Statistical Programming

Statistical Programming works to carry out planned analyses which include creation and analysis of datasets and tables, listings, and figures

A primary function of the statistical programming is to create programs to generate the Tables, Listings, and Figures needed for the analysis and reporting of the study. Data are individual pieces of factual information recorded and used for the purpose of analysis. It is the raw information from which statistics are created. Statistics are the results of data analysis - its interpretation and presentation. In other words some computation has taken place that provides some understanding of what the data means. Statistics are often, though they don't have to be, presented in the form of a table, chart, or graph.

Statistics is basically a science that involves data collection, data interpretation and finally, data validation. Statistical Data analysis is a procedure of performing various statistical operations. It is a kind of quantitative research, which seeks to quantify the data, and typically, applies some form of statistical analysis. Quantitative data basically involves descriptive data, such as survey data and observational data.

Data analysis is a process of inspecting, cleansing, transforming, and modelling data with the goal of discovering useful information, informing conclusions, and supporting decision-making.

The data is classified into majorly four categories:

- Nominal data
- Ordinal data
- Discrete data
- Continuous data

Nominal Data

- Nominal data is one of the types of qualitative information which helps to label the variables without providing the numerical value. Nominal data is also called the nominal scale. It cannot be ordered and measured. But sometimes, the data can be qualitative and quantitative. Examples of nominal data are letters, symbols, words, gender etc.
- The nominal data are examined using the grouping method. In this method, the data are grouped into categories, and then the frequency or the percentage of the data can be calculated. These data are visually represented using the pie charts.

Ordinal Data

- Ordinal data/variable is a type of data which follows a natural order. The significant feature of the nominal data is that the difference between the data values is not determined. This variable is mostly found in surveys, finance, economics, questionnaires, and so on.
- The ordinal data is commonly represented using a bar chart. These data are investigated and interpreted through many visualisation tools. The information may be expressed using tables in which each row in the table shows the distinct category.

Quantitative or Numerical Data

- Quantitative data is also known as numerical data which represents the numerical value (i.e., how much, how often, how many). Numerical data gives information about the quantities of a specific thing. Some examples of numerical data are height, length, size, weight, and so on. The quantitative data can be classified into two different types based on the data sets. The two different classifications of numerical data are discrete data and continuous data.

Discrete Data

- Discrete data can take only discrete values. Discrete information contains only a finite number of possible values. Those values cannot be subdivided meaningfully. Here, things can be counted in whole numbers.
- Example: Number of students in the class

Continuous Data

Continuous data is data that can be calculated. It has an infinite number of probable values that can be selected within a given specific range.

Example: Temperature range

Measurements of data:

1. Nominal scale of measurement

The nominal scale of measurement defines the identity property of data. This scale has certain characteristics, but doesn't have any form of numerical meaning. The data can be placed into categories but can't be multiplied, divided, added or subtracted from one another. It's also not possible to measure the difference between data points.

Examples of nominal data include eye colour and country of birth. Nominal data can be broken down again into three categories:

- Nominal without order: Nominal data can also be sub-categorised as nominal without order, such as male and female.
- Dichotomous: Dichotomous data is defined by having only two categories or levels, such as "yes' and 'no'.

2. Ordinal scale of measurement

The ordinal scale defines data that is placed in a specific order. While each value is ranked, there's no information that specifies what differentiates the categories from each other. These values can't be added to or subtracted from.

An example of this kind of data would include satisfaction data points in a survey, where 'one = happy, two = neutral, and three = unhappy.' Where someone finished in a race also describes ordinal data. While first place, second place or third place shows what order the

runners finished in, it doesn't specify how far the first-place finisher was in front of the second-place finisher.

3. Interval scale of measurement
The interval scale contains properties of nominal and ordered data, but the difference between data points can be quantified. This type of data shows both the order of the variables and the exact differences between the variables. They can be added to or subtracted from each other, but not multiplied or divided. For example, 40 degrees is not 20 degrees multiplied by two.

This scale is also characterised by the fact that the number zero is an existing variable. In the ordinal scale, zero means that the data does not exist. In the interval scale, zero has meaning – for example, if you measure degrees, zero has a temperature.

Data points on the interval scale have the same difference between them. The difference on the scale between 10 and 20 degrees is the same between 20 and 30 degrees. This scale is used to quantify the difference between variables, whereas the other two scales are used to describe qualitative values only. Other examples of interval scales include the year a car was made or the months of the year.

4. Ratio scale of measurement
Ratio scales of measurement include properties from all four scales of measurement. The data is nominal and defined by an identity, can be classified in order, contains intervals and can be broken down into exact value. Weight, height and distance are all examples of ratio variables. Data in the ratio scale can be added, subtracted, divided and multiplied.
Ratio scales also differ from interval scales in that the scale has a 'true zero'. The number zero means that the data has no value point. An example of this is height or weight, as someone cannot be zero centimetres tall or weigh zero kilos – or be negative centimetres or negative kilos. Examples of the use of this scale are calculating shares or sales. Of all types of data on the scales of measurement, data scientists can do the most with ratio data points.

R language

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.

- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

R Command Prompt

Once you have R environment setup, then it's easy to start your R command prompt by just typing the following command at your command prompt −

$ R

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows −

> myString <- "Hello, World!"

> print (myString)

[1] "Hello, World!"

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

R Script File

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called Rscript. So let's start with writing following code in a text file called test.R as under −

My first program in R Programming

myString <- "Hello, World!"

print (myString)

Save the above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

$ Rscript test.R

When we run the above program, it produces the following result.

[1] "Hello, World!"

The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are −

Vectors

Lists

Matrices

Arrays

Data Frames

The simplest of these objects is the vector object and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type

Logical TRUE, FALSE

v <- TRUE

print(class(v))

it produces the following result −

[1] "logical"

Numeric 12.3, 5, 999

v <- 23.5

print(class(v))

it produces the following result −

[1] "numeric"

Integer 2L, 34L, 0L

v <- 2L

print(class(v))

it produces the following result −

[1] "integer"

Complex 3 + 2i

v <- 2+5i

print(class(v))

it produces the following result −

[1] "complex"

Character 'a' , '"good", "TRUE", '23.4'

v <- "TRUE"

print(class(v))

it produces the following result −

[1] "character"

Raw "Hello" is stored as 48 65 6c 6c 6f

```
v <- charToRaw("Hello")

print(class(v))
```

it produces the following result −

[1] "raw"

In R programming, the very basic data types are the R-objects called vectors which hold elements of different classes as shown above. Please note in R the number of classes is not confined to only the above six types. For example, we can use many atomic vectors and create an array whose class will become array.

Vectors

When you want to create vector with more than one element, you should use c() function which means to combine the elements into a vector.

```
# Create a vector.

apple <- c('red','green',"yellow")

print(apple)

# Get the class of the vector.

print(class(apple))
```

When we execute the above code, it produces the following result −

[1] "red" "green" "yellow"

[1] "character"

Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

```
# Create a list.

list1 <- list(c(2,5,3),21.3,sin)

# Print the list.

print(list1)
```

When we execute the above code, it produces the following result −

[[1]]

[1] 2 5 3

[[2]]

[1] 21.3

[[3]]

function (x) .Primitive("sin")

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

```
# Create a matrix.
M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)
print(M)
```

When we execute the above code, it produces the following result −

```
     [,1] [,2] [,3]
[1,] "a"  "a"  "b"
[2,] "c"  "b"  "a"
```

Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

```
# Create an array.
a <- array(c('green','yellow'),dim = c(3,3,2))
print(a)
```

When we execute the above code, it produces the following result −

```
, , 1

     [,1]     [,2]     [,3]
[1,] "green"  "yellow" "green"
[2,] "yellow" "green"  "yellow"
[3,] "green"  "yellow" "green"

, , 2

     [,1]     [,2]     [,3]
[1,] "yellow" "green"  "yellow"
[2,] "green"  "yellow" "green"
```

[3,] "yellow" "green" "yellow"

Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the data.frame() function.

```
# Create the data frame.
BMI <-        data.frame(
   gender = c("Male", "Male","Female"),
   height = c(152, 171.5, 165),
   weight = c(81,93, 78),
   Age = c(42,38,26)
)
print(BMI)
```

When we execute the above code, it produces the following result −

```
  gender height weight Age
1   Male  152.0     81  42
2   Male  171.5     93  38
3 Female  165.0     78  26
```

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many Robjects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name,	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using print() or cat() function. The cat() function combines multiple items into a continuous print output.

```
# Assignment using equal operator.

var.1 = c(0,1,2,3)

# Assignment using leftward operator.

var.2 <- c("learn","R")

# Assignment using rightward operator.

c(TRUE,1) -> var.3

print(var.1)

cat ("var.1 is ", var.1 ,"\n")

cat ("var.2 is ", var.2 ,"\n")

cat ("var.3 is ", var.3 ,"\n")
```

When we execute the above code, it produces the following result −

```
[1] 0 1 2 3

var.1 is  0 1 2 3

var.2 is  learn R

var.3 is  1 1
```

Note − The vector c(TRUE,1) has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

```
var_x <- "Hello"

cat("The class of var_x is ",class(var_x),"\n")

var_x <- 34.5

cat("  Now the class of var_x is ",class(var_x),"\n")

var_x <- 27L

cat("   Next the class of var_x becomes ",class(var_x),"\n")
```

When we execute the above code, it produces the following result −

```
The class of var_x is  character
  Now the class of var_x is  numeric
    Next the class of var_x becomes  integer
```

Finding Variables

To know all the variables currently available in the workspace we use the ls() function. Also the ls() function can use patterns to match the variable names.

```
print(ls())
```

When we execute the above code, it produces the following result −

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

Note − It is a sample output depending on what variables are declared in your environment.

The ls() function can use patterns to match the variable names.

```
# List the variables starting with the pattern "var".
print(ls(pattern = "var"))
```

When we execute the above code, it produces the following result −

```
[1] "my var"     "my_new_var" "my_var"     "var.1"
[5] "var.2"      "var.3"      "var.name"   "var_name2."
[9] "var_x"      "varname"
```

The variables starting with dot(.) are hidden, they can be listed using "all.names = TRUE" argument to ls() function.

```
print(ls(all.name = TRUE))
```

When we execute the above code, it produces the following result −

```
[1] ".cars"        ".Random.seed" ".var_name"    ".varname"     ".varname2"
[6] "my var"       "my_new_var"   "my_var"       "var.1"        "var.2"
[11]"var.3"        "var.name"     "var_name2."   "var_x"
```

Deleting Variables

Variables can be deleted by using the rm() function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

```
rm(var.3)
```

print(var.3)

When we execute the above code, it produces the following result −

[1] "var.3"

Error in print(var.3) : object 'var.3' not found

All the variables can be deleted by using the rm() and ls() function together.

rm(list = ls())

print(ls())

When we execute the above code, it produces the following result −

character(0)

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operators

We have the following types of operators in R programming −

Arithmetic Operators

Relational Operators

Logical Operators

Assignment Operators

Miscellaneous Operators

Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

Operator Description Example

+ Adds two vectors

v <- c(2,5.5,6)

t <- c(8, 3, 4)

print(v+t)

it produces the following result −

[1] 10.0 8.5 10.0

− Subtracts second vector from the first

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v-t)
```

it produces the following result −

```
[1] -6.0  2.5  2.0
```

* Multiplies both vectors

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v*t)
```

it produces the following result −

```
[1] 16.0 16.5 24.0
```

/ Divide the first vector with the second

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v/t)
```

When we execute the above code, it produces the following result −

```
[1] 0.250000 1.833333 1.500000
```

%% Give the remainder of the first vector with the second

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v%%t)
```

it produces the following result −

```
[1] 2.0 2.5 2.0
```

%/% The result of division of first vector with second (quotient)

```
v <- c( 2,5.5,6)
t <- c(8, 3, 4)
print(v%/%t)
```

it produces the following result −

[1] 0 1 1

^ The first vector raised to the exponent of second vector

v <- c(2,5.5,6)

t <- c(8, 3, 4)

print(v^t)

it produces the following result −

[1] 256.000 166.375 1296.000

Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator Description Example

> Checks if each element of the first vector is greater than the corresponding element of the second vector.

v <- c(2,5.5,6,9)

t <- c(8,2.5,14,9)

print(v>t)

it produces the following result −

[1] FALSE TRUE FALSE FALSE

< Checks if each element of the first vector is less than the corresponding element of the second vector.

v <- c(2,5.5,6,9)

t <- c(8,2.5,14,9)

print(v < t)

it produces the following result −

[1] TRUE FALSE TRUE FALSE

== Checks if each element of the first vector is equal to the corresponding element of the second vector.

v <- c(2,5.5,6,9)

t <- c(8,2.5,14,9)

print(v == t)

it produces the following result −

[1] FALSE FALSE FALSE TRUE

<= Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9)

t <- c(8,2.5,14,9)

print(v<=t)
```

it produces the following result −

[1] TRUE FALSE TRUE TRUE

>= Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9)

t <- c(8,2.5,14,9)

print(v>=t)
```

it produces the following result −

[1] FALSE TRUE FALSE TRUE

!= Checks if each element of the first vector is unequal to the corresponding element of the second vector.

```
v <- c(2,5.5,6,9)

t <- c(8,2.5,14,9)

print(v!=t)
```

it produces the following result −

[1] TRUE TRUE TRUE FALSE

Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator Description Example

& It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.

```
v <- c(3,1,TRUE,2+3i)

t <- c(4,1,FALSE,2+3i)

print(v&t)
```

it produces the following result −

```
[1]  TRUE  TRUE FALSE  TRUE
```

| It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.

```
v <- c(3,0,TRUE,2+2i)

t <- c(4,0,FALSE,2+3i)

print(v|t)
```

it produces the following result −

```
[1]  TRUE FALSE  TRUE  TRUE
```

! It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.

```
v <- c(3,0,TRUE,2+2i)

print(!v)
```

it produces the following result −

```
[1] FALSE  TRUE FALSE FALSE
```

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator Description Example

&& Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.

```
v <- c(3,0,TRUE,2+2i)

t <- c(1,3,TRUE,2+3i)

print(v&&t)
```

it produces the following result −

```
[1] TRUE
```

|| Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.

```
v <- c(0,0,TRUE,2+2i)
t <- c(0,3,TRUE,2+3i)
print(v||t)
```

it produces the following result −

```
[1] FALSE
```

Assignment Operators

These operators are used to assign values to vectors.

Operator

<− or=or

<<−

Called Left Assignment

```
v1 <- c(3,1,TRUE,2+3i)
v2 <<- c(3,1,TRUE,2+3i)
v3 = c(3,1,TRUE,2+3i)
print(v1)
print(v2)
print(v3)
```

it produces the following result −

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

->

or

->>

Called Right Assignment

```
c(3,1,TRUE,2+3i) -> v1
```

```
c(3,1,TRUE,2+3i) ->> v2
print(v1)
print(v2)
```

it produces the following result −

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

Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator

: Colon operator. It creates the series of numbers in sequence for a vector.

```
v <- 2:8
print(v)
```

it produces the following result −

```
[1] 2 3 4 5 6 7 8
```

%in% This operator is used to identify if an element belongs to a vector.

```
v1 <- 8
v2 <- 12
t <- 1:10
print(v1 %in% t)
print(v2 %in% t)
```

it produces the following result −

```
[1] TRUE
[1] FALSE
```

%*% This operator is used to multiply a matrix with its transpose.

```
M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)
t = M %*% t(M)
```

print(t)

it produces the following result −

```
     [,1] [,2]
[1,]   65   82
[2,]   82  117
```

Decision making

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −

An if statement consists of a Boolean expression followed by one or more statements.

Syntax

The basic syntax for creating an if statement in R is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
}
```

If the Boolean expression evaluates to be true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to be false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

R if statement

Example

```
x <- 30L
if(is.integer(x)) {
   print("X is an Integer")
```

```
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "X is an Integer"
```

An if statement can be followed by an optional else statement which executes when the boolean expression is false.

Syntax

The basic syntax for creating an if...else statement in R is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
} else {
   // statement(s) will execute if the boolean expression is false.
}
```

If the Boolean expression evaluates to be true, then the if block of code will be executed, otherwise else block of code will be executed.

R if...else statement

Example

```
x <- c("what","is","truth")
if("Truth" %in% x) {
   print("Truth is found")
} else {
   print("Truth is not found")
}
```

When the above code is compiled and executed, it produces the following result −

[1] "Truth is not found"

Here "Truth" and "truth" are two different strings.

The if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are few points to keep in mind.

An if can have zero or one else and it must come after any else if's.

An if can have zero to many else if's and they must come before the else.

Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The basic syntax for creating an if...else if...else statement in R is −

```
if(boolean_expression 1) {
   // Executes when the boolean expression 1 is true.
} else if( boolean_expression 2) {
   // Executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {
   // Executes when the boolean expression 3 is true.
} else {
   // executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")

if("Truth" %in% x) {
   print("Truth is found the first time")
} else if ("truth" %in% x) {
   print("truth is found the second time")
} else {
   print("No truth found")
}
```

When the above code is compiled and executed, it produces the following result −

[1] "truth is found the second time"

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The basic syntax for creating a switch statement in R is −

switch(expression, case1, case2, case3....)

The following rules apply to a switch statement −

If the value of expression is not a character string it is coerced to integer.

You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

If the value of the integer is between 1 and nargs()−1 (The max number of arguments)then the corresponding element of case condition is evaluated and the result returned.

If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.

If there is more than one match, the first matching element is returned.

No Default argument is available.

In the case of no match, if there is a unnamed element of ... its value is returned. (If there is more than one such argument an error is returned.)

R switch statement

Example

```
x <- switch(
  3,
  "first",
  "second",
  "third",
  "fourth"
)
print(x)
```

When the above code is compiled and executed, it produces the following result −

[1] "third"

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

Loops

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages −

R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

Sr.No.	Loop Type & Description
1	repeat loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

2	while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

3	for loop

Like a while statement, except that it tests the condition at the end of the loop body.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description

1 break statement

Terminates the loop statement and transfers execution to the statement immediately following the loop.

2 Next statement

The next statement simulates the behavior of R switch.

The Repeat loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a repeat loop in R is −

```
repeat {
   commands
   if(condition) {
      break
   }
}
```

R Repeat Statement

Example

```
v <- c("Hello","loop")
cnt <- 2

repeat {
   print(v)
   cnt <- cnt+1

   if(cnt > 5) {
      break
   }
```

```
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

The Repeat loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a repeat loop in R is −

```
repeat {
   commands
   if(condition) {
      break
   }
}
```

R Repeat Statement

Example

```
v <- c("Hello","loop")
cnt <- 2

repeat {
   print(v)
   cnt <- cnt+1
```

```
  if(cnt > 5) {
    break
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
[1] "Hello" "loop"
```

The While loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a while loop in R is −

```
while (test_expression) {
  statement
}
```

R while loop

Here key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
v <- c("Hello","while loop")
cnt <- 2

while (cnt < 7) {
```

```
   print(v)
   cnt = cnt + 1
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
[1] "Hello"  "while loop"
```

A For loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The basic syntax for creating a for loop statement in R is −

```
for (value in vector) {
   statements
}
```

R for loop

R's for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

Example

```
v <- LETTERS[1:4]
for ( i in v) {
   print(i)
}
```

When the above code is compiled and executed, it produces the following result −

[1] "A"

[1] "B"

[1] "C"

[1] "D"

The next statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax

The basic syntax for creating a next statement in R is −

```
next
```

R Next Statement

Example

```
v <- LETTERS[1:6]
for ( i in v) {

  if (i == "D") {
    next
  }
  print(i)
}
```

When the above code is compiled and executed, it produces the following result −

[1] "A"

[1] "B"

[1] "C"

[1] "E"

[1] "F"

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

Functions

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

Function Definition

An R function is created by using the keyword function. The basic syntax of an R function definition is as follows −

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

Function Components

The different parts of a function are −

Function Name − This is the actual name of the function. It is stored in R environment as an object with this name.

Arguments − An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.

Function Body − The function body contains a collection of statements that defines what the function does.

Return Value − The return value of a function is the last expression in the function body to be evaluated.

R has many in-built functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as user defined functions.

Built-in Function

Simple examples of in-built functions are seq(), mean(), max(), sum(x) and paste(...) etc. They are directly called by user written programs. You can refer most widely used R functions.

```
# Create a sequence of numbers from 32 to 44.
print(seq(32,44))

# Find mean of numbers from 25 to 82.
print(mean(25:82))

# Find sum of numbers frm 41 to 68.
print(sum(41:68))
```

When we execute the above code, it produces the following result −

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
```

```
new.function <- function(a) {
   for(i in 1:a) {
      b <- i^2
      print(b)
   }
}
```

Calling a Function

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
   for(i in 1:a) {
      b <- i^2
      print(b)
   }
}

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

When we execute the above code, it produces the following result −

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

Calling a Function without an Argument

```
# Create a function without an argument.
new.function <- function() {
```

```
   for(i in 1:5) {
      print(i^2)
   }
}

# Call the function without supplying an argument.
new.function()
```

When we execute the above code, it produces the following result −

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

```
# Create a function with arguments.
new.function <- function(a,b,c) {
   result <- a * b + c
   print(result)
}

# Call the function by position of arguments.
new.function(5,3,11)

# Call the function by names of the arguments.
new.function(a = 11, b = 5, c = 3)
```

When we execute the above code, it produces the following result −

[1] 26

[1] 58

Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

```
# Create a function with arguments.
new.function <- function(a = 3, b = 6) {
   result <- a * b
   print(result)
}

# Call the function without giving any argument.
new.function()

# Call the function with giving new values of the argument.
new.function(9,5)
```

When we execute the above code, it produces the following result −

[1] 18

[1] 45

Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

```
# Create a function with arguments.
```

```
new.function <- function(a, b) {
   print(a^2)
   print(a)
   print(b)
}

# Evaluate the function without supplying one of the arguments.
new.function(6)
```

When we execute the above code, it produces the following result −

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

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

Following are the characteristics of a data frame.

The column names should be non-empty.

The row names should be unique.

The data stored in a data frame can be of numeric, factor or character type.

Each column should contain same number of data items.

Create Data Frame

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),
```

```
   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Print the data frame.
print(emp.data)
```

When we execute the above code, it produces the following result −

```
 emp_id    emp_name     salary     start_date
1     1     Rick        623.30     2012-01-01
2     2     Dan         515.20     2013-09-23
3     3     Michelle    611.00     2014-11-15
4     4     Ryan        729.00     2014-05-11
5     5     Gary        843.25     2015-03-27
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using str() function.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Get the structure of the data frame.
str(emp.data)
```

When we execute the above code, it produces the following result −

```
'data.frame':   5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "Rick" "Dan" "Michelle" "Ryan" ...
 $ salary    : num  623 515 611 729 843
 $ start_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11" ...
```

Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying summary() function.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Print the summary.
print(summary(emp.data))
```

When we execute the above code, it produces the following result −

```
     emp_id    emp_name            salary        start_date
 Min.   :1   Length:5           Min.   :515.2   Min.   :2012-01-01
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                      Mean   :664.4   Mean   :2014-01-14
```

```
 3rd Qu.:4                    3rd Qu.:729.0   3rd Qu.:2014-11-15
 Max.   :5                    Max.   :843.2   Max.   :2015-03-27
```

Extract Data from Data Frame

Extract specific column from a data frame using column name.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Extract Specific columns.
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

When we execute the above code, it produces the following result −

```
  emp.data.emp_name emp.data.salary
1              Rick          623.30
2               Dan          515.20
3          Michelle           611.00
4              Ryan           729.00
5              Gary          843.25
```

Extract the first two rows and then all columns

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Extract first two rows.
result <- emp.data[1:2,]
print(result)
```

When we execute the above code, it produces the following result −

```
  emp_id    emp_name   salary    start_date
1      1     Rick       623.3     2012-01-01
2      2     Dan        515.2     2013-09-23
```

Extract 3rd and 5th row with 2nd and 4th column

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

         start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
```

```
)

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

When we execute the above code, it produces the following result −

```
  emp_name start_date
3 Michelle 2014-11-15
5     Gary 2015-03-27
```

Expand Data Frame

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

Add Column

Just add the column vector using a new column name.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)

# Add the "dept" coulmn.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
```

```
v <- emp.data
print(v)
```

When we execute the above code, it produces the following result −

```
  emp_id   emp_name    salary    start_date       dept
1     1    Rick        623.30    2012-01-01       IT
2     2    Dan         515.20    2013-09-23       Operations
3     3    Michelle    611.00    2014-11-15       IT
4     4    Ryan        729.00    2014-05-11       HR
5     5    Gary        843.25    2015-03-27       Finance
```

Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the rbind() function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the first data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   dept = c("IT","Operations","IT","HR","Finance"),
   stringsAsFactors = FALSE
)

# Create the second data frame
```

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

# Bind the two data frames.
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

When we execute the above code, it produces the following result −

	emp_id	emp_name	salary	start_date	dept
1	1	Rick	623.30	2012-01-01	IT
2	2	Dan	515.20	2013-09-23	Operations
3	3	Michelle	611.00	2014-11-15	IT
4	4	Ryan	729.00	2014-05-11	HR
5	5	Gary	843.25	2015-03-27	Finance
6	6	Rasmi	578.00	2013-05-21	IT
7	7	Pranab	722.50	2013-07-30	Operations
8	8	Tusar	632.80	2014-06-17	Fianance

Previous Page Print Page

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called "library" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at R Packages.

Below is a list of commands to be used to check, verify and use the R packages.

Check Available R Packages

Get library locations containing R packages

.libPaths()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

[2] "C:/Program Files/R/R-3.2.2/library"

Get the list of all the packages installed

library()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

Packages in library ‘C:/Program Files/R/R-3.2.2/library’:

base The R Base Package

boot Bootstrap Functions (Originally by Angelo Canty for S)

class Functions for Classification

cluster "Finding Groups in Data": Cluster Analysis Extended Rousseeuw et al.

codetools Code Analysis Tools for R

compiler The R Compiler Package

datasets The R Datasets Package

foreign Read Data Stored by 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...

graphics The R Graphics Package

grDevices The R Graphics Devices and Support for Colours and Fonts

grid The Grid Graphics Package

KernSmooth Functions for Kernel Smoothing Supporting Wand & Jones (1995)

lattice Trellis Graphics for R

MASS Support Functions and Datasets for Venables and Ripley's MASS

Matrix Sparse and Dense Matrix Classes and Methods

methods Formal Methods and Classes

mgcv Mixed GAM Computation Vehicle with GCV/AIC/REML Smoothness Estimation

nlme Linear and Nonlinear Mixed Effects Models

nnet Feed-Forward Neural Networks and Multinomial Log-Linear Models

parallel Support for Parallel computation in R

rpart Recursive Partitioning and Regression Trees

spatial Functions for Kriging and Point Pattern Analysis

splines Regression Spline Functions and Classes

stats The R Stats Package

stats4 Statistical Functions using S4 Classes

survival Survival Analysis

tcltk Tcl/Tk Interface

tools Tools for Package Development

utils The R Utils Package

Get all packages currently loaded in the R environment

search()

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

[1] ".GlobalEnv" "package:stats" "package:graphics"

[4] "package:grDevices" "package:utils" "package:datasets"

[7] "package:methods" "Autoloads" "package:base"

Install a New Package

There are two ways to add new R packages. One is installing directly from the CRAN directory and another is downloading the package to your local system and installing it manually.

Install directly from CRAN

The following command gets the packages directly from CRAN webpage and installs the package in the R environment. You may be prompted to choose a nearest mirror. Choose the one appropriate to your location.

```
install.packages("Package Name")
```

```
# Install the package named "XML".
install.packages("XML")
```

Install package manually

Go to the link R Packages to download the package needed. Save the package as a .zip file in a suitable location in the local system.

Now you can run the following command to install this package in the R environment.

```
install.packages(file_name_with_path, repos = NULL, type = "source")
```

```
# Install the package named "XML"
install.packages("E:/XML_3.98-1.3.zip", repos = NULL, type = "source")
```

Load Package to Library

Before a package can be used in the code, it must be loaded to the current R environment. You also need to load a package that is already installed previously but not available in the current environment.

A package is loaded using the following command −

library("package Name", lib.loc = "path to library")

Load the package named "XML"

install.packages("E:/XML_3.98-1.3.zip", repos = NULL, type = "source")

Data Categorization

1. How to Categorize Numeric Data with cut() Function

In this section, we learn cut() function to convert numerical data into categories. The cut() function includes breaks argument. We can specify the break points or the number of categories. For example, in our example, we want to categorize the data in three groups. Therefore, we can specify the break points as 30 and 60. Also, we need to specify end points. For example, we write -Inf and Inf as end points. Then, we specify the breaks argument as three. Moreover, we can define the labels of the categories with labels argument. For instance, we define the labels as low, medium and high.

Categories <- cut(data, breaks = c(-Inf,30,60,Inf), labels = c("Low","Medium","High"))

table(Categories)

Categories

Low Medium High

15 15 15

Categories <- cut(data, breaks = 3, labels = c("Low","Medium","High"))

table(Categories)

Categories

Low Medium High

15 15 15

Also Check: How to Handle Missing Values in R

2. How to Categorize Numeric Data with discretize() Function

In this part, we learn the use of discretize() function available in arules package (Hahsler et al., 2021). We specify the number of categories. For example, we want to categorize our variable in three groups. Therefore, we specify the breaks argument as three. Moreover, we can define the labels of classes with labels argument.

```
Categories <- arules::discretize(data, breaks = 3, labels = c("Low","Medium","High"))

table(Categories)

## Categories

##    Low Medium   High

##     15     15     15
```

Also Check: How to Recode Character Variables in R

3. How to Categorize Numeric Data with group_var() Function

In this section, we learn how to use group_var() function available in sjmisc package (Ludecke, 2018) to convert the numerical variable into classes. We need to specify the range of each category with size argument. Also, as.num argument should be set to FALSE if the labels of categories want to be specifed.

```
Categories <- sjmisc::group_var(data, size = 30, as.num = FALSE)

levels(Categories) <- c("Low","Medium","High")

table(Categories)

## Categories

##    Low Medium   High

##     15     15     15
```

4. How to Categorize Numeric Data with frq() Function

In this part, we learn how to categorize numeric variables with frq() function available in sjmisc package (Ludecke, 2018). We specify the number of categories with auto.grp argument. With this function, we can also construct a frequency table including frequency and (raw, valid and cumulative) percentages. Also, the function returns mean and standard deviation of the data. The frq() function presents the number of NA values in table. Moreover, we can define the names of groups using frq() function together with group_var() function.

```
library(sjmisc)
frq(data, auto.grp = 3)

## x <numeric>
## # total N=45  valid N=45  mean=45.00  sd=26.27

## Value | Label |  N | Raw % | Valid % | Cum. %
## ------------------------------------------
##     1 |  1-30 | 15 | 33.33 |   33.33 |  33.33
##     2 | 31-60 | 15 | 33.33 |   33.33 |  66.67
##     3 | 61-90 | 15 | 33.33 |   33.33 | 100.00
##  <NA> |  <NA> |  0 |  0.00 |    <NA> |   <NA>

Categories <- group_var(data, size = 30, as.num = FALSE)
levels(Categories) <- c("Low","Medium","High")
frq(Categories)

## x <categorical>
## # total N=45  valid N=45  mean=2.00  sd=0.83

## Value  |  N | Raw % | Valid % | Cum. %
## ---------------------------------------
## Low    | 15 | 33.33 |   33.33 |  33.33
## Medium | 15 | 33.33 |   33.33 |  66.67
## High   | 15 | 33.33 |   33.33 | 100.00
```

Data Management

Sorting Data

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

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

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

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

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

detach(mtcars)
```

Merging Data

Adding Columns

To merge two data frames (datasets) horizontally, use the merge function. In most cases, you join two data frames by one or more common key variables (i.e., an inner join).

```
# merge two data frames by ID
total <- merge(data frameA,data frameB,by="ID")

# merge two data frames by ID and Country
total <- merge(data frameA,data frameB,by=c("ID","Country"))
```

Adding Rows

To join two data frames (datasets) vertically, use the rbind function. The two data frames must have the same variables, but they do not have to be in the same order.

```
total <- rbind(data frameA, data frameB)
```

If data frameA has variables that data frameB does not, then either:

Delete the extra variables in data frameA or

Create the additional variables in data frameB and set them to NA (missing)

before joining them with rbind( ).

Aggregating Data

It is relatively easy to collapse data in R using one or more BY variables and a defined function.

```
# aggregate data frame mtcars by cyl and vs, returning means
# for numeric variables
attach(mtcars)
aggdata <-aggregate(mtcars, by=list(cyl,vs),
  FUN=mean, na.rm=TRUE)
print(aggdata)
detach(mtcars)
```

Subsetting Data

R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations. The following code snippets demonstrate ways to keep or delete variables and observations and to take random samples from a dataset.

Selecting (Keeping) Variables

```
# select variables v1, v2, v3
myvars <- c("v1", "v2", "v3")
newdata <- mydata[myvars]

# another method
myvars <- paste("v", 1:3, sep="")
newdata <- mydata[myvars]
```

```
# select 1st and 5th thru 10th variables
newdata <- mydata[c(1,5:10)]
```

To practice this interactively, try the selection of data frame elements exercises in the Data frames chapter of this introduction to R course.

Excluding (DROPPING) Variables

```
# exclude variables v1, v2, v3
myvars <- names(mydata) %in% c("v1", "v2", "v3")
newdata <- mydata[!myvars]

# exclude 3rd and 5th variable
newdata <- mydata[c(-3,-5)]

# delete variables v3 and v5
mydata$v3 <- mydata$v5 <- NULL
```

Selecting Observations

```
# first 5 observations
newdata <- mydata[1:5,]

# based on variable values
newdata <- mydata[ which(mydata$gender=='F'
& mydata$age > 65), ]

# or
attach(mydata)
newdata <- mydata[ which(gender=='F' & age > 65),]
detach(mydata)
```

Selection using the Subset Function

The subset( ) function is the easiest way to select variables and observations. In the following example, we select all rows that have a value of age greater than or equal to 20 or age less then 10. We keep the ID and Weight columns.

```
# using subset function

newdata <- subset(mydata, age >= 20 | age < 10,

select=c(ID, Weight))
```

In the next example, we select all men over the age of 25 and we keep variables weight through income (weight, income and all columns between them).

```
# using subset function (part 2)

newdata <- subset(mydata, sex=="m" & age > 25,

select=weight:income)
```

To practice the subset() function, try this this interactive exercise. on subsetting data.tables.

Random Samples

Use the sample( ) function to take a random sample of size n from a dataset.

```
# take a random sample of size 50 from a dataset mydata

# sample without replacement

mysample <- mydata[sample(1:nrow(mydata), 50,

  replace=FALSE),]
```

When using the aggregate() function, the by variables must be in a list (even if there is only one). The function can be built-in or user provided.