```
#Chriss Jordan Oboa
#Professsor Dimitriglou
#CS474
#October 24th, 2022


                          #1) Basic Data Types


      #a) Numeric

# assign a decimal value
x = 43.6
# print the value of x
x
# print the class name of x
class(x)

#Furthermore, even if we assign an integer to a variable k, it is still being saved as a
numeric value.

y = 4
# print the value of y
y
# print the class name of y
class(y)

#The fact that k is not an integer can be confirmed with the is.integer function

# is y an integer?
is.integer(y)


      #b) Interger

#In order to create an integer variable in R, we invoke the integer function. We can be
assured that y is indeed an integer by applying the is.integer function.

z = as.integer(129)
# print the value of z
z

# print the class name of z
class(z)

# is z an integer?
is.integer(z)

#We can also declare an integer by appending an L suffix.

b = 3L

# is y an integer?
is.integer(b)

#Incidentally, we can coerce a numeric value into an integer with the as.integer function.

# coerce a numeric value
as.integer(3.14)

#And we can parse a string for decimal values in much the same way.

# coerce a decimal string
as.integer("4.47")
```

```
#On the other hand, it is erroneous trying to parse a non-decimal string.
# coerce an non-decimal string
as.integer("Chriss")

#Often, it is useful to perform arithmetic on logical values. Just like the C language,
TRUE has the value 1, while FALSE has value 0.
# the numeric value of TRUE
as.integer(TRUE)

# the numeric value of FALSE
as.integer(FALSE)


      #c) Interger

A complex value in R is defined via the pure imaginary value i.

# create a complex number
w = 5 + 4i

# print the value of w
w

# print the class name of w
class(w)


#The following gives an error as -1 is not a complex value.

# square root of -1
sqrt(-1)


#Instead, we have to use the complex value -1 + 0i.

# square root of -1+0i
sqrt(-1+0i)

#An alternative is to coerce -1 into a complex value.

sqrt(as.complex(-1))


      #d) Logical

#A logical value is often created via comparison between variables.
# sample values
x = 4; y = 7

# is x larger than y?
z = x > y

# print the logical value
z

# print the class name of z
class(z)

#Standard logical operations are "&" (and), "|" (or), and "!" (negation).

u = TRUE; v = FALSE

# u AND v
u & v
```

```
# u OR v
u | v
# negation of u
!v

      #e) Character

x = as.character(6.74)
# print the character string
x
# print the class name of x
class(x)

#Two character values can be concatenated with the paste function.
fname = "Chriss"; lname ="Oboa"
paste(fname, lname)

#However, it is often more convenient to create a readable string with the sprintf
function, which has a C language syntax.
sprintf("%s has %d dollars", "Chriss", 1000000)

#To extract a substring, we apply the substr function. Here is an example showing how to
extract the substring between the third and twelfth positions in a string.
substr("Chriss is a great deal.", start=2, stop=13)

#And to replace the first occurrence of the word "little" by another word "big" in the
string, we apply the sub function.
sub("great", "big", "Chriss is a great deal")


                            #2) Vector
#a vector containing three numeric values 2, 3 and 5.
c(32, 332, 455)
#a vector of logical values.
c(TRUE, FALSE, TRUE, FALSE, FALSE)
#The number of members in a vector is given by the length function
length(c("elephant", "baby", "dogs"))

      #a) Combining Vectors

#Vectors can be combined via the function c.
n = c(3, 7)
s = c("aa", "dd")
c(n, s)

      #b) Vector Arithmetics

a = c(5, 4)
b = c(1, 4)

#we multiply a by 7
7 * b

#add a and b together
a + b

#subtraction, multiplication and division
a - b
a * b
a / b

#Recycling Rule
#If two vectors are of unequal length, the shorter one will be recycled in order to match
the longer vector.
```

```
u = c(100, 200)
v = c(1, 2, 3, 4, 5, 6)
u + v

      #c) Vector index

#We retrieve values in a vector by declaring an index inside a single square bracket "[]"
operator.
s = c("aa", "bb")
s[2]

#Negative Index
#If the index is negative, it would strip the member whose position has the same absolute
value as the negative index.
s[-2]
s[-3]
s[-4]
#Out-of-Range Index
#If an index is out-of-range, a missing value will be reported via the symbol NA.
s[10]

      #d) Numeric Index Vector

#A new vector can be sliced from a given vector with a numeric index vector, which
consists of member positions of the original vector to be retrieved.
s = c("bb","dd", "ee")
s[c(3,1)]

#Duplicate Indexes
#The index vector allows duplicate values. Hence the following retrieves a member twice in
one operation
s[c(1, 2, 3, 2)]

#Out-of-Order Indexes
#The index vector can even be out-of-order. Here is a vector slice with the order of first
and second members reversed.
s[c(2, 1, 3)]
#Range Index
#To produce a vector slice between two indexes, we can use the colon operator ":". This
can be convenient for situations involving large vectors.
s[2:3]

      #e) Named Vector Members

#We can assign names to vector members.
v = c("Chriss", "Jordan")
#We now name the first member as First, and the second as Last.
names(v) = c("First", "Last")
v
#Then we can retrieve the first member by its name.
v["First"]
#Furthermore, we can reverse the order with a character string index vector.
v[c("Last", "First")]


                          #3) Matrix

#A matrix is a collection of data elements arranged in a two-dimensional rectangular
layout.
Z = matrix(
  c(8, 2, 4, 3, 5, 7),
  nrow=2,
  ncol=3,
  byrow = TRUE)
Z
```

```
#An element at the mth row, nth column of A can be accessed by the expression A[m, n].
Z[2,3]
#entire mth row A can be extracted as A[m, ].
Z[2, ]       #second row
#Similarly, the entire nth column A can be extracted as A[ ,n].
Z[ ,3]       # the 3rd column
#We can also extract more than one rows or columns at a time.
Z[ ,c(1,3)]  # the 1st and 3rd columns
#If we assign names to the rows and columns of the matrix, than we can access the elements
by names.
dimnames(Z) = list(
c("row1", "row2"),         # row names
c("col1", "col2", "col3")) # column names
Z                  # print Z
Z["row2", "col3"] # element at 2nd row, 3rd column

            #a)Matrix Construction

#There are various ways to construct a matrix. When we construct a matrix directly with
data elements, the matrix content is filled along the column orientation by default.
B = matrix(
c(8, 2, 4, 3, 5, 7),
nrow=3,
ncol=2)
B

#Transpose
#We construct the transpose of a matrix by interchanging its columns and rows with the
function t .
t (B)

#Combining Matrices
#The columns of two matrices having the same number of rows can be combined into a larger
matrix.
C = matrix(
c(10, 2, 7),
nrow=3,
ncol=1)
C
#Then we can combine the columns of B and C with cbind.
cbind(B, C)
```

Matrix Construction

There are various ways to construct a matrix. When we construct a matrix directly with data elements, the matrix content is filled along the column orientation by default. For example, in the following code snippet, the content of B is filled along the columns consecutively.

```
> B = matrix(
  +   c(2, 4, 3, 1, 5, 7),
  +   nrow=3,
  +   ncol=2)

> B             # B has 3 rows and 2 columns
[,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
```

Transpose

We construct the transpose of a matrix by interchanging its columns and rows with the function t .

```
> t(B)          # transpose of B
[,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
```

Combining Matrices
The columns of two matrices having the same number of rows can be combined into a larger matrix. For example, suppose we have another matrix C also with 3 rows.

```
> C = matrix(
  +   c(7, 4, 2),
  +   nrow=3,
  +   ncol=1)

> C             # C has 3 rows
[,1]
[1,]    7
[2,]    4
[3,]    2
```

Then we can combine the columns of B and C with cbind.

```
> cbind(B, C)
[,1] [,2] [,3]
[1,]    2    1    7
[2,]    4    5    4
[3,]    3    7    2
#Similarly, we can combine the rows of two matrices if they have the same number of
columns with the rbind function.
A = matrix(
c(6, 2),
nrow=1,
ncol=2)

A
rbind(B, A)
#Deconstruction
#We can deconstruct a matrix by applying the c function, which combines all column vectors
into one.
c(B)


                              #4) List

#A list is a generic vector containing other objects.
n = c(4, 5, 6)
e = c("hope", "greatness", "limitless")
w = c(TRUE, TRUE, FALSE, FALSE)
x = list(n, e, w, 2)
#List Slicing
#We retrieve a list slice with the single square bracket "[]" operator. The following is a
slice containing the second member of x, which is a copy of s.
x[2]
#With an index vector, we can retrieve a slice with multiple members. Here a slice
containing the second and fourth members of x.
x[c(2, 4)]
#Member Reference
#In order to reference a list member directly, we have to use the double square bracket "
[[]]" operator. The following object x[[2]] is the second member of x. In other words,
x[[2]] is a copy of s, but is not a slice containing s or its copy.
x[[2]]
#We can modify its content directly.
x[[2]][1] = "you"
x[[2]]
e #is unaffected

            #a) Named List Members
#We can assign names to list members, and reference them by names instead of numeric
indexes.
v = list(Chriss=c(2, 3, 5), Jordan=c("aa", "bb"))
v
```

```
#List Slicing
#We retrieve a list slice with the single square bracket "[]" operator. Here is a list
slice containing a member of v named "bob".
v["Chriss"]
#With an index vector, we can retrieve a slice with multiple members. Here is a list slice
with both members of v. Notice how they are reversed from their original positions in v.
v[c("Jordan", "Chriss")]
#Member Reference
#In order to reference a list member directly, we have to use the double square bracket "
[[]]" operator. The following references a member of v by name.
v[["Chriss"]]
#A named list member can also be referenced directly with the "$" operator in lieu of the
double square bracket operator.
v$Chriss
#Search Path Attachment
#We can attach a list to the R search path and access its members without explicitly
mentioning the list. It should to be detached for cleanup.
attach(v)
Chriss
detach(v)

                            #5) Data Frame
#A data frame is used for storing data tables. It is a list of vectors of equal length.
For example, the following variable df is a data frame containing three vectors n, s, b.
n = c(2, 3, 5)
s = c("aa", "bb", "cc")
b = c(TRUE, FALSE, TRUE)
df = data.frame(n, s, b)       # df is a data frame


            #a) Data Frane Column Vector

#A data frame is used for storing data tables. It is a list of vectors of equal length.
n = c(200, 250, 275)
s = c("Mustang", "Corvette", "Ferrari")
b = c(TRUE, FALSE, TRUE)
df = data.frame(n, s, b)       # df is a data frame

#Built-in Data Frame
#We use built-in data frames in R for our tutorials. For example, here is a built-in data
frame in R, called mtcars.
mtcars
#the cell value from the first row, second column of mtcars
mtcars[1, 2]
mtcars[1, 4]

#Moreover, we can use the row and column names instead of the numeric coordinates.
mtcars["Mazda RX4", "cyl"]
#Lastly, the number of data rows in the data frame is given by the nrow function.
nrow(mtcars)
#And the number of columns of a data frame is given by the ncol function.
ncol(mtcars)
#Preview
#Instead of printing out the entire data frame, it is often desirable to preview it with
the head function beforehand.
head(mtcars)

            #a) Data Frame Column Vector

#We reference a data frame column with the double square bracket "[[]]" operator.
mtcars[[9]]
#We can retrieve the same column vector by its name.
mtcars[["am"]]
#We can also retrieve with the "$" operator in lieu of the double square bracket operator.
mtcars$am
```

```
#Yet another way to retrieve the same column vector is to use the single square bracket "
[]" operator. We prepend the column name with a comma character, which signals a wildcard
match for the row position.
mtcars[,"am"]

            #b) Data Frame Column Slice

#We retrieve a data frame column slice with the single square bracket "[]" operator.
#Numeric Indexing
#The following is a slice containing the first column of the built-in data set mtcars.
mtcars[1]
#Name Indexing
#We can retrieve the same column slice by its name.
mtcars["mpg"]
#To retrieve a data frame slice with the two columns mpg and hp, we pack the column names
in an index vector inside the single square bracket operator.
mtcars[c("mpg", "hp", "gear")]

            #c) Data Frame Row Slice

#We retrieve rows from a data frame with the single square bracket operator, just like
what we did with columns. However, in additional to an index vector of row positions, we
append an extra comma character. This is important, as the extra comma signals a wildcard
match for the second coordinate for column positions.
#Numeric Indexing
mtcars[30,]
#To retrieve more than one rows, we use a numeric index vector.
mtcars[c(2, 8),]
#Name Indexing
#We can retrieve a row by its name.
mtcars["Ferrari Dino",]
#And we can pack the row names in an index vector in order to retrieve multiple rows.
mtcars[c("Ferrari Dino", "Camaro Z28"),]
#Logical Indexing
#Lastly, we can retrieve rows with a logical index vector. In the following vector L, the
member value is TRUE if the car has automatic transmission, and FALSE if otherwise.
L = mtcars$am == 0
L
#Here is the list of vehicles with automatic transmission.
mtcars[L,]
#And here is the gas mileage data for automatic transmission.
mtcars[L,]$mpg

            #d) Data Import

#It is often necessary to import sample textbook data into R before you start working on
your homework.
#Excel File
#Quite frequently, the sample data is in Excel format, and needs to be imported into R
prior to use. For this, we can use the function read.xls from the gdata package. It reads
from an Excel spreadsheet and returns a data frame. The following shows how to load an
Excel spreadsheet named "mydata.xls". This method requires Perl runtime to be present in
the system.
library(gdata)                  # load gdata package
help(read.xls)                  # documentation
mydata = read.xls("mydata.xls")  # read from first sheet
#Alternatively, we can use the function loadWorkbook from the XLConnect package to read
the entire workbook, and then load the worksheets with readWorksheet. The XLConnect
package requires Java to be pre-installed.
library(XLConnect)              # load XLConnect package
wk = loadWorkbook("mydata.xls")
df = readWorksheet(wk, sheet="Sheet1")
#Minitab File
#If the data file is in Minitab Portable Worksheet format, it can be opened with the
function read.mtp from the foreign package. It returns a list of components in the Minitab
```

```
worksheet.
library(foreign)                # load the foreign package
help(read.mtp)                  # documentation
mydata = read.mtp("mydata.mtp")  # read from .mtp file
#SPSS File
#For the data files in SPSS format, it can be opened with the function read.spss also from
the foreign package. There is a "to.data.frame" option for choosing whether a data frame
is to be returned. By default, it returns a list of components instead.
library(foreign)                # load the foreign package
help(read.spss)                 # documentation
mydata = read.spss("myfile", to.data.frame=TRUE)
#Table File
#A data table can resides in a text file. The cells inside the table are separated by
blank characters. Here is an example of a table with 4 rows and 3 columns.


#Now copy and paste the table above in a file named "mydata.txt" with a text editor. Then
load the data into the workspace with the function read.table.
mydata = read.table("mydata.txt")  # read text file
mydata                             # print data frame

#CSV File
#The sample data can also be in comma separated values (CSV) format. Each cell inside such
data file is separated by a special character, which usually is a comma, although other
characters can be used as well.
#The first row of the data file should contain the column names inst

#After we copy and paste the data above in a file named "mydata.csv" with a text editor,
we can read the data with the function read.csv.
mydata = read.csv("mydata.csv")  # read csv file
mydata

#Working Directory
#Finally, the code samples above assume the data files are located in the R working
directory, which can be found with the function getwd.
getwd()               # get current working directory
#You can select a different working directory with the function setwd(), and thus avoid
entering the full path of the data files.
setwd("<new path>")   # set working directory
#Note that the forward slash should be used as the path separator even on Windows
platform.
setwd("C:/MyDoc")
```