Week 1 Cheat Sheet

Statistics and Data Analysis with R

Course Link: https://www.coursera.org/learn/statistics-and-data-analysis-with-r/

Charlie Nuttelman

Here, I provide the functions in R required to perform various calculations in Week 1 of the course. The headings represent the screencasts in which you will find those concepts and examples.

Basic Calculations in R

We can use R to perform basic calculations, much like a calculator. If you've used any other programming language or scientific calculator, you will be familiar with order of operations. The following bullet points and examples will show you important aspects of basic calculations in R:

- The assignment operator in R is **<-** (less than sign followed by a hyphen). In many other programming languages, the assignment operator is simply the equal sign. In R, the equal sign will work, but it is preferable to use `<-`.
- The assignment operator takes whatever is on the right side of it and places it into the variable or object that is on the left side of it.
- For example, if we want to assign the value of 5 to the variable x, we can use: `x <- 5`
- Similarly, if we want to assign the value of 10 to the variable y, we can use: `y <- 10`
- It would <u>not</u> make sense to assign the value of y to the variable 4: `4 <- y`
- If we want to assign the value that stored in variable b to variable a, we can use: `a <- b`
- If we want to assign the sum of vector **z** to the variable p, we can use: `p <- sum(z)`
- We can assign the sum of variables x and y to the variable z: `z <- x+y`
- If you just type x <- 5 into the Console, there is no "echo" that is characteristic of many other programming languages:

```
> x <- 5
>
```

- If at any point you want to display or print the value of a variable in the Console, you can just type in the variable name or use the `print()` function:

```
> x
[1] 5
> print(x)
[1] 5
```

- You can also go up to the Environment pane in the upper right corner to see what the current variables are as well as their current values.

- Order of operations dictate which mathematical calculations occur before others. In R, the following characters are used for mathematical calculations and they are listed in the order of operations (from highest to lowest precedence):
    - Parentheses: ( )
    - Exponentiation: ^ or **
    - Multiplication: *
    - Division: /
    - Addition: +
    - Subtraction: -
- For example, `2+3*4` (=14) is not the same as `(2+3)*4` (=20)
- As another example, `5+6^2/3` (=17) is not the same as `(5+6)^2/3` (=40.33)
- `rm(x)` can be used to clear the variable/value of x from the Environment pane.
- To clear all variables in the Environment pane, you can click on the broom in the Environment pane:
  
- To refresh/clear everything from the Console without clearing the variables/values from the Environment pane, you can click in the Console then use Ctrl-L.
- A vector is a single variable that has multiple values. We can create a vector in R using the `c()` function, which concatenates (joins) values into a single vector.
- For example: `x <- c(10,20,30,40,50,60,70)` or `y <- c(1,2.5,3.8,3.4,4.8,5.1,6.2)`
- (You will learn more about how to reference items in a vector in the "Vectors and Matrices" screencast.)

Using Script Files

Script files (.R files) are nice because multiple lines of code can be placed into the file without actually executing those lines. Then, lines can be executed as needed or executed all at once. Additionally, script files can be used to store large code blocks that are used frequently, and the files can be emailed to others to use.

Script files can be created (File → New File → R Script) and edited in the Scripts pane. When lines of code are entered into the Scripts pane, they are not automatically executed when Enter is pressed. Instead, to run a line of code in the Scripts pane, one must place the cursor on the line that is to be run and Ctrl-Enter must be pressed. To execute multiple lines of code, those lines must be highlighted then Ctrl-Enter can be pressed. In either case, the lines are executed in the Console and variable values stored in the Environment pane.

We can toggle back and forth between the Console and Scripts pane using Ctrl-1 (Scripts pane) and Ctrl-2 (Console).

You will get a lot of experience using script files in this course, especially if you purchase the Course Certificate and can download the .R starter files each week.

Vectors and Matrices

The following examples will show you how to create, edit, and work with vectors and matrices:

- `d <- 5` creates a vector of size 1; all variables in R, by default, are vectors.
- `x <- c(12,17,24,8,22)` creates a vector:
```
> x <- c(12,17,24,8,22)
> x
[1] 12 17 24  8 22
```
  `c` stands for "concatenate", which is fancy for "join". By default, vectors created using `c()` are neither row nor column vectors.
- We can determine the length of a vector using the length function: `length(x)` results in 5
- `x[1]` outputs the first element of the vector `x`, which is 12
- `x[4]` outputs the fourth element of the vector `x`, which is 8.
- `x[2:3]` outputs elements 2 through 3 (2 and 3 in this case) of the vector `x`, which is a vector comprised of 17 and 24:
```
> x[2:3]
[1] 17 24
```
- `x[-2]` outputs the vector `x` with the second element removed:
```
> x[-2]
[1] 12 24  8 22
```
- `x[-c(3,4)]` removes the 3rd and 4th elements of vector `x`:
```
> x[-c(3,4)]
[1] 12 17 22
```
- `rev(x)` reverses the order of vector `x`:
```
> rev(x)
[1] 22  8 24 17 12
```
- We can create a vector of elements between a starting value and an ending value using the colon (:) symbol; importantly, the spacing is either 1 if the starting value is less than the ending value or -1 if the starting value is greater than the ending value:
```
> v <- 1:10
> v
 [1]  1  2  3  4  5  6  7  8  9 10
> w <- 20:1
> w
 [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```
- The `seq` function (base library) can be used to create a sequence when the increment is not necessarily 1:
```
> s <- seq(1,3,1)
> s
[1] 1 2 3
```
- The `rep` function repeats a smaller vector a specified number of times:
```
> rep(s,5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```
- We can also create vectors of strings:
```
> dwarfs <- c("Sleepy","Dopey","Doc","Grumpy","Happy","Bashful","Sneezy")
> dwarfs
[1] "Sleepy"  "Dopey"   "Doc"     "Grumpy"  "Happy"   "Bashful" "Sneezy"
```

- If we want to create a vector of all dwarfs except for Grumpy, we can use the following:

```
> dwarfs <- dwarfs[dwarfs!="Grumpy"]
> dwarfs
[1] "Sleepy"  "Dopey"   "Doc"     "Happy"   "Bashful" "Sneezy"
```

- Alternatively:

```
> dwarfs <- dwarfs[-4]
> dwarfs
[1] "Sleepy"  "Dopey"   "Doc"     "Happy"   "Bashful" "Sneezy"
```

- If we want to add back Grumpy, we can concatenate the new **dwarfs** vector with "Grumpy":

```
> dwarfs
[1] "Sleepy"  "Dopey"   "Doc"     "Happy"   "Bashful" "Sneezy"  "Grumpy"
```

  (Note that the order has been changed from the original **dwarfs** vector.)

- To create a matrix from individual vectors (`x`, `y`, and `z`), we can use the `matrix` function. By default, the `matrix` function just creates one long column vector:

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> z <- c(7,8,9)
> A <- matrix(c(x,y,z))
> A
      [,1]
 [1,]    1
 [2,]    2
 [3,]    3
 [4,]    4
 [5,]    5
 [6,]    6
 [7,]    7
 [8,]    8
 [9,]    9
```

- The `length` function will output the total number of elements in a vector or matrix:

```
> length(A)
[1] 9
```

- The `dim` function by itself will specify the dimensions (rows and columns) of a vector or matrix:

```
> dim(A)
[1] 9 1
```

- We can permanently change the above vector **A** to a matrix of specified dimensions using the `dim` function with the assignment operator:

```
> dim(A) <- c(3,3)
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

  (Note that the items have been placed column-wise into matrix **A**.)

- We can see that the dimensions have changed:

```
> dim(A)
[1] 3 3
```

- We can specify the size of the matrix formed by the `matrix` function using `nrow` and `ncol` inside the matrix function:

```
> B <- matrix(c(x,y,z),nrow=3,ncol=3)
> B
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

  (This creates a 3x3 matrix. Note that the items have also been placed column-wise into matrix **A**.)
- If we want to place the items row-wise into the matrix, we can use "`byrow=TRUE`" inside the matrix function:

```
> C <- matrix(c(x,y,z),nrow=3,ncol=3,byrow=TRUE)
> C
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

- Let's revisit matrix **A**:

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

- To extract the 3rd item from matrix **A**, we can use:

```
> A[3]
[1] 3
```

- As another example, to extract the 7th item from matrix **A**, we can use:

```
> A[7]
[1] 7
```

  Note that if there is only one item inside the square brackets, this refers to the index number in the matrix. To output `A[n]`, we start in the first column and count down (top to bottom), proceed to the next column and count down (top to bottom), etc., until we are at the nth item of the matrix.
- If we wish to display only a specific row of a matrix, we can specify the row number in square brackets followed by a comma (empty column number):

```
> A[3,]
[1] 3 6 9
```

  This can be interpreted as row 3 and all columns of matrix **A**. Note that the output is a vector.
- If we wish to display only a specific column of a matrix, we can leave row empty followed by a comma and the column index number inside square brackets:

```
> A[,2]
[1] 4 5 6
```

  This can be interpreted as all rows of column 2 of matrix **A**. Note that the output is a vector.

- To specify a range of rows or columns, we can use the colon (:):

```
> A[2:3,1:2]
     [,1] [,2]
[1,]    2    5
[2,]    3    6
```

  This can be interpreted as rows 2 to 3 and columns 1 to 2 of matrix **A**.
- As another example, `A[c(1,3),]` refers to all columns of rows 1 and 3 of matrix **A**:

```
> A[c(1,3),]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    3    6    9
```

- As one more example, `A[c(1,3),2:3]` refers to rows 1 and 3 but only columns 2 and 3 of matrix **A**:

```
> A[c(1,3),2:3]
     [,1] [,2]
[1,]    4    7
[2,]    6    9
```

- We can also create matrices using `cbind` and `rbind`. First, let's show `cbind`, which takes vectors and adds them as columns into a new matrix:

```
> x
[1] 1 2 3
> y
[1] 4 5 6
> z
[1] 7 8 9
> D <- cbind(x,y,z)
> D
     x y z
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

- We can add row names to matrix **D**:

```
> rownames(D) <- c("Jim","George","Liz")
> D
       x y z
Jim    1 4 7
George 2 5 8
Liz    3 6 9
```

- And we can output specific elements of matrix **D** using one of the two methods here:

```
> D[2,3]
[1] 8
> D["George","z"] # same thing as above line
[1] 8
```

- The `rbind` function takes vectors and adds them as new rows to a matrix:

```
> E <- rbind(x,y,z)
> E
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
z    7    8    9
```

- We can add column names to matrix **E** (the colors blue, red, and green here):

```
> q <- c("blue","red","green")
> colnames(E) <- q
> E
  blue red green
x    1   2     3
y    4   5     6
z    7   8     9
```

- And finally, we can refer to specific elements of matrix **E** using one of two methods:

```
> E["z","blue"]
[1] 7
> E[3,1]
[1] 7
```

- The `t` function in R will take the transpose of a matrix (flips rows and columns):

```
> t(E)
      x y z
blue  1 4 7
red   2 5 8
green 3 6 9
```

- We can edit a matrix permanently using one of the following (both of these open up the **Data Editor**, allowing the user to change values of items in the matrix much like an Excel spreadsheet):

```
> E <- edit(E)
> fix(E)
```

Installing and Loading Packages

Base R (what you install initially) contains several built-in libraries (some examples are the **stats**, **utils**, and **graphics** libraries, among others). However, many other functions and tools are available through add-on packages. The term "packages" generally refers to the set of functions and tools available to download off the internet, and once these packages have been installed on your computer, we refer to them as "libraries".

It is important to note that packages, other than those that are installed with base R, only need to be installed on your computer once (unless you reinstall RStudio), but they *do* need to be loaded into RStudio during each session.

You can install new packages in RStudio by going to the **Packages** tab/pane on the lower right, click on **Install**, then you can type in the package of interest to search for it and install it. Alternatively, if you know the name of the package, you can use the `install.packages()` function.

To load a package during each RStudio session, you can either go to the **Packages** tab/pane on the lower right and check mark the box next to the package that you would like to load. This will automatically type "`library(<package>)`" into the **Console** pane. Or you can simply type into the **Console** pane (or in a script file) "`library(<package>)`" to load that library into the current session.

The screencast "Installing and Loading Packages" will take you through a few examples of installing and loading packages in RStudio.

Data Frames and Tibbles

Statisticians typically use data in tabular form, oftentimes provided in .xlsx, .csv, or .txt files (see next section for how to import data). Data frames are nice structures in R and other programming languages to work with these tabular data sets. For example, data is typically structured into columns and rows, and it is nice to keep this format and be able to use it in an easy-to-use manner.

- We can easily create a data frame from individual vectors using the `data.frame` function (base R):

```
Month <- c("January","February","March","April","May","June","July","August",
           "September","October","November","December")
Jimmy <- c(290,310,420,280,370,440,480,430,300,260,410,250)
Sue <- c(250,310,390,400,320,480,470,310,410,410,340,210)
John <- c(280,320,420,300,450,320,450,380,390,310,380,350)
Sally <- c(260,390,380,350,390,360,390,480,290,380,320,340)
Gilbert <- c(330,240,360,280,360,520,390,390,350,380,270,230)
sales <- data.frame(Month,Jimmy,Sue,John,Sally,Gilbert)
```

This results in the following:

```
> sales
       Month Jimmy Sue John Sally Gilbert
1    January   290 250  280   260     330
2   February   310 310  320   390     240
3      March   420 390  420   380     360
4      April   280 400  300   350     280
5        May   370 320  450   390     360
6       June   440 480  320   360     520
7       July   480 470  450   390     390
8     August   430 310  380   480     390
9  September   300 410  390   290     350
10   October   260 410  310   380     380
11  November   410 340  380   320     270
12  December   250 210  350   340     230
```

- We can use the `is.data.frame` function to determine if a particular variable is a data frame:

```
> is.data.frame(sales)
[1] TRUE
```

- Note that the **sales** data frame has column names (**colnames**), which by default are the individual vector names from which the data frame was formed. You also notice that, by default, row indices or names (**rownames**) have been added (1 through 12) on the far left. To display vectors of row names and column names, we can use:

```
> rownames(sales)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
> colnames(sales)
[1] "Month"   "Jimmy"   "Sue"     "John"    "Sally"   "Gilbert"
```

- You can also change the row names and/or column names by assigning a new vector of names to either **rownames** or **colnames**. For example, we could change the row names to “A” through “L” using:

```
> rownames(sales) <- c("A","B","C","D","E","F","G","H","I","J","K","L")
> sales
      Month Jimmy Sue John Sally Gilbert
A   January   290 250  280   260     330
B  February   310 310  320   390     240
C     March   420 390  420   380     360
D     April   280 400  300   350     280
E       May   370 320  450   390     360
F      June   440 480  320   360     520
G      July   480 470  450   390     390
H    August   430 310  380   480     390
I September   300 410  390   290     350
J   October   260 410  310   380     380
K  November   410 340  380   320     270
L  December   250 210  350   340     230
```

- Let's go back to our original **sales** data frame (with indices 1-12 as row names). The following code shows how we can refer to individual items, columns, rows, or sub-matrices of the data frame:
    - `sales[2]` refers to John's entire column (it's the 4th column of the data frame), stored as a data frame:

      ```
      > sales[2]
        Jimmy
      1   290
      2   310
      3   420
      4   280
      5   370
      6   440
      7   480
      8   430
      9   300
      10  260
      11  410
      12  250
      ```

    - `sales["Jimmy"]` does the exact same thing (output is a data frame) as the above example since "**Jimmy**" is the 2nd column of the **sales** data frame.
    - `sales$Jimmy` outputs the "**Jimmy**" column but as a vector, not a data frame as in the two examples above.
    - Whereas `sales[3]` displays all items of column 3, `sales[3,]` displays all columns of row 3:

      ```
      > sales[3,]
        Month Jimmy Sue John Sally Gilbert
      3 March   420 390  420   380     360
      ```

    - `sales[,3]` displays all rows of column 3 (similar to `sales[3]` or `sales["Sue"]`) but stored as a vector, not a data frame:

      ```
      > sales[,3]
       [1] 250 310 390 400 320 480 470 310 410 410 340 210
      ```

- `sales[1,3]` outputs the item of the **sales** data frame in row 1, column 3:

```
> sales[1,3]
[1] 250
```

- `sales[,1:2]` displays all rows of columns 1 through 2 (1 and 2):

```
> sales[,1:2]
       Month Jimmy
1    January   290
2   February   310
3      March   420
4      April   280
5        May   370
6       June   440
7       July   480
8     August   430
9  September   300
10   October   260
11  November   410
12  December   250
```

- sales[1:2,] displays all columns of rows 1 through 2 (1 and 2):

```
> sales[1:2,]
     Month Jimmy Sue John Sally Gilbert
1  January   290 250  280   260     330
2 February   310 310  320   390     240
```

- `sales[c(3,4,5)]` displays columns 3, 4, and 5 of the **sales** data frame:

```
> sales[c(3,4,5)]
   Sue John Sally
1  250  280   260
2  310  320   390
3  390  420   380
4  400  300   350
5  320  450   390
6  480  320   360
7  470  450   390
8  310  380   480
9  410  390   290
10 410  310   380
11 340  380   320
12 210  350   340
```

- `sales[c("Sue","John","Sally")]` does the exact same thing as the line above.

- `sales[-c(2,4)]` refers to the sales data frame ***without*** columns 2 and 4:

```
> sales[-c(2,4)]
       Month Sue Sally Gilbert
1    January 250   260     330
2   February 310   390     240
3      March 390   380     360
4      April 400   350     280
5        May 320   390     360
6       June 480   360     520
7       July 470   390     390
8     August 310   480     390
9  September 410   290     350
10   October 410   380     380
11  November 340   320     270
12  December 210   340     230
```

- In situations where we have numerical data in most columns but have a specifier/index column (**Month** in the above **sales** data frame), we oftentimes would like to change the row index (**rownames**) to be that column (**Month** here). To do this, we can use the `column_to_row` function, which is in the **tidyverse** library (be sure to run "`library(tidyverse)`" prior to the following commands):

```
> sales <- column_to_rownames(sales, var="Month")
> sales
          Jimmy Sue John Sally Gilbert
January     290 250  280   260     330
February    310 310  320   390     240
March       420 390  420   380     360
April       280 400  300   350     280
May         370 320  450   390     360
June        440 480  320   360     520
July        480 470  450   390     390
August      430 310  380   480     390
September   300 410  390   290     350
October     260 410  310   380     380
November    410 340  380   320     270
December    250 210  350   340     230
```

You notice that we no longer have indices 1-12 over on the far left. Instead, these indices (**rownames**) have been changed to be the **Month** column.

- Using this new data frame:
  - `sales[3]` refers to the entire 3rd column (the **John** column) and output as a data frame:

```
> sales[3]
          John
January    280
February   320
March      420
April      300
May        450
June       320
July       450
August     380
September  390
October    310
November   380
December   350
```

  - sales[,"John"] refers to the **John** column but output as a vector, not a data frame:

```
> sales[,"John"]
 [1] 280 320 420 300 450 320 450 380 390 310 380 350
```

  - `sales$John` is equivalent to the above line (output as a vector).

- `summary(sales)` provides a nice statistical summary of all columns of a data frame:

```
> summary(sales)
     Jimmy            Sue             John           Sally          Gilbert     
 Min.   :250.0   Min.   :210.0   Min.   :280.0   Min.   :260.0   Min.   :230.0  
 1st Qu.:287.5   1st Qu.:310.0   1st Qu.:317.5   1st Qu.:335.0   1st Qu.:277.5  
 Median :340.0   Median :365.0   Median :365.0   Median :370.0   Median :355.0  
 Mean   :353.3   Mean   :358.3   Mean   :362.5   Mean   :360.8   Mean   :341.7  
 3rd Qu.:422.5   3rd Qu.:410.0   3rd Qu.:397.5   3rd Qu.:390.0   3rd Qu.:382.5  
 Max.   :480.0   Max.   :480.0   Max.   :450.0   Max.   :480.0   Max.   :520.0  
```

- We can convert a data frame to a tibble using the `as_tibble` function, which is in the **tibble** library (be sure to use "`library(tibble)`" prior to running the following lines:

```
> sales.tibble <- as_tibble(sales)
> sales.tibble
# A tibble: 12 × 5
   Jimmy   Sue  John Sally Gilbert
   <dbl> <dbl> <dbl> <dbl>   <dbl>
 1   290   250   280   260     330
 2   310   310   320   390     240
 3   420   390   420   380     360
 4   280   400   300   350     280
 5   370   320   450   390     360
 6   440   480   320   360     520
 7   480   470   450   390     390
 8   430   310   380   480     390
 9   300   410   390   290     350
10   260   410   310   380     380
11   410   340   380   320     270
12   250   210   350   340     230
```

  A tibble is similar to a data frame, yet there are many advantages for using tibbles that are beyond the scope of this course. Primarily, tibbles are useful when working with **tidyverse** library functions, as they are optimized for use with tibbles.

- We can edit the values in a data frame or tibble using the **edit** or **fix** functions (fix is preferred because the change is permanent).

The following shows how we can create a data frame directly from a matrix using the **as.data.frame** function.

- First, we can create a 3x3 matrix from 3 individual vectors using the **matrix** function:

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> z <- c(7,8,9)
> A <- matrix(c(x,y,z),nrow=3,ncol=3) # Creates 3x3 matrix
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

- Next, we can convert the matrix to a data frame using the **as.data.frame** function:

```
> A <- as.data.frame(A) # Converts A to a data frame
> A
  V1 V2 V3
1  1  4  7
2  2  5  8
3  3  6  9
```

- We can change the row names (rownames) and column names (colnames) if desired:

```
> rownames(A) <- c("First","Second","Third"
> colnames(A) <- c("Larry","Curly","Moe")
> A
       Larry Curly Moe
First      1     4   7
Second     2     5   8
Third      3     6   9
```

The following example shows how we can create column names directly when we create the data frame from individual vectors. Also, the example shows how we can add a column of data to an existing data frame.

- First, we create the individual vectors of data (assignments here) and a column with the names of the students:

```
> assignment.1 <- c(8,9.5,10,9,8.5)
> assignment.2 <- c(10,10,9,10,9.5)
> assignment.3 <- c(7.5,9.5,10,8,9.5)
> assignment.4 <- c(10,8,8,8.5,9)
> student.names <- c("Jenny","Chris","Pat","Logan","Mila")
```

- Next, we can create the data frame, specifying the column names (each column name before the equal sign):

```
> HW.scores <- data.frame(Student = student.names,
+                         HW1 = assignment.1,
+                         HW2 = assignment.2,
+                         HW3 = assignment.3,
+                         HW4 = assignment.4)
> HW.scores
  Student  HW1  HW2  HW3  HW4
1   Jenny  8.0 10.0  7.5 10.0
2   Chris  9.5 10.0  9.5  8.0
3     Pat 10.0  9.0 10.0  8.0
4   Logan  9.0 10.0  8.0  8.5
5    Mila  8.5  9.5  9.5  9.0
```

- We can change the row indices/names from integers 1-5 to the **Student** column, if desired:

```
> HW.scores <- column_to_rownames(HW.scores, var="Student")
> HW.scores
       HW1  HW2  HW3  HW4
Jenny  8.0 10.0  7.5 10.0
Chris  9.5 10.0  9.5  8.0
Pat   10.0  9.0 10.0  8.0
Logan  9.0 10.0  8.0  8.5
Mila   8.5  9.5  9.5  9.0
```

- To add an assignment (**assignment.5**) to the **HW.scores** data frame, we can do the following:

```
> assignment.5 <- c(9.5,8,10,8.5,9.5)
> HW.scores["HW5"] <- assignment.5
> HW.scores
       HW1  HW2  HW3  HW4  HW5
Jenny  8.0 10.0  7.5 10.0  9.5
Chris  9.5 10.0  9.5  8.0  8.0
Pat   10.0  9.0 10.0  8.0 10.0
Logan  9.0 10.0  8.0  8.5  8.5
Mila   8.5  9.5  9.5  9.0  9.5
```

Importing Data

Three of the most common types of files that we wish to import into R are Excel (.xlsx) files, comma separated value (.csv) files, and text (.txt) files. In order to import data, we first must make sure that the file path is correct. Let's assume that you have saved the file (a .xlsx file here) that you wish to import in **Documents → Folder1 → Folder2**, and the file is named "**ToImport.xlsx**". I will show two methods for how we can import this file and how we can set the correct file path:

1. We can change the working directory to **Folder2** using **Session → Set Working Directory → Choose Directory**. After navigating to **Folder2**, select **Open** and the working directory should be changed (you can verify the path in the Console). Now, all we need to do in the import options below is to use "**ToImport.xlsx**" as the file path to import.
2. Keep (or set) the working directory as **Documents** but specify the full file path of the file when you import the file. To obtain the full file path on a Windows machine, you can right-click on the

file in the location that it is saved on your computer. Then, select “**Copy as path**”. In RStudio (either the Scripts pane or the Console), you can paste that file path and use it in one of the import approaches below. Importantly, you must change the single forward slashes (/) in the file path to double forward slashes (//) OR replace the single forward slashes with single back slashes (\). You must also delete **Documents** (the current working directory) and any higher-level directories from the file path. For example, if your working directory were Documents and the full copied file path were "**C:\Users\abcde\OneDrive\Documents\Folder1\Folder2\ToImport.xlsx**", then the file path that you would use in the import functions below would be “**Folder1\Folder2\ToImport.xlsx**”.

To import .xlsx files, we can use the **read_excel** function that is found in the **readxl** library (part of the **tidyverse**). The data is imported as a tibble:

- library(readxl)
  data.tibble <- read_excel(<file path>)

To import .csv files, we can use the **read.csv** function in base R (utils library), which imports the data as a data frame, or we can use the **read_csv** function in the **readr** library (also part of the **tidyverse**), which is much faster than the read.csv function and imports the data as a tibble:

- read.csv(<file path>)   OR:
- library(readr)
  read_csv(<file path>)

To import .txt files, we can use: **read.table** in base R (utils library), which imports the data as a data frame (be sure to use “header=TRUE” if you have headers in your data); **read.delim** in base R (utils library), which imports data as a data frame; or **read_delim** in the **readr** library (part of the tidyverse), which imports the data as a tibble:

- read.table(<file path>)   OR
- read.delim(<file path>   OR
- library(readr)
  read_delim(<file path>

Built-In Functions

There are hundreds if not thousands of functions that are either built-in to base R or are available in various packages to install on your computer. You will learn many of the statistics related functions later in this course, but I wanted to explain how to use the arguments and argument names.

Let’s consider an example, the **dbinom** function, which you’ll learn about and use later in the course. If you type help(dbinom) in the Console, you will get the following (truncated here, but this has the most useful information):

Usage

```
dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`size`	number of trials (zero or more).
`prob`	probability of success on each trial.

Many of the arguments are optional, and the help menu has information related to these optional arguments. In the "**Usage**" section, if an argument contains and = sign (for example, "`log=FALSE`"), that means that that argument is optional. If left off, the default value will be whatever that argument name is equal to (in "`log=FALSE`", the probabilities are not given as log(p)). Therefore, we need to specify 3 arguments to the **dbinom** function, and I'll work through an example here:

- To calculate the probability of getting exactly 2 successes out of 10 trials if the probability of success is 0.7, we can write: `dbinom(x=2, size=10, prob=0.7)`
  When executed, this works fine.
- A slightly simpler way to write this, but only if we are specifying the arguments in the order that they appear in the help menu (for **dbinom** this is **x**, **size**, **prob**), is: `dbinom(2,10,0.7)`
  This works fine, as well.
- However, we cannot switch the order of the arguments if we do not specify the argument names. The following will NOT work as expected: `dbinom(2,0.7,10)`. In this case, the function will try to use 0.7 as the size argument and 10 as the prob argument, and this is not what we want.
- If we provide argument names, however, we can change the order of the arguments. The following will work properly: `dbinom(2, prob=0.7, n=10)`

As another example, the **plot** function has lots of optional arguments. It would be difficult to memorize or continually have to look up the order of these arguments, so it's much easier to refer to most of the arguments (other than the first two) by name:

- `plot(x,y,type="b",lty=2,col="red",xlab="Time",ylab="Stock Price")`

User-Defined Functions

There are occasions where you might not find a function that suits your needs. If this is the case, you can always create a custom, user-defined function (UDF) in RStudio. These can be written in either a script file or in the Console, but probably better to write as in a script file if you will be using the function often.

If you need to store the output result of a UDF in a variable, then your function output needs to include the return function. Whatever is inside the **return** function inside your UDF will be output by the function.

As an example, the myfunction function below has one argument (x) and the function simply squares x and adds 5:

```
myfunction = function(x){
  result <- x^2+5
  return(result)
}
```

This could also have been written as:

```
myfunction = function(x){
  return(x^2+5)
}
```

If in a script file, the cursor must be placed in the first line of the function and executed (you'll see it stored in the Environment pane) before it can be used.

This function works as follows on two different arguments:

```
> myfunction(3)
[1] 14
> myfunction(10)
[1] 105
```

As another example, which doesn't allow us to store the output values since the **return** function is not used, will display the mean and standard deviation of a vector **x**:

```
statistics = function(x){
  m = mean(x)
  s = sd(x)
  cat(" Mean =",m,"\n","Standard deviation =",s)
}
```

And when executed in the Console with a vector of data:

```
> data <- c(230,315,340,375,325,280,295,290,325,355,380,425)
> statistics(data)
 Mean = 327.9167
 Standard deviation = 52.06894
```