Data Analysis and Visualization Exercise 1

Shabnam Sadegh, Leonhard Wachutka, Elaine Zosa, Vicente Yepez

28 October 2019

Abstract

Basic R data structures

1 Vectors

1.1 Question

First, create three named numeric vectors of size 10, 11 and 12 respectively in the following manner:

- One vector with the "colon" approach: *from:to*
- One vector with the `seq()` function: *seq(from, to)*
- And one vector with the `seq()` function and the `by` argument: *seq(from, to, by)*

For easier naming you can use the vectors `letters` or `LETTERS` which contain the latin alphabet in lowercase and capital, respectively. In order to select specific letters use e.g. `letters[1:4]` to get the first four letters. Check their types. What is the outcome? Where do you think the difference between these three comes from?

Then combine all three vectors in a list called 'myList'. Check the attributes of the vectors and the list. What is the difference and why?

Hint: If the elements of a list have no names, we can access them with the double brackets and an index, e.g. `myList[[1]]`

```r
# Answer:

# A. Create vectors
aa <- 1:10
names(aa) <- letters[aa]
bb <- seq(1, 11)
names(bb) <- letters[bb]
cc <- seq(1, 12, by = 1)
names(cc) <- letters[cc]

typeof(aa)
## [1] "integer"
typeof(bb)
## [1] "integer"
typeof(cc)
```

```
## [1] "double"
# Because the 'by' argument can take any number, it returns a float
# If we want it to return an integer, use 'by = 1L'

# B. Combine in a list
myList <- list(aa, bb, cc)
attributes(aa)
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
attributes(bb)
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
attributes(cc)
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
attributes(myList)
## NULL

# The list itself doesn't have names. Each of its elements have names:
myList[[1]]
##  a  b  c  d  e  f  g  h  i  j
##  1  2  3  4  5  6  7  8  9 10
attributes(myList[[1]])
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

1.2 Question:

Assume you have a lookup table as `lookup <- c(a = "sun", b = "rain", c = "wind", u = NA)`. Generate the weekly weather predictions `c("sun","sun","rain", NA, "rain", "rain", "wind")` out of this lookup table.

```
# Answer:
lookup <- c(a = "sun", b = "rain", c = "wind", u = NA)
x<- c("a", "a", "b", "u", "b", "b", "c")
unname(lookup[x])
## [1] "sun"  "sun"  "rain" NA     "rain" "rain" "wind"
```

2 Factors

2.1 Question:

What is the difference between a factor and a vector?

```
# A factor is a vector that can contain only predefined values,
# and is used to store categorical data. It is stored as an integer with
# a character string associated with each integer value
```

2.2 Question

Create a factor vector of length 30 with the three levels *Rita Repulsa*, *Lord Zedd* and *Rito Revolto* and equal length for each level. What happens if you replace the second element of the vector with *Shredder*? *Hint*: `?gl`

```
x <- gl(n = 3, k = 10, labels = c("Rita Repulsa", "Lord Zedd", "Rito Revolto"))
x
##  [1] Rita Repulsa Rita Repulsa Rita Repulsa Rita Repulsa Rita Repulsa
##  [6] Rita Repulsa Rita Repulsa Rita Repulsa Rita Repulsa Rita Repulsa
## [11] Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd
## [16] Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd
## [21] Rito Revolto Rito Revolto Rito Revolto Rito Revolto Rito Revolto
## [26] Rito Revolto Rito Revolto Rito Revolto Rito Revolto Rito Revolto
## Levels: Rita Repulsa Lord Zedd Rito Revolto

# It doesn't work, as we can only use one of the predefined labels.
x[2] <- "Shredder"
x
##  [1] Rita Repulsa <NA>         Rita Repulsa Rita Repulsa Rita Repulsa
##  [6] Rita Repulsa Rita Repulsa Rita Repulsa Rita Repulsa Rita Repulsa
## [11] Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd
## [16] Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd    Lord Zedd
## [21] Rito Revolto Rito Revolto Rito Revolto Rito Revolto Rito Revolto
## [26] Rito Revolto Rito Revolto Rito Revolto Rito Revolto Rito Revolto
## Levels: Rita Repulsa Lord Zedd Rito Revolto
```

2.3 Question:

Create the following 3 factor vectors f1, f2 and f3:

```
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

The function `rev` reverses the order of an order-able object. What is the difference between f1, f2 and f3? Why?

```
# Answer:

f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
# f1 goes from z - a, but the underlying encoding goes from z = 1 to a = 26
# We create the vector with the letters a to z and the mapped integer
# structure 1 to 26. THEN we reverse the levels = the mapping. As 'z' becomes 1
# and 'a' becomes 26 the letters are mapped back to the unchanged integer
# structure and hence reversed.
f1
##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a

f2 <- rev(factor(letters))
# f2 goes from z - a, but the underlying encoding goes from a = 1 to z = 26
# We create the vector with the letters a to z and the mapped integer
# structure 1 to 26. Then we reverse the vector, i.e. the underlying integers,
# hence the vector gets reversed, but not the levels.
f2
##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z

f3 <- factor(letters, levels = rev(letters))
# f3 goes from a - z, but the underlying encoding goes from z = 1 to a = 26.
# We create the vector with the letters a to z BUT the mapped integer
# structure 26 to 1. Hence the levels but not the vector are reversed.
f3
##  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a

# Reversing f3 will give f1
rev(f3)
##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
```

3 Matrices

3.1 Question :

Create a 3 by 4 matrix that contains the numbers 1 to 12 and then convert it into a data frame.

```
# Answer:
x <- matrix(1:12, 3, 4)
x
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
x <- as.data.frame(x)
x
##   V1 V2 V3 V4
## 1  1  4  7 10
## 2  2  5  8 11
## 3  3  6  9 12
```

3.2 Question:

Create a 10 by 5 matrix which contains the numbers from 1 to 50 column-wisely. Name the rows as 'row_n' and columns as 'col_n' where n is the row or column number. Compute the mean and sum of each row and column. Add vector seq(60,100,10) as another row to the matrix.

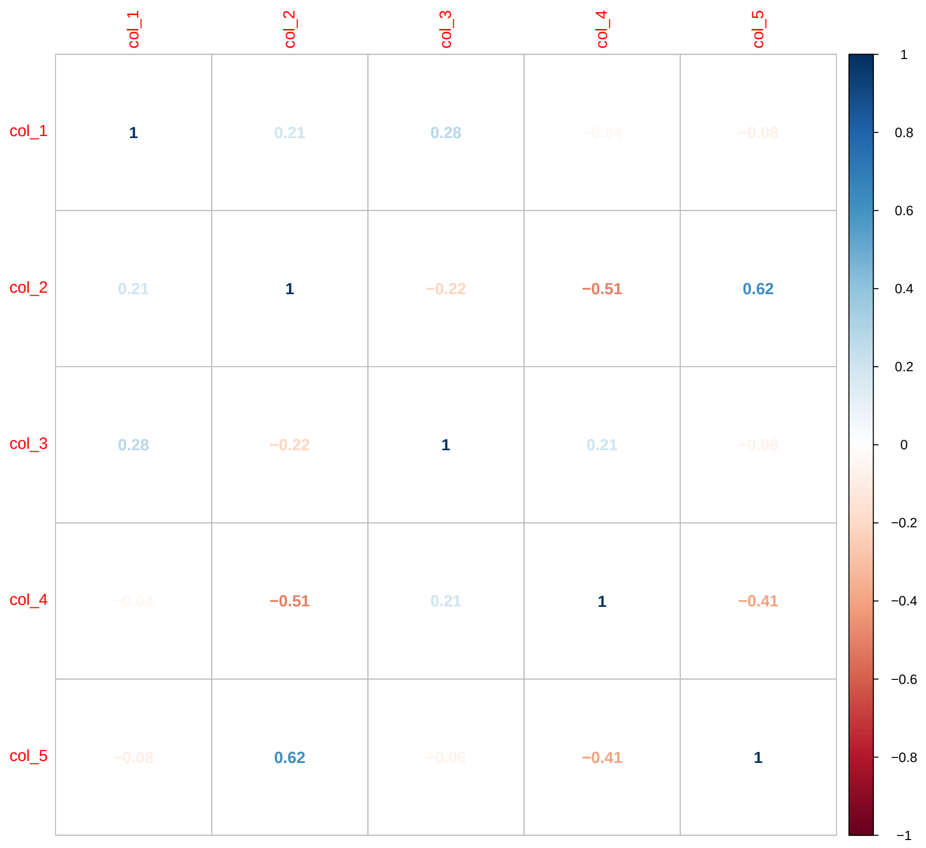
Generate another matrix with the same dimensions, containing random numbers between 1 and 100. Subtract this matrix from the first one.

Compute the matrix columns' correlation using the spearman method. Then, plot the resulting correlation matrix.

Hint: Check the functions paste0(), colMeans(), rowMeans(), colSums(), rowSums(), sample(), cor() and corrplot() (in package 'corrplot')

```
# Answer:
m <- matrix(1:50, ncol = 5)
rownames(m) <- paste0("row_", 1:nrow(m))
colnames(m) <- paste0("col_", 1:ncol(m))
colMeans(m)
## col_1 col_2 col_3 col_4 col_5
##   5.5  15.5  25.5  35.5  45.5
rowMeans(m)
##  row_1  row_2  row_3  row_4  row_5  row_6  row_7  row_8  row_9 row_10
##     21     22     23     24     25     26     27     28     29     30
colSums(m)
## col_1 col_2 col_3 col_4 col_5
##    55   155   255   355   455
rowSums(m)
##  row_1  row_2  row_3  row_4  row_5  row_6  row_7  row_8  row_9 row_10
##    105    110    115    120    125    130    135    140    145    150

m <- rbind(m,seq(60,100,10))
n <- matrix(sample(1:100, length(m), replace=T), ncol = ncol(m))
m <- m-n
M <- cor(m, method = "spearman")
corrplot::corrplot(M, method = "number")
```



4 Lists and data.frames

4.1 From lists to data.frames

Take myList from the first question. Coerce it to a `data.frame` with `as.data.frame()`. Why does it fail and how can we fix it? What happened to the names?

```
# Answer:
# Coerce to data.frame
# df <- as.data.frame(myList) ## fails
# The lengths of the elements of the list are not the same,
# so we need to fix that by inserting NAs
myList[[1]] <- c(myList[[1]], NA, NA)
myList[[2]] <- c(myList[[2]], NA)

df <- as.data.frame(myList)

# Add column names
names(df) <- LETTERS[1:3]
df
##    A  B  C
## a  1  1  1
## b  2  2  2
## c  3  3  3
## d  4  4  4
## e  5  5  5
## f  6  6  6
## g  7  7  7
## h  8  8  8
## i  9  9  9
## j 10 10 10
## k NA 11 11
##   NA NA 12
```

4.2 Creating data.frames

Create a data.frame with 26 rows like the one shown below. Only the first six and the last six rows are displayed.

Hint: Instead of the workaround with `list` you can also use simply `data.frame(column_name = column_vector, ...)`

```
# Head: first 6 lines
head(df)
##   V1 V2 V3
## 1  1  4  a
## 2  2  8  a
## 3  3 12  c
## 4  4 16  c
## 5  5 20  e
## 6  6 24  e
# Tail: last 6 lines
tail(df)
##    V1  V2 V3
## 21 21  84  u
## 22 22  88  u
## 23 23  92  w
## 24 24  96  w
## 25 25 100  y
## 26 26 104  y
```

```
# Answer:
aa <- seq(1,26)
bb <- seq(4, 104, by = 4)
cc <- rep(seq(1, 26, 2), each = 2)
df <- data.frame(V1 = aa, V2 = bb, V3 = letters[cc])
df
##    V1  V2 V3
## 1   1   4  a
## 2   2   8  a
## 3   3  12  c
## 4   4  16  c
## 5   5  20  e
## 6   6  24  e
## 7   7  28  g
## 8   8  32  g
## 9   9  36  i
## 10 10  40  i
## 11 11  44  k
## 12 12  48  k
## 13 13  52  m
## 14 14  56  m
## 15 15  60  o
## 16 16  64  o
## 17 17  68  q
## 18 18  72  q
## 19 19  76  s
## 20 20  80  s
## 21 21  84  u
## 22 22  88  u
## 23 23  92  w
## 24 24  96  w
## 25 25 100  y
## 26 26 104  y
```

4.3 Attributes

Take again the `data.frame` from the previous question and make a copy of it.

- Change the row names and the column names of the data.frame to capital letters (or small letters, if they are already capital).
- Change the `class` attribute to *list*. What happens?
- Change it now to any name you like. What happens now? What happens if you remove the `class` attribute?

```
# Answer:

df2 <- df

# A
# One possible way through attributes
attributes(df2) ## first check attributes
## $names
## [1] "V1" "V2" "V3"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [25] 25 26
```

```
attr(df2, "names") <- letters[1:3]
attr(df2, "row.names") <- LETTERS[1:26]

df2
##    a   b c
## A  1   4 a
## B  2   8 a
## C  3  12 c
## D  4  16 c
## E  5  20 e
## F  6  24 e
## G  7  28 g
## H  8  32 g
## I  9  36 i
## J 10  40 i
## K 11  44 k
## L 12  48 k
## M 13  52 m
## N 14  56 m
## O 15  60 o
## P 16  64 o
## Q 17  68 q
## R 18  72 q
## S 19  76 s
## T 20  80 s
## U 21  84 u
## V 22  88 u
## W 23  92 w
## X 24  96 w
## Y 25 100 y
## Z 26 104 y

# Or through accessor functions

names(df2) <- LETTERS[1:3]
rownames(df2) <- letters[1:26]
df2
##    A   B C
## a  1   4 a
## b  2   8 a
## c  3  12 c
## d  4  16 c
## e  5  20 e
## f  6  24 e
## g  7  28 g
## h  8  32 g
## i  9  36 i
## j 10  40 i
## k 11  44 k
## l 12  48 k
## m 13  52 m
## n 14  56 m
## o 15  60 o
## p 16  64 o
## q 17  68 q
## r 18  72 q
## s 19  76 s
## t 20  80 s
## u 21  84 u
## v 22  88 u
## w 23  92 w
## x 24  96 w
## y 25 100 y
## z 26 104 y

# B
# the data.frame is coerced to a list
attr(df2, "class") <- "list"
df2
## $A
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [25] 25 26
##
## $B
##  [1]   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
## [19]  76  80  84  88  92  96 100 104
##
## $C
##  [1] a a c c e e g g i i k k m m o o q q s s u u w w y y
## Levels: a c e g i k m o q s u w y
##
## attr(,"class")
## [1] "list"
## attr(,"row.names")
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
## [19] "s" "t" "u" "v" "w" "x" "y" "z"

# C
# nothing happens
attr(df2, "class") <- "Power Rangers"
df2
## $A
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [25] 25 26
##
## $B
##  [1]   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72
## [19]  76  80  84  88  92  96 100 104
##
## $C
##  [1] a a c c e e g g i i k k m m o o q q s s u u w w y y
## Levels: a c e g i k m o q s u w y
##
## attr(,"class")
## [1] "Power Rangers"
## attr(,"row.names")
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
## [19] "s" "t" "u" "v" "w" "x" "y" "z"

# 'class' is a generic mechanism for a simple object-oriented style of
# programming in R. In this course we will never really need it, as all
# higher objects in R provide accessor functions to extract information
# from them. However from the exercise you can already see that data.frame
# is a class which is build on list. Whereas list itself is not a class.
```

4.4 Combining data.frames

Now take the previous data.frame and reproduce the following data.frame. Only the first and the last six rows are shown.

Hint: In order to combine to data.frames by column you can use `cbind(df1, df2, ...)`

```
# Head: first 6 lines
head(df_df2)
##   V4    V5 V2 V3
## 1 25 0.000  4  a
## 2 25 0.064  8  a
## 3 23 0.128 12  c
## 4 23 0.192 16  c
## 5 21 0.256 20  e
## 6 21 0.320 24  e
# Tail: last 6 lines
tail(df_df2)
##    V4    V5  V2 V3
## 21  5 1.280  84  u
## 22  5 1.344  88  u
## 23  3 1.408  92  w
## 24  3 1.472  96  w
## 25  1 1.536 100  y
## 26  1 1.600 104  y
```

```
# Answer:
df[,1] <- NULL
dd <- rev(rep(seq(1, 26, 2), each = 2))
ee <- seq(0,1.6,length.out = 26)
df2 <- data.frame(V4 = dd, V5 = ee)
df_df2 <- cbind(df2, df)
df_df2
##    V4    V5 V3
## 1  25 0.000  a
## 2  25 0.064  a
## 3   23 0.128  c
## 4   23 0.192  c
## 5   21 0.256  e
## 6   21 0.320  e
## 7   19 0.384  g
## 8   19 0.448  g
## 9   17 0.512  i
## 10  17 0.576  i
## 11  15 0.640  k
## 12  15 0.704  k
## 13  13 0.768  m
## 14  13 0.832  m
## 15  11 0.896  o
## 16  11 0.960  o
## 17   9 1.024  q
## 18   9 1.088  q
## 19   7 1.152  s
## 20   7 1.216  s
## 21   5 1.280  u
## 22   5 1.344  u
## 23   3 1.408  w
## 24   3 1.472  w
## 25   1 1.536  y
## 26   1 1.600  y
```

4.5 Operations on data.frames

Create the data.frame *df* using `df <- as.data.frame(matrix(runif(9e6), 3e3, 3e3))`

This will create a data.frame with 3000 columns and rows and a total of 9 million values.

Now compute the sum of any row, then compute the sum of any column. Measure the time for both operations. Why are the times different although the size is the same?

- **Hint 1:** The time is measured with the function `system.time(my_function_call)`
- **Hint 2:** The sum can be computed with the sum function `sum(my_vector)`
- **Hint 3:** Columns and rows are selected by single brackets. Rows: `df[row_number, ]`, Columns: `df[, column_number]`

```
# Answer:
# Create a data.frame
df <- as.data.frame(matrix(runif(9e6), 3e3, 3e3))

# Compare runtime
# Rows
system.time(res <- sum(df[1,]))
##    user  system elapsed
##   0.056   0.000   0.056
res
## [1] 1509.093
```

```
# Columns
system.time(res <- sum(df[,1]))
##    user  system elapsed
##       0       0       0
res
## [1] 1482.014

# Look at the structure of the objects over which we are computing the sum
# Column
str(df[,1])
##  num [1:3000] 0.3137 0.6046 0.4493 0.0106 0.522 ...

# Row
str(df[1,])
## 'data.frame':    1 obs. of  3000 variables:
##  $ V1   : num 0.314
##  $ V2   : num 0.185
##  $ V3   : num 0.999
##  $ V4   : num 0.841
##  $ V5   : num 0.881
##  $ V6   : num 0.682
##  $ V7   : num 0.555
##  $ V8   : num 0.882
##  $ V9   : num 0.965
##  $ V10  : num 0.458
##  $ V11  : num 0.479
##  $ V12  : num 0.251
##  $ V13  : num 0.688
##  $ V14  : num 0.362
##  $ V15  : num 0.172
##  $ V16  : num 0.942
##  $ V17  : num 0.265
##  $ V18  : num 0.822
##  $ V19  : num 0.209
##  $ V20  : num 0.149
##  $ V21  : num 0.664
##  $ V22  : num 0.645
##  $ V23  : num 0.89
##  $ V24  : num 0.597
##  $ V25  : num 0.312
##  $ V26  : num 0.646
##  $ V27  : num 0.234
##  $ V28  : num 0.246
##  $ V29  : num 0.672
##  $ V30  : num 0.565
##  $ V31  : num 0.0489
##  $ V32  : num 0.787
##  $ V33  : num 0.757
##  $ V34  : num 0.126
##  $ V35  : num 0.373
##  $ V36  : num 0.405
##  $ V37  : num 0.934
##  $ V38  : num 0.00877
##  $ V39  : num 0.618
##  $ V40  : num 0.494
##  $ V41  : num 0.545
##  $ V42  : num 0.054
##  $ V43  : num 0.118
##  $ V44  : num 0.13
##  $ V45  : num 0.758
##  $ V46  : num 0.378
##  $ V47  : num 0.563
##  $ V48  : num 0.0907
##  $ V49  : num 0.617
##  $ V50  : num 0.0808
##  $ V51  : num 0.454
##  $ V52  : num 0.844
##  $ V53  : num 0.105
##  $ V54  : num 0.22
##  $ V55  : num 0.859
##  $ V56  : num 0.662
##  $ V57  : num 0.88
##  $ V58  : num 0.344
##  $ V59  : num 0.936
##  $ V60  : num 0.734
##  $ V61  : num 0.672
##  $ V62  : num 0.871
##  $ V63  : num 0.395
##  $ V64  : num 0.875
##  $ V65  : num 0.932
##  $ V66  : num 0.0739
##  $ V67  : num 0.118
##  $ V68  : num 0.488
##  $ V69  : num 0.0648
##  $ V70  : num 0.2
##  $ V71  : num 0.174
##  $ V72  : num 0.705
##  $ V73  : num 0.649
##  $ V74  : num 0.269
##  $ V75  : num 0.936
##  $ V76  : num 0.972
##  $ V77  : num 0.151
##  $ V78  : num 0.839
##  $ V79  : num 0.574
##  $ V80  : num 0.254
##  $ V81  : num 0.668
##  $ V82  : num 0.624
##  $ V83  : num 0.253
##  $ V84  : num 0.66
##  $ V85  : num 0.965
##  $ V86  : num 0.71
##  $ V87  : num 0.613
##  $ V88  : num 0.515
##  $ V89  : num 0.972
##  $ V90  : num 0.574
##  $ V91  : num 0.0883
##  $ V92  : num 0.314
##  $ V93  : num 0.761
##  $ V94  : num 0.819
##  $ V95  : num 0.291
##  $ V96  : num 0.476
##  $ V97  : num 0.901
##  $ V98  : num 0.511
##  $ V99  : num 0.955
##   [list output truncated]

# As we can see the extracted column is a numeric vector. But the extracted
# row is a list. Under the hood the sum function is iterating in C/Fortran
# over the specific structure. Iterating over a native array of doubles is
# faster, than iterating over a structure, where at each position, the value
# has to be retrieved from an object possibly strored somewhere further away
# in memory.
```

5 Subsetting

5.1 Question:

Use the data.frame you created from the 3 by 4 matrix in the earlier question (Question 3.1) for the next three questions.

1. Select the elements on the second row and the second and fourth columns.

```
# Answer:
x <- data.frame(matrix(1:12,3,4))
x[2, c(2,4)]
##   X2 X4
## 2  5 11
```

2. Set the rownames to “row1”, “row2”, “row3” and column names to “col1”, “col2”, “col3” and “col4”. (Hint: use the function “paste0”)

```
# Answer:
x <- data.frame(matrix(1:12,3,4))
rownames(x) <- paste0("row", 1:3)
colnames(x) <- paste0("col", 1:4)
```

3. Assign 0 to all the elements in columns “col3” and “col4”.

```
# Answer:
x <- data.frame(matrix(1:12,3,4))
```

```
colnames(x) <- paste0("col",1:4)
x[,paste0("col",3:4)] <- 0
```

5.2 Question:

Considering x <- c("a"=1, "b"=2, "c"=3, "d"=4, "e"=5), select the third and fifth elements of x by using positive integers, negative integers, a logical vector, and a character vector.

```
# Answer:
x <- c("a"=1, "b"=2, "c"=3, "d"=4, "e"=5)
x[c(3,5)]
## c e
## 3 5
x[-c(1,2,4)]
## c e
## 3 5
x[c(F,F,T,F,T)]
## c e
## 3 5
x[c("c","e")]
## c e
## 3 5
```

5.3 Question:

Why are vals[c(2, 5)] and vals[2, 5] different where `vals <- outer(1:5, 1:5, FUN = "/")`?

```
# Answer:

# Because when you subset matrix with a vector, the 2d matrix behaves
# like a vector and vals[c(2, 5)] returns the elements at indices 2
# and 5 in column-major order. vals[2, 5] returns the element at row 2, column 5.

vals <- outer(1:5, 1:5, FUN = "/")
vals[c(2, 5)]
## [1] 2 5
vals[2, 5]
## [1] 0.4
```

5.4 Question:

Assume `x <- matrix(1:20, ncol=2)`. What is the difference between `x[1, , drop = T]` and `x[1, , drop = F]`? Now let `y <- as.data.frame(x)`. What is the difference between `y[,1]` , `y[[1]]` and `y[1]`?

```
# Answer:

# When drop=T the output is simplified to a vector when
# a single row or column is subsetted. However when drop=F, the
# the input data structure is preserved at the output.

#Likewise y[,1] and y[[1]] simplifies the output and returns a vector where y[1]
#preserves the input data structure and returns a data frame with one column.
```

5.5 Question:

Now assume the weather in winter lookup table is a data frame as below and we have the predictions for the next week as stored in weeklyCast. How would you create "weeklyTable" by the use of rownames function? How would you create it using the `match` function? How would you order the rows of lookup table by 'desc' column?

```
lookup <- data.frame(
  averageTemperature = c(5, 7, 10, 0, 3),
  desc = c("cloudy", "rainy", "sunny", "snowy", "windy"),
  goodForSki = c(T, F, T, F, F)
)

weeklyCast <- c("rainy", "rainy", "cloudy", "windy", "snowy", "cloudy", "sunny")

weeklyTable <- data.frame(averageTemperature=c(7,7,5,3,0,5,10),
                          desc=c("rainy", "rainy", "cloudy", "windy", "snowy", "cloudy", "sunny"),
                          goodForSki=c(F,F,T,F,F,T,T))
```

```
# Answer:

lookup <- data.frame(
  averageTemperature = c(5, 7, 10, 0, 3),
  desc = c("cloudy", "rainy", "sunny", "snowy", "windy"),
  goodForSki = c(T, F, T, F, F)
)

weeklyCast <- c("rainy", "rainy", "cloudy", "windy", "snowy", "cloudy", "sunny")

#alternative 1
weeklyTable <- data.frame(averageTemperature=c(7,7,5,3,0,5,10),
                          desc=c("rainy", "rainy", "cloudy", "windy", "snowy", "cloudy", "sunny"),
                          goodForSki=c(F,F,T,F,F,T,T))
#alternative 2
rownames(lookup) <- lookup$desc
lookup[weeklyCast,]
##          averageTemperature   desc goodForSki
## rainy                     7  rainy      FALSE
## rainy.1                   7  rainy      FALSE
## cloudy                    5 cloudy       TRUE
## windy                      3  windy      FALSE
## snowy                      0  snowy      FALSE
## cloudy.1                   5 cloudy       TRUE
## sunny                     10  sunny       TRUE

id <- match(weeklyCast, lookup$desc)
lookup[id,]
##            averageTemperature   desc goodForSki
## rainy                      7  rainy      FALSE
## rainy.1                    7  rainy      FALSE
## cloudy                     5 cloudy       TRUE
## windy                      3  windy      FALSE
## snowy                      0  snowy      FALSE
## cloudy.1                   5 cloudy       TRUE
## sunny                     10  sunny       TRUE

lookup[order(lookup$desc),]
##          averageTemperature   desc goodForSki
## cloudy                     5 cloudy       TRUE
## rainy                      7  rainy      FALSE
## snowy                      0  snowy      FALSE
## sunny                     10  sunny       TRUE
## windy                      3  windy      FALSE
```

5.6 Question:

For the next questions, consider the bigDF data.frame which has 1,500 columns and rows.

1. Select the even numbered columns named such as “Column_2”, “Column_4”, etc.

```
bigDF <- as.data.frame(matrix(0, ncol=1500, nrow=1500))
colnames(bigDF) <- paste0("Column_", 1:ncol(bigDF))
```

```
# Answer:
selectedCols <- paste0("Column_", seq(2, 1500, 2))
a <- bigDF[selectedCols]
```

2. Select all the columns other than column 76.

```
b <- bigDF[setdiff(names(bigDF), "Column_76")]
```

3. Assign the number 1 to 500 randomly selected diagonal indices.

```
randSample <- sample(1:1500, 500)   # Sample 500 out of the 1500 diagonal elements
bigDF[cbind(randSample,randSample)] <- 1
```

4. Retrieve the row and column indices of the elements which have been assigned a 1. Select the rows where columns Column_1 or Column_2 are 1 using the subset() function.

```
c <- which(bigDF == 1, arr.ind = T)

d <- subset(bigDF, Column_1==1 | Column_2 == 1)
d[,1:10]
##  [1] Column_1  Column_2  Column_3  Column_4  Column_5  Column_6  Column_7
##  [8] Column_8  Column_9  Column_10
## <0 rows> (or 0-length row.names)
```

5.7 (Optional) Data.frames: Titanic

Compute the number of women who survived the Titanic. Start by loading the data into a data.frame using the following command:

```
tab <- read.csv("extdata/titanic.csv")
```

```
tab$isfemale <- grepl("Mrs|Miss",tab$name)
print("total number of women")
## [1] "total number of women"
sum(tab$isfemale)
## [1] 459
print("total number of people survived")
## [1] "total number of people survived"
sum(tab$survived)
## [1] 500
print("total number of women who survived")
## [1] "total number of women who survived"
sum(tab[tab$survived==1,"isfemale"])
## [1] 333
```