Introduction to R

Lan Wen & Katherine Li (July 2020, modified May 2022)

Outline

1. Installing R/RStudio
2. Introduction to R and RStudio
3. Manipulations in R
4. Conditional statements
5. Data set description using data frames
6. Data cleaning
7. Transforming dataset by subsetting and Merging

1. Installing R, RStudio

1.1 To install R

1. Go to `www.r-project.org`

2. Click the "download R" link in the middle of the page under "Getting Started."

3. Select a CRAN location (a mirror site) and click the corresponding link.

4. If you are a Mac user: Click on the "Download R for (Mac) OS X" link at the top of the page, and click the .pkg file containing the latest version of R under "Latest Release". Save the .pkg file, double-click it to open, and follow the installation instructions.

If you are a Windows user: Click on the "Download R for Windows" link, and click on the "install R for the first time" link at the top of the page. Click "Download R for Windows" and save the .exe file somewhere on your computer. Run the .exe file and follow the installation instructions.

5. Now that R is installed, you need to download and install RStudio.

1.2 To install RStudio

1. Go to `www.rstudio.com` and click on the "Download" button on the top.

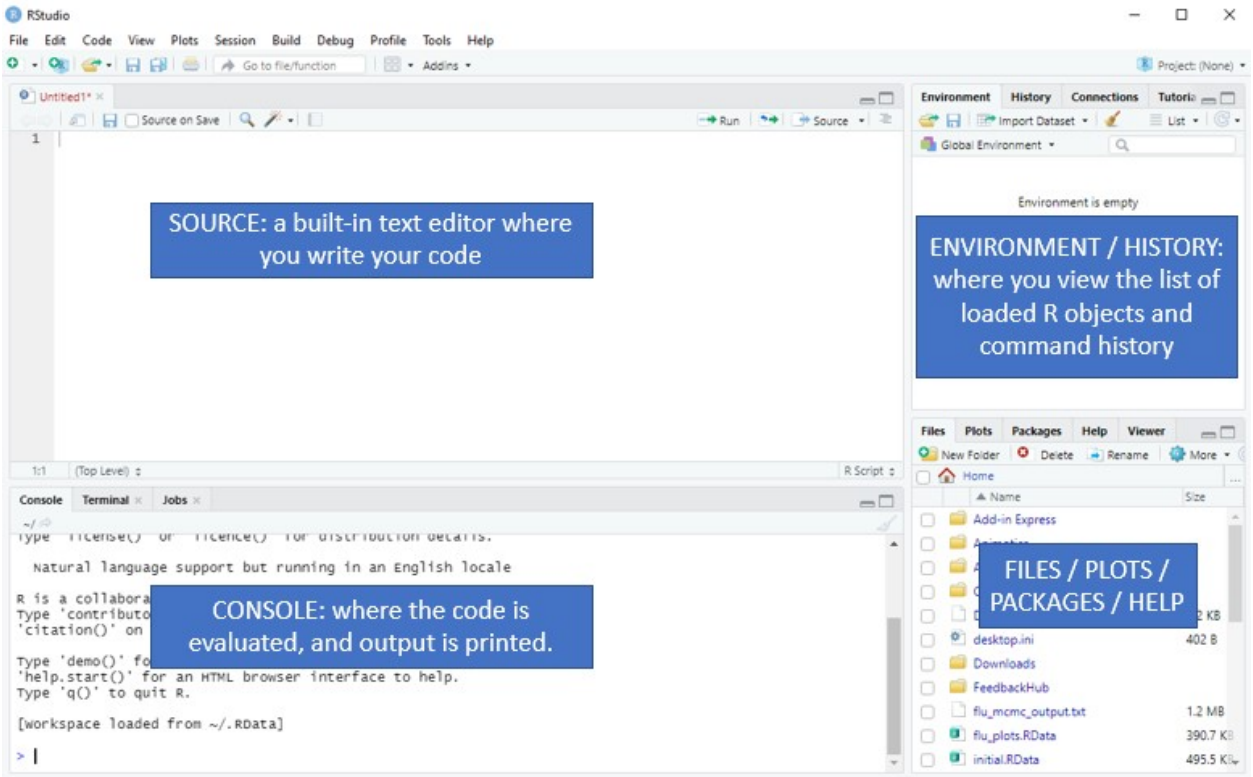
2. Click on the "Download" button under RStudio Desktop.

3. Click on the version recommended for your system, and save the .dmg file (if using Mac OS) or the .exe file (if using Windows OS). Run the file and follow the installation instructions.

2. Introduction to R and RStudio

R is a free, open-source programming language for statistical computing and graphics. RStudio is an integrated development environment (IDE) for R with a more user-friendly interface and powerful programming tools.

When you open RStudio, you will see three main windows in the interface. Click File / New File / R Script to open the fourth window, and the interface will look like this:



3. Some simple manipulations in R

The entities that R creates and manipulates are known as objects. In R, all types of data are treated as objects, with various structures, including vectors, matrices, arrays, lists, and data frames.

Vectors are the most basic data structure in R, which is an single entity that consists of a collection of like elements. Note that R does not have scalar types that hold only one value at a time. Individual numbers or strings that are usually considered scalars, are treated as vectors of length one in R.

The elements could be of classes/types that include integer, double (usually called "numeric" in R), logical values and characters (string). We can assign values to a vector by using the arrow <- symbol (or equivalently with = symbol), and use class() to check the type of data:

```
# Commenting is a good way to keep your code organized.
# Just use a hash symbol # in front of the text that you want to comment out.
# R will not run any commented text.

> vec_double <- c(1, 2.3, 4.56)
#By using "<-", we created the vector but it does not provide an output.
#To display the object, we just type its name and run it
> vec_double
[1] 1.00 2.30 4.56
#Check data type
> class(vec_double)
[1] "numeric"

#With the L suffix right after a whole number,
#the number will be an integer rather than a double
> vec_integer <- c(1L, 2L, 3L)
> vec_integer
[1] 1 2 3
> class(vec_integer)
[1] "integer"

#note that in the third element here we are using a logical expression 2<3
#NA stands for "not available" or "missing value" and is a logical element
#T and F are variables which are set to TRUE and FALSE by default
> vec_logical <- c(TRUE, F, 2<3, NA)
> vec_logical
[1]  TRUE FALSE  TRUE    NA
> class(vec_logical)
[1] "logical"

> vec_character <- c("some", "characters")
> vec_character
[1] "some"       "characters"
> class(vec_character)
[1] "character"
```

If you try to combine different types of elements, they will be coerced to the most flexible type, as all elements need to be of the same type in a vector. The most to least flexible types are: character, double, integer, logical.

For example, if we want to combine a character and a numeric value, we get a vector of character type:

```
> vec_coercion <- c("abc", 1.2)
> vec_coercion
[1] "abc" "1.2"
> class(vec_coercion)
[1] "character"
```

We can also perform mathematical operations using R. For example, if we want to take the sum of all the elements in the vector "vec_double" and then take the square-root of the sum:

```
> sum_sqrt_vec_double <- sqrt(sum(vec_double))
> sum_sqrt_vec_double
[1] 2.803569
```

Vectors can also be used in arithmetic expressions, and the operations will be performed element by element. For example, if we would like to sum up the vectors "vec_double" and "vec_integer" element by element:

```
> sum_int_dbl <- vec_integer + vec_double
> sum_int_dbl
```

```
[1] 2.00 4.30 7.56
```

Arrays are objects that can store data in more than two dimensions. A **matrix** is a special case of array with two dimensions. Arrays and Matrices are created with `matrix()` and `array()`. Their dimensions can be modified by setting the dimension `dim()`.

```
# input one vector (1:18) and create two 3x3 matrices each with 3 rows and 3 columns
> array1 <- array(1:18, c(3,3,2))
> array1
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

# input one vector (1:6) and specify numbers of rows and columns
> matrix1 <- matrix(1:6, ncol=3, nrow=2)
> matrix1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

# modify an object by specifying dim()
> vector_ram <- 1:6
> dim(vector_ram) <- c(3,2)
> vector_ram
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
# we can again re-modify the dimension of this matrix
> dim(vector_ram) <- c(2,3)
> vector_ram
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

For these arrays and matrices, we can identify rows, columns and elements using subscripts. An element at the r^{th} row, c^{th} column of `A` can be extracted by the expression `A[r, c]`.

```
> matrix1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
# row 2 of column 3
> matrix1[2,3]
[1] 6
# row 2 of columns 1,2
> matrix1[2, 1:2]
[1] 2 4
```

If we omit `r` in the expression, i.e. `A[,c]`, the entire c^{th} column of `A` will be extracted:

```
> matrix1[,3] # 3rd column of matrix
[1] 5 6
```

If we omit `c` in the expression, i.e. `A[r,]`, the entire r^{th} row of `A` will be extracted:

```
> matrix1[1,] # 1st row of matrix
[1] 1 3 5
```

We can also multiply two conformable matrices together using `%*%` as follows:

```
> matrix2 <- matrix(1:6, ncol=2, nrow=3)
> matrix1 %*% matrix2
     [,1] [,2]
[1,]   22   49
[2,]   28   64
```

A data frame is the most common way of storing data in R. It is similar to a matrix but more general, as different columns can have different class/type of data. For example:

```
> d <- c(1,2,3,4)
> e <- c("Mary", "James", "Dan", "Rose")
> f <- c(TRUE,TRUE,TRUE,FALSE)
> dataframe1 <- data.frame(d,e)
> dataframe1 <- cbind(dataframe1, f) # Add a new column
> names(dataframe1) <- c("ID","Name","Attendance") # variable names
> dataframe1
  ID  Name Attendance
1  1  Mary       TRUE
2  2 James       TRUE
3  3   Dan       TRUE
4  4  Rose      FALSE
```

4. Conditional Statements in R

Conditional statements are a programming construct where the execution of some code is based on a logical expression. Common statements in R include `if()` statement, `if()...else()` statement, and `ifelse()` statement. The syntax of `if` statement looks like this. The statement will only be executed when the logical expression is TRUE. If the expression is FALSE, nothing happens.

```
if (logical expression) { statement }

#For example
> x <- 5
> if (x > 0) { print(``It is a positive number'') }
[1] ``It is a positive number''
```

The `if()...else()` statement is similar to the `if()` statement, but with an additional statement to `else()`. The statement2 will only be executed when the logical expression is FALSE:

```
if (logical expression) {
statement1
} else {
statement2
}

#For example
> x <- -1
> if (x > 0) {print(``It is a positive number'')
+ } else {print(``it is a non-positive number'')}
[1] ``it is a non-positive number''
```

A simplified statement for this is `ifelse()`. The syntax is:

```
ifelse(logical expression, statement1, statement2)

#The last example could be re-written as:
> x <- -1
> ifelse(x > 0, ''It is a positive number'', ''It is a non-positive number'')
[1] ''It is a non-positive number''
```

If there are more than two alternatives, we can build an if...else ladder, with multiple statements. For example, an if...else ladder with four statements looks like this:

```
if (logical expression 1) {statement1
} else if (logical expression 2) {statement2
} else if (logical expression 3) {statement3
} else {statement4}
```

Some useful operators in logical expressions are provided below:

Operator	Description
&	And
|	Or
!	Not
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to

5. Describing data frames/tables in R

Data frames (tables) are commonly used in R to read, manipulate and store data. In this part of the tutorial, we will learn to read (and write), describe and summarize, and generate new random variables for a dataset. In particular, we will look at the 'sleep' dataset from the 'VIM' R package. This is a simple dataset on the sleeping patterns of mammals. Let's first install/load the package:

```
> install.packgages("VIM") # install package
> library(VIM) # load package for use
```

Note that capitalization matters in R. For example, R distinguishes between object such as 'mySleep' and 'mysleep'. Hence, any command that you write must have the proper capitalization. Base commands in R are generally in lower case.

5.1 Loading and Inspecting the Data

You can store the dataset in your computer so that it can be reopened for later use. In order to do this, first specify a working directory (a folder on your computer that allows you to read and store

data). It is good practice to organize your working directories in a way that enables you to access your dataset easily (e.g. by project). To set the working directory in R, we can write setwd("file path name here in quotes"). For example:

```
> setwd("/Users/lanwen/Documents/Tutorial_R")
```

Alternatively, a manual way to set the working directory is to go to the menu bar of R studio and click 'Session' $\longrightarrow$ 'Set Working Directory'$\longrightarrow$ 'Choose Directory', which allows you to choose the working directory by navigating manually on your computer. You can also check your current working directory in R by typing:

```
> getwd()
```

Now that we have set the working directory, we will store the 'sleep' data set in our directory as a CSV file. To do this, we can type the command `write.csv(dataset, ‘‘name of file.csv")`. For example we can type the following:

```
> write.csv(sleep, "Sleep_data.csv")
```

Now that we have a dataset in our directory, we will import our data in R. We can read our CSV file in R with the command `read.csv(‘‘name of file.csv", header=TRUE)`. If your working directory does not contain the specified CSV file, you will get the following error message in R:

```
Error: attempt to use zero-length variable name
```

By importing the Sleep_data.csv file, we have created an object in R called a data frame. Assigning it as, e.g. 'sample_data', allows us to change and manipulate the dataset later on. Another type of object for datasets is data table (inherits from data frame but can be faster/more efficient).

```
> sample_data <- read.csv("Sleep_data.csv", header=TRUE)
```

Now that we have loaded the sleep data, we can take a look at the entire dataset in a separate window in R studio by calling:

```
> View(sample_data)
```

To look at the data in the current console, we can type

```
> sample_data
```

However, it is impractical to view an entire dataset if it is large. Instead, we can inspect the first few lines of the dataset by typing:

```
> head(sample_data)
> head(sample_data, 6)
> sample_data[1:6,]
```

These all give the first 6 lines of the data frame. By default, the command `head(sample_data)` will display the first 6 lines of the data frame. In `head(sample_data, 6)`, the number 6 in the second argument in parenthesis says that you want to see the first 6 lines of the data frame. To call an item in a data frame, traditional R syntax requires one to specify the `[Row, Column]`. In `sample_data[1:6,]`, we are specifying that we want to see the first 6 rows of the data, and ALL of the columns, since we have left everything to the right of the comma blank.

We can type the following to obtain the dimension of the data frame:

```
> dim(sample_data)
[1] 62 11 # we have 62 observations (rows) and 11 variables (columns)
```

In a data frame, each row is an observation and each variable has its own column. In the sleep dataset, each observation is a type of mammal, identified with a unique ID denoted by the variable 'X'. Here is a full list of the variables:

- X: Species of mammal
- BodyWgt: body weight of the mammal (kg)
- BrainWgt: weight of the mammal's brain (g)
- NonD: Hours of slow wave ('nondreaming') sleep per day
- Dream: Hours of paradoxical ('dreaming') sleep per day
- Sleep: Total hours of sleep per day (NonD plus Dream)
- Span: Life span of mammal (years)
- Gest: Gestation time of mammal (days)
- Pred: Predation index [1–5]; 1 = least likely to be preyed upon
- Exp: Sleep exposure index [1–5]; 1 = most well-protected sleeping environment
- Danger: Overall danger index [1–5]; 1 = safest from other animals

To see the names of the variables, we can type into R the following:

```
> names(sample_data)
```

To look at an entire variable in the dataset (i.e. an entire column), we can use the $ operator in the following way:

```
> sample_data$BodyWgt
```

This allows one to look at one variable at a time. To see multiple columns and/or rows, we can use the `sample_data[row,column]` syntax.

5.2 Generating new variables

Suppose now we want to identify and store in our dataset the species of mammals that have a total of more than 8 hours of sleep per day. To do this we can create a new variable called 'sleep_cat'. This variable equals 0 if the species of mammals has ≤ 8 hours of sleep per day and 1 if it has > 8 hours of sleep per day. We can add this random variable in two ways using the $ operator:

```
# First solution
> sample_data$sleep_cat[sample_data$Sleep > 8] = 1
> sample_data$sleep_cat[sample_data$Sleep <= 8] = 0
# Second solution
> sample_data$sleep_cat = ifelse(sample_data$Sleep > 8, 1, 0)
```

The first way to do this is to generate a new random variable called 'sleep_cat' and assign it a value 1 for observations with more than 8 hours of sleep (first row), and a value 0 for observations with less than or equal to 8 hours of sleep (row 2). The second way allows us to assign 'sleep_cat' values in the same way with the ifelse() syntax (try it yourself).

Notice that there are missing values in the 'Sleep' variable. Hence the observations that are missing in the 'Sleep' variable will also be missing in the 'sleep_cat' variable (we will come back to this later). Nevertheless, we can calculate the number of species (that are observed) with more than 8 hours of sleep. To do this we can use the following command:

```
> table(sample_data$sleep_cat)
 0  1
15 43
```

The following command tells us the proportion of (observed) species than have more than 8 hours of total sleep:

```
> table(sample_data$sleep_cat)/sum(table(sample_data$sleep_cat))
        0         1
0.2586207 0.7413793
```

By using `sum(table(sample_data$sleep_cat))`, we can calculate the total number of observed species in the table. Dividing the number of species with more than 8 hours of sleep by this number will give us the correct proportions. Alternatively, we can call the following command to give us the same results:

```
> prop.table(table(sample_data$sleep_cat))
        0         1
0.2586207 0.7413793
```

5.3 Summary statistics and simple plots

5.3.1 Define mean, standard deviation, median, interquartile range, percentiles

We noticed that there were missing data from the inspection of the variable 'Sleep'. To see if any of the other variables have missing data, we can type the command:

```
> summary(sample_data)
```

This tells us that the variables BodyWgt, BrainWgt, Pred, Exp, Danger do not have any missing rows, but NonD, Sleep, Dream, Span, Gest, sleep_cat, have at least one missing row. To see how many observations in each variable is missing, we can simply look at the number of NA's in each variable. The `summary()` command also tells us the summary statistics (i.e. minimum, 1st quartile, median, mean, 3rd quartile, and maximum) for each variable in the dataset. To look at the summary statistic for just one variable, we can also just call `summary(sample_data$variable_name)`. Now let's take a look at the 'BodyWgt' variable:

```
> summary(sample_data$BodyWgt)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
   0.005    0.600    3.342  198.790   48.203 6654.000
```

Other commands that tell us this information are of the following:

```
> mean(sample_data$BodyWgt)
[1] 198.79
> median(sample_data$BodyWgt)
[1] 3.3425
> quantile(sample_data$BodyWgt)
      0%      25%      50%      75%     100%
  0.0050   0.6000   3.3425  48.2025 6654.0000
```

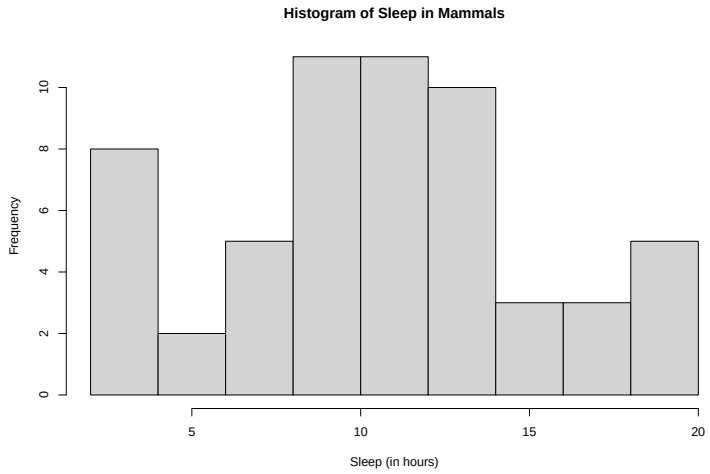
```
> min(sample_data$BodyWgt)
[1] 0.005
> max(sample_data$BodyWgt)
[1] 6654
> var(sample_data$BodyWgt)
[1] 808485.1
> sd(sample_data$BodyWgt)
[1] 899.158
> quantile(sample_data$BodyWgt, c(.1, .2, .3))
   10%    20%    30%
0.0776 0.3090 0.9060
```

These individual commands allow us to obtain more statistics that are not given by the summary command such as variance (var()), standard deviation (sd()) and percentile (quantile()).

5.3.2 Graphs and plots

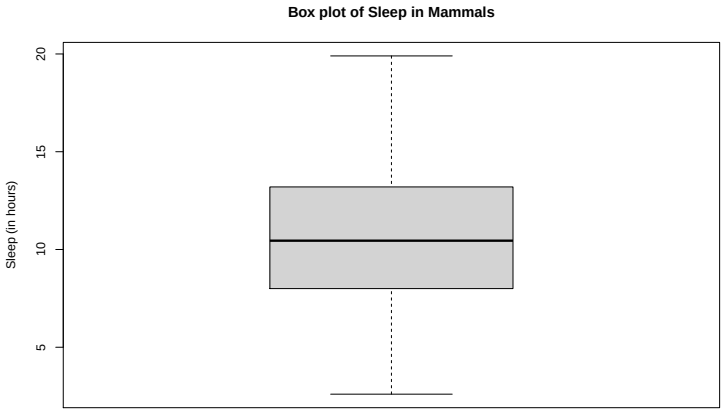
We can also use graphics in R to describe and summarize our data. In this tutorial, we will look as histograms, box plots and scatterplot to describe the 'Sleep' variable. To obtain the histogram of `Sleep`, we can use the following command:

```
> hist((sample_data$Sleep), xlab="Sleep (in hours)",
        main="Histogram of Sleep in Mammals"))
```



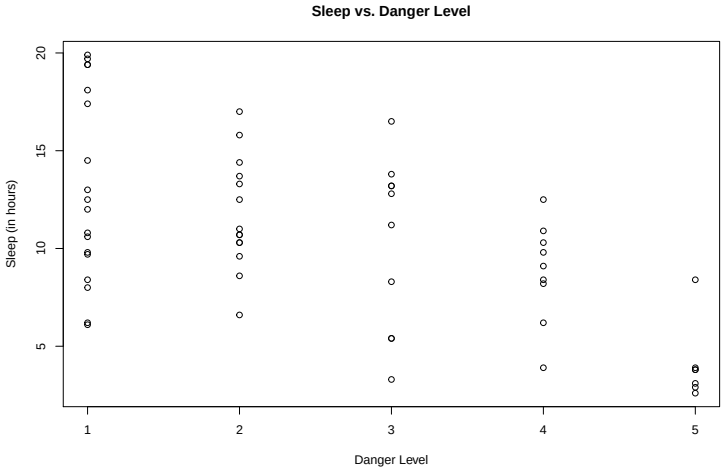
To obtain the boxplot of `Sleep`, we can use the following command:

```
> boxplot((sample_data$Sleep), ylab="Sleep (in hours)",
        main="Box plot of Sleep in Mammals")
```



To obtain the scatterplot of `Sleep` vs. `Danger`, we can use the following command:

```
> plot((sample_data$Danger), (sample_data$Sleep), ylab="Sleep (in hours)",
        xlab="Danger Level",main = "Sleep vs. Danger Level")
```



6. Model fitting using linear regression

To run a linear model in R regressing `Sleep` on `Danger`, we can use the `lm()` function:

```
## Run a linear model using lm()
> my_model <- lm( Sleep ~ Danger, data=sample_data)
summary(my_model)
> summary(my_model)

Call:
lm(formula = Sleep ~ Danger, data = sample_data)

Residuals:
    Min      1Q  Median      3Q     Max
-7.4177 -2.8070 -0.7225  3.0865  6.8728
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  15.4630     1.0326  14.975  < 2e-16 ***
Danger       -1.9452     0.3578  -5.436 1.23e-06 ***
---
...
Residual standard error: 3.76 on 56 degrees of freedom
  (4 observations deleted due to missingness)
Multiple R-squared:  0.3454,    Adjusted R-squared:  0.3338
F-statistic: 29.55 on 1 and 56 DF,  p-value: 1.23e-06
```

7. Data cleaning: handling missing values

7.1 Inspection of the variable of interest

We will be focusing on the 'Sleep' (total hours of sleep per day) variable in this section. An initial inspection of the 'Sleep' variable using the summary command tells us that there are 4 missing observations:

```
> summary(sample_data$Sleep)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NAs
   2.60    8.05   10.45   10.53   13.20   19.90       4
```

Alternatively, we can find out how many observations are missing in 'Sleep' by using the following command:

```
> sum(is.na(sample_data$Sleep))
[1] 4
```

Note: The `is.na()` command returns a logical vector that will be TRUE for every element that meets the criteria, and FALSE for every element that does not. The `sum()` command then adds up the total number of elements that met this criteria. In the above command, we are summing up the total number of observations that had missing 'Sleep'.

To see if each observation in the dataset is missing, we can write try the following:

```
> is.na(sample_data)
```

If we only want observations (rows) from the dataset that have complete data (i.e. no missing data in any columns), we can write the following command that omits any rows of data with at least one NA's:

```
> na.omit(sample_data)
```

or

```
> sample_data[complete.cases(sample_data),]
```

However, as we can see, doing this would erases 20 rows of observations in the data. We could omit the rows or variables in our data analysis, but often times this is not the most efficient use of data.

Say we want to analyze the total sleep variables in the mammals. This variable is made up of the sum of the 'NonD' and 'Dream' variables. Let's take a look at the missing values in all three variables:

```
> sum(is.na(sample_data$Sleep))
[1] 4
> sum(is.na(sample_data$Dream))
[1] 12
> sum(is.na(sample_data$NonD))
[1] 14
```

Are there any observations that are missing 'Dream' AND 'NonD' AND 'Sleep'?

```
> sample_data[is.na(sample_data$Dream) &
              is.na(sample_data$NonD) & is.na(sample_data$Sleep),]
```

Note: This '&' in the command tells us the rows that are missing both of these variables, and the syntax in the brackets [,] tells us to select these rows and ALL of the columns.

Are there any rows where 'Sleep' is observed, but only one of 'Dream' (Hours of paradoxical sleep per day) or 'NonD' (Hours of slow wave sleep per day) is observed (i.e. 'Dream' is missing and 'NonD' is observed, OR 'Dream' is observed and 'NonD' is missing)?

```
> sample_data[!is.na(sample_data$Dream) &
        is.na(sample_data$NonD) & !is.na(sample_data$Sleep),]
...
<0 rows> (or 0-length row.names)
> sample_data[is.na(sample_data$Dream) &
        !is.na(sample_data$NonD) & !is.na(sample_data$Sleep),]
...
<0 rows> (or 0-length row.names)
```

Note: The Boolean operator NOT is signified by '!'.

There is no observation that meets this criteria, otherwise we can simply fill in the missing value without imputation. In what follows, we will use the `dplyr` package in R that allows us to clean our data. First we will install and load the package into your R workspace:

```
> install.packages("dplyr")
> library(dplyr)
```

7.2 Dplyr data manipulation

The dplyr package allows us to apply the very useful % > % operator called the pipe. The pipe operator is similar to a grammatical element of a sentence that means "and then do". It is followed by command "verbs" that allow quick execution of many tasks. The "verbs" that we will see in this tutorial are `filter`, `select`, and `summarise`.

To filter out observations that have missing 'Sleep' variables, we first identify the species of animals that have this missing variable and assign it a vector. Then we would use the % > % take the sample_data data frame and then filter out 'X' that are in this vector using the %in% operator:

```
> miss_sleep <- sample_data[is.na(sample_data$Sleep),]$X
> sample_data_miss_sleep <- sample_data %>% filter((X %in% miss_sleep))
> dim(sample_data_miss_sleep)
[1]  4 12
> nrow(sample_data_miss_sleep)
[1] 4
```

Similarly, we can use the syntax [Row,Column] to do the same operation without % > %. Let's first identify the species of animals that do not have this missing variable and assign it a vector. We will tell R to take the data frame and give us the rows where 'X' is NOT ('!') in the vector.

```
> obs_sleep <- sample_data[!is.na(sample_data$Sleep),]$X
> sample_data_miss_sleep2 <- sample_data[!(sample_data$X %in% obs_sleep),]
> dim(sample_data_miss_sleep2)
[1]  4 12
> nrow(sample_data_miss_sleep2)
[1] 4
```

Now that we have identified the four observations with the missing sleep patterns, we can either remove them from analysis or we can try to impute the missing values. There are fancy imputation techniques in R (e.g. multiple imputation using chained equations, joint multivariate normal multiple imputation), however for simplicity we will impute the missing 'Sleep' values based on 'Exp' (sleep exposure index). We see that of the four observations, two have 'Exp' value of 1 ($X = 62$) and two have 'Exp' value of 5 ($X = \{21,\ 31,\ 41\}$).

First, let's subset the data to only the observations with 'Exp' value of 1, excluding $X = 62$. Standard R syntax:

```
> sample_data[(sample_data$Exp==1)&(sample_data$X!=62),]
```

Dplyr pipe:

```
> sample_data %>% filter((Exp==1)&(X!=62))
> sample_data %>% filter(Exp==1) %>% filter(X!=62)
```

From here, it's easy to find the average 'Sleep' of this subset of data and then assign it to $X = 62$ using the dplyr pipe. Here are a few ways to do this:

Option 1: Dplyr pipe % > % with saved output to compute mean

```
> Exp_equal_1 <- sample_data  %>% filter(Exp==1) %>% filter(X!=62)
> mean(Exp_equal_1$Sleep)
[1] 12.94615
```

Option 2: Dplyr pipe % > % with summarise() command to compute mean

```
> sample_data  %>% filter(Exp==1) %>% filter(X!=62) %>% summarise(mean(Sleep))
  mean(Sleep)
1    12.94615
```

Option 3: Standard R syntax with saved output to compute mean

```
> Exp_equal_1_2 <- sample_data[(sample_data$Exp==1)&(sample_data$X!=62),]
> mean(Exp_equal_1_2$Sleep)
[1] 12.94615
```

We can now assign this average as the 'Sleep' for $X = 62$.

```
> sample_data$Sleep[sample_data$X==62] <- mean(Exp_equal_1_2$Sleep)
```

Did it work? Check the data:

```
> sample_data[sample_data$X==62,]
```

Now try it on your own: following the procedures above, try to impute the missing 'Sleep' values for $X = \{21,\ 31,\ 41\}$ by finding the average hours of sleep in the mammals that have 'Exp' equal to 5 (*Hint*: you can create a vector `c(21, 31, 41)` and make use the `filter()` and %in% functions). You should obtain an average of 4.19. Let's assign this value to $X = \{21,\ 31,\ 41\}$:

```
> sample_data$Sleep[sample_data$X==21] <- 4.19
> sample_data$Sleep[sample_data$X==31] <- 4.19
> sample_data$Sleep[sample_data$X==41] <- 4.19
```

It's good practice to keep the original dataset (sample_data) in your R workspace, and create a new dataset after data manipulation. We will simply assign a new name to the imputed dataset:

```
>sample_data_2 <- sample_data
```

Let's remove the unnecessary data frames from our workspace so that we don't get confused:

```
>rm(Exp_equal_1, Exp_equal_1_2, sample_data_miss_sleep, sample_data_miss_sleep2)
```

Another good practice is to use garbage collection in R. This automatically releases memory when an object is no longer used. To do this, we simply call

```
>gc()
         used (Mb) gc trigger (Mb) limit (Mb) max used (Mb)
Ncells 2521749 134.7    4501002 240.4         NA  4501002 240.4
Vcells 4993764  38.1   10146329  77.5      16384 10085159  77.0
```

8. More subsetting, and Merging Data

We have now imputed the 'Sleep' variable and created the sample_data_2 dataset. Next, we will learn how to isolate particular subsets of variables, and merge data frames. Our first goal is to subset the data and selecting the variables of interest: first, we will isolate only those species of animals that have more than 8 hours of total sleep per day. Note that this is a binary variable (i.e. it takes two values: 1 and 0). In base R, the command is `subset()`, while in dplyr, we tell R to filter all of the observations with `sleep_cat==1`.

```
> sleep_lot <- subset(sample_data_2, sleep_cat==1)
> sleep_lot <- sample_data_2 %>% filter(sleep_cat==1)
```

Let's take a look at this data frame:

```
head(sleep_lot)
```

Next, let's restrict to three relevant variables in sleep_lot data frame: 'X' , 'sleep_cat' and 'Sleep'. With dplyr, we use `select()` to choose the columns using `c()`. Using base R, we must use quotations around the variable names to indicate that we are specifying objects (variables) within sleep_lot. In dplyr, the $\% > \%$ operator allow us to avoid the quotations because it let's R know that we are looking inside the sleep_lot data.

```
> sleep_lot_subset <- sleep_lot[,c("X", "sleep_cat", "Sleep")] # OR
> sleep_lot_subset <- sleep_lot %>% select(c(X, sleep_cat, Sleep))
```

We also want to get from sample_data_2 species of animals of 'Exp' of 1. To do this we will subset and then take only two relevant variables: 'X' and 'Exp':

```
> low_exp <- sample_data_2 %>% filter(Exp==1)
> low_exp_subset <- low_exp %>% select(c(X, Exp))
```

Our second goal is to merge data: Say we are interested in looking at mammals who have 'Exp' equal to 1 among those species who sleep more than 8 hours a day. We will perform a merge of

the low_exp_subset data with the sleep_lot_subset data. R uses an unique identifier so it knows how and which the data will be merged. In the two data frames, this unique identifier is the 'X' variable (species). To perform a merge, the base R command `merge()`. In dplyr, the command is `inner_join()`, which is much faster than `merge()`. This makes a difference when working with huge datasets. We can tell R which variable(s) to merge on by using `by=''variable name''`. In our example, the common variable `X`.

```
> my_interested_data <- merge(low_exp_subset, sleep_lot_subset, by="X", all=FALSE)
> my_interested_data <- inner_join(low_exp_subset, sleep_lot_subset, by="X")
```

To merge by 2 or more variables, we would specify `by=c(''var1'',''var2'',...)`, where var1 and var2 and so forth are the different variable names.