] 20 40	[,1] [,2] [1,] 10 30 [2,] 30 40	[,1] [,2] [1,] 10 30 [2,] 20 40
What would be the output of the following code ? > x <- 1:4 > y <- 6:9 > x/y	0.1666667 0.2857143 0.4444444	0.1666667 0.2857143 0.3750000 0.4444444	0.2857143 0.3750000 0.4444444	0.2857143 0.3750000 0.1666667	0.1666667 0.2857143 0.3750000 0.4444444
What would be the output of the following code ? > x <- as.Date("1970-01-01") > x	“1970-01-01”	“1970-01-02”	“1970-02-01”	“1970-02-02”	“1970-01-01”
What would be the output of the following code ? > x <- as.Date("2012-01-01") > y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S") > x-y	Time difference of 356.3095 days	Warning	NULL	Error	Warning
What would be the output of the following code ? > x <- as.POSIXct("2012-10-25 01:00:00") > y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT") > y-x	Time difference of 10 sec	Time difference of 1 sec	Time difference of 1 min	Time difference of 1 hour	Time difference of 1 hour
Which of the following code generate a uniform random number ?	x <- runif(1, 0, 10) if(x > 3) { y <- 10 } else { y <- 0 }	x <- run(1, 0, 10) if(x > 3) { y <- 10 } else { y <- 0 }	x <- random(1, 0, 10) if(x > 3) { y <- 10 } else { y <- 0 }	x <-runn(1, 0, 10) if(x > 3) { y <- 10 } else { y <- 0 }	x <- runif(1, 0, 10) if(x > 3) { y <- 10 } else { y <- 0 }
Point out the wrong stateme	for will execute a loop a fixed number of times	break will execute a loop while a condition is true	if and else tests a condition and acting on it	break will execute a loop while a condition is false	break will execute a loop while a condition is true
_____initiates an infinite loop right from the start.	next	for	repeat	while	repeat

_____is used to exit a loop immediately, regardless of what iteration the loop may be on.	next	break	repeat	while	break
____loops begin by testing a condition.	next	break	repeat	while	while
The_____function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object	seq()	seq_long()	seq_along()	seq_alo()	seq_along()
The_____function is used to extract subsets of rows from a data frame.	arrange()	filter()	select()	mutate()	filter()
The_____function is used to reorder rows of a data frame according to one of the variables/- columns	arrange()	filter()	select()	mutate()	arrange()
The_____function is designed to make this process easier.	arrange()	rename()	select()	mutate()	rename()
The_____function is used to generate summary statistics from the data frame within strata defined by a variable.	subset()	summarize()	group_by()	group()	group_by()
The_____package provides a concise set of operations for managing data frames.	summarize	dlyr	dpl	dplyr	dplyr

UNIT-V

SYLLABUS

Debugging: Something's Wrong! - Figuring Out What's Wrong - Debugging Tools in R. Using traceback() - Using debug() - Using recover(). **Profiling R Code:** Using system.time() . Timing Longer Expressions - The R Profiler – Using summaryRprof().**Simulation:** Generating Random Numbers - Setting the random number seed -Simulating a Linear Model - Random Sampling .

DEBUGGING

SOMETHING'S WRONG!

- R has a number of ways to indicate to you that something's not right. There are different levels of indication that can be used, ranging from mere notification to fatal error.
- Executing any function in R may result in the following conditions.
  • message: A generic notification/diagnostic message produced by the message() function;
  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.
- Here is an example of a warning that you might receive in the course of using R.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

- This warning lets you know that taking the log of a negative number results in a NaN value because you can't take the log of negative numbers. Nevertheless, R doesn't give an error, because it has a useful value that it can return, the NaN value. The warning is just there to let you know that something unexpected happens. Depending on what you are programming, you may have intentionally taken the log of a negative number in order to move on to another section of code.
- Here is another function that is designed to print a message to the console depending on the nature of its input.

```
> printmessage <- function(x) {
+ if(x > 0)
+ print("x is greater than zero")
+ else
+ print("x is less than or equal to zero")
```

```
+ invisible(x)
+ }
```

- This function is simple—it prints a message telling you whether x is greater than zero or less than or equal to zero. It also returns its input invisibly, which is a common practice with ―print‖ functions.
- Returning an object invisibly means that the return value does not get auto-printed when the function is called.
- Take a hard look at the function above and see if you can identify any bugs or problems.
- We can execute the function as follows.

```
> printmessage(1)
[1] "x is greater than zero"
```

- The function seems to work fine at this point. No errors, warnings, or messages.

```
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print("x is less than o\
r equal to zero"): missing value where TRUE/FALSE needed
```

- **What happened?**
  - Well, the first thing the function does is test if x > 0. But you can't do that test if x is a NA or NaN value.
  - R doesn't know what to do in this case so it stops with a fatal error.
  - We can fix this problem by anticipating the possibility of NA values and checking to see if the input is NA with the is.na() function.

```
> printmessage2 <- function(x) {
+ if(is.na(x))
+ print("x is a missing value!")
+ else if(x > 0)
+ print("x is greater than zero")
+ else
+ print("x is less than or equal to zero")
+ invisible(x)
+ }
```

- Now we can run the following.

```
> printmessage2(NA)
[1] "x is a missing value!"
```

And all is fine.

- Now what about the following situation.

```
> x <- log(c(-1, 2))
Warning in log(c(-1, 2)): NaNs produced
> printmessage2(x)
Warning in if (is.na(x)) print("x is a missing value!") else if (x > 0)
```

print("x is greater than zero") else print("x is less than or equal to zero"): the condition has length > 1 and only the first element will be used
[1] "x is a missing value!"

- Now what?? Why are we getting this warning? The warning says –the condition has length > 1 and only the first element will be used‖.
- The problem here is that I passed printmessage2() a vector x that was of length 2 rather than length
    1. Inside the body of printmessage2() the expression is.na(x) returns a vector that is tested in the if statement. However, if cannot take vector arguments so you get a warning. The fundamental problem here is that printmessage2() is not vectorized.
    2. We can solve this problem two ways. One is by simply not allowing vector arguments. The other way is to vectorize the printmessage2() function to allow it to take vector arguments.
- For the first way, we simply need to check the length of the input.

```
> printmessage3 <- function(x) {
+ if(length(x) > 1L)
+ stop("'x' has length > 1")
+ if(is.na(x))
+ print("x is a missing value!")
+ else if(x > 0)
+ print("x is greater than zero")
+ else
+ print("x is less than or equal to zero")
+ invisible(x)
+ }
```

- Now when we pass printmessage3() a vector we should get an error.
  > printmessage3(1:2)
  Error in printmessage3(1:2): 'x' has length > 1
  Vectorizing the function can be accomplished easily with the Vectorize() function.

```
> printmessage4 <- Vectorize(printmessage2)
> out <- printmessage4(c(-1, 2))
[1] "x is less than or equal to zero"
[1] "x is greater than zero"
```

- You can see now that the correct messages are printed without any warning or error. Note that I stored the return value of printmessage3() in a separate R object called out. This is because when I use the Vectorize() function it no longer preserves the invisibility of the return value

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 your code (or somebody else's), it's important first 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.

DEBUGGING TOOLS IN R

- R provides a number of tools to help you with debugging your code. The primary tools for debugging functions in R are
- traceback(): prints out the function call stack after an error occurs; does nothing if there's no error
- debug(): flags a function for ‑debug‖ mode which allows you to step through execution of a function one line at a time
- browser(): suspends the execution of a function wherever it is called and puts the function in debug mode
- trace(): allows you to insert debugging code into a function a specific places
- recover(): allows you to modify the error behavior so that you can browse the function call stack
- These functions are interactive tools specifically designed to allow you to pick through a function.
- There's also the more blunt technique of inserting print() or cat() statements in the function.

Using traceback()

The traceback() function prints out the function call stack after an error has occurred. The function
call stack is the sequence of functions that was called before the error occurred.
For example, you may have a function a() which subsequently calls function b() which calls c() and
then d(). If an error occurs, it may not be immediately clear in which function the error occurred.

The tracback() function shows you how many levels deep you were when the error occurred.

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
```

Here, it's clear that the error occurred inside the mean() function because the object x does not exist.

The traceback() function must be called immediately after an error occurs. Once another function is called, you lose the traceback.

Here is a slightly more complicated example using the lm() function for linear modeling.

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

You can see now that the error did not get thrown until the 7th level of the function call stack, in which case the eval() function tried to evaluate the formula y ~ x and realized the object y did not exist.

Looking at the traceback is useful for figuring out roughly where an error occurred but it's not useful
for more detailed debugging. For that you might turn to the debug() function.

Using debug()

The debug() function initiates an interactive debugger (also known as the "browser" in R) for a function. With the debugger, you can step through an R function one expression at a time to pinpoint
exactly where an error occurs.

The debug() function takes a function as its first argument. Here is an example of debugging the lm() function.

```
> debug(lm) ## Flag the 'lm()' function for interactive debugging
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
ret.x <- x
ret.y <- y
cl <- match.call()
...
if (!qr)
```

z$qr <- NULL
z
}
Browse[2]>
Now, every time you call the lm() function it will launch the interactive debugger. To turn this behavior off you need to call the undebug() function.
The debugger calls the browser at the very top level of the function body. From there you can step
through each expression in the body. There are a few special commands you can call in the browser:
• n executes the current expression and moves to the next expression
• c continues execution of the function and does not stop until either an error or the function exits
• Q quits the browser
Here's an example of a browser session with the lm() function.
Browse[2]> n ## Evalute this expression and move to the next one
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action", "offset"), names(mf), 0L)

While you are in the browser you can execute any other R function that might be available to you in a regular session. In particular, you can use ls() to see what is in your current environment (the function environment) and print() to print out the values of R objects in the function environment.
You can turn off interactive debugging with the undebug() function.
undebug(lm) ## Unflag the 'lm()' function for debugging

Using recover()

The recover() function can be used to modify the error behavior of R when an error occurs. Normally, when an error occurs in a function, R will print out an error message, exit out of the function, and return you to your workspace to await further commands.
With recover() you can tell R that when an error occurs, it should halt execution at the exact point
at which the error occurred. That can give you the opportunity to poke around in the environment in which the error occurred. This can be useful to see if there are any R objects or data that have been corrupted or mistakenly modified.

> options(error = recover) ## Change default R error behavior
> read.csv("nosuchfile") ## This code doesn't work
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
cannot open file 'nosuchfile': No such file or directory
Enter a frame number, or 0 to exit
1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec =
3: file(file, "rt")
Selection:
The recover() function will first print out the function call stack when an error occurrs. Then, you can choose to jump around the call stack and investigate the problem. When you choose a frame number, you will be put in the browser (just like the interactive debugger triggered with debug()) and will have the ability to poke around.

PROFILING R CODE

- R comes with a profiler to help you optimize your code and improve its performance. In generally, it's usually a bad idea to focus on optimizing your code at the very beginning of development. Rather, in the beginning it's better to focus on translating your ideas into code and writing code that's coherent and readable. The problem is that heavily optimized code tends to be obscure and difficult to read, making it harder to debug and revise. Better to get all the bugs out first, and then focus on optimizing.
- Of course, when it comes to optimizing code, the question is what should you optimize? Well, clearly should optimize the parts of your code that are running slowly, but how do we know what parts those are? This is what the profiler is for. Profiling is a systematic way to examine how much time is spent in different parts of a program.
- Sometimes profiling becomes necessary as a project grows and layers of code are placed on top of each other. Often you might write some code that runs fine once. But then later, you might put that same code in a big loop that runs 1,000 times. Now the original code that took 1 second to run is taking 1,000 seconds to run! Getting that little piece of original code to run faster will help the entire loop.
- It's tempting to think you just know where the bottlenecks in your code are. I mean, after all, you write it! But trust me, I can't tell you how many times I've been surprised at where exactly my code is spending all its time. The reality is that profiling is better than guessing. Better to collect some data than to go on hunches alone. Ultimately, getting the biggest impact on speeding up code depends on knowing where the code spends most of its time. This cannot be done without some sort of rigorous performance analysis or profiling.

- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil —Donald Knuth
- The basic principles of optimizing your code are:
  - Design first, then optimize
  - Remember: Premature optimization is the root of all evil
  - Measure (collect data), don't guess.
  - If you're going to be scientist, you need to apply the same principles here!

Using system.time()

- They system.time() function takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression. The system.time() function computes the time (in seconds) needed to execute an expression and if there's an error, gives the time until the error occurred. The function returns an object of class proc_time which contains two useful bits of information:
  - user time: time charged to the CPU(s) for this expression
  - elapsed time: "wall clock" time, the amount of time that passes for you as you're sitting there Usually, the user time and elapsed time are relatively close, for straight computing tasks. But there are a few situations where the two can diverge, sometimes dramatically.

  The elapsed time may be greater than the user time if the CPU spends a lot of time waiting around.

  This commonly happens if your R expression involes some input or output, which depends on the activity of the file system and the disk (or the Internet, if using a network connection).

  The elapsed time may be smaller than the user time if your machine has multiple cores/processors (and is capable of using them).
- For example, multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL) can greatly speed up linear algebra calculations and are commonly installed on even desktop systems these days. Also, parallel processing done via something like the parallel package can make the elapsed time smaller than the user time.
- When you have multiple processors/- cores/machines working in parallel, the amount of time that the collection of CPUs spends working on a problem is the same as with a single CPU, but because they are operating in parallel, there is a savings in elapsed time.
- Here's an example of where the elapsed time is greater than the user time.

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
user system elapsed
0.004 0.002 0.431
```

- Most of the time in this expression is spent waiting for the connection to the web server and waiting for the data to travel back to my computer. This doesn't involve the CPU and so the CPU simply waits around for things to get done. Hence, the user time is small.
- In this example, the elapsed time is smaller than the user time.

```
## Elapsed time < user time
> hilbert <- function(n) {
+ i <- 1:n
+ 1 / outer(i - 1, i, "+")
+ }
> x <- hilbert(1000)
> system.time(svd(x))
user system elapsed
1.035 0.255 0.462
```

- In this case I ran singular value decomposition on the matrix in x, which is a common linear algebra procedure. Because my computer is able to split the work across multiple processors, the elapsed time is about half the user time.

TIMING LONGER EXPRESSIONS

- You can time longer expressions by wrapping them in curly braces within the call to system.time().

```
> system.time({
+ n <- 1000
+ r <- numeric(n)
+ for(i in 1:n) {
+ x <- rnorm(n)
+ r[i] <- mean(x)
+ }
+ })
user system elapsed
0.086 0.001 0.088
```

- If your expression is getting pretty long (more than 2 or 3 lines), it might be better to either break it into smaller pieces or to use the profiler. The problem is that if the expression is too long, you won't be able to identify which part of the code is causing the bottleneck.

THE R PROFILER

- Using system.time() allows you to test certain functions or code blocks to see if they are taking excessive amounts of time. However, this approach assumes that you already

know where the problem is and can call system.time() on it that piece of code. What if you don't know where to start?

- This is where the profiler comes in handy. The Rprof() function starts the profiler in R. Note that R must be compiled with profiler support (but this is usually the case). In conjunction with Rprof(), we will use the summaryRprof() function which summarizes the output from Rprof() (otherwise it's not really readable). Note that you should NOT use system.time() and Rprof() together, or you will be sad.
- Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function. By default, the profiler samples the function call stack every 0.02 seconds. This means that if your code runs very quickly (say, under 0.02 seconds), the profiler is not useful. But of your code runs that fast, you probably don't need the profiler.
- The profiler is started by calling the Rprof() function.

  > Rprof() ## Turn on the profiler
- You don't need any other arguments. By default it will write its output to a file called Rprof.out. You can specify the name of the output file if you don't want to use this default.
- Once you call the Rprof() function, everything that you do from then on will be measured by the profiler. Therefore, you usually only want to run a single R function or expression once you turn on the profiler and then immediately turn it off. The reason is that if you mix too many function calls together when running the profiler, all of the results will be mixed together and you won't be able to sort out where the bottlenecks are. In reality, I usually only run a single function with the profiler on.
- The profiler can be turned off by passing NULL to Rprof().

  > Rprof(NULL) ## Turn off the profiler
- The raw output from the profiler looks something like this. Here I'm calling the lm() function on some data with the profiler running.

lm(y ~ x)
sample.interval=10000
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"

"lm.fit" "lm"
"lm.fit" "lm"
"lm.fit" "lm"

- At each line of the output, the profiler writes out the function call stack. For example, on the very first line of the output you can see that the code is 8 levels deep in the call stack. This is where you need the summaryRprof() function to help you interpret this data.

Using summaryRprof()

- The summaryRprof() function tabulates the R profiler output and calculates how much time is spendin which function. There are two methods for normalizing the data.
  - ―by.total‖ divides the time spend in each function by the total run time
  - ―by.self‖ does the same as ―by.total‖ but first subtracts out time spent in functions above the current function in the call stack. I personally find this output to be much more useful.
- Here is what summaryRprof() reports in the ―by.total‖ output.

$by.total
total.time total.pct self.time self.pct
"lm" 7.41 100.00 0.30 4.05
"lm.fit" 3.50 47.23 2.99 40.35
"model.frame.default" 2.24 30.23 0.12 1.62
"eval" 2.24 30.23 0.00 0.00
"model.frame" 2.24 30.23 0.00 0.00
"na.omit" 1.54 20.78 0.24 3.24
"na.omit.data.frame" 1.30 17.54 0.49 6.61
"lapply" 1.04 14.04 0.00 0.00
"[.data.frame" 1.03 13.90 0.79 10.66
"[" 1.03 13.90 0.00 0.00
"as.list.data.frame" 0.82 11.07 0.82 11.07
"as.list" 0.82 11.07 0.00 0.00

- Because lm() is the function that I called from the command line, of course 100% of the time is spent somewhere in that function. However, what this doesn't show is that if lm() immediately calls another function (like lm.fit(), which does most of the heavy lifting), then in reality, most of the time is spent in that function, rather than in the top-level lm() function.
- The ―by.self‖ output corrects for this discrepancy.

$by.self
self.time self.pct total.time total.pct
"lm.fit" 2.99 40.35 3.50 47.23
"as.list.data.frame" 0.82 11.07 0.82 11.07
"[.data.frame" 0.79 10.66 1.03 13.90
"structure" 0.73 9.85 0.73 9.85

"na.omit.data.frame" 0.49 6.61 1.30 17.54
"list" 0.46 6.21 0.46 6.21
"lm" 0.30 4.05 7.41 100.00
"model.matrix.default" 0.27 3.64 0.79 10.66
"na.omit" 0.24 3.24 1.54 20.78
"as.character" 0.18 2.43 0.18 2.43
"model.frame.default" 0.12 1.62 2.24 30.23
"anyDuplicated.default" 0.02 0.27 0.02 0.27

- Now you can see that only about 4% of the runtime is spent in the actual lm() function, whereas over 40% of the time is spent in lm.fit(). In this case, this is no surprise since the lm.fit() function is the function that actually fits the linear model.
- You can see that a reasonable amount of time is spent in functions not necessarily associated with linear modeling (i.e. as.list.data.frame, [.data.frame). This is because the lm() function does a bit of pre-processing and checking before it actually fits the model. This is common with modeling functions—the preprocessing and checking is useful to see if there are any errors. But those two functions take up over 1.5 seconds of runtime. What if you want to fit this model 10,000 times?
- You're going to be spending a lot of time in preprocessing and checking.
- The final bit of output that summaryRprof() provides is the sampling interval and the total runtime.

$sample.interval
[1] 0.02
$sampling.time
[1] 7.41

SIMULATION

GENERATING RANDOM NUMBERS

- Simulation is an important (and big) topic for both statistics and for a variety of other areas where there is a need to introduce randomness. Sometimes you want to implement a statistical procedure that requires random number generation or samplie (i.e. Markov chain Monte Carlo, the bootstrap, random forests, bagging) and sometimes you want to simulate a system and random number generators can be used to model random inputs.
- R comes with a set of pseudo-random number generators that allow you to simulate from well known probability distributions like the Normal, Poisson, and binomial. Some example functions for probability distributions in R

  • rnorm: generate random Normal variates with a given mean and standard deviation

  • dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)

- pnorm: evaluate the cumulative distribution function for a Normal distribution
- rpois: generate random Poisson variates with a given rate

- For each probability distribution there are typically four functions available that start with a –r‖, –d‖, –p‖, and –q‖. The –r‖ function is the one that actually simulates random numbers from that distribution. The other functions are prefixed with a
  - d for density
  - r for random number generation
  - p for cumulative distribution
  - q for quantile function (inverse cumulative distribution)
- If you're only interested in simulating random numbers, then you will likely only need the –r‖ functions and not the others. However, if you intend to simulate from arbitrary probability distributions using something like rejection sampling, then you will need the other functions too.
- Probably the most common probability distribution to work with the Normal distribution (also known as the Gaussian). Working with the Normal distributions requires using these four functions

dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)

- Here we simulate standard Normal random numbers with mean 0 and standard deviation 1.

```
> ## Simulate standard Normal random numbers
> x <- rnorm(10)
> x
[1] 0.01874617 -0.18425254 -1.37133055 -0.59916772 0.29454513
[6] 0.38979430 -1.20807618 -0.36367602 -1.62667268 -0.25647839
```

- We can modify the default parameters to simulate numbers with mean 20 and standard deviation 2.

```
> x <- rnorm(10, 20, 2)
> x
[1] 22.20356 21.51156 19.52353 21.97489 21.48278 20.17869 18.09011
[8] 19.60970 21.85104 20.96596
> summary(x)
Min. 1st Qu. Median Mean 3rd Qu. Max.
18.09 19.75 21.22 20.74 21.77 22.20
```

- If you wanted to know what was the probability of a random Normal variable of being less than, say, 2, you could use the pnorm() function to do that calculation.

```
> pnorm(2)
[1] 0.9772499
```

- You never know when that calculation will come in handy

SETTING THE RANDOM NUMBER SEED

- When simulating any random numbers it is essential to set the random number seed. Setting the random number seed with set.seed() ensures reproducibility of the sequence of random numbers.
- For example, I can generate 5 Normal random numbers with rnorm()

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
```

Note that if I call rnorm() again I will of course get a different set of 5 random numbers.

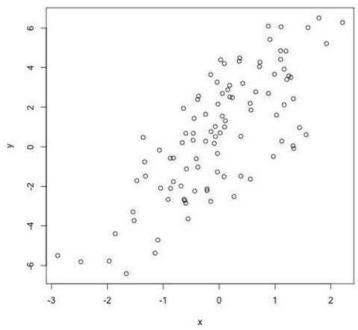
```
> rnorm(5)
[1] -0.8204684 0.4874291 0.7383247 0.5757814 -0.3053884
```

If I want to reproduce the original set of random numbers, I can just reset the seed with set.seed().

```
> set.seed(1)
> rnorm(5) ## Same as before
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
```

- In general, you should always set the random number seed when conducting a simulation!
- Otherwise, you will not be able to reconstruct the exact numbers that you produced in an analysis. It is possible to generate random numbers from other probability distributions like the Poisson. The Poisson distribution is commonly used to model data that come in the form of counts.

```
> rpois(10, 1) ## Counts with a mean of 1
[1] 0 0 1 1 2 1 1 4 1 2
> rpois(10, 2) ## Counts with a mean of 2
[1] 4 1 2 0 1 1 0 1 4 1
> rpois(10, 20) ## Counts with a mean of 20
[1] 19 19 24 23 22 24 23 20 11 22
```



plot of chunk Linear Model

SIMULATING A LINEAR MODEL

- Simulating random numbers is useful but sometimes we want to simulate values that come from a specific model. For that we need to specify the model and then simulate from it using the functions described above.
- Suppose we want to simulate from the following linear model

y = β0 + β1x + ε
where ε ~ N (0, 2
2
). Assume x ~ N (0, 1
2
), β0 = 0.5 and β1 = 2. The variable x might represent
an important predictor of the outcome y. Here's how we could do that in R.

```
> ## Always set your seed!
> set.seed(20)
>
> ## Simulate predictor variable
> x <- rnorm(100)
>
> ## Simulate the error term
> e <- rnorm(100, 0, 2)
>
> ## Compute the outcome via the model
> y <- 0.5 + 2 * x + e
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-6.4080 -1.5400 0.6789 0.6893 2.9300 6.5050
```

We can plot the results of the model simulation.

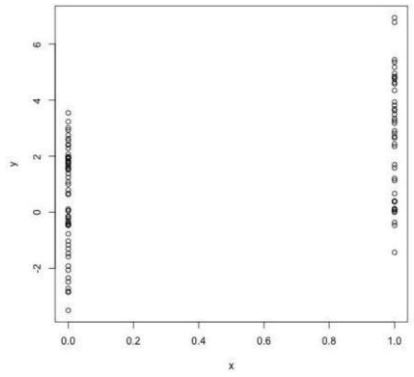
```
> plot(x, y)
```

- What if we wanted to simulate a predictor variable x that is binary instead of having a Normal distribution. We can use the rbinom() function to simulate binary random variables.

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> str(x) ## 'x' is now 0s and 1s
int [1:100] 1 0 0 1 0 0 0 0 1 0 ...
```

Then we can procede with the rest of the model as before.

```
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> plot(x, y)
```



plot of chunk Linear Model Binary

- We can also simulate from generalized linear model where the errors are no longer from a Normal distribution but come from some other distribution. For examples, suppose we want to simulate from a Poisson log-linear model where

$$Y \sim Poisson(\mu)$$

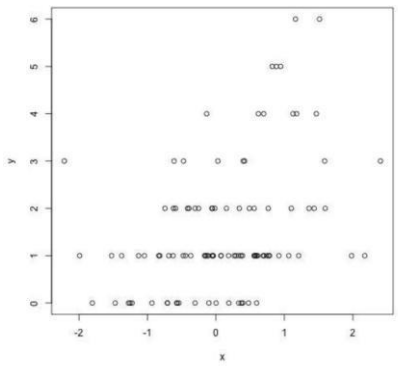
$$\log \mu = \beta 0 + \beta 1 x$$

and β0 = 0.5 and β1 = 0.3. We need to use the rpois() function for this

```
> set.seed(1)
>
> ## Simulate the predictor variable as before
> x <- rnorm(100)
```

Now we need to compute the log mean of the model and then exponentiate it to get the mean to pass to rpois().

```
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.00 1.00 1.00 1.55 2.00 6.00
> plot(x, y)
```



plot of chunk Poisson Log-Linear Model

- You can build arbitrarily complex models like this by simulating more predictors or making transformations of those predictors (e.g. squaring, log transformations, etc.).

RANDOM SAMPLING

- The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
>
> ## Doesn't have to be numbers
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
>
> ## Do a random permutation
> sample(1:10)
[1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
[1] 2 3 4 1 9 5 10 8 6 7
>
> ## Sample w/replacement
> sample(1:10, replace = TRUE)
[1] 2 9 7 8 2 8 5 9 7 8
```

- To sample more complicated things, such as rows from a data frame or a list, you can sample the indices into an object rather than the elements of the object itself.
- Here's how you can sample rows from a data frame.

```
> library(datasets)
> data(airquality)
> head(airquality)
```

Ozone Solar.R Wind Temp Month Day
1 41 190 7.4 67 5 1
2 36 118 8.0 72 5 2
3 12 149 12.6 74 5 3
4 18 313 11.5 62 5 4
5 NA NA 14.3 56 5 5
6 28 NA 14.9 66 5 6

- Now we just need to create the index vector indexing the rows of the data frame and sample directly from that index vector.

```
> set.seed(20)
>
> ## Create index vector
> idx <- seq_len(nrow(airquality))
>
> ## Sample from the index vector
> samp <- sample(idx, 6)
> airquality[samp, ]
```

Ozone Solar.R Wind Temp Month Day
135 21 259 15.5 76 9 12
117 168 238 3.4 81 8 25
43 NA 250 9.2 92 6 12
80 79 187 5.1 87 7 19
144 13 238 12.6 64 9 21
146 36 139 10.3 81 9 23

Other more complex objects can be sampled in this way, as long as there's a way to index the sub elements of the object.

POSSIBLE QUESTIONS

UNIT – V

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is Debugging?
2. Define Random Samplings
3. What is the use of sample ()?
4. When the random number seed set?
5. Give some examples for probability distributions in R.
6. What are the Debugging tools in R programming
7. Define recover()
8. What is the process of debug ()?
9. What is meant by Simulation?
10. What is the use of traceback()?

PART – C (6 MARKS)

1. Explain the process of Debugging
2. Discuss the Debugging tools in R
3. Explain the process of traceback ()
4. Discuss in detail (i) recover () (ii) debug ()
5. Explain about system.time() with suitable examples
6. Explain about the R profiler
7. Explain how to simulate a linear model
8. Explain about Random Samplings
9. Explain about Simulation and its process
10. Explain the process of Using summaryRprof()

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2018 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - IV : (Objective Type Multiple choice Questions each Question carries one Mark)

R PROGRAMMING [18CAU404A]

PART - A (Online Examination)

Questions	Opt1	Opt2	Opt3	Opt4	Key
Which of the following is apply function in R ?	apply()	tapply()	fapply()	sapply()	tapply()
Functions are defined using the_____directive and are stored as R objects	function()	funct()	functions()	func()	function()
Point out the wrong statement :	Functions in R are “second class objects”	The writing of a function allows a developer to create an interface to the code, that is explicitly specified with a set of parameters	Functions provides an abstraction of the code to potential users	Functions in R are “first class objects”	Functions in R are “second class objects”
What will be the output of the following code ? > f <- function() { + ## This is an empty function + } > class(f)	“data"	“procedure”	“function”	“class”	“function”
The_____function returns a list of all the	formals()	funct()	formal()	function()	formals()

Point out the wrong statement :	Functions can be nested, so that you can define a function inside of another function	The value returned by the call to function is not a function	A formal argument can be a symbol, a statement of the form 'symbol = expression', or the special formal argument	The first component of the function declaration is the keyword function	A formal argument can be a symbol, a statement of the form 'symbol = expression', or the special formal argument
You can check to see whether an R object is NULL	is.null()	is.nullobj()	null()	is.obj()	is.null()
Which of the following code will print NULL ?	> args(pastebin)	> args(paste)	> args(pastebin)	> argc(pastebin)	> args(paste)
What will be the output of following code snippet ?	"a+b"	"a=b"	"a:b"	"a-b"	"a:b"
What will be the output of following code ? > f <- function(a, b) { + print(a) + print(b) + }	32	42	52	45	45
_____is an indication that a fatal problem has occurred and execution of the function stops	message	error	warning	stop	error
What would be the value of following expression ? log(-1)	0	Null	Warning in log(-1): NaNs produced	1	Warning in log(-1): NaNs produced
Warnings are generated by the_____function	warning()	error()	run()	runif()	warning()

Point out the correct statement :	POSIX represents a portable operating system interface, primarily for UNIX systems	There are different levels of indication that can be used, ranging from mere notification to fatal error	The default input format for POSIX dates consists of the month, followed by the year and day, separated by slashes or dashes	Dates are not stored in the POSIX format are date/time values	POSIX represents a portable operating system interface, primarily for UNIX systems
To get the current date, the_____function will return a Date object which can be converted to a	Sys.Time	Sys.Date	Sys.DateTime	Sys.TimeDate	Sys.Date
Which of the followin code represents internal representation of a Date object ?	class(as.Date("1970-01-02"))	classint(as.Date("1970-01-02"))	unclass(as.Date("1970-01-02"))	unclassint(as.Date("1970-01-02"))	unclass(as.Date("1970-01-02"))
What will be the output of following code snippet ? > lm <- function(x) { x * x }	function(x) { x * x }	func(x) { x * x }	function(x) { x / x }	funct(x) { x / x }	function(x) { x * x }
Point out the correct statement :	The search list can be found by using the searchlist() function	The search list can be found by using the search() function	The global environment or the user's workspace is always the second element of the search list	The search can be found by using the searchlt() function	The search list can be found by using the search() function
A function, together with an environment, makes up what is called a_____closure.	formal	function	reflective	unformal	function
R uses_____scoping or static scoping.	reflective	transitive	lexical	formal	lexical
The only environment without a parent is the	full	half	null	empty	empty
The_____for R are the main feature that make it different from the original S language	scoping rules	closure rules	environment rules	lexical rules	scoping rules
The_____function is a kind of "constructor function" that can be used to construct other	make.pow()	make.power()	keep.power()	keep.pow()	make.power()

What will be the output of following code ? > g <- function(x) { + a <- 3 + x+a+y + ## 'y' is a free variable + }	9	42	8	Error	Error
_____functions can be “built which contain all of the necessary data for evaluating the function	Objective	reflective	Nested	lexical	Objective
_____require you to pass a function whose argument is a vector of parameters (	optimize()	optimise()	opt()	oplt()	opt()
The_____function is used to plot negative	plot()	graph()	graph.plot()	plot.graph()	plot()
_____loop over a list and evaluate a function on	apply()	lapply()	sapply()	mapply()	apply()
Point out the wrong statement :	Multi-line expressions with curly braces are just not that easy to sort through when working on the command line	lappy() loops over a list, iterating over each element in that list	lapply() does not always returns a list	lapply() always returns a list, regardless of the class of the input.	lapply() does not always returns a list
_____function is same as lapply in R	apply()	lapply()	sapply()	mapply()	sapply()
Which of the following is multivariate version of	apply()	lapply()	sapply()	mapply()	mapply()
Point out the correct statement :	lapply() takes elements of the list and passes them as the first argument of the function you are applying	You can use lapply() to evaluate a function multiple times each with a different argument	Functions that you pass to lapply() may have other arguments	The lapply() function and its friends make heavy use of anonymous functions.	The lapply() function and its friends make heavy use of anonymous functions.
_____applies a function over the margins of an	apply()	lapply()	sapply()	mapply()	apply()
_____is used to apply a function over subsets of	apply()	lapply()	tapply()	mapply()	tapply()

lappy functions takes_____arguments in R	two	three	four	five	four
Point out the wrong statement :	The sapply() function behaves similarly to lapply()	With multiple factors and many levels, creating an interaction can result in many levels that are empty	apply() can be thought of as a combination of split() and sapply() for vectors only	tapply() can be thought of as a combination of split() and sapply() for vectors only.	apply() can be thought of as a combination of split() and sapply() for vectors only
The_____function takes a vector or other objects and splits it into groups determined by a factor or	apply()	lsplit()	split()	mapply()	split()
What will be the output of the following code ? > nLL <- make.NegLogLik(normals, c(1, FALSE)) > optimize(nLL, c(1e-6, 10))$minimum	1.217775	1.800596	3.73424	empty	1.800596
Point out the correct statement :	An environment is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value	If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the child environment	After the top-level environment, the search continues down the search list until we hit the parent environment	Every environment has a parent environment and it is not possible for an environment to have multiple “children”.	An environment is a collection of (symbol, value) pairs, i.e. x is a symbol and 3.14 might be its value
Point out the wrong statement :	Dynamic scoping turns out to be particularly useful for simplifying statistical computations	Lexical scoping turns out to be particularly useful for simplifying statistical computations	The scoping rules of a language determine how values are assigned to free variables	Free variables are not formal arguments and are not local variables	Dynamic scoping turns out to be particularly useful for simplifying statistical computations

What would be the output of the following code ? > printmessage <- function(x) { + if(x > 0) + print("x is greater than zero") + else + print("x is less than or equal to zero") + invisible(x) + } > printmessage(NA)	Error	Warning	Messages	Data	Error
Arguments to functions are evaluated_____, so they are evaluated only as needed in the body of	completely	lazily	directly	inversely	lazily
In R the calling environment is known as the	data frame	child fram	parent frame	called frame	parent frame
_____turns out to be particularly useful for simplifying statistical computations	scoping rules	Lexical scoping	dynamic scoping	scoping	Lexical scoping
Optimization routines in R like_____,_____and _____ require you to pass a function whose argument is a vector of parameters	opti(), lm(), and optimize()	opt(), nm(), and optimi()	optim(), nlm(), and optimize()	optim(), lmn(), and optimize()	optim(), nlm(), and optimize()
Optimization functions in R_____functions, so you need to use the negative loglikelihood.	minimize	maximize	calling	return	minimize
The mapply() function can be use to automatically	minimize	maximize	vectorize	calling	vectorize
The_____function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop	apply()	lsplit()	split()	mapply()	split()
_____expressions with curly braces are just not that easy to sort through when working on the	looping	Multi-line	lexical	Single-line	Multi-line
we are passing the_____function as an	mode()	median()	mean()	split()	mean()
The lapply() function and its friends make heavy use	calling	unanonymous	anonymous	member	anonymous
What will be the output of the following code ? > f <- function() { + ## This is an empty function + }	0	No result	NULL	Error	NULL

Which of the following code will print “Hello, world!” ?	> f <- function() { cat("Hello, world!\n") } > f()	> f <- function() { cat("Hello, World!\n") } > f()	> f <- function() { cat("Hello world!\n") } > f()	> f <- function() { cat("hello World!\n") } > f()	> f <- function() { cat("Hello, world!\n") } > f()

What will be the output of following code ? > f <- function(num) {
+ for(i in seq_len(num)) {
+ cat("Hello, world!\n")
+ }
+ }
	Hello, world!
	Hello, world!
		Hello, world!
		Hello, world!
		Hello, world!
			Hello, world!
			Hello, world!
			Hello, world!
			Hello, world!
				Hello, world!	Hello, world!
					Hello, world!
					Hello, world!

What will be the output of the following code ? > f <- function(num = 1) {
+ hello <- "Hello, world!\n"
+ for(i in seq_len(num)) {
+ cat(hello)
+ }
+ chars <- nchar(hello) * num
+ chars
+ }
	Hello, world! [1] 14	Hello, world! [1] 15	Hello, world! [1] 16	Hello, world! [1] 17	Hello, world! [1] 14

What will be the output of following code ? > f <- function(a, b) { + a^2 + }	4	3	2	1	4
What will be the output of following code ? > f <- function(a, b) { + print(a) + print(b) + }	32	42	52	45	45
What would be the output of the following code ? > p <- as.POSIXlt(x) > names(unclass(p))	1	2	3	4	1
_____will not simplify the result and will	apply()	lapply()	tapply()	mapply()	tapply()

_____keeps track of the function call stack at regularly sampled intervals and tabulates how much	summaryRprof()	Rprof()	system.time()	prof()	Rprof()

UNIT-V

SYLLABUS

Debugging: Something's Wrong! - Figuring Out What's Wrong - Debugging Tools in R. Using traceback() - Using debug() - Using recover(). **Profiling R Code:** Using system.time() . Timing Longer Expressions - The R Profiler – Using summaryRprof().**Simulation:** Generating Random Numbers - Setting the random number seed -Simulating a Linear Model - Random Sampling .

DEBUGGING

SOMETHING'S WRONG!

- R has a number of ways to indicate to you that something's not right. There are different levels of indication that can be used, ranging from mere notification to fatal error.
- Executing any function in R may result in the following conditions.
  • message: A generic notification/diagnostic message produced by the message() function;
  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.
- Here is an example of a warning that you might receive in the course of using R.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

- This warning lets you know that taking the log of a negative number results in a NaN value because you can't take the log of negative numbers. Nevertheless, R doesn't give an error, because it has a useful value that it can return, the NaN value. The warning is just there to let you know that something unexpected happens. Depending on what you are programming, you may have intentionally taken the log of a negative number in order to move on to another section of code.
- Here is another function that is designed to print a message to the console depending on the nature of its input.

```
> printmessage <- function(x) {
+ if(x > 0)
+ print("x is greater than zero")
+ else
+ print("x is less than or equal to zero")
```

```
+ invisible(x)
+ }
```

- This function is simple—it prints a message telling you whether x is greater than zero or less than or equal to zero. It also returns its input invisibly, which is a common practice with ―print‖ functions.
- Returning an object invisibly means that the return value does not get auto-printed when the function is called.
- Take a hard look at the function above and see if you can identify any bugs or problems.
- We can execute the function as follows.

```
> printmessage(1)
[1] "x is greater than zero"
```

- The function seems to work fine at this point. No errors, warnings, or messages.

```
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print("x is less than o\
r equal to zero"): missing value where TRUE/FALSE needed
```

- **What happened?**
  - Well, the first thing the function does is test if x > 0. But you can't do that test if x is a NA or NaN value.
  - R doesn't know what to do in this case so it stops with a fatal error.
  - We can fix this problem by anticipating the possibility of NA values and checking to see if the input is NA with the is.na() function.

```
> printmessage2 <- function(x) {
+ if(is.na(x))
+ print("x is a missing value!")
+ else if(x > 0)
+ print("x is greater than zero")
+ else
+ print("x is less than or equal to zero")
+ invisible(x)
+ }
```

- Now we can run the following.

```
> printmessage2(NA)
[1] "x is a missing value!"
```

And all is fine.

- Now what about the following situation.

```
> x <- log(c(-1, 2))
Warning in log(c(-1, 2)): NaNs produced
> printmessage2(x)
Warning in if (is.na(x)) print("x is a missing value!") else if (x > 0)
```

print("x is greater than zero") else print("x is less than or equal to zero"): the condition has length > 1 and only the first element will be used
[1] "x is a missing value!"

- Now what?? Why are we getting this warning? The warning says –the condition has length > 1 and only the first element will be used‖.
- The problem here is that I passed printmessage2() a vector x that was of length 2 rather than length
    1. Inside the body of printmessage2() the expression is.na(x) returns a vector that is tested in the if statement. However, if cannot take vector arguments so you get a warning. The fundamental problem here is that printmessage2() is not vectorized.
    2. We can solve this problem two ways. One is by simply not allowing vector arguments. The other way is to vectorize the printmessage2() function to allow it to take vector arguments.
- For the first way, we simply need to check the length of the input.

```
> printmessage3 <- function(x) {
+ if(length(x) > 1L)
+ stop("'x' has length > 1")
+ if(is.na(x))
+ print("x is a missing value!")
+ else if(x > 0)
+ print("x is greater than zero")
+ else
+ print("x is less than or equal to zero")
+ invisible(x)
+ }
```

- Now when we pass printmessage3() a vector we should get an error.
  > printmessage3(1:2)
  Error in printmessage3(1:2): 'x' has length > 1
  Vectorizing the function can be accomplished easily with the Vectorize() function.

```
> printmessage4 <- Vectorize(printmessage2)
> out <- printmessage4(c(-1, 2))
[1] "x is less than or equal to zero"
[1] "x is greater than zero"
```

- You can see now that the correct messages are printed without any warning or error. Note that I stored the return value of printmessage3() in a separate R object called out. This is because when I use the Vectorize() function it no longer preserves the invisibility of the return value

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 your code (or somebody else's), it's important first 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.

DEBUGGING TOOLS IN R

- R provides a number of tools to help you with debugging your code. The primary tools for debugging functions in R are
- traceback(): prints out the function call stack after an error occurs; does nothing if there's no error
- debug(): flags a function for ‑debug‖ mode which allows you to step through execution of a function one line at a time
- browser(): suspends the execution of a function wherever it is called and puts the function in debug mode
- trace(): allows you to insert debugging code into a function a specific places
- recover(): allows you to modify the error behavior so that you can browse the function call stack
- These functions are interactive tools specifically designed to allow you to pick through a function.
- There's also the more blunt technique of inserting print() or cat() statements in the function.

Using traceback()

The traceback() function prints out the function call stack after an error has occurred. The function
call stack is the sequence of functions that was called before the error occurred.
For example, you may have a function a() which subsequently calls function b() which calls c() and
then d(). If an error occurs, it may not be immediately clear in which function the error occurred.

The tracback() function shows you how many levels deep you were when the error occurred.

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
```

Here, it's clear that the error occurred inside the mean() function because the object x does not exist.

The traceback() function must be called immediately after an error occurs. Once another function is called, you lose the traceback.

Here is a slightly more complicated example using the lm() function for linear modeling.

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

You can see now that the error did not get thrown until the 7th level of the function call stack, in which case the eval() function tried to evaluate the formula y ~ x and realized the object y did not exist.

Looking at the traceback is useful for figuring out roughly where an error occurred but it's not useful
for more detailed debugging. For that you might turn to the debug() function.

Using debug()

The debug() function initiates an interactive debugger (also known as the "browser" in R) for a function. With the debugger, you can step through an R function one expression at a time to pinpoint
exactly where an error occurs.

The debug() function takes a function as its first argument. Here is an example of debugging the lm() function.

```
> debug(lm) ## Flag the 'lm()' function for interactive debugging
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
ret.x <- x
ret.y <- y
cl <- match.call()
...
if (!qr)
```

z$qr <- NULL
z
}
Browse[2]>
Now, every time you call the lm() function it will launch the interactive debugger. To turn this behavior off you need to call the undebug() function.
The debugger calls the browser at the very top level of the function body. From there you can step
through each expression in the body. There are a few special commands you can call in the browser:
• n executes the current expression and moves to the next expression
• c continues execution of the function and does not stop until either an error or the function exits
• Q quits the browser
Here's an example of a browser session with the lm() function.
Browse[2]> n ## Evalute this expression and move to the next one
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action", "offset"), names(mf), 0L)

While you are in the browser you can execute any other R function that might be available to you in a regular session. In particular, you can use ls() to see what is in your current environment (the function environment) and print() to print out the values of R objects in the function environment.
You can turn off interactive debugging with the undebug() function.
undebug(lm) ## Unflag the 'lm()' function for debugging

Using recover()

The recover() function can be used to modify the error behavior of R when an error occurs. Normally, when an error occurs in a function, R will print out an error message, exit out of the function, and return you to your workspace to await further commands.
With recover() you can tell R that when an error occurs, it should halt execution at the exact point
at which the error occurred. That can give you the opportunity to poke around in the environment in which the error occurred. This can be useful to see if there are any R objects or data that have been corrupted or mistakenly modified.

> options(error = recover) ## Change default R error behavior
> read.csv("nosuchfile") ## This code doesn't work
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
cannot open file 'nosuchfile': No such file or directory
Enter a frame number, or 0 to exit
1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec =
3: file(file, "rt")
Selection:
The recover() function will first print out the function call stack when an error occurrs. Then, you can choose to jump around the call stack and investigate the problem. When you choose a frame number, you will be put in the browser (just like the interactive debugger triggered with debug()) and will have the ability to poke around.

PROFILING R CODE

- R comes with a profiler to help you optimize your code and improve its performance. In generally, it's usually a bad idea to focus on optimizing your code at the very beginning of development. Rather, in the beginning it's better to focus on translating your ideas into code and writing code that's coherent and readable. The problem is that heavily optimized code tends to be obscure and difficult to read, making it harder to debug and revise. Better to get all the bugs out first, and then focus on optimizing.
- Of course, when it comes to optimizing code, the question is what should you optimize? Well, clearly should optimize the parts of your code that are running slowly, but how do we know what parts those are? This is what the profiler is for. Profiling is a systematic way to examine how much time is spent in different parts of a program.
- Sometimes profiling becomes necessary as a project grows and layers of code are placed on top of each other. Often you might write some code that runs fine once. But then later, you might put that same code in a big loop that runs 1,000 times. Now the original code that took 1 second to run is taking 1,000 seconds to run! Getting that little piece of original code to run faster will help the entire loop.
- It's tempting to think you just know where the bottlenecks in your code are. I mean, after all, you write it! But trust me, I can't tell you how many times I've been surprised at where exactly my code is spending all its time. The reality is that profiling is better than guessing. Better to collect some data than to go on hunches alone. Ultimately, getting the biggest impact on speeding up code depends on knowing where the code spends most of its time. This cannot be done without some sort of rigorous performance analysis or profiling.

- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil —Donald Knuth
- The basic principles of optimizing your code are:
  - Design first, then optimize
  - Remember: Premature optimization is the root of all evil
  - Measure (collect data), don't guess.
  - If you're going to be scientist, you need to apply the same principles here!

Using system.time()

- They system.time() function takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression. The system.time() function computes the time (in seconds) needed to execute an expression and if there's an error, gives the time until the error occurred. The function returns an object of class proc_time which contains two useful bits of information:
  - user time: time charged to the CPU(s) for this expression
  - elapsed time: "wall clock" time, the amount of time that passes for you as you're sitting there Usually, the user time and elapsed time are relatively close, for straight computing tasks. But there are a few situations where the two can diverge, sometimes dramatically.

  The elapsed time may be greater than the user time if the CPU spends a lot of time waiting around.

  This commonly happens if your R expression involes some input or output, which depends on the activity of the file system and the disk (or the Internet, if using a network connection).

  The elapsed time may be smaller than the user time if your machine has multiple cores/processors (and is capable of using them).
- For example, multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL) can greatly speed up linear algebra calculations and are commonly installed on even desktop systems these days. Also, parallel processing done via something like the parallel package can make the elapsed time smaller than the user time.
- When you have multiple processors/- cores/machines working in parallel, the amount of time that the collection of CPUs spends working on a problem is the same as with a single CPU, but because they are operating in parallel, there is a savings in elapsed time.
- Here's an example of where the elapsed time is greater than the user time.

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
user system elapsed
0.004 0.002 0.431
```

- Most of the time in this expression is spent waiting for the connection to the web server and waiting for the data to travel back to my computer. This doesn't involve the CPU and so the CPU simply waits around for things to get done. Hence, the user time is small.
- In this example, the elapsed time is smaller than the user time.

```
## Elapsed time < user time
> hilbert <- function(n) {
+ i <- 1:n
+ 1 / outer(i - 1, i, "+")
+ }
> x <- hilbert(1000)
> system.time(svd(x))
user system elapsed
1.035 0.255 0.462
```

- In this case I ran singular value decomposition on the matrix in x, which is a common linear algebra procedure. Because my computer is able to split the work across multiple processors, the elapsed time is about half the user time.

TIMING LONGER EXPRESSIONS

- You can time longer expressions by wrapping them in curly braces within the call to system.time().

```
> system.time({
+ n <- 1000
+ r <- numeric(n)
+ for(i in 1:n) {
+ x <- rnorm(n)
+ r[i] <- mean(x)
+ }
+ })
user system elapsed
0.086 0.001 0.088
```

- If your expression is getting pretty long (more than 2 or 3 lines), it might be better to either break it into smaller pieces or to use the profiler. The problem is that if the expression is too long, you won't be able to identify which part of the code is causing the bottleneck.

THE R PROFILER

- Using system.time() allows you to test certain functions or code blocks to see if they are taking excessive amounts of time. However, this approach assumes that you already

know where the problem is and can call system.time() on it that piece of code. What if you don't know where to start?

- This is where the profiler comes in handy. The Rprof() function starts the profiler in R. Note that R must be compiled with profiler support (but this is usually the case). In conjunction with Rprof(), we will use the summaryRprof() function which summarizes the output from Rprof() (otherwise it's not really readable). Note that you should NOT use system.time() and Rprof() together, or you will be sad.
- Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spent inside each function. By default, the profiler samples the function call stack every 0.02 seconds. This means that if your code runs very quickly (say, under 0.02 seconds), the profiler is not useful. But of your code runs that fast, you probably don't need the profiler.
- The profiler is started by calling the Rprof() function.

  > Rprof() ## Turn on the profiler
- You don't need any other arguments. By default it will write its output to a file called Rprof.out. You can specify the name of the output file if you don't want to use this default.
- Once you call the Rprof() function, everything that you do from then on will be measured by the profiler. Therefore, you usually only want to run a single R function or expression once you turn on the profiler and then immediately turn it off. The reason is that if you mix too many function calls together when running the profiler, all of the results will be mixed together and you won't be able to sort out where the bottlenecks are. In reality, I usually only run a single function with the profiler on.
- The profiler can be turned off by passing NULL to Rprof().

  > Rprof(NULL) ## Turn off the profiler
- The raw output from the profiler looks something like this. Here I'm calling the lm() function on some data with the profiler running.

lm(y ~ x)
sample.interval=10000
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"

"lm.fit" "lm"
"lm.fit" "lm"
"lm.fit" "lm"

- At each line of the output, the profiler writes out the function call stack. For example, on the very first line of the output you can see that the code is 8 levels deep in the call stack. This is where you need the summaryRprof() function to help you interpret this data.

Using summaryRprof()

- The summaryRprof() function tabulates the R profiler output and calculates how much time is spendin which function. There are two methods for normalizing the data.
  - ―by.total‖ divides the time spend in each function by the total run time
  - ―by.self‖ does the same as ―by.total‖ but first subtracts out time spent in functions above the current function in the call stack. I personally find this output to be much more useful.
- Here is what summaryRprof() reports in the ―by.total‖ output.

$by.total
total.time total.pct self.time self.pct
"lm" 7.41 100.00 0.30 4.05
"lm.fit" 3.50 47.23 2.99 40.35
"model.frame.default" 2.24 30.23 0.12 1.62
"eval" 2.24 30.23 0.00 0.00
"model.frame" 2.24 30.23 0.00 0.00
"na.omit" 1.54 20.78 0.24 3.24
"na.omit.data.frame" 1.30 17.54 0.49 6.61
"lapply" 1.04 14.04 0.00 0.00
"[.data.frame" 1.03 13.90 0.79 10.66
"[" 1.03 13.90 0.00 0.00
"as.list.data.frame" 0.82 11.07 0.82 11.07
"as.list" 0.82 11.07 0.00 0.00

- Because lm() is the function that I called from the command line, of course 100% of the time is spent somewhere in that function. However, what this doesn't show is that if lm() immediately calls another function (like lm.fit(), which does most of the heavy lifting), then in reality, most of the time is spent in that function, rather than in the top-level lm() function.
- The ―by.self‖ output corrects for this discrepancy.

$by.self
self.time self.pct total.time total.pct
"lm.fit" 2.99 40.35 3.50 47.23
"as.list.data.frame" 0.82 11.07 0.82 11.07
"[.data.frame" 0.79 10.66 1.03 13.90
"structure" 0.73 9.85 0.73 9.85

"na.omit.data.frame" 0.49 6.61 1.30 17.54
"list" 0.46 6.21 0.46 6.21
"lm" 0.30 4.05 7.41 100.00
"model.matrix.default" 0.27 3.64 0.79 10.66
"na.omit" 0.24 3.24 1.54 20.78
"as.character" 0.18 2.43 0.18 2.43
"model.frame.default" 0.12 1.62 2.24 30.23
"anyDuplicated.default" 0.02 0.27 0.02 0.27

- Now you can see that only about 4% of the runtime is spent in the actual lm() function, whereas over 40% of the time is spent in lm.fit(). In this case, this is no surprise since the lm.fit() function is the function that actually fits the linear model.
- You can see that a reasonable amount of time is spent in functions not necessarily associated with linear modeling (i.e. as.list.data.frame, [.data.frame). This is because the lm() function does a bit of pre-processing and checking before it actually fits the model. This is common with modeling functions—the preprocessing and checking is useful to see if there are any errors. But those two functions take up over 1.5 seconds of runtime. What if you want to fit this model 10,000 times?
- You're going to be spending a lot of time in preprocessing and checking.
- The final bit of output that summaryRprof() provides is the sampling interval and the total runtime.

$sample.interval
[1] 0.02
$sampling.time
[1] 7.41

SIMULATION

GENERATING RANDOM NUMBERS

- Simulation is an important (and big) topic for both statistics and for a variety of other areas where there is a need to introduce randomness. Sometimes you want to implement a statistical procedure that requires random number generation or samplie (i.e. Markov chain Monte Carlo, the bootstrap, random forests, bagging) and sometimes you want to simulate a system and random number generators can be used to model random inputs.
- R comes with a set of pseudo-random number generators that allow you to simulate from well known probability distributions like the Normal, Poisson, and binomial. Some example functions for probability distributions in R

  • rnorm: generate random Normal variates with a given mean and standard deviation

  • dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)

- pnorm: evaluate the cumulative distribution function for a Normal distribution
- rpois: generate random Poisson variates with a given rate

- For each probability distribution there are typically four functions available that start with a –r‖, –d‖, –p‖, and –q‖. The –r‖ function is the one that actually simulates random numbers from that distribution. The other functions are prefixed with a
  - d for density
  - r for random number generation
  - p for cumulative distribution
  - q for quantile function (inverse cumulative distribution)
- If you're only interested in simulating random numbers, then you will likely only need the –r‖ functions and not the others. However, if you intend to simulate from arbitrary probability distributions using something like rejection sampling, then you will need the other functions too.
- Probably the most common probability distribution to work with the Normal distribution (also known as the Gaussian). Working with the Normal distributions requires using these four functions

dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)

- Here we simulate standard Normal random numbers with mean 0 and standard deviation 1.

```
> ## Simulate standard Normal random numbers
> x <- rnorm(10)
> x
[1] 0.01874617 -0.18425254 -1.37133055 -0.59916772 0.29454513
[6] 0.38979430 -1.20807618 -0.36367602 -1.62667268 -0.25647839
```

- We can modify the default parameters to simulate numbers with mean 20 and standard deviation 2.

```
> x <- rnorm(10, 20, 2)
> x
[1] 22.20356 21.51156 19.52353 21.97489 21.48278 20.17869 18.09011
[8] 19.60970 21.85104 20.96596
> summary(x)
Min. 1st Qu. Median Mean 3rd Qu. Max.
18.09 19.75 21.22 20.74 21.77 22.20
```

- If you wanted to know what was the probability of a random Normal variable of being less than, say, 2, you could use the pnorm() function to do that calculation.

```
> pnorm(2)
[1] 0.9772499
```

- You never know when that calculation will come in handy

SETTING THE RANDOM NUMBER SEED

- When simulating any random numbers it is essential to set the random number seed. Setting the random number seed with set.seed() ensures reproducibility of the sequence of random numbers.
- For example, I can generate 5 Normal random numbers with rnorm()

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
```

Note that if I call rnorm() again I will of course get a different set of 5 random numbers.

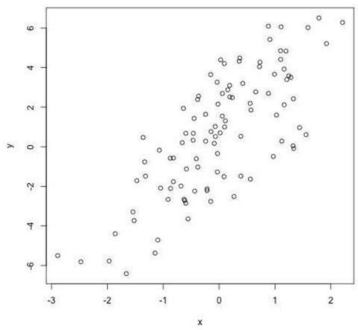
```
> rnorm(5)
[1] -0.8204684 0.4874291 0.7383247 0.5757814 -0.3053884
```

If I want to reproduce the original set of random numbers, I can just reset the seed with set.seed().

```
> set.seed(1)
> rnorm(5) ## Same as before
[1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078
```

- In general, you should always set the random number seed when conducting a simulation!
- Otherwise, you will not be able to reconstruct the exact numbers that you produced in an analysis. It is possible to generate random numbers from other probability distributions like the Poisson. The Poisson distribution is commonly used to model data that come in the form of counts.

```
> rpois(10, 1) ## Counts with a mean of 1
[1] 0 0 1 1 2 1 1 4 1 2
> rpois(10, 2) ## Counts with a mean of 2
[1] 4 1 2 0 1 1 0 1 4 1
> rpois(10, 20) ## Counts with a mean of 20
[1] 19 19 24 23 22 24 23 20 11 22
```



plot of chunk Linear Model

SIMULATING A LINEAR MODEL

- Simulating random numbers is useful but sometimes we want to simulate values that come from a specific model. For that we need to specify the model and then simulate from it using the functions described above.
- Suppose we want to simulate from the following linear model

y = β0 + β1x + ε
where ε ~ N (0, 2
2
). Assume x ~ N (0, 1
2
), β0 = 0.5 and β1 = 2. The variable x might represent
an important predictor of the outcome y. Here's how we could do that in R.

```
> ## Always set your seed!
> set.seed(20)
>
> ## Simulate predictor variable
> x <- rnorm(100)
>
> ## Simulate the error term
> e <- rnorm(100, 0, 2)
>
> ## Compute the outcome via the model
> y <- 0.5 + 2 * x + e
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
-6.4080 -1.5400 0.6789 0.6893 2.9300 6.5050
```

We can plot the results of the model simulation.

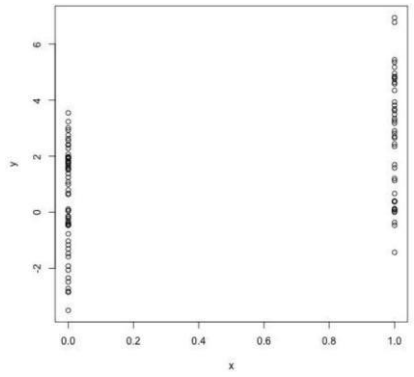
```
> plot(x, y)
```

- What if we wanted to simulate a predictor variable x that is binary instead of having a Normal distribution. We can use the rbinom() function to simulate binary random variables.

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> str(x) ## 'x' is now 0s and 1s
int [1:100] 1 0 0 1 0 0 0 0 1 0 ...
```

Then we can procede with the rest of the model as before.

```
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> plot(x, y)
```



plot of chunk Linear Model Binary

- We can also simulate from generalized linear model where the errors are no longer from a Normal distribution but come from some other distribution. For examples, suppose we want to simulate from a Poisson log-linear model where

$$Y \sim Poisson(\mu)$$

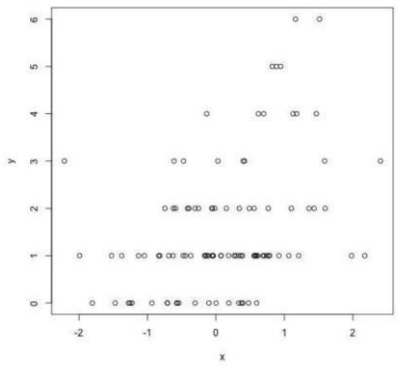
$$\log \mu = \beta 0 + \beta 1 x$$

and β0 = 0.5 and β1 = 0.3. We need to use the rpois() function for this

```
> set.seed(1)
>
> ## Simulate the predictor variable as before
> x <- rnorm(100)
```

Now we need to compute the log mean of the model and then exponentiate it to get the mean to pass to rpois().

```
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.00 1.00 1.00 1.55 2.00 6.00
> plot(x, y)
```



plot of chunk Poisson Log-Linear Model

- You can build arbitrarily complex models like this by simulating more predictors or making transformations of those predictors (e.g. squaring, log transformations, etc.).

RANDOM SAMPLING

- The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
>
> ## Doesn't have to be numbers
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
>
> ## Do a random permutation
> sample(1:10)
[1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
[1] 2 3 4 1 9 5 10 8 6 7
>
> ## Sample w/replacement
> sample(1:10, replace = TRUE)
[1] 2 9 7 8 2 8 5 9 7 8
```

- To sample more complicated things, such as rows from a data frame or a list, you can sample the indices into an object rather than the elements of the object itself.
- Here's how you can sample rows from a data frame.

```
> library(datasets)
> data(airquality)
> head(airquality)
```

Ozone Solar.R Wind Temp Month Day
1 41 190 7.4 67 5 1
2 36 118 8.0 72 5 2
3 12 149 12.6 74 5 3
4 18 313 11.5 62 5 4
5 NA NA 14.3 56 5 5
6 28 NA 14.9 66 5 6

- Now we just need to create the index vector indexing the rows of the data frame and sample directly from that index vector.

```
> set.seed(20)
>
> ## Create index vector
> idx <- seq_len(nrow(airquality))
>
> ## Sample from the index vector
> samp <- sample(idx, 6)
> airquality[samp, ]
```

Ozone Solar.R Wind Temp Month Day
135 21 259 15.5 76 9 12
117 168 238 3.4 81 8 25
43 NA 250 9.2 92 6 12
80 79 187 5.1 87 7 19
144 13 238 12.6 64 9 21
146 36 139 10.3 81 9 23

Other more complex objects can be sampled in this way, as long as there's a way to index the sub elements of the object.

POSSIBLE QUESTIONS

UNIT – V

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is Debugging?
2. Define Random Samplings
3. What is the use of sample ()?
4. When the random number seed set?
5. Give some examples for probability distributions in R.
6. What are the Debugging tools in R programming
7. Define recover()
8. What is the process of debug ()?
9. What is meant by Simulation?
10. What is the use of traceback()?

PART – C (6 MARKS)

1. Explain the process of Debugging
2. Discuss the Debugging tools in R
3. Explain the process of traceback ()
4. Discuss in detail (i) recover () (ii) debug ()
5. Explain about system.time() with suitable examples
6. Explain about the R profiler
7. Explain how to simulate a linear model
8. Explain about Random Samplings
9. Explain about Simulation and its process
10. Explain the process of Using summaryRprof()

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2018 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - V : (Objective Type Multiple choice Questions each Question carries one Mark)

R PROGRAMMING [18CAU404A]

PART - A (Online Examination)

Questions	Opt1	Opt2	Opt3	Opt4	Key
_____is an indication that a fatal problem has occurred and execution of the function stops	message	error	warning	stop	error
What will be the value of following expression ?	Warning in log(c(-1, 2)): NaNs produced	Error in log(c(-1, 2)): NaNs produced	Message	error	Error in log(c(-1, 2)): NaNs produced
_____prints out the function call stack after	trace()	traceback()	back()	backerror()	traceback()
Point out the wrong statement :	The primary task of debugging any R code is correctly diagnosing what the problem is	R provides only two tools to help you with debugging your code	print statement can be used for debugging purpose	The traceback() function must be called immediately after an error occurs	R provides only two tools to help you with debugging your code
Which of the following is primary tool for	debug()	trace()	browser()	traceback()	debug()
_____allows you to insert debugging code into	debug()	trace()	browser()	traceback()	trace()

Point out the correct statement :	The traceback() function must be called immediately after an error occurs	The debugger calls the browser at the very low level of the function body	Every time you call the mod() function it will launch the interactive debugger	R provides only two tools to help you with debugging your code	The traceback() function must be called immediately after an error occurs
_____allows you to modify the error behavior so that you can browse the function call stack	debug()	trace()	recover()	traceback()	recover()
_____suspends the execution of a function wherever it is called and puts the function in debug	debug()	trace()	recover()	browser()	browser()
debug() flags a function for_____mode in R	debug	run	compile	recover	run
What would be the output of the following code ? > mean(x) Error in mean(x) : object 'x' not found	1: mean(x)	Null	0	1	1: mean(x)
The recover() function will first print out the function call stack when an_____occurs.	Error	Warning	Messages	stop	Error
_____is a systematic way to examine how much time is spent in different parts of a program.	Profiling	Monitoring	Logging	Scheduling	Profiling
Point out the correct statement :	The Rprofiler() function starts the profiler in R	Using system.time() allows you to test certain functions or code blocks to see if they are taking excessive amounts of time	R must not be compiled with profiler support	Rprofiler() tabulates how much time is spent inside each function	Using system.time() allows you to test certain functions or code blocks to see if they are taking excessive amounts of time
R comes with a_____to help you optimize your code and improve its performance.	debugger	monitor	browser	profiler	debugger

The_____function computes the time (in seconds) needed to execute an expression.	system.timedeb()	system.time()	system.datetime()	system.timedate()	system.time()
Point out the correct statement :	Rprofiler() tabulates how much time is spent inside each function	Rprof() keeps track of the function call stack at regularly sampled intervals	By default, the profiler samples the function call stack every 2 seconds	R must not be compiled with profiler support	Rprof() keeps track of the function call stack at regularly sampled intervals
system.time function returns an object of class _____which contains two useful bits of	debug_time	proc_time	procedure_time	proced_time	proc_time
_____time is time charged to the CPU(s) for	elapsed	user	response	request	elapsed
The elapsed time may be_____than the user time if your machine has multiple cores/processors	smaller	greater	equal to	not equal to	smaller
Parallel processing is done via_____package can make the elapsed time smaller than the user	parallel	statistics	distributed	equal	parallel
You can time_____expressions by wrapping them in curly braces within the call to	smaller	longer	error	warning	longer
The profiler can be turned off by passing	0	1	2	NULL	NULL
Point out the correct statement :	Rprof() is used to turn off the profiler	At each line of the output, the profiler writes out the function call stack	The summaryprof() function tabulates the R profiler output	R must not be compiled with profiler support	At each line of the output, the profiler writes out the function call stack
How many methods exist for normalizing the data ?	one	two	three	profiler	two
_____divides the time spend in each function by	“by.sum”	“by.total”	“by.self”	“by.mull”	“by.total”

Point out the correct statement :	"by.total" first subtracts out time spent in functions above the current function in the call stack	The summaryRprof() function calculates how much time is spend in which function	By default, the profiler samples the function call stack every 0.02 seconds	R must not be compiled with profiler support	By default, the profiler samples the function call stack every 0.02 seconds
Which of the following function actually fits the	lm.time()	lm.date()	lm.fit()	lm.day()	lm.fit()
_____time is time charged to the CPU(s) for	elapsed	user	response	request	elapsed
The final bit of output that summaryRprof() provides is the_____interval and the total	response	sampling	processing	request	sampling
Which of the following statement gives sampling interval ?	$sampling.interval	$sampling.time	$sampling.date	$sampling.day	$sampling.time
Which of the following code is not profiled ?	C	C++	Java	.Net	C
_____generate random Normal variates with a given mean and standard deviation	dnorm	rnorm	pnorm	rpois	rnorm
Point out the correct statement :	R comes with a set of pseudo-random number generators	Random number generators cannot be used to model random inputs	Statistical procedure does not require random number generation	For each probability distribution there are typically three functions	R comes with a set of pseudo-random number generators
_____evaluate the cumulative distribution function	dnorm	rnorm	pnorm	rpois	pnorm
_____generate random Poisson variates with a	dnorm	rnorm	pnorm	rpois	rpois
Point out the wrong statement :	For each probability distribution there are typically three functions	For each probability distribution there are typically four functions	r function is sufficient for simulating random numbers	R comes with a set of pseudo-random number generators	For each probability distribution there are typically three functions
Which of the following evaluate the Normal probability density (with a given mean/SD) at a	dnorm	rnorm	pnorm	rpois	dnorm

_____is the most common probability	Gaussian	Parametric	Paradox	paradix	Gaussian
What will be the output of the following code ? >	0.9772499	1.9772499	0.6772499	0.8772499	0.9772499
_____ensures reproducibility of the sequence of random numbers.	sets.seed()	set.seed()	set.seedvalue()	seedvalue()	set.seed()
Point out the correct statement :	It is not possible to generate random numbers from other probability distributions like the Poisson	When simulating any random numbers it is not essential to set the random number seed	You should always set the random number seed when conducting a simulation	The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers	You should always set the random number seed when conducting a simulation
5 Normal random numbers can be generated with rnorm() by setting seed value to :	1	2	3	4	1
_____function is used to simulate binary	dnorm	rbinom	binom	rpois	rbinom
Point out the wrong statement :	Drawing samples from specific probability distributions can be done with “s” functions	The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers	The sampling() function draws randomly from a specified set of objects	You should always set the random number seed when conducting a simulation	The sample() function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions of numbers
What will be the output of the following code ? > set.seed(10) > x <- rbinom(100, 1, 0.5)	int [1:100] 1 0 0 1 0 0 0 0 1 0 ...	int [1:100] 10 0 01 1 0 0 01 0 1 0 ...	int [1:100] 1 03 0 1 0 0 0 02 1 0 ...	int [1:100] 1 2 3 1 1 0 0 0 1 0 ...	int [1:100] 1 0 0 1 0 0 0 0 1 0 ...
_____distribution is commonly used to model data that come in the form of counts.	Gaussian	Parametric	Poisson	Paradox	Poisson

What will be the output of the following code ? > rpois(10, 1)	[1] 7 0 1 1 2 1 1 4 1 2	[1] 0 8 1 1 2 1 1 4 1 2	[1] 0 0 1 1 2 1 1 4 1 2	[1] 0 9 1 1 2 1 1 5 1 2	[1] 0 0 1 1 2 1 1 4 1 2
Which of the following code represents count with	rpois(10, 2)	rpois(10, 20)	rpois(20, 2)	rpois(2, 20)	rpois(10, 2)
The_____function draws randomly from a specified set of (scalar) objects allowing you to	sam()	seed()	sample()	samp()	sample()
_____is an important (and big) topic for both statistics and for a variety of other areas where	Simulation	samplie	distribution	normal	Simulation
Setting the_____number generator seed via set.seed() is critical for reproducibility	arbitrary	sample	random	sequence	random
The_____function tabulates the R profiler output and calculates how much time is spend in	prof()	summaryRprof()	Rprof()	Rpro()	summaryRprof()
Interactive debugging tools _____,_____,_____,_____and _____can be used to find problematic code in functions	trace, debug, browser, backtrace, and recover	traceback, debug, browser, trace, and recover	traceback, debug, browser, trace, and request	traceback, debug, browser, request, and recover	traceback, debug, browser, trace, and recover
The_____function will first print out the function call stack when an error occurrs.	debug()	trace()	recover()	traceback()	recover()
In simulating linear model can also simulate from_____where the errors are no longer from a Normal distribution but come from	generalized model	generalized linear model	linear model	ungeneralized linear model	generalized linear model
Simulating_____numbers is useful but sometimes we want to simulate values that come	arbitrary	sample	random	sequence	random
The function call stack is the_____of functions that was called before the error occurred.	arbitrary	sample	random	sequence	sequence
In which case the_____function tried to evaluate the formula y x and realized the object y	debug()	trace()	eval()	traceback()	eval()
_____time charged to the CPU(s) for this	sample.time	user time	elapsed time	system.time	user time