Exercise 3 - R Expressions and Data Structures

Now that we understand the basic R environment let's create some data structures and use R expressions with the data. In this session of exercise, the following things are implemented in R

- Create R data structures using different data types.
- Perform descriptive statistics on data within R
- Create, access, and modify data in vectors, lists, and matrices.
- Use various R expressions and functions on data (eg. mean, length)
- Convert data between different data types.

Note the text shaded (which you will found as you go through the exercises) with grey are commands.
`This is command`

I. Objects and Expressions

In the R console window, let's create some variables, data structures and use a few common R expressions. Before starting this section ensure you have cleared your R workspace.

```
rm(list = ls())
```

This will remove everything shown by `ls()`

a) Creating and Working with R objects

Creating Scalar objects

The following R functions should be used
`<-, ls(), str(), typeof(), length(), nchar(), as.integer()`

1. Create three R **objects** to hold information. Use the assignment operator (`<-`) to store values in the object.

```
product <- "shirt"
quantity <- 5
price <- 12.45
```

Let's verify that the objects are in memory.

2. List the objects in R and verifying the structure of objects

```
ls()
```

Output: `[1] "price" "product" "quantity"`

```
str(product)
```

Output: `chr "shirt"`

```
str(quantity)
```

Output: `num 5`

```
str(price)
```
Output: `num 12.4`

Note that the default numeric data type is **double**.

```
typeof(price)
```
Output: `[1] "double"`

3. Converting the data type of quantity to integer.
```
quantity <- as.integer(quantity)
```

```
typeof(quantity)
```
Output: `[1] "integer"`

4. Find the number of letters in the product name.

Since product is a single element vector, `length()` returns the number of elements in the vector. You can use the `nchar()` function instead.
```
length(product)
```
Output: `[1] 1`

```
nchar(product)
```
Output: `[1] 5`

Create a logical object

5. The available object represents whether the product is available or not.
```
available <- TRUE
```

```
typeof(available)
```
Output: `[1] "logical"`

b) Working with Vectors

Using **vector** data structures. The following R functions should be used
`c(), mean()`

6. Create a vector of numeric values and checking the structure of the vectors

```
prices <- c(12.45, 10.50, 13.30)
```

```
str(prices)
```

Output: `num [1:3] 12.4 10.5 13.3`

7. Compute the average or mean of the elements in the vector.

   `mean(prices)`
   Output: `[1] 12.08333`

8. Access data stored in vectors using numeric indexing.

   `prices[1]` **# retrieve the first value**
   Output: `[1] 12.45`

   `prices[2:3]` **# range of indexes 2 through 3**
   Output: `[1] 10.5 13.3`

   `prices[-2]` **# all items except the second**
   Output: `[1] 12.45 13.30`

   `prices[length(prices))` **# retrieve the last value**
   Output: `[1] 13.3`

9. Access data stored in a vector using logical indexing. and print out the products that are priced over $12.

   `print ( prices[ prices>12] )`
   Output: `[1] 12.45 13.30`

10. Creating a character vector (represents the types of products sold).

    `types <- c("shirt", "sock", "pants")`

11. Creating a integer vector (represents the amount of each product sold).

    `amounts <- as.integer(c(3, 6, 7))`

12. Let's do a bit of math and try to compute the total value of our inventory. Performing vector operations.

    `values <- prices * amounts`

    `print (values)`
    Output: `[1] 37.35 63.00 93.10`

c) Working with Matrices

13. Using matrix data structures

    Now let's look at how a matrix could be used to track sales across multiple stores. In R, a **matrix** is a data structure of values. The data structure has well defined dimensions (number of rows, number of columns) and the values **must be of a single datatype**.

We currently have a vector called ***amounts*** which represents the number of products sold at one store. Let's extend or append values to the vector ***amounts*** to represent the number of products sold in a second store. The product types are defined in the vector called ***types*** (shirts, socks, pants)
Append items to a vector (representing sales in another store)

```
amounts <- append ( amounts, c (4,5,6) )
```

```
amounts
```

Output: `[1] 3 6 7 4 5 6`

Let's assume that the data in the vector represents the following data:
store 1 sold 3 shirts / store 2 sold 6 shirts
store 1 sold 7 socks / store 2 sold 4 socks
store 1 sold 5 pants / store 2 sold 6 pants

We want to create a matrix where each row represents the sales of the products for a given store and each column represents the sales by product.

We will use the `matrix()` function to accomplish this task.
Values for a matrix are populated in column order by default. Basically, R will store the values in column 1 and then continue across the entire data structure.

14. Create a matrix from a vector

```
store.sales <- matrix (amounts, nrow=2, ncol=3)
store.sales
```

Output:
```
[,1] [,2] [,3]
[1,] 3 7 5
[2,] 6 4 6
```

15. Calculate the store sales using matrix multiplication.

```
sales <- store.sales %*% prices
```

Output:
```
[,1]
[1,] 177.35
[2,] 196.50
```

Now we know that the products sales for store 1 was $177.35 and sales for store 2 was $196.50.

You can access matrix elements using numeric indexing. Matrix indexing is always performed in row and column order.

16. Find the number of socks sold by store 2.

```
store.sales[2,2]
```

Output:
```
[1] 4
```

Accessing elements in a matrix using numeric index can be a bit frustrating as the rows and columns represent "real" information. In R, it is possible to name the columns and rows.

17. Naming the columns and rows in a matrix.
    We will assume the first store is in Toronto and the second store is in Paris.

```
dimnames(store.sales) <- list( c("toronto", "paris"), types)
store.sales
```

Output:
```
shirt sock pants
toronto 3 7 5
paris 6 4 6
```

18. How many pants were sold in Paris?
    To know that, we are accessing elements in a matrix using **named indexing** as follows

```
store.sales["paris","pants"]
```

Output: `[1] 6`

19. Let's determine how many total socks were sold. We will use matrix slicing notation and the `sum()` function to accomplish this task.

`sum(store.sales[,2])` **# method using numeric index**

`sum(store.sales[,"sock"])` **# method using column name**

R provides row and column summary statistics functions including: `colSums(), rowSums(), colMeans(), rowMeans()`

```
colSums(store.sales)
```

Output:
```
shirt sock pants
9 11 11
```

```
rowSums(store.sales)
```

Output:
```
toronto paris
```

```
15 16
```

The store in Toronto sold 15 items and Paris sold 16 items.

d) Working with Lists

When you have **mixed data types** you can store them in a single data structure called a **list**.
The lengths of data structures stored in lists can also have different number of elements (**varying lengths**).

20. Create a list of employees and salaries for our stores.

```
employees <- c("Mary", "Bob","Cindy")
salaries <- c( 34000, 45000, 76000)
hr <- list (employees, salaries)
hr
```

Output:

```
[[1]]
[1] "Mary" "Bob" "Cindy"

[[2]]
[1] 34000 45000 76000
```

21. Accessing data in a list.

If you use the **single indexing method []** you are provided with the corresponding list elements(s), except they're wrapped up as lists themselves.
If you use a **double indexing method [[]]** you are provided a vector of values as is.

```
str( hr[1] )
```

Output:

```
List of 1
$ : chr [1:3] "Mary" "Bob" "Cindy"
```

```
str( hr [ [1] ] )
```

Output:

```
chr [1:3] "Mary" "Bob" "Cindy"
```

22. Let's find the mean salary for the employees.
In this example we will use numeric indexing with lists to determine the mean salary of our employees.

```
mean(hr[[2]])
```

Output:

```
[1] 51666.67
```

23. Let's change the names of the vectors stored in our list.

```
names(hr) <- c("emp","sal")
str(hr)
```

Output:
```
$emp
[1] "Mary" "Bob" "Cindy"
$sal
[1] 34000 45000 76000
```

24. Find the highest salary using access by member name.

```
max(hr$sal)
```

Output:
```
[1] 76000
```

e) Working with Data Frames

Data is frequently provided in the form of a 2 dimensional set of values of mixed data types. This type of data structure is frequently called a table of values. In R, this data structure is known as a **data frame**.

25. Create a data frame of students and test scores**.**

```
df1 <- data.frame (students = c("Mary", "Jane", "Eva"), test1 =
c(80,90,70) )
```

or you can use the following command

```
df1<-data.frame(c("Mary", "Jane", "Eva"),c(80,90,70))
names(df1)<-c("students","test1")
```

In this scenario we have named the columns "**students**" and "**test1**".There are row names associated with data frames, but they are often not used for analysis.

```
str(df1)
```

Output:
```
'data.frame': 3 obs. of 2 variables:
$ students: Factor w/ 3 levels "Eva","Jane","Mary": 3 2 1
$ test1 : num 80 90 70
```
In R, column values are referred to as "**variables"** and the number of rows is the number of "**observations**".
R automatically determines the data types. In this scenario, test scores are numeric and student names are Factors.

In R, a factor is considered a categorical or enumerated type. Our student names should not be considered Factors as we would not consider the phrase "Mary" is not a category. If your data included a categorical type of data (eg. gender) then a factor data type would make sense.

26. Recoding a variable. Let's change the students data from Factor to character data type.

```
df1$students <- as.character(df1$students)
```

27. Find the distribution of data for test scores.

```
summary (df1$test1)
```

Output:
```
Min. 1st Qu. Median Mean 3rd Qu. Max.
70 75 80 80 85 90
```

28. Adding a new variable to a data frame

In this example we will add a variable for student birthdates. The `cbind()` function can be used to accomplish this task.

```
df1<- cbind(df1, birth = as.Date(c("1990-08-01", "1995-07-23", "1993-12-13")))
```

R has a built-in class called **Date** that can be used to store date information. The default format is used in this example, but other formats can be provided.

```
str(df1)
```

Output:
```
'data.frame': 3 obs. of 3 variables:
$ students: chr "Mary" "Jane" "Eva"
$ test1 : num 80 90 70
$ birth : Date, format: "1990-08-01" "1995-07-23" "1993-12-13"
```

29. Adding a new observation to a data frame. The `rbind()` function can be used to accomplish this task.

```
df1<-rbind( df1,c("Leon", 95, "1990-08-09") )
```

Output:
```
'data.frame': 3 obs. of 3 variables:
$ students: chr "Mary" "Jane" "Eva" "Leon"
$ test1 : num 80 90 70 95
$ birth : Date, format: "1990-08-01" "1995-07-23" "1993-12-13" "1990-08-09"
```

30. Now it's time to view some of the data. The `head()` function can be used to examine the first 'n' rows of data in our data frame. A similar `tail()` function can be used to examine the last few rows of data. Viewing data in a data frame

```
head(df1)
```

Output:

```
students test1 birth
1 Mary 80 1990-08-01
2 Jane 90 1995-07-23
3 Eva 70 1993-12-13
4 Leon 95 1990-08-09
```

Note the row names that are automatically assigned to each observation.

```
tail(df1,1)
```

Output:

```
students test1 birth
4 Leon 95 1990-08-09
```