Chapter 6 Basic Operations in R

6.1 Basic Commands

To get current working directory:-

```
getwd()
## [1] "C:/Users/mp64310/Desktop/100 Page R Book/bookdown-demo-master"
```

To set a new working directory use setwd()

List all the objects in your local workspace using ls() command. List all the files in your working directory using list.files() or dir().

We will start with the basic operations in R. Programming is best learnt hands on and hence I will strongly suggest to start typing following commands in your R Studio Console.

6.2 Airthmetic Operations in R

Addition, Substraction,multiplication and Division are some of the basic Airthmetic operations in R.

6.2.1 Addition & Substraction

```
2+2  # Addition
## [1] 4
2-2  # Substraction
## [1] 0
```

6.2.2 Multiplication & Division

```
2*2 # Multiplication
## [1] 4
2/2 # Division
## [1] 1
```

6.2.4 Square Root

```
2^(1/2)
## [1] 1.414214
```

Alternatively you can use sqrt function

```
sqrt(2)
## [1] 1.414214
```

Here ^ are the power opertors It can be used to calculate a^ b , i.e. ‘a’ rasie to the power of ‘b’

Here ^ are the power opertors It can be used to calculate a^ b , i.e. ‘a’ rasie to the power of ‘b’

6.2.5 Integer Division- Returns Integer after division

```
5%/%2
## [1] 2
```

6.2.6 Modulus- Returns Remainder after division

```
5%%2
## [1] 1
```

6.2.7 Operators Precedence and Associativity

We can also see the operator associativity in R Associativity means which of the operations wil take precedence. Lets say you have both division and multiplication then which operation should take precdence.

```
#Should the result be 21 or 11?
5+2*3
## [1] 11
```

As you might have guessed from the operator here the multiplication operator “*" takes precedence over “+” addition operator.

```
# What if we include brackets? What will be the result then?
(5+2)*3
## [1] 21
# What about Division "/" , Addition and Multiplication "*" operator.
# Check the result one.
5*5/2+3
## [1] 15.5
```

6.3 Relational Operators in R

Relational operators are frequently used for conditional filtering of Data in R. We can assign a value to a variable using “<-”. A variable is named unit of data that can be assigned a value

```
x=5 # we have assigned x value of 5
y=7  # we have assigned y value of 7
# To check value of X just type x in console
x
## [1] 5
y
## [1] 7
#The '>' operator is caled relational operation and mean greater than
x>y
## [1] FALSE
y>x
```

```
## [1] TRUE
```

As you can see the output is logical .

```
5>=5 # Left Value Greater than or Equal to RightValue. IF TRUE the output will be a logical - TRUE .
## [1] TRUE
5>=7 #IF FALSE output will be logical FALSE
## [1] FALSE
10==5 # Compare two values whether they are equal or not. IF Equal the output will be equal.
## [1] FALSE
5==5
## [1] TRUE
10!=5 # Similarly "!=" is used for notequal to . If the values are not equal it will give output TRUE
## [1] TRUE
```

6.4 Logical Operations in R

A logical operation will return an output of True or False and is frequently used for conditional filtering and other operations in R.

```
#Character "!" is Logical NOT
# so a not TRUE will evaluate to FALSE
!TRUE
## [1] FALSE
# Symbol "&" is used for elementwise Logical AND for example TRUE & TRUE will evaluate to TRUE, while TRUE and FALSE will evaluate to FALSE
TRUE & TRUE
## [1] TRUE
TRUE & FALSE
## [1] FALSE
# Similarly
FALSE & FALSE
## [1] FALSE
```

Symbol “|” is used for Logical Operations If any of the condition is TRUE then TRUE ELSE FALSE

```
TRUE | TRUE # TRUE OR TRUE will return TRUE
## [1] TRUE
TRUE | FALSE # TRUE OR FALSE will return TRUE
## [1] TRUE
FALSE | FALSE # FALSE OR FALSE Will return FALSE
## [1] FALSE
```

The ‘&’ and ‘|’ symbol perform logical operations elementwise.

```
TRUE && TRUE
## [1] TRUE
TRUE & FALSE
## [1] FALSE
TRUE & FALSE
## [1] FALSE
FALSE && FALSE
## [1] FALSE
```

Control Flow

- Create a conditional using `if`, `else if`, and `else`

```
if(x > 0){
        print("value is positive")
} else if (x < 0){
        print("value is negative")
} else{
        print("value is neither positive nor negative")
}
```

- create a `for` loop to process elements in a collection one at a time

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

This will print:

```
1
2
3
4
5
```

- Use `==` to test for equality
    - `3 == 3`, will return `TRUE`,
    - `'apple' == 'orange'` will return `FALSE`
- `X & Y` is `TRUE` is both X and Y are true
- `X | Y` is `TRUE` if either X or Y, or both are true

Functions

- Defining a function:

```
is_positive <- function(integer_value){
        if(integer_value > 0){
           TRUE
        }
        else{
           FALSE
        {
}
```

In R, the last executed line of a function is automatically returned

- Specifying a default value for a function argument

```
increment_me <- function(value_to_increment, value_to_increment_by = 1){
        value_to_increment + value_to_increment_by
}
```

`increment_me(4)`, will return 5

`increment_me(4, 6)`, will return 10

- Call a function by using `function_name(function_arguments)`
- apply family of functions: `apply()`, `sapply()`, `lapply()`, and `mapply()`

`apply(dat, MARGIN = 2, mean)` will return the average (`mean`) of each column in `data`

5 Getting Data In and Out of R

5.1 Reading and Writing Data

Watch a video of this section

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).
-

5.2 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). I know, I know, everyone always says to read the help file, but this one is actually worth reading.
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 arguments

```
> 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).

5.3 Reading in Larger Datasets with read.table

Watch a video of this section

With much larger datasets, there are a few things that you can do that will make your life easier and will prevent R from choking.

- 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)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

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 ar you using? Some operating systems can limit the amount of memory a single process can access

Addition operator (+):
The values at the corresponding positions of both the operands are added. Consider the following R snippet to add two vectors:

```
Input : a <- c (1, 0.1)
        b <- c (2.33, 4)
        print (a+b)
Output : 3.33  4.10
```

Subtraction Operator (-):
The second operand values are subtracted from the first. Consider the following R snippet to subtract two variables:

```
Input : a <- 6
        b <- 8.4
        print (a-b)
Output : -2.4
```

Multiplication Operator (*):
The multiplication of corresponding elements of vectors and Integers are multiplied with the use of the '*' operator.

```
Input : B= matrix(c(4,6i),nrow=1,ncol=2)
        C= matrix(c(2,2i ),nrow=1, ncol=2)
        print (B*C)
Output : 8+0i  -12+0i
The elements at corresponding positions of matrices are multiplied.
```

Division Operator (/):
The first operand is divided by the second operand with the use of the '/' operator.

```
Input : a <- 1
        b <- 0
        print (a/b)
Output : -Inf
```

Power Operator (^):

The first operand is raised to the power of the second operand.

```
Input : list1 <- c(2, 3)
        list2 <- c(2,4)
        print(list1^list2)
Output : 4  81
```

Modulo Operator (%%):

The remainder of the first operand divided by the second operand is returned.

```
Input : list1<- c(2, 3)
        list2<-c(2,4)
        print(list1%%list2)
Output : 0  3
```

The following R code illustrates the usage of all Arithmetic Operators in R:

R

```
# R program to illustrate

# the use of Arithmetic operators

vec1 <- c(0, 2)

vec2 <- c(2, 3)

# Performing operations on Operands

cat ("Addition of vectors :", vec1 + vec2, "\n")

cat ("Subtraction of vectors :", vec1 - vec2, "\n")

cat ("Multiplication of vectors :", vec1 * vec2, "\n")

cat ("Division of vectors :", vec1 / vec2, "\n")

cat ("Modulo of vectors :", vec1 %% vec2, "\n")

cat ("Power operator :", vec1 ^ vec2)
```

Output:

```
Addition of vectors : 2 5

Subtraction of vectors : -2 -1

Multiplication of vectors : 0 6

Division of vectors : 0 0.6666667

Modulo of vectors : 0 2

Power operator : 0 8
```

Logical Operators

Logical operations simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non-zero integer value is considered as a TRUE value, be it a complex or real number.

Element-wise Logical AND operator (&):

Returns True if both the operands are True.

```
Input : list1 <- c(TRUE, 0.1)
        list2 <- c(0,4+3i)
        print(list1 & list2)
Output : FALSE   TRUE
```

Any non zero integer value is considered as a TRUE value, be it complex or real number.

Element-wise Logical OR operator (|):

Returns True if either of the operands is True.

```
Input : list1 <- c(TRUE, 0.1)
        list2 <- c(0,4+3i)
        print(list1|list2)
Output : TRUE  TRUE
```

NOT operator (!):

A unary operator that negates the status of the elements of the operand.

```
Input : list1 <- c(0,FALSE)
        print(!list1)
Output : TRUE  TRUE
```

Logical AND operator (&&):

Returns True if both the first elements of the operands are True.

```
Input : list1 <- c(TRUE, 0.1)
        list2 <- c(0,4+3i)
        print(list1 && list2)
Output : FALSE
```

Compares just the first elements of both the lists.

Logical OR operator (||):

Returns True if either of the first elements of the operands is True.

```
Input : list1 <- c(TRUE, 0.1)
        list2 <- c(0,4+3i)
        print(list1||list2)
Output : TRUE
```

The following R code illustrates the usage of all Logical Operators in R:

R

```
# R program to illustrate

# the use of Logical operators
```

```
vec1 <- c(0,2)

vec2 <- c(TRUE,FALSE)

# Performing operations on Operands

cat ("Element wise AND :", vec1 & vec2, "\n")

cat ("Element wise OR :", vec1 | vec2, "\n")

cat ("Logical AND :", vec1 && vec2, "\n")

cat ("Logical OR :", vec1 || vec2, "\n")

cat ("Negation :", !vec1)
```

Output:

```
Element wise AND : FALSE FALSE

Element wise OR : TRUE TRUE

Logical AND : FALSE

Logical OR : TRUE

Negation : TRUE FALSE
```

Relational Operators

The relational operators carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE.

Less than (<):

Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.

```
Input :  list1 <- c(TRUE, 0.1,"apple")
         list2 <- c(0,0.1,"bat")
         print(list1<list2)
Output :  FALSE FALSE  TRUE
```

Less than equal to (<=):

Returns TRUE if the corresponding element of the first operand is less than or equal to that of the second operand. Else returns FALSE.

```
Input :  list1 <- c(TRUE, 0.1,"apple")
         list2 <- c(0,0.1,"bat")
         print(list<=list2)
Output : FALSE TRUE TRUE
```

Greater than (>):

Returns TRUE if the corresponding element of the first operand is greater than that of the second operand. Else returns FALSE.

```
Input :  list1 <- c(TRUE, 0.1,"apple")
         list2 list2)
Output : TRUE FALSE FALSE
```

Greater than equal to (>=):

Returns TRUE if the corresponding element of the first operand is greater or equal to that of the second operand. Else returns FALSE.

```
Input :  list1 <- c(TRUE, 0.1,"apple")
         list2 =list2)
Output : TRUE TRUE FALSE
```

Not equal to (!=):

Returns TRUE if the corresponding element of the first operand is not equal to the second operand. Else returns FALSE.

```
Input : list1 <- c(TRUE, 0.1,""apple")
        list2 <- c(0,0.1,"bat")
        print(list1!=list2)
Output : TRUE FALSE TRUE
```

The following R code illustrates the usage of all Relational Operators in R:

R

```
# R program to illustrate

# the use of Relational operators

vec1 <- c(0, 2)

vec2 <- c(2, 3)

# Performing operations on Operands

cat ("Vector1 less than Vector2 :", vec1 < vec2, "\n")

cat ("Vector1 less than equal to Vector2 :", vec1 <= vec2, "\n")

cat ("Vector1 greater than Vector2 :", vec1 > vec2, "\n")

cat ("Vector1 greater than equal to Vector2 :", vec1 >= vec2, "\n")

cat ("Vector1 not equal to Vector2 :", vec1 != vec2, "\n")
```

Output:

```
Vector1 less than Vector2 : TRUE TRUE

Vector1 less than equal to Vector2 : TRUE TRUE

Vector1 greater than Vector2 : FALSE FALSE

Vector1 greater than equal to Vector2 : FALSE FALSE

Vector1 not equal to Vector2 : TRUE TRUE
```

Assignment Operators

Assignment operators are used to assigning values to various data objects in R. The objects may be integers, vectors, or functions. These values are then stored by the assigned variable names. There are two kinds of assignment operators: Left and Right

Left Assignment (<- or <<- or =):

Assigns a value to a vector.

```
Input : vec1 = c("ab", TRUE)
        print (vec1)
Output : "ab"   "TRUE"
```

Right Assignment (-> or ->>):

Assigns value to a vector.

```
Input : c("ab", TRUE) ->> vec1
        print (vec1)
Output : "ab"   "TRUE"
```

The following R code illustrates the usage of all Relational Operators in R:

R

```
# R program to illustrate

# the use of Assignment operators

vec1 <- c(2:5)

c(2:5) ->> vec2

vec3 <<- c(2:5)

vec4 = c(2:5)

c(2:5) -> vec5

# Performing operations on Operands

cat ("vector 1 :", vec1, "\n")

cat("vector 2 :", vec2, "\n")

cat ("vector 3 :", vec3, "\n")

cat("vector 4 :", vec4, "\n")

cat("vector 5 :", vec5)
```

Output:

```
vector 1 : 2 3 4 5

vector 2 : 2 3 4 5

vector 3 : 2 3 4 5
```

```
vector 4 : 2 3 4 5

vector 5 : 2 3 4 5
```

Miscellaneous Operators

These are the mixed operators that simulate the printing of sequences and assignment of vectors, either left or right-handed.

%in% Operator:

Checks if an element belongs to a list and returns a boolean value TRUE if the value is present else FALSE.

```
Input : val <- 0.1
        list1 <- c(TRUE, 0.1,"apple")
        print (val %in% list1)
Output : TRUE
Checks for the value 0.1 in the specified list. It exists,
therefore, prints TRUE.
```

Colon Operator(:):

Prints a list of elements starting with the element before the colon to the element after it.

```
Input :  print (1:5)
Output : 1 2 3 4 5
Prints a sequence of the numbers from 1 to 5.
```

%*% Operator:

This operator is used to multiply a matrix with its transpose. Transpose of the matrix is obtained by interchanging the rows to columns and columns to rows. The number of columns of the first matrix must be equal to the number of rows of the second matrix. Multiplication of the matrix A with its transpose, B, produces a square matrix.

```
Input :  mat = matrix(c(1,2,3,4,5,6),nrow=2,ncol=3)
         print (mat)
         print( t(mat))
         pro = mat %*% t(mat)
         print(pro)
Output :     [,1] [,2] [,3]      #original matrix of order 2x3
        [1,]   1    3    5
        [2,]   2    4    6
             [,1] [,2]           #transposed matrix of order 3x2
        [1,]    1    2
        [2,]    3    4
        [3,]    5    6
             [,1] [,2]          #product matrix of order 2x2
        [1,]   35   44
        [2,]   44   56
```

The following R code illustrates the usage of all Miscellaneous Operators in R:

R

```
# R program to illustrate

# the use of Miscellaneous operators

mat <- matrix (1:4, nrow = 1, ncol = 4)

print("Matrix elements using : ")

print(mat)


product = mat %*% t(mat)

print("Product of matrices")

print(product,)

cat ("does 1 exist in prod matrix :", "1" %in% product)
```

Output:

```
[1] "Matrix elements using : "
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4

[1] "Product of matrices"
     [,1]
[1,]   30
```

```
# defining vector
x <- c(7, 15, 23, 12, 44, 56, 32)

# output to be present as PNG file
png(file = "barplot.png")

# plotting vector
barplot(x, xlab = "GeeksforGeeks Audience",
              ylab = "Count", col = "white",
              col.axis = "darkgreen",
              col.lab = "darkgreen")
```

```
# saving the file
dev.off()


# defining vector x with number of articles
x <- c(210, 450, 250, 100, 50, 90)

# defining labels for each value in x
names(x) <- c("Algo", "DS", "Java", "C", "C++", "Python")

# output to be present as PNG file
png(file = "piechart.png")

# creating pie chart
pie(x, labels = names(x), col = "white",
main = "Articles on GeeksforGeeks", radius = -1,
col.main = "darkgreen")

# saving the file
dev.off()


# defining vector
x <- c(21, 23, 56, 90, 20, 7, 94, 12,
       57, 76, 69, 45, 34, 32, 49, 55, 57)

# output to be present as PNG file
png(file = "hist.png")

# hist(x, main = "Histogram of Vector x",
              xlab = "Values",
              col.lab = "darkgreen",
```

```
              col.main = "darkgreen")

# saving the file
dev.off()



# taking input from dataset Orange already
# present in R
orange <- Orange[, c('age', 'circumference')]

# output to be present as PNG file
png(file = "plot.png")

# plotting
plot(x = orange$age, y = orange$circumference, xlab = "Age",
ylab = "Circumference", main = "Age VS Circumference",
col.lab = "darkgreen", col.main = "darkgreen",
col.axis = "darkgreen")

# saving the file
dev.off()



# output to be present as PNG file
png(file = "plotmatrix.png")

# plotting scatterplot matrix
# using dataset Orange
pairs(~age + circumference, data = Orange,
col.axis = "darkgreen")
```

```
# saving the file
dev.off()



# defining vector with ages of employees
x <- c(42, 21, 22, 24, 25, 30, 29, 22,
       23, 23, 24, 28, 32, 45, 39, 40)

# output to be present as PNG file
png(file = "boxplot.png")

# plotting
boxplot(x, xlab = "Box Plot", ylab = "Age",
col.axis = "darkgreen", col.lab = "darkgreen")

# saving the file
dev.off()







# Data Inspection in EDA
# loading the required packages
library(aqp)
library(soilDB)

# Load from the loafercreek dataset
data("loafercreek")
```

```
# Construct generalized horizon designations
n < - c("A", "BAt", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p < - c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
             "Cr", "R")

# Compute genhz labels and
# add to loafercreek dataset
loafercreek$genhz < - generalize.hz(
       loafercreek$hzname,
       n, p)

# Extract the horizon table
h < - horizons(loafercreek)

# Examine the matching of pairing of
# the genhz label to the hznames
table(h$genhz, h$hzname)

vars < - c("genhz", "clay", "total_frags_pct",
             "phfield", "effclass")
summary(h[, vars])

sort(unique(h$hzname))
h$hzname < - ifelse(h$hzname == "BT",
                                  "Bt", h$hzname)



# EDA
# Descriptive Statistics
```

```
# Measures of Central Tendency

#loading the required packages
library(aqp)
library(soilDB)

# Load from the loafercreek dataset
data("loafercreek")

# Construct generalized horizon designations
n <- c("A", "BAt", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p <- c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
       "Cr", "R")

# Compute genhz labels and
# add to loafercreek dataset
loafercreek$genhz <- generalize.hz(
                                    loafercreek$hzname,
                                    n, p)

# Extract the horizon table
h <- horizons(loafercreek)

# Examine the matching of pairing
# of the genhz label to the hznames
table(h$genhz, h$hzname)

vars <- c("genhz", "clay", "total_frags_pct",
              "phfield", "effclass")
summary(h[, vars])
```

```
sort(unique(h$hzname))
h$hzname <- ifelse(h$hzname == "BT",
                           "Bt", h$hzname)

# first remove missing values
# and create a new vector
clay <- na.exclude(h$clay)

mean(clay)
median(clay)
sort(table(round(h$clay)),
       decreasing = TRUE)[1]
table(h$genhz)
# append the table with
# row and column sums
addmargins(table(h$genhz,
                           h$texcl))

# calculate the proportions
# relative to the rows, margin = 1
# calculates for rows, margin = 2 calculates
# for columns, margin = NULL calculates
# for total observations
round(prop.table(table(h$genhz, h$texture_class),
                           margin = 1) * 100)
knitr::kable(addmargins(table(h$genhz, h$texcl)))

aggregate(clay ~ genhz, data = h, mean)
aggregate(clay ~ genhz, data = h, median)
aggregate(clay ~ genhz, data = h, summary)
```

```
# EDA
# Descriptive Statistics
# Measures of Dispersion

# loading the packages
library(aqp)
library(soilDB)

# Load from the loafercreek dataset
data("loafercreek")

# Construct generalized horizon designations
n <- c("A", "BAt", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p <- c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
       "Cr", "R")

# Compute genhz labels and add
# to loafercreek dataset
loafercreek$genhz <- generalize.hz(
                                   loafercreek$hzname,
                                   n, p)

# Extract the horizon table
h <- horizons(loafercreek)

# Examine the matching of pairing of
# the genhz label to the hznames
```

```
table(h$genhz, h$hzname)

vars <- c("genhz", "clay", "total_frags_pct",
              "phfield", "effclass")
summary(h[, vars])

sort(unique(h$hzname))
h$hzname <- ifelse(h$hzname == "BT",
                           "Bt", h$hzname)

# first remove missing values
# and create a new vector
clay <- na.exclude(h$clay)
var(h$clay, na.rm=TRUE)
sd(h$clay, na.rm = TRUE)
cv <- sd(clay) / mean(clay) * 100
cv
quantile(clay)
range(clay)
IQR(clay)




# EDA
# Descriptive Statistics
# Correlation

# loading the required packages
library(aqp)
library(soilDB)
```

```
# Load from the loafercreek dataset
data("loafercreek")

# Construct generalized horizon designations
n <- c("A", "BAt", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p <- c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
       "Cr", "R")

# Compute genhz labels and add
# to loafercreek dataset
loafercreek$genhz <- generalize.hz(
                                  loafercreek$hzname,
                                  n, p)

# Extract the horizon table
h <- horizons(loafercreek)

# Examine the matching of pairing
# of the genhz label to the hznames
table(h$genhz, h$hzname)

vars <- c("genhz", "clay", "total_frags_pct",
             "phfield", "effclass")
summary(h[, vars])

sort(unique(h$hzname))
h$hzname <- ifelse(h$hzname == "BT",
                           "Bt", h$hzname)

# first remove missing values
```

```
# and create a new vector
clay <- na.exclude(h$clay)

# Compute the middle horizon depth
h$hzdepm <- (h$hzdepb + h$hzdept) / 2
vars <- c("hzdepm", "clay", "sand",
              "total_frags_pct", "phfield")
round(cor(h[, vars], use = "complete.obs"), 2)
```

```
# EDA Graphical Method Distributions

# loading the required packages
library("ggplot2")
library(aqp)
library(soilDB)

# Load from the loafercreek dataset
data("loafercreek")

# Construct generalized horizon designations
n <- c("A", "BAt", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p <- c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
       "Cr", "R")
```

```
# Compute genhz labels and add
# to loafercreek dataset
loafercreek$genhz <- generalize.hz(
                                    loafercreek$hzname, n, p)

# Extract the horizon table
h <- horizons(loafercreek)

# Examine the matching of pairing
# of the genhz label to the hznames
table(h$genhz, h$hzname)

vars <- c("genhz", "clay", "total_frags_pct",
              "phfield", "effclass")
summary(h[, vars])

sort(unique(h$hzname))
h$hzname <- ifelse(h$hzname == "BT",
                           "Bt", h$hzname)

# graphs
# bar plot
ggplot(h, aes(x = texcl)) +geom_bar()

# histogram
ggplot(h, aes(x = clay)) +
geom_histogram(bins = nclass.Sturges(h$clay))

# density curve
ggplot(h, aes(x = clay)) + geom_density()
```

```
# box plot
ggplot(h, (aes(x = genhz, y = clay))) +
geom_boxplot()

# QQ Plot for Clay
ggplot(h, aes(sample = clay)) +
geom_qq() +
geom_qq_line()



# EDA
# Graphical Method
# Scatter and Line plot

# loading the required packages
library("ggplot2")
library(aqp)
library(soilDB)

# Load from the loafercreek dataset
data("loafercreek")

# Construct generalized horizon designations
n <- c("A", "BAt", "Bt1", "Bt2", "Cr", "R")

# REGEX rules
p <- c("A", "BA|AB", "Bt|Bw", "Bt3|Bt4|2B|C",
       "Cr", "R")

# Compute genhz labels and add
# to loafercreek dataset
loafercreek$genhz <- generalize.hz(
```

```
                                    loafercreek$hzname, n, p)

# Extract the horizon table
h <- horizons(loafercreek)

# Examine the matching of pairing
# of the genhz label to the hznames
table(h$genhz, h$hzname)

vars <- c("genhz", "clay", "total_frags_pct",
              "phfield", "effclass")
summary(h[, vars])

sort(unique(h$hzname))
h$hzname <- ifelse(h$hzname == "BT",
                           "Bt", h$hzname)

# graph
# scatter plot
ggplot(h, aes(x = clay, y = hzdepm)) +
geom_point() +
ylim(100, 0)

# line plot
ggplot(h, aes(y = clay, x = hzdepm,
                                    group = peiid)) +
geom_line() + coord_flip() + xlim(100, 0)
```