# Introduction to R

Johnny Lo

Feb 2023

## Contents

To start the session, open a blank R script bygo to the toolbar in R Studio and select **File > New File > R Script** or **Ctrl+Shift+N** on your keyboard. Next, make sure you load the **tidyverse** package using the *library(.)* command. Then commence coding by following the notes below.

# 1 Fundamentals of Programming in R

Everything that we create in R is an **object**. An object is a data structure having some attributes and methods which act on its attributes. For beginners of R, it is very important that you have a strong understanding of **variables**, and the basic **data types** and **data structures**. These are the types of objects that you will create, operate on, and manipulate on a day-to-day basis. Once you have the basics down, then you are well on your way to being a competent R programmer.

## 1.1 Variable assignment

In mathematics and statistics, a variable is a symbol that can take on any value. In R, a variable allows the user to store a value, a data structure, a function or sub-routine, and even plots. We can later use the variable's name to access the value or the object that is stored within it. In R, we can use either **<-** or = for assigning or storing a value or an object to a variable. A variable name can be a single letter, or a combination of letters, numbers and symbols. However, there are two rules when defining a variable name in R.

1. It **must** start with a letter (lower or upper case), or full stop (.) but the latter cannot be followed by a number.
2. It **must not** contain any symbol other than full stop and/or underscore(_).

In the example below, the number 35 is assigned to the variable **x**. Note that when you run this command, the object or variable **x** is created but nothing will be displayed.

```
x <- 35
```

To recall or print out the value that is assigned to variable **x**, you just need to run variable name as a command.

```
x
```

```
## [1] 35
```

You can overwrite the stored value by assigning another value to the same variable. For example,

```
x <- 18; x
```

```
## [1] 18
```

Suppose we want to add 8 to x and store the sum in a new object, say y. This can be done as follows.

```
y <- x + 8; y
```

```
## [1] 26
```

Suppose now we have a linear equation given by y = 3x + 5. Since x = 18, then we have,

```
y <- 3*x+5; y
```

```
## [1] 59
```

If we wish to evaluate y for another value of x, say 17 then all we need to do is overwrite the value of x, i.e. x <- 17, and re-run the above command for y. The ability to write an equation as a code, such as the one above, allows us perform computations much more efficiently.

When coding, it is always a good idea to give meaningful names to variables and better yet, ones that are self explanatory. Providing meaningful names to variables will help the reader and reduce the amount of commentaries required. For example, suppose we want to calculate the area of a circle (i.e. $\pi r^2$), given that its diameter is 10 cm. Here we have,

```
diam <- 10  #diameter = 10 cm
radius <- diam/2  #radius is half of the diameter
area.circle <- pi*radius^2  #area of the circle
```

## 1.2 Data Types

There are six basic data types in R, however, we will only discuss four of them in this workshop, namely,

- **Character**
- **Numeric**
- **Integer**
- **Logical**

as they are the most frequently used. The other two types are *complex* and *raw*.

### 1.2.1 Character

A *character* is a string value and is defined by using the quotes (" "). A character can be a number (e.g. 1, 2, 3), letter (e.g. a, b, C), symbol (e.g. #, $, @), or a combination of the three (e.g. a word or sentence). Note that if a number is defined as a character in R, then you will not be able to perform any mathematical operation on it.

Here are some examples.

```
"a"
```

```
## [1] "a"
```

```
"abc123"
```

```
## [1] "abc123"
```

```
"apples"
```

```
## [1] "apples"
```

```
"I hate apples"
```

```
## [1] "I hate apples"
```

You can also run multiple commands with a single line of code by using using semi-colon (;) as the separator.

```
"a"; "abc"; "apples"; "I hate apples"
```

```
## [1] "a"
## [1] "abc"
## [1] "apples"
## [1] "I hate apples"
```

### 1.2.2 Numeric

A *numeric* is any real or decimal value. You can define them in R simply by typing in the value and executing it. For example,

```
5.5; 2.75; pi
```

```
## [1] 5.5
## [1] 2.75
## [1] 3.141593
```

We can also perform mathematical operations on the numerics just as you would do in a calculator.

```
5.5+2.7  #Addition
```

```
## [1] 8.2
```

```
5.5-2.7  #Subtraction
```

```
## [1] 2.8
```

```
5.5/2  #Division
```

```
## [1] 2.75
```

```
5.5*2  #Multiplication
```

```
## [1] 11
```

```
5.5^2  #Squaring
```

```
## [1] 30.25
```

```
5.5^4  #To the power of 4
```

```
## [1] 915.0625
```

```
sqrt(5.5)  #Square root
```

```
## [1] 2.345208
```

```
exp(5.5)  #Exponential
```

```
## [1] 244.6919
```

```r
log(5.5)  #Natural log
```

```
## [1] 1.704748
```

### 1.2.3 Integer

An *integer* is a whole number that can be positive or negative. To define an integer in R, we just need to type **L** after a whole number. For example,

```r
5L  #5
```

```
## [1] 5
```

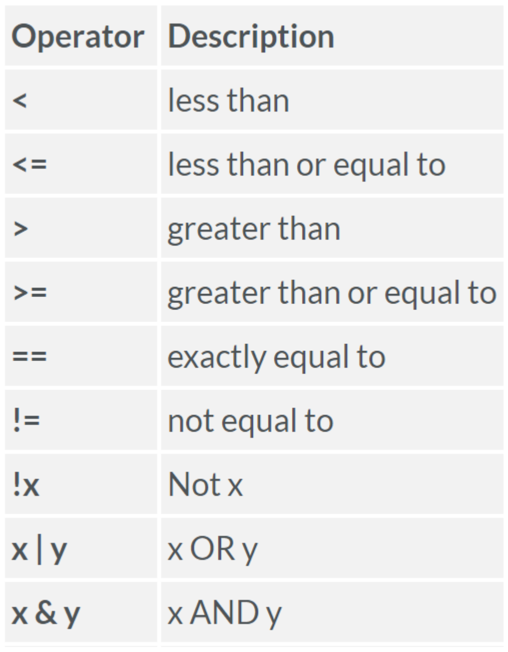
```r
-3L  #-3
```

```
## [1] -3
```

The mathematical operations that were performed previously on numerics can also be applied to integers.

### 1.2.4 Logical

A *logical* value is to indicate whether an item/statement is TRUE or FALSE, i.e. a Boolean value. A list of the R logical operators are given in Figure 1.



| Operator | Description |
|---|---|
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | exactly equal to |
| != | not equal to |
| !x | Not x |
| x \| y | x OR y |
| x & y | x AND y |

Figure 1: Logical operators in R

Here are some examples.

```r
5==5  #Does 5 equal 5?
```

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

```
5==2  #Does 5 equal 2?
```

```
## [1] FALSE
```

```
5!=2  #Does 5 not equal 2?;
```

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

```
5>2  #Is 5 greater than 2?
```

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

```
5>=2  #Is 5 greater than or equals to 2
```

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

```
5<2  #Is 5 less than 2?
```

```
## [1] FALSE
```

```
5<=2  #Is 5 less than or equals to 2
```

```
## [1] FALSE
```

What about for the following? See if you can determine the answers to them.

```
(5>2)&(5>4)
(5>2)&(5<4)
(5>2)|(5<4)
```

Now run the code and check your answers.

## 1.3 Data Structures

Data structures are tools for holding multiple values. In data analysis, one typically works with groups of numbers and/or characters, and rarely with single values one-at-a-time. Data structures allows us to store and manipulate multiple or groups of values simultaneously. The basic data structures used in R include vectors, factors, matrices, data frames (tibbles) and lists.

### 1.3.1 Vector

A *vector* is a collection of elements of the **same** data type. A vector can be created by using the *c(.)* command.

```
c()  #A null vector
```

```
## NULL
```

```
c(1,2,3)  #A numeric vector
```

```
## [1] 1 2 3
```

```
c("a","b","c")  #A character vector
```

```
## [1] "a" "b" "c"
```

```
c(TRUE,FALSE,FALSE)  #A logical vector
```

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

Let us create a vector containing the integers from 1 to 5.

```
a <- c(1:5); a
```

```
## [1] 1 2 3 4 5
```

We use square brackets, i.e. [.], if we wish to access particular element(s) within the vector, rather than the whole vector.

```
a[2]  #2nd element
```

```
## [1] 2
```

```
a[3]  #3rd element
```

```
## [1] 3
```

```
a[3:5]  #3rd to 5th element
```

```
## [1] 3 4 5
```

```
a[c(2,5)]  #2nd and 5th elements
```

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

We can combine vectors together to form a larger vector.

```
b <- c(6:10); b  #Create a new vector b containing the integers 6, 7, 8, 9 and 10.
```

```
## [1]  6  7  8  9 10
```

```
c <- c(a,b); c  #Combine vectors a and b, and assign the vector sum to a new vector, c.
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

We can also perform mathematical operations on vectors.

```
a*0.25  #Multiply the elements of vector a by a constant, i.e. 0.25
```

```
## [1] 0.25 0.50 0.75 1.00 1.25
```

```
a+b  #Add the elements of vector a and vector b together
```

```
## [1]  7  9 11 13 15
```

```
b-a  #Subtract the elements of vector a from vector b
```

```
## [1] 5 5 5 5 5
```

```
a*b  #Multiply the elements of vector a and vector b together
```

```
## [1]  6 14 24 36 50
```

```
b/a  #Divide the elements of vector b by the elements of vector a
```

```
## [1] 6.000000 3.500000 2.666667 2.250000 2.000000
```

Notice how that the operations are performed element to element. For instance with a+b, the 1st element of vector a is added to the 1st element of vector b to form the 1st element of the vector sum, then the 2nd element of vector a is added to the 2nd element of vector b to form the 2nd element of the vector sum, and so on. This is also the case for the other mathematical operations.

### 1.3.2 Factor

A *factor* is similar to a vector, but it is a collection of elements from a finite set of values, i.e. categorical variable. To create a factor, we start by creating a numeric or character vector then convert it to a factor using the *factor(.)* command.

```
f1 <- c(1:5);  #A numeric vector
f1 <- factor(f1); f1  #Convert f1 to a factor
```

```
## [1] 1 2 3 4 5
## Levels: 1 2 3 4 5
```

```
f2 <- c("Male","Female","Female","Male","Female");  #A charactor vector
f2 <- factor(f2); f2  #Convert f2 to a factor
```

```
## [1] Male   Female Female Male   Female
## Levels: Female Male
```

With f2, we can see that it is a factor with 2 levels, i.e. Male or Female.

*Note that by default, the levels of a factor are ordered alphabetically.* This is often not ideal in cases where there is an order to said levels. For example, suppose we have a ordinal factor (f3) with three levels; low(L), medium (M) and High (H).

```
f3 <- factor(c("L","M","H","H","M","L")); f3
```

```
## [1] L M H H M L
## Levels: H L M
```

As you can see, the default order for the levels is H, L, M. This can be annoying if you want to summarise or plot a set of data accordingly for these levels and compare them. Naturally, one would prefer to have the order be L, M, H. We can ensure this is the case by including the *levels* argument to the *factor(.)* command, and specifying the desired order.

```
#re-order the levels of f3 and overwrite the original f3.
f3 <- factor(f3,levels=c("L","M","H")); f3
```

```
## [1] L M H H M L
## Levels: L M H
```

### 1.3.3 Matrix

A matrix (plural: matrices) is a 2-dimensional array whose elements are all of the **same** data type. A matrix is defined using the *matrix(.)*, and requiring the data and matrix dimension as inputs. By default, data are read into the matrix in a column-wise manner.

Below is an example of a 3 x 3 matrix.

```
Mat.A <- matrix(c(1:9),nrow=3,ncol=3,byrow=FALSE); Mat.A
```

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

We can also read the data in a row-wise manner change the *byrow* argument to **TRUE**. That is,

```
Mat.B <- matrix(c(1:9),nrow=3,ncol=3,byrow=TRUE); Mat.B
```

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

Note that in some instances, the command may have several input arguments and can be difficult to read. To improve the readability of an extended command, we can break it up and write it across multiple lines. This works because R will examine the next line if the current command line is incomplete. Another advantage of this is that we can add commentary to each of the input arguments. We will use the previous example to illustrate this.

```
Mat.A <- matrix(c(1:9),  #9 entries to be read in
                nrow=3,  #3 rows
                ncol=3,  #3 columns
                byrow=FALSE  #data to be read in column-wise (default)
                );
Mat.A
```

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

Another way to create a matrix, is by binding multiple vectors of the same length together. We can either bind the vectors together as column vectors using the *cbind(.)* command, or as row vectors using the *rbind(.)* command.

```
v1 <- c(1:3)  #Vector 1
v2 <- c(4:6)  #Vector 2
v3 <- c(7:9)  #Vector 3
```

```
Mat.A <- cbind(v1,v2,v3); Mat.A  #binding vectors as column vectors
```

```
##      v1 v2 v3
## [1,]  1  4  7
## [2,]  2  5  8
## [3,]  3  6  9
```

```
Mat.B <- rbind(v1,v2,v3); Mat.B  #binding vectors as row vectors
```

```
##    [,1] [,2] [,3]
## v1    1    2    3
## v2    4    5    6
## v3    7    8    9
```

The elements of two or more matrices can be multiplied together only if they have the same dimension. Since **Mat.A** and **Mat.B** are both 3 × 3 matrices, we can multiply them by one another. Note that the multiplication is done element-to-element. In this instance, A × B = B × A.

```
Mat.A*Mat.B
```

```
##      v1 v2 v3
## [1,]  1  8 21
## [2,]  8 25 48
## [3,] 21 48 81
```

However in matrix multiplication, say A × B, the number of columns in matrix A must equal the number of rows in matrix B, and each row of matrix A are multiplied by each column of matrix B, and summed. Click **here** for a Khan Academy online tutorial in manual matrix multiplication. Matrix multiplication in R is performed using the %*% command. In matrix multiplication, A × B ≠ B × A.

```
Mat.A%*%Mat.B  #A matrix multiply B
```

```
##      [,1] [,2] [,3]
## [1,]   66   78   90
## [2,]   78   93  108
## [3,]   90  108  126
```

```
Mat.B%*%Mat.A  #B matrix multiply A
```

```
##    v1  v2  v3
## v1 14  32  50
## v2 32  77 122
## v3 50 122 194
```

To access the elements of a matrix, say A, we need to specify the corresponding row(s) and columns(s) within the square brackets, i.e. **A**[***row(s)*,*column(s)***]. If the rows are **not** specified, but the columns are, then all the rows will be extracted and vice versa.

Here are some examples.

```
Mat.A[2,3]  #Access the element located in row 2 and column 3
```

```
## v3
##  8
```

```
Mat.A[1:2,3]  #Access the elements located in rows 1 and 2 along column 3
```

```
## [1] 7 8
```

```
Mat.A[1,2:3]  #Access the elements located along row 1 and in columns 2 and 3
```

```
## v2 v3
##  4  7
```

```
Mat.A[1,]  #Access all elements in row 1
```

```
## v1 v2 v3
##  1  4  7
```

```
Mat.A[,3]  #Access all elements in column 3
```

```
## [1] 7 8 9
```

```
Mat.A[,c(1,3)]  #Access all elements in columns 1 and 3
```

```
##      v1 v3
## [1,]  1  7
## [2,]  2  8
## [3,]  3  9
```

```
Mat.A[,-1]  #Access all data, except those in column 1
```

```
##      v2 v3
## [1,]  4  7
## [2,]  5  8
## [3,]  6  9
```

### 1.3.4 Data Frame and Tibble

A data frame similar to a matrix, but the columns do not have to be the same data type. However, all columns must have the **same length**, and the elements within each column must be of the **same type**. Most datasets, when imported into R, are defined as data frames and hence they are the most commonly used data structure. In R, data frames are created using the *data.frame(.)* command on a list of vectors, which can be of different data types. Let us first create three vectors of different types.

```
Name <- c("John","Sarah","Zach","Beth","Lachlan");  #Name - Character vector
Age <- c(35,28,33,55,43); #Age - Numeric vector
Gender <- factor(c("Male","Female","Male","Female","Male"));  #Gender - factor
```

Now we can combined the three vectors to form a data frame.

```
df <- data.frame(Name,Age,Gender); df
```

```
##      Name Age Gender
## 1    John  35   Male
## 2   Sarah  28 Female
## 3    Zach  33   Male
## 4    Beth  55 Female
## 5 Lachlan  43   Male
```

We can add new columns(s) to an existing data frame by using either the *data.frame(.)* or *cbind(.)* command.

```
Coffee.Drinker <- c(TRUE,TRUE,FALSE,TRUE,FALSE);  #Drinks coffee? - logical vector

data.frame(df,Coffee.Drinker)  #Add new column to the data frame - Method 1
```

```
##      Name Age Gender Coffee.Drinker
## 1    John  35   Male           TRUE
## 2   Sarah  28 Female           TRUE
## 3    Zach  33   Male          FALSE
## 4    Beth  55 Female           TRUE
## 5 Lachlan  43   Male          FALSE
```

```
cbind(df,Coffee.Drinker)  ##Add new column to the data frame - Method 2
```

```
##      Name Age Gender Coffee.Drinker
## 1    John  35   Male           TRUE
## 2   Sarah  28 Female           TRUE
## 3    Zach  33   Male          FALSE
## 4    Beth  55 Female           TRUE
## 5 Lachlan  43   Male          FALSE
```

Data frames are accessed in the same way as matrices.

```
df[1,c(1:3)]
```

```
##   Name Age Gender
## 1 John  35   Male
```

```
df[2:3,]
```

```
##    Name Age Gender
## 2 Sarah  28 Female
## 3  Zach  33   Male
```

```
df[,c(1,3)]
```

```
##      Name Gender
## 1    John   Male
## 2   Sarah Female
## 3    Zach   Male
## 4    Beth Female
## 5 Lachlan   Male
```

We can also access the columns by using their names.

```
df[,"Name"]
```

```
## [1] "John"    "Sarah"   "Zach"    "Beth"    "Lachlan"
```

```r
df[,c("Name","Gender")]
```

```
##      Name Gender
## 1    John   Male
## 2   Sarah Female
## 3    Zach   Male
## 4    Beth Female
## 5 Lachlan   Male
```

Another way is to use the **$** syntax and the column name to access the column. However, we can only access one column at a time using this approach.

```r
df$Name  #Access and display the "Name" column
```

```
## [1] "John"    "Sarah"   "Zach"    "Beth"    "Lachlan"
```

```r
df$Age  #Access and display the "Age" column
```

```
## [1] 35 28 33 55 43
```

The **$** syntax can also be used, along with a new name, to add a new column to an existing data frame, without having to overwrite the variable assigned with the original data frame, unlike with the *data.frame(.)* or *cbind(.)* command. Note that the overwriting process was not done previously with the *Coffee.Drinker* vector and therefore the vector is actually not a part of the data frame *df* at the moment.

```r
#Add the new variable, Coffee.Drinker, to df and recall the data frame
df$Coffee.Drinker <- c(TRUE,TRUE,FALSE,TRUE,FALSE); df
```

```
##      Name Age Gender Coffee.Drinker
## 1    John  35   Male           TRUE
## 2   Sarah  28 Female           TRUE
## 3    Zach  33   Male          FALSE
## 4    Beth  55 Female           TRUE
## 5 Lachlan  43   Male          FALSE
```

```r
#Add the new variable, Diabetes, to df and recall the data frame
df$Diabetes <- factor(c("Yes","No","No","No","Yes")); df
```

```
##      Name Age Gender Coffee.Drinker Diabetes
## 1    John  35   Male           TRUE      Yes
## 2   Sarah  28 Female           TRUE       No
## 3    Zach  33   Male          FALSE       No
## 4    Beth  55 Female           TRUE       No
## 5 Lachlan  43   Male          FALSE      Yes
```

Try examining your data frame with the following commands:

1. Structure: *str(.)*;
2. Class: *class(.)*;
3. Dimension: *dim(.)*;
4. Number of rows: *nrow(.)*;
5. Number of columns: *ncol(.)*; and
6. Names of the columns: *colnames(.)*.

The *str(.)* command is particularly informative as it provides a summary of the data frame and its data.

A *tibble* is a modern take on a data frame, without the annoying features. A tibble is defined using the *tibble(.)* command.

```
tib1 <- tibble(Name,Age,Gender,Coffee.Drinker); tib1
```

```
## # A tibble: 5 x 4
##   Name      Age Gender Coffee.Drinker
##   <chr>   <dbl> <fct>  <lgl>
## 1 John       35 Male   TRUE
## 2 Sarah      28 Female TRUE
## 3 Zach       33 Male   FALSE
## 4 Beth       55 Female TRUE
## 5 Lachlan    43 Male   FALSE
```

One of the ways that *tibble(.)* and *data.frame(.)* differ is how the data frames are printed to screen. If you recall a data frame in R, it will attempt print all the rows and columns until it reaches the maximum allowed. When you print a tibble, only the first ten rows, and all the columns that fit on one screen will be displayed and this makes it easier to work with large datasets. Furthermore, an abbreviated description of the column type is provided under the column names.

Another key difference is in subsetting. Subsetting a tibble will alway return a tibble by default, even if only one column is accessed. On the hand, when you access a single column in a data frame, the end result is either a vector or a factor. The latter is true if the column that is accessed is a factor. This difference often leads to codes in older packages not working properly due to mis-matching of data types.

Thankfully, we can convert from one structure to the other (and vice versa) to overcome this issue.

```
as.data.frame(tib1)  #Convert the tibble, tib1, to a data frame
```

```
##      Name Age Gender Coffee.Drinker
## 1    John  35   Male           TRUE
## 2   Sarah  28 Female           TRUE
## 3    Zach  33   Male          FALSE
## 4    Beth  55 Female           TRUE
## 5 Lachlan  43   Male          FALSE
```

```
as_tibble(df)  #Convert the data frame, df, to a tibble
```

```
## # A tibble: 5 x 5
##   Name      Age Gender Coffee.Drinker Diabetes
##   <chr>   <dbl> <fct>  <lgl>          <fct>
## 1 John       35 Male   TRUE           Yes
## 2 Sarah      28 Female TRUE           No
## 3 Zach       33 Male   FALSE          No
## 4 Beth       55 Female TRUE           No
## 5 Lachlan    43 Male   FALSE          Yes
```

Refer to the link here for more details regarding other differences between these two structures.

### 1.3.5 List

A list is a collection of data structures. It is the most flexible data structure in R in that each component of a list can be of different type, dimension or length to the other components. You can even store a list within a list. A list is created using the *list(.)* command.

Let us create a list consisting of a vector, a matrix and a data frame that were defined previously.

```
list1 <- list(c,Mat.A,df); list1
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##      v1 v2 v3
## [1,]  1  4  7
## [2,]  2  5  8
## [3,]  3  6  9
##
## [[3]]
##      Name Age Gender Coffee.Drinker Diabetes
## 1    John  35   Male           TRUE      Yes
## 2   Sarah  28 Female           TRUE       No
## 3    Zach  33   Male          FALSE       No
## 4    Beth  55 Female           TRUE       No
## 5 Lachlan  43   Male          FALSE      Yes
```

```
str(list1)  #examine the structure of the list and its components.
```

```
## List of 3
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "v1" "v2" "v3"
##  $ :'data.frame':    5 obs. of  5 variables:
##   ..$ Name          : chr [1:5] "John" "Sarah" "Zach" "Beth" ...
##   ..$ Age           : num [1:5] 35 28 33 55 43
##   ..$ Gender        : Factor w/ 2 levels "Female","Male": 2 1 2 1 2
##   ..$ Coffee.Drinker: logi [1:5] TRUE TRUE FALSE TRUE FALSE
##   ..$ Diabetes      : Factor w/ 2 levels "No","Yes": 2 1 1 1 2
```

To access the components of a list, we use the double square brackets [[.]].

```
list1[[1]]  #Access the 1st component, i.e. vector c
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
list1[[2]]  #Access the 2nd component, i.e. matrix Mat.A
```

```
##      v1 v2 v3
## [1,]  1  4  7
## [2,]  2  5  8
## [3,]  3  6  9
```

```
list1[[3]]  #Access the 3rd component, i.e. data frame df
```

```
##      Name Age Gender Coffee.Drinker Diabetes
## 1    John  35   Male           TRUE      Yes
## 2   Sarah  28 Female           TRUE       No
## 3    Zach  33   Male          FALSE       No
## 4    Beth  55 Female           TRUE       No
## 5 Lachlan  43   Male          FALSE      Yes
```

If the components in a list are labelled, then we can access them by using the **$** syntax as well.

```
#Create the same list, but each component is labelled
list1 <- list(VecC=c,MatA=Mat.A,DatFrame=df)

str(list1)  #Examine the structure of the list
```

```
## List of 3
##  $ VecC    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ MatA    : int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "v1" "v2" "v3"
##  $ DatFrame:'data.frame':    5 obs. of  5 variables:
##   ..$ Name          : chr [1:5] "John" "Sarah" "Zach" "Beth" ...
##   ..$ Age           : num [1:5] 35 28 33 55 43
##   ..$ Gender        : Factor w/ 2 levels "Female","Male": 2 1 2 1 2
##   ..$ Coffee.Drinker: logi [1:5] TRUE TRUE FALSE TRUE FALSE
##   ..$ Diabetes      : Factor w/ 2 levels "No","Yes": 2 1 1 1 2
```

```
list1$VecC  #Access the vector component
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
list1$MatA  #Access the matrix component
```

```
##      v1 v2 v3
## [1,]  1  4  7
## [2,]  2  5  8
## [3,]  3  6  9
```

```
list1$DatFrame  #Access the data frame component
```

```
##      Name Age Gender Coffee.Drinker Diabetes
## 1    John  35   Male           TRUE      Yes
## 2   Sarah  28 Female           TRUE       No
## 3    Zach  33   Male          FALSE       No
## 4    Beth  55 Female           TRUE       No
## 5 Lachlan  43   Male          FALSE      Yes
```

## 1.4 Coercion of Data Types

Recall that the elements of a vector or matrix must be of the **same type**. What would happen if we attempt to create a vector or matrix whose elements are of different types?

In such instances, the elements whose data type is the most flexible will be coerced into the least flexible data type. For the four data types that were discussed, the order of flexibility (least to most) is (1) *Character*, (2) *Numeric*, (3) *Integer* and (4) *Logical*.

Here are some examples.

```
c(1,"a")  #numeric value is coerced into a character
```

```
## [1] "1" "a"
```

```
c(TRUE,"a")  #logical value coerced into a character
```

```
## [1] "TRUE" "a"
```

```
c(TRUE,1)  #logical value is coerced into a numeric; 1 = TRUE, 0 = FALSE by default
```

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

```
#The 1st three elements are coerced into characters.
matrix(c(5,FALSE,4.6,"No"),nrow=2,ncol=2,byrow=FALSE)
```

```
##      [,1]    [,2] 
## [1,] "5"     "4.6"
## [2,] "FALSE" "No"
```

Sometimes coercions are done intentionally, and other times they are not. For the latter, it can be a frustrating experience if you are unaware that this is happening, and in particular, when this leads to coding errors later on.

## 2 Create, Manipulate and Summarise Data

In this section, you will be introduced to some of the basic and commonly used functions in R.This includes generating, importing, exporting, subsetting, summarising and plotting data.

### 2.1 Generating Numbers and Sequences

#### 2.1.1 Random numbers

```
#20 random numbers between 1 and 10 (inclusive)
x1 <- sample(1:10,size=20,replace=TRUE); x1
```

```
##  [1]  9 10  9  3 10  3  1  5  5  4  8  2  7  3  3 10  4 10  2 10
```

```
#20 random numbers between 5 and 15 from a Uniform distribution
x2 <- runif(20,5,15); x2
```

```
##  [1] 10.146141 13.603996 10.120195  8.484670  9.808949  9.899311 11.608314
##  [8] 10.578226 12.632671 12.877863 10.342623 14.600668  6.581835 10.508629
## [15] 12.130340  9.814027 14.612239 12.997693  9.080456 14.485755
```

```
#20 random numbers from a normal distribution with mean=12 and sd=3
x3 <- rnorm(20,mean=12,sd=3); x3
```

```
##  [1] 10.564230 13.409124 11.928950 11.806775  6.476582  7.134383 13.582144
##  [8]  8.787177 12.550854 15.752268 16.405761 16.338473 19.899509 11.739384
## [15] 12.759802 12.835594 16.644072  9.088407  8.747706 12.945742
```

**Note:** The numbers show here are likely to be different to what you will have as these are randomly generated numbers.

### 2.1.2 Sequences and Repeats

```
#A sequence from 1 to 20 with a 0.5 increment
x4 <- seq(1,20,by=0.5); x4
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
## [16]  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0 15.5
## [31] 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0
```

```
#Repeat the sequence {1,2,3,4,5} 5 times over
x5 <- rep(c(1:5),times=5); x5
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
#Repeat each element of the sequence {1,2,3,4,5} by 5 times
x6 <- rep(c(1:5),each=5); x6
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5
```

```
#Repeat each element of the sequence {1,2,3,4,5} by 3 times, and then repeat
#that sequence the 2nd time. Note that "each" has precedence over "times".
x7 <- rep(c(1:5),times=2,each=3); x7
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

### 2.1.3 Alphabets

```
letters[1:10]     #List first 10 letters (in lower case) of the alphabet
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
LETTERS[1:10]     #List first 10 letters (in UPPER case) of the alphabet
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

## 2.2 Importing and Exporting Data

The easiest way to import a dataset into R is by using the **Import Dataset** option under the **Environment** tab in the top-right panel of your R Studio session (see Figure 2). With this method, you can import data in various formats, including *CSV*, *Excel* and *SPSS* files. Whenever possible, try and work with *CSV* files.

If you have not already done so, download the *ElderlyPopWA.csv* dataset from Blackboard to your working directory. Then import this data into R using this approach with either the "From text (base)" or "From text (readr)" option. The first option imports the data and stores it as a data frame, whereas the 2$^{\text{nd}}$ option stores it as a tibble.

Once the dataset is imported, R Studio will open it (courtesy of the *View(.)* command) in a new tab for you to view. Furthermore, the following command line(s) (depending on which option you went

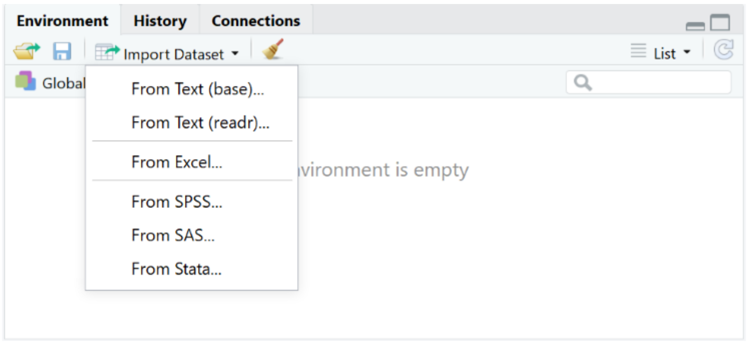


Figure 2: Importing Data

with) will appear in the **Console**. Notice that the 2nd approach requires the *readr* package to be loaded first.

```
#If using the 1st option
ElderlyPopWA <- read.csv("ElderlyPopWA.csv")
View(ElderlyPopWA)

#If using the 2nd option
library(readr)
ElderlyPopWA <- read_csv("ElderlyPopWA.csv")
View(ElderlyPopWA)
```

Copy the above command lines and paste them into your R script. This way, you do not need to browse for your data file again if you ever need to restart your session and/or re-run your code.

Suppose you have amended the dataset and would like to export the updated data frame/tibble to a new *CSV* file. To do this, we can use either the *write.csv(.)* or *write_csv(.)* command.

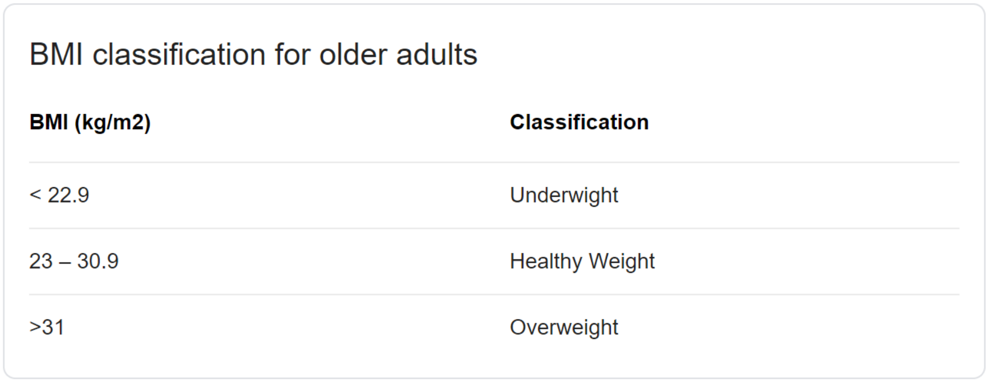
```
#Method 1
write.csv(ElderlyPopWA,  #Name of the data frame/tibble to be exported
          "ElderlyPopWA_updated.csv"  #Name of the new CSV file to write the data to.
          )

#Method 2
write_csv(ElderlyPopWA,  #Name of the data frame/tibble to be exported
          "ElderlyPopWA_updated.csv"  #Name of the new CSV file to write the data to.
          )
```

## 2.3 Data Manipulation

In pre-processing of data, we often want to categorise a group of individuals (which may be people, animals or objects) based on certain feature(s). Doing so often makes it easier to analyse the data and interpret the subsequent results.

For example, suppose we wish to categorise the elderly female partipants (from the ElderlyPopWA dataset) into their respective BMI classes (see Figure 3).



BMI classification for older adults

| BMI (kg/m2) | Classification |
|---|---|
| < 22.9 | Underwight |
| 23 – 30.9 | Healthy Weight |
| >31 | Overweight |

Figure 3: BMI classification

The *cut(.)* command will allow us to do this. Note that by default, the left side of the interval is open, and the right side is closed, i.e. (a,b]. Type and run the command *?cut* for help if you want to know how to change this setting.

```
#Create BMI categories for the elderly female participants

mBMI <- max(ElderlyPopWA$BMI)  #maximum BMI value within the sample

ElderlyPopWA$BMI.class <- cut(ElderlyPopWA$BMI,
                              breaks=c(0,23,31,mBMI), #Set the intervals for the classes
                              labels=c("Underweight",
                                       "Healthy_Weight",
                                       "Overweight")) #Add labels to the classes
```

**Exercise:** Within the sample, how many elderly females are in each of the three BMI classes?

In many instances, we are often interested in having a closer examination of a sub-sample of individuals based on particular group(s) they belong to (e.g. those of a particular gender, ethnicity, income bracket, etc), or by some thresholds (e.g. those who are 35 years old or younger). The *subset(.)* command allows us to perform this task.

```
#Subsetting by a categorical variable

#Underweight individuals only
Underweight <- subset(ElderlyPopWA,BMI.class=="Underweight");

#Under and overweight individuals
Unhealthy.weight <- subset(ElderlyPopWA,BMI.class=="Underweight"|BMI.class=="Overweight");

#Subsetting by a condition on a continuous variable
Age.LT75 <- subset(ElderlyPopWA,Age<75);  #Select those under 75 years of age.
```

Make sure you view your subsets, i.e. *View(.)*, to ensure you have done this correctly.

We can also use the *which(.)* command for this purpse. This command outputs the indices of the elements for this the set condition is true, which we can set as the rows of data that we wish to select.

```
#Underweight individuals only
Underweight <- ElderlyPopWA[which(ElderlyPopWA$BMI.class=="Underweight"),]

#Select those under 75 years of age.
Age.LT75 <- ElderlyPopWA[which(ElderlyPopWA$Age<75),]

#Select those that are not under 75 years of age.
Age.GTE75 <- ElderlyPopWA[-which(ElderlyPopWA$Age<75),]
```

## 2.4 Descriptive Statistics

Once your dataset is pre-processed and cleaned, then it is a good idea to obtain some summary statistics on the variables to get a sense of how the data are distributed.

### 2.4.1 Summarising Continuous Data

When describing a *continuous* variable (e.g. age, weight, height, etc), you should always address three aspects; the (1) centre, (2) spread and (3) shape.

```
#Measures of centre
mean(ElderlyPopWA$Age)  #Mean (i.e. average) age
```

```
## [1] 74.31748
```

```
median(ElderlyPopWA$Age)  #Median age
```

```
## [1] 74.08767
```

```
#Measures of spread
sd(ElderlyPopWA$Age)  #Standard deviation of age
```

```
## [1] 2.596677
```

```
range(ElderlyPopWA$Age)  #range of age, i.e. minimum and maximum
```

```
## [1] 70.30411 79.96712
```

```
IQR(ElderlyPopWA$Age)  #Interquartile range
```

```
## [1] 4.158219
```

```
fivenum(ElderlyPopWA$Age)  #Five-number summary, i.e. min, Q1, Q2, Q3, max.
```

```
## [1] 70.30411 72.09589 74.08767 76.28356 79.96712
```

Skewness and kurtosis are two commonly used measures of shape. Skewness is a measure of symmetry where,

1. Skewness $= 0$, distribution is symmetrical;
2. Skewness $< 0$, distribution is left- or negatively-skewed, i.e. longer left tail;
3. Skewness $> 0$, distribution is right- or positively-skewed, i.e. longer right tail.

Kurtosis is a measure of "tailness" in a distribution relative to a normal distribution.

1. Kurtosis = 3, tails are that of a normal distribution, i.e. mesokurtic;
2. Kurtosis < 3, tails are comparatively shorter than a normal distribution, i.e. platykurtic
3. Kurtosis > 3, tails are comparatively longer than a normal distribution, i.e. leptokurtic.

To compute these two measures in R, we require the **moments** package.

```
#install.packages("moments");  #De-comment to install the package
library(moments)  #Load the pacakge

#Measures of shape
skewness(ElderlyPopWA$Age)
```

```
## [1] 0.2670584
```

```
kurtosis(ElderlyPopWA$Age)
```

```
## [1] 2.021052
```

The above values indicate that the variable Age is approximately symmetrically distributed with a slight positive skewness, and is platykurtic.

**Exercise:** Standardised the Age variable (i.e. (x-mean)/sd), add it to the existing data frame (call it z_Age), and summarise it.

There are other functions that allow you to obtain the summary statistics in a more efficient, in particular for large data frames.

```
colMeans(data.frame(ElderlyPopWA[,2:8]))  # Column means
```

```
##         Age         BMI       Waist         Hip      Tricep    ArmGirth 
##    74.31748    26.62886    88.41351   104.32973    28.86149    31.24932 
## Pc_Body_Fat 
##    38.29458
```

Note that *colMeans(.)* function only works for data frames. If you data is stored as a tibble (as in the case here), then you will need to covert it to a data frame beforehand.

Another way is by using the *apply(.)* function.

```
apply(ElderlyPopWA[,2:8],  # Data
      MARGIN=2,  # Apply the function in a column-wise manner. MARGIN = 1 implies row-wise.
      FUN=mean  # The function to be applied.
      )
```

```
##         Age         BMI       Waist         Hip      Tricep    ArmGirth 
##    74.31748    26.62886    88.41351   104.32973    28.86149    31.24932 
## Pc_Body_Fat 
##    38.29458
```

The *apply(.)* functionis much more versatile here as you can apply other functions to your data frame besides *mean(.)*. Suppose now, we wish to determine the standard deviations for a data frame. All we need to do here is set **FUN** to *sd*.

```
apply(ElderlyPopWA[,2:8],  # Data
      MARGIN=2,  # Apply the function in a column-wise manner. MARGIN = 1 implies row-wise.
      FUN=sd  # The function to be applied.
      )
```

```
##         Age        BMI      Waist        Hip     Tricep   ArmGirth
##    2.596677   3.925148   9.081308   8.987062   7.654855   3.388104
## Pc_Body_Fat
##    5.302573
```

**Exercise:** Using the *apply(.)* function, find the five number summary for each of the continuous variables in the *ElderlyPopWA* dataset.

### 2.4.2 Summarising Discrete and Categorical Data

Frequencies and proportions are more appropriate When describing categorical or ordinal data. The latter is often treated and described like a continuous variable, but this should only be the case if the range of finite values is relatively large; for example, your final grade in the unit (as recorded in the system), which ranges between 0 and 100 and does not include any decimal value. If, say for example, a variable described by a 5-point Likert scale is to be summarised, then frequencies and proportions should be reported.

To obtain these measures, we use the *table(.)* and *prop.table(.)* commands.

```
BMI.freq <- table(ElderlyPopWA$BMI.class); BMI.freq  #Number of participants in each BMI class
```

```
##
##    Underweight Healthy_Weight     Overweight
##             27            100             21
```

```
BMI.prop <- prop.table(BMI.freq); BMI.prop  #Proportions of sample for each BMI class
```

```
##
##    Underweight Healthy_Weight     Overweight
##      0.1824324      0.6756757      0.1418919
```

We can also use the *table(.)* command to crosstabulate two or more categorical/ordinal variables, and then use *prop.table(.)* to compute the marginal proportions.

```
#Create another categorical variable, i.e. age group.
ElderlyPopWA$Age.grp <- cut(ElderlyPopWA$Age,c(0,74.99,100),
                            labels=c("<75years","75+years"))

#Cross tabulate age group with BMI
tab <- table(ElderlyPopWA$Age.grp,ElderlyPopWA$BMI.class); tab  #2 by 2 contigency table
```

```
##
##            Underweight Healthy_Weight Overweight
##   <75years          16             62         11
##   75+years          11             38         10
```

```
prop.table(tab,1)  #Proportions by row, i.e. within each age group
```

```
##
##             Underweight Healthy_Weight Overweight
##   <75years   0.1797753      0.6966292  0.1235955
##   75+years   0.1864407      0.6440678  0.1694915
```

```
prop.table(tab,2)  #Proportions by column, i.e. within each BMI class
```

```
##
##             Underweight Healthy_Weight Overweight
##   <75years   0.5925926      0.6200000  0.5238095
##   75+years   0.4074074      0.3800000  0.4761905
```

```
tab/sum(tab)  #Proportions relative to overall sample size
```

```
##
##             Underweight Healthy_Weight Overweight
##   <75years  0.10810811     0.41891892 0.07432432
##   75+years  0.07432432     0.25675676 0.06756757
```

### 2.4.3 Summarising Continuous Data Across Sub-Groups

In addition to summarising the characteristics or features of a sample as a whole, we often want to summarise and compare the individuals across variaous sub-groups. We can split the data initially using the *subset(.)* command and then compute the relevant summary statistics, however, there is a quicker way to do this via the *aggregate(.)* command.

```
#Summarise the waist circumference of the individuals across the three BMI classes

#Compute the mean
aggregate(Waist~BMI.class,  #Formula to subset the Waist data by BMI classes
          data=ElderlyPopWA,  #Define the relevant data
          FUN=mean)  #Define the function to compute the desired statistic(s), i.e. mean
```

```
##        BMI.class     Waist
## 1    Underweight  76.78519
## 2 Healthy_Weight  88.96800
## 3     Overweight 100.72381
```

```
#Compute the standard deviation
aggregate(Waist~BMI.class,data=ElderlyPopWA,FUN=sd)
```

```
##        BMI.class    Waist
## 1    Underweight 5.717362
## 2 Healthy_Weight 6.081282
## 3     Overweight 6.115383
```

We can also compute the summary statistics across the combination of two or more categorical/ordinal variables.

```
#Summarise the waist circumference of individuals across all combinations between
#BMI classes and age groups.
```

```
#Compute the mean
aggregate(Waist~BMI.class+Age.grp,data=ElderlyPopWA,FUN=mean)
```

```
##        BMI.class  Age.grp     Waist
## 1    Underweight <75years  77.01250
## 2 Healthy_Weight <75years  88.98871
## 3     Overweight <75years 102.80909
## 4    Underweight 75+years  76.45454
## 5 Healthy_Weight 75+years  88.93421
## 6     Overweight 75+years  98.43000
```

```
#Compute the standard deviation
aggregate(Waist~BMI.class+Age.grp,data=ElderlyPopWA,FUN=sd)
```

```
##        BMI.class  Age.grp    Waist
## 1    Underweight <75years 5.188175
## 2 Healthy_Weight <75years 6.227915
## 3     Overweight <75years 6.333317
## 4    Underweight 75+years 6.664138
## 5 Healthy_Weight 75+years 5.916412
## 6     Overweight 75+years 5.232387
```

We can also use the *tapply(.)* command to obtain these statistics.

```
#Mean waist circumference by BMI class
tapply (ElderlyPopWA$Waist,
        INDEX=ElderlyPopWA$BMI.class,
        FUN=mean)
```

```
##    Underweight Healthy_Weight     Overweight
##       76.78519       88.96800      100.72381
```

```
#Mean waist circumference by BMI class and age group
tapply (ElderlyPopWA$Waist,
        INDEX=list(ElderlyPopWA$BMI.class,ElderlyPopWA$Age.grp),
        FUN=mean)
```

```
##                 <75years 75+years
## Underweight     77.01250 76.45454
## Healthy_Weight  88.98871 88.93421
## Overweight     102.80909 98.43000
```

**Exercise:** Determine the mean and standard deviation of hip, triceps and arm girth across all BMI classes.

You can also use the data manipulation and summary functions from the **dplyr** package (which is included within **tidyverse** package), and some may find them easier to implement. A copy of the cheat sheet for these functions is available by clicking here. There are also plenty of online resources available to assist you, such as https://dplyr.tidyverse.org/.

## 3 Piping

In modern R scripts, you will often find codes written with the %>% syntax. This is the most popular pipe operator in R. Pipes allow us to clearly express a sequence of multiple operations, without having to nest these functions within each other. The latter will often times involve multiple sets of parentheses and can make the code hard to read and understand. This is where pipes can help.

**Note:** In order to use pipes, we need to installed and load the **magrittr** package, or we can just call the **tidyverse** package, i.e. *library(tidyverse)*, as it includes the **magrittr** package.

Let us return to the *ElderlyPopWA* dataset. **Suppose that we wish to find the proportions of individuals in each of the BMI classes to 3 decimal places.** We can do just that with a single line of code with nested commands (see below), however, another person reading this code may need some time before he/she can determine its purpose.

```
round(prop.table(table(ElderlyPopWA$BMI.class)),3)
```

```
##
##    Underweight Healthy_Weight     Overweight
##          0.182          0.676          0.142
```

Another way to do this is as follows.

```
BMI.freq <- table(ElderlyPopWA$BMI.class);  #Number of participants in each BMI class
BMI.prop <- prop.table(BMI.freq);  #Proportions of sample for each BMI class
round(BMI.prop,3)  #Display the proportions to 3 d.p.
```

```
##
##    Underweight Healthy_Weight     Overweight
##          0.182          0.676          0.142
```

There is nothing wrong with the above approach as the sequence of operations is clear. The issue here, in partilcuar in loops (to be introduced later), is that a new variable is created to store the value at each of the first two steps, thus requiring memory space. Pipes are a way for us to avoid these consequences. Here is how we would do this.

```
ElderlyPopWA$BMI.class %>%
  table(.) %>%
  prop.table(.) %>%
  round(.,digits=3)
```

```
## .
##    Underweight Healthy_Weight     Overweight
##          0.182          0.676          0.142
```

**Note:** The dot "." acts as an argument placeholder for the object created from the previous step. By default, the created object is always assumed to be the first argument to be passed to the command in the proceeding step. Therefore, it is actually not necessary to include the dot if this applies to your situation.

That is, we can easily write this as:

```
ElderlyPopWA$BMI.class %>%
  table() %>%
  prop.table() %>%
  round(digits=3)
```

```
## .
##    Underweight Healthy_Weight     Overweight
##          0.182          0.676          0.142
```

In fact, you can even exclude the brackets if there is only one argument to pass through. Having said this, it is probably better to include . as a beginner user of pipes and to avoid any confusion later.

The use of . is particular useful if you wish to adjust the argument placeholder. Suppose that we want to diplay $\pi$ to 5 decimal places.

```
5 %>% round(pi,digits=.)
```

```
## [1] 3.14159
```

Here, notice how the value of 5 is passed as the 2$^{\text{nd}}$ argument using ., instead of first.

There are other forms of pipes that you may find useful. Click here if you want to find out more.

## 4 Functions and Control Structures

Functions are used to encapsulate a sequence of commands that need to be executed multiple times, but perhaps under slightly different conditions. Functions allow one to automate common tasks in a more powerful and general way than copy-and-pasting, which can often lead to incidental mistakes (i.e. updating a variable name in one place, but not in another). Further, functions allows a developer to create an interface to the code, and communicate to the user the most important aspects of the code, rather than having to know every detail of the code.

Control structures in R allows you to control the flow of execution of a series of R commands, and not having run the same set of commands repeatedly. Control structures are typically, though not necessarily, used inside of functions. The inclusion of control structures and functions will often improve the readability of the code.

In this section, you will learn how to write your own function. You will be also be introduced to two of the more commonly used control structures (if and else and for loops). Other forms of control structures are discussed but not emphasized.

### 4.1 Functions

Functions are defined using the **function(.)** directive. Like anything else in R, functions are stored as R objects. Let's begin with a simple function that prints **Hello! My names is [insert your name here]** when called. We will call this function **Greeting**.

```r
Greeting <- function()
{
  print("Hello! My name is XXXX")
}
```

The command(s) between the curly brackets {} form the body of a function. For the above function, it is just a simple *print(.)* command. When you execute the above code, the **Greeting** function is created (check your Global Environment panel). To call or execute a function, just type the function name along with the empty brackets, and run.

```r
Greeting()
```

```
## [1] "Hello! My name is XXXX"
```

The next aspect of a basic function in R is the function *arguments*. These are the options that you can specify to the user that the user may explicity set for the function. The argument list is defined within the brackets of the **function(.)** directive. Values assigned to an argument list are passed to, and are typically used by the code within the body of a function. Otherwise there is no point in having an argument list.

Suppose we wish to create a function that adds 3 to any value that we pass into the function and outputs the sum.

```r
add3 <- function(x)
{
  x + 3
}
```

Here, a user can assign any value to the argument **x** which function will then add 3 to. The sum is then outputted as the function value.

```r
add3(5)
```

```
## [1] 8
```

```r
add3(10)
```

```
## [1] 13
```

```r
add3(15)
```

```
## [1] 18
```

This is the basis of how all the built-in functions in R are created, i.e. *mean(.)*, *sd(.)*, etc.

Let's step it up a bit and create a function that finds the sum, product and ratio between any two values, say **x** and **y**. The two values are to be specified by the user.

```
add_mult <- function(x,y)
{
  sum_xy <- x + y  #Calculate the sum of x and y
  prod_xy <- x * y  #Calculate the product of x and y
  ratio_xy <- x / y  #Calculate the ratio of x to y
}
```

Let's assign the values, 3 and 4 to **x** and **y** respectively, and pass them into the function.

```
x <- 3; y <- 4;

add_mult(x,y) %>% #Call the function
  print()  #Print the value of the function
```

```
## [1] 0.75
```

Notice how ***only*** the ratio of **x** to **y** is given, i.e. 0.75. This is because, by default, a function in R will always output the result of the last line of code within the body of a function. In this case, the result of the final line is the creation of the variable **ratio_xy**, which contains the ratio of **x** to **y**. If we want the function to return all three measures, then we just need to add a command (as the final line of code) that concatenates them.

```
add_mult <- function(x,y)
{
  sum_xy <- x + y  #Calculate the sum of x and y
  prod_xy <- x * y  #Calculate the product of x and y
  ratio_xy <- x / y  #Calculate the ratio of x to y

  c(sum_xy,prod_xy,ratio_xy) #A vector consisting of the sum, product and ratio
}
```

Here, the value of the function is a vector with 3 elements, the first of which is the sum, the second is the product and the third is the ratio of **x** and **y**.

```
add_mult(x=3,y=4)
```

```
## [1]  7.00 12.00  0.75
```

R function arguments can be matched positionally or by name. With the above command, the arguments are matched by name. In this instance, the order of the argument does not matter. We can specify the **y** argument before **x** if we want, and the result will be the same.

```
add_mult(y=4,x=3)
```

```
## [1]  7.00 12.00  0.75
```

Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc. For example,

```
add_mult(3,4)
```

```
## [1]  7.00 12.00  0.75
```

Here, the first value, i.e. 3, is assigned to the first argument (i.e. **x**) and the second value (i.e. 4) is assigned to the 2nd argument.

Note that with this approach, the order in which you specify the values does matter. For the previous code, swap the value order of the two values around and see what the difference is between their output.

This concludes the introduction to functions in R. Click here if you would like to find out more about functions in R and their nuances that are not discussed here.

**Exercise:** Write a function that accepts a vector, say **vec_x** of length greater than 2, and returns the mean and standard deviation of that vector.

## 4.2 Control Structures

There are a number of ways in which we can control the flow of execution of R commands. The common ones include:

1. **if-and-else**: used to test a condition and execute based on the condition;
2. **for** loop: used to execute a loop for a fixed number of iterations;
3. **while** loop: used to execute a loop while a condition is true;
4. **break**: used to break the execution of a loop;
5. **next**: used to skip an iteration of a loop;
6. **return**: used to exit a function, and return a given value.

Care needs to be taken when using a **while** loop. When the condition no longer holds true, a **while** loop then permits exit. However, if you are not careful and the condition is never untrue, then then loop will never end. The for loop, on the other hand, will stop after a set number of iterations. Furthermore, what is done with a while loop can be achieved with a for loop. Given this, the **while** loop will not be discussed further here. We will also not discuss **next** and **return** controls as they are not often used. Click here for more information about **while** loops and other control structures in R.

### 4.2.1 If-and-Else

The **if-and-else** control structures are used to execute a set of commands only when a particular condition is satisfied, and performs the alternate set of commands if the conditions is not met. The **else** clause is not mandatory, and would apply in intances where no action is taken if the condition is not met.

To start, let's create an **if** statement that checks if a given integer $> 0$ is odd, and prints a statement that indicates this if it is true.

```
x <- 5  # Set an arbitrary x value

#If statement
```

```r
if (x%%2 != 0)
{
  cat(paste(x," is an odd integer\n",sep="")) #print command
}
```

```
## 5 is an odd integer
```

**Note:** %% is the **modulus** operator and != is the **not equals to** logical operator.

If **x** is an even number, then no action is taken. Try this out and change **x** to, say 22, and re-run the above code.

Let's now extend the above **if** statement to include *else*. Suppose now we want the code to print a statement that an integer > 0 is odd if it is true, otherwise print a statement that the integer is even.

```r
x <- 10  # Set an arbitrary x value

if (x%%2 != 0)
{
  cat(paste(x," is an odd integer\n",sep=""))
}else{
  cat(paste(x," is an even integer\n",sep=""))
}
```

```
## 10 is an even integer
```

We can also include multiple conditions in an **if-and-else** statement by using **else if**. For example, suppose you wish to know your grade (i.e. N, P, C, D and HD) for a unit based on your final score.

```r
score <- 65  #Set an arbitrary score

if (score < 0)
{
  print("Invalid score!")  #Cannot have a negative score
}else if (score < 50){
  print("Your final grade is N.")
}else if (score <60){
  print("Your final grade is P.")
}else if (score <70){
  print("Your final grade is C.")
}else if (score <80){
  print("Your final grade is D.")
}else if (score <=100){
  print("Your final grade is HD.")
}else{
  print("Invalid score!") #Cannot have a score greater than 100%
}
```

```
## [1] "Your final grade is C."
```

### 4.2.2 For Loop

A **for** loop executes repetitive code statements for a definite number of iterations. Loops are commonly used to iterate over the elements of vectors, columns (or rows for matrices) of data frames and components of lists.

Let's begin with a loop that repeats itself 10 time over.

```
#For loop
for (I in 1:10)
{
  print(I)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

The variable **I** in the for loop acts as a counter that starts at 1, the command(s) (in this instance it is just a print command) within the loop are then executed. **I** then increases by 1 to be 2, and the commands are executed again. This process continues until **I** equals to 10. Note that **I** is just a variable name, and you can quite easily use another name. Furthermore, the count does not necessary have to start at 1.

We often take advantage of the counter and incorporate it into the commands and run them systematically through, say a data frame or list. For example, suppose we want to generate the means for the continous measurements (i.e. columns 2 to 8) in the ElderlyPopWA dataset. We can systematically move from one column to the next and generate their means using a for loop.

```
#Create a copy of ElderlyPopWA and store it as a data frame
dat <- data.frame(ElderlyPopWA)

#For loop that starts at 2
for (J in 2:8)
{
  mean(dat[,J]) %>%  #Find the mean of the jth column
    print()  #Print the mean
}
```

```
## [1] 74.31748
## [1] 26.62886
## [1] 88.41351
## [1] 104.3297
## [1] 28.86149
## [1] 31.24932
## [1] 38.29458
```

**If-and-else** statements are often defined within a for loop to control the execution of certain sets of commands. For example, suppose we want to print out all numbers between 1 and 100 that are divisible by 7 only.

```r
#For loop
for (I in 1:100)
{
  #If statement to check if the number is divisible by 7
  if (I%%7 == 0)
  {
    print(I)
  }
}
```

```
## [1] 7
## [1] 14
## [1] 21
## [1] 28
## [1] 35
## [1] 42
## [1] 49
## [1] 56
## [1] 63
## [1] 70
## [1] 77
## [1] 84
## [1] 91
## [1] 98
```

**If-and-else** statements in for loops are often used to break from the loop when a condition is met. For example, suppose we want to print out the first 10 numbers between 500 and 800 that are divisible by 13.

```r
count <- 0   #Set the count of numbers divisible by 13 to 0

for (I in 500:800)
{
  #1st if statement: Checks if the number is divisible by 13
  if (I%%13 == 0)
  {
    print(I)  #Print the number
    count <- count + 1   #Add 1 to count
  }

  #2nd if statement: Checks if the count of numbers divisible by 13 is 10.
  #                  If so, break from the for loop
  if (count==10)
  {
    break
  }
}
```

```
## [1] 507
## [1] 520
## [1] 533
## [1] 546
## [1] 559
## [1] 572
## [1] 585
## [1] 598
## [1] 611
## [1] 624
```

THE END