Chapter 1. Introduction to R for Finance

1. Installation of R and Packages

The way to obtain **R** is to download it from one of the CRAN (Comprehensive **R** Archive Network) websites. The main website is http://cran.r-project.org/.

Binary distributions: As of March 2020, the version for recent variants of Microsoft Windows comes as a single R-3.6.3-win.exe file, on which you simply double-click with the mouse and then follow the on-screen instructions.

RStudio: RStudio is a free and open-source Integrated Development Environment (IDE) for **R**, which contains a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging, and workspace management. You can download its most recent version at https://www.rstudio.com/.

Package installation: To work through the examples and exercises in this course, you should install some packages. If you are connected to the Internet, you can start **R** or **RStudio** and from the Windows or Mac versions, use their convenient menu interfaces.

Or you can type (for example, if you want to install package "Quantlib")

```
> install.packages("Quantlib")
```

This will give off a harmless warning and install the package in the default location. If your **R** machine is not connected to the Internet, you can also download the package as a file via a different computer.

2. Getting Started with R Programming

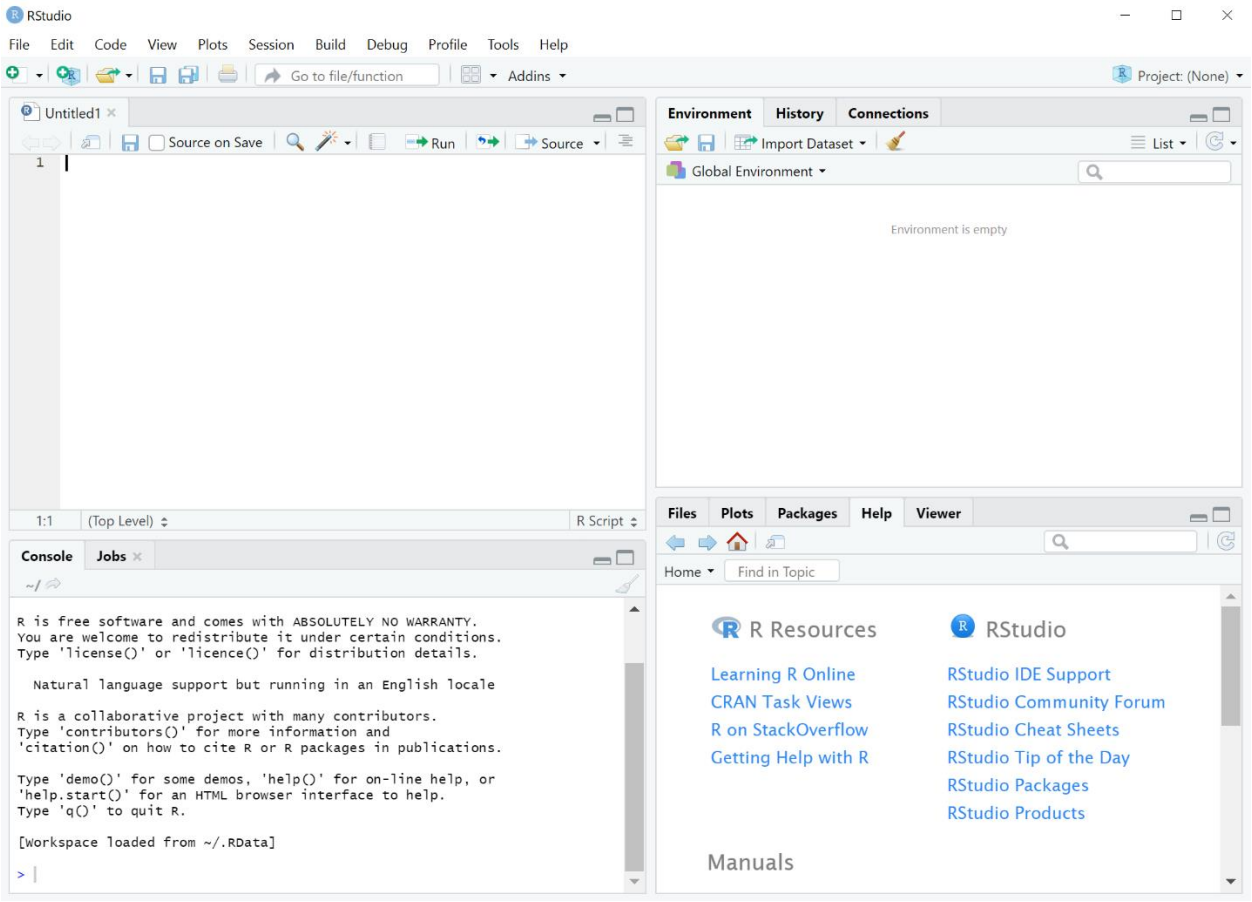
Your screen should look like below. You have four work areas (also called panes).

- Source: The top-left corner of the screen contains a text editor that lets you work with source script files. Here, you can enter multiple lines of code, save your script file to disk, and perform other tasks on your script. This code editor works a bit like every other text editor you have ever seen, but it is smart. It recognizes and highlights various elements of your code, for example (using different colors for different elements), and it also helps you find matching brackets in your scripts.

- Console: In the bottom-left corner, you find the console. This is where you do all the interactive work with R.

- Environment and History: The top-right corner is a handy overview of your environment, where you can inspect the variables you created in your session, as well as their values. This is also the area where you can see a history of the commands you have issued in R.

- Files, Plots, Packages, Help, and Viewer: In the bottom-right corner, you have access to several tools.
  a. Files: This is where you can browse the folders and files on your computer.
  b. Plots: This is where R displays your plots (charts or graphs). We discuss plots in later chapters.
  c. Packages: You can view a list of all installed packages. A package is a self-contained set of code that adds functionality to R, similar to the way that add-ins add functionality to Microsoft Excel.
  d. Help: This is where you can browse R's built-in Help system.
  e. Viewer: This is where R Studio displays previews of some advanced features, such as dynamic web pages and presentations that you can create with R and add-on packages.

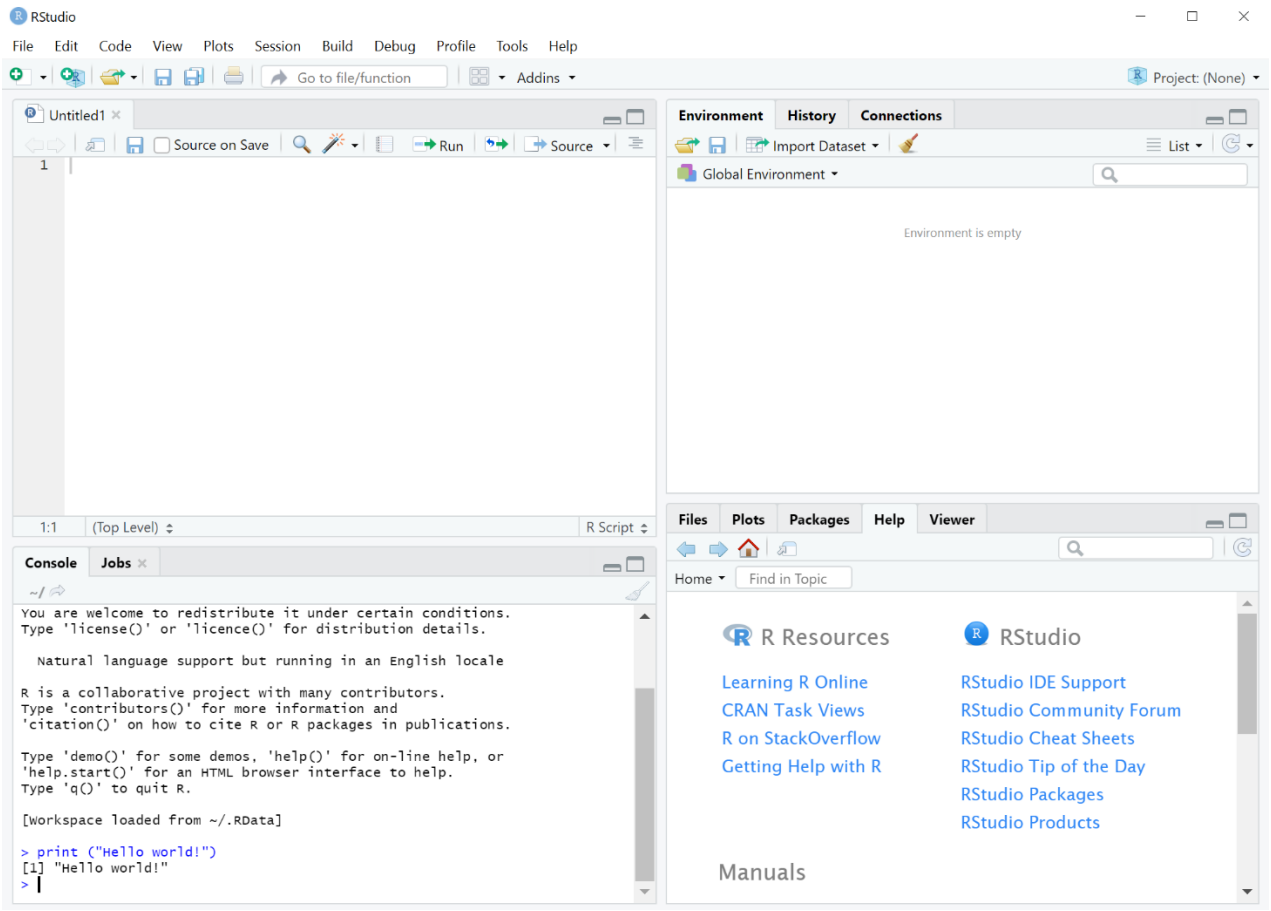


3. Starting First R Session – Saying hello to the world

Programming books typically start with a very simple program. Often, this first program creates the message "Hello world!". In R, hello world program consists of one line of code.

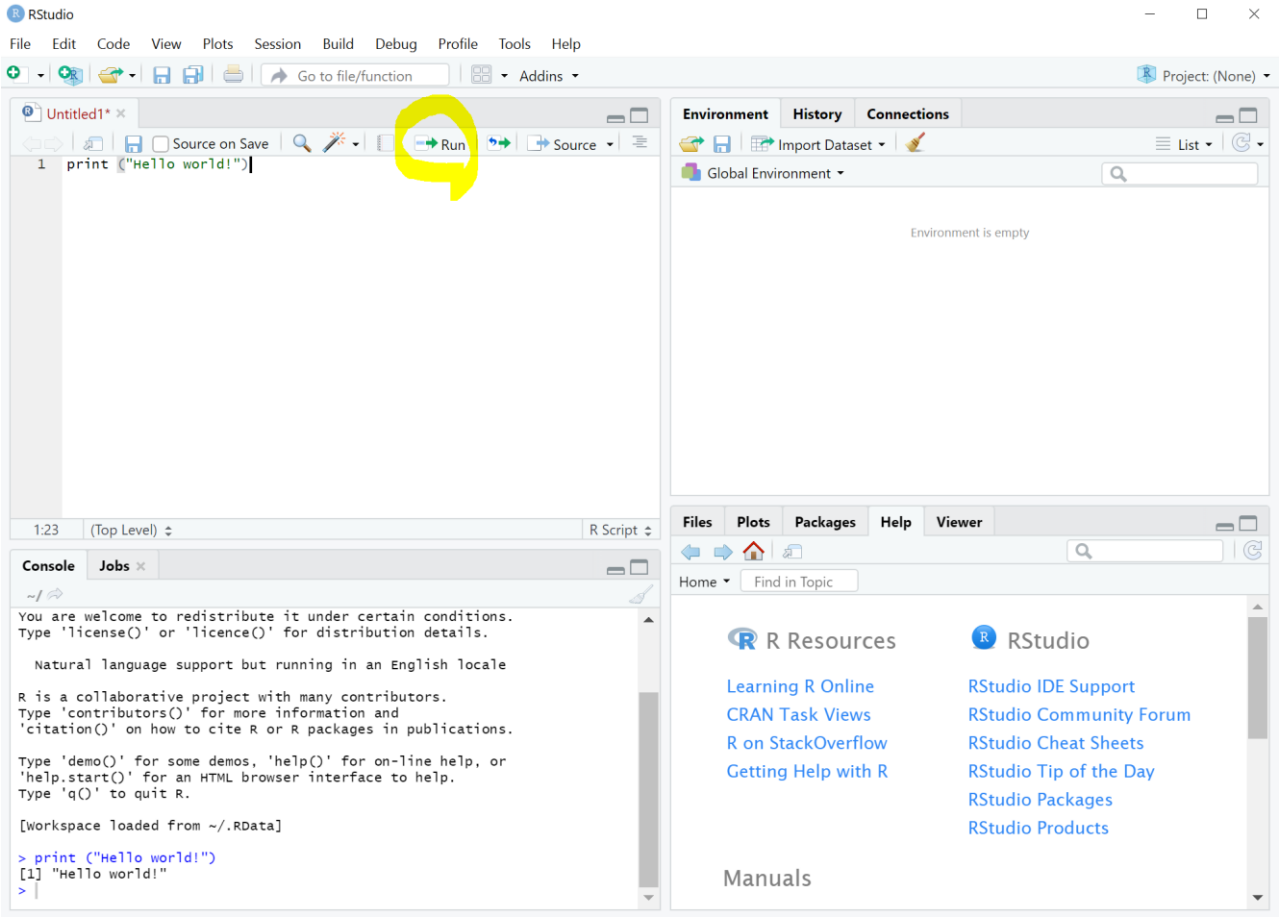
Type the following in your Console, and press Enter:

```
> print ("Hello world!")
  [1] "Hello world!"
```



In R Studio, you have another way to obtain and see the results of your codes.

You can type in your Source area, put the cursor on the line of the command, and then click “Run”.

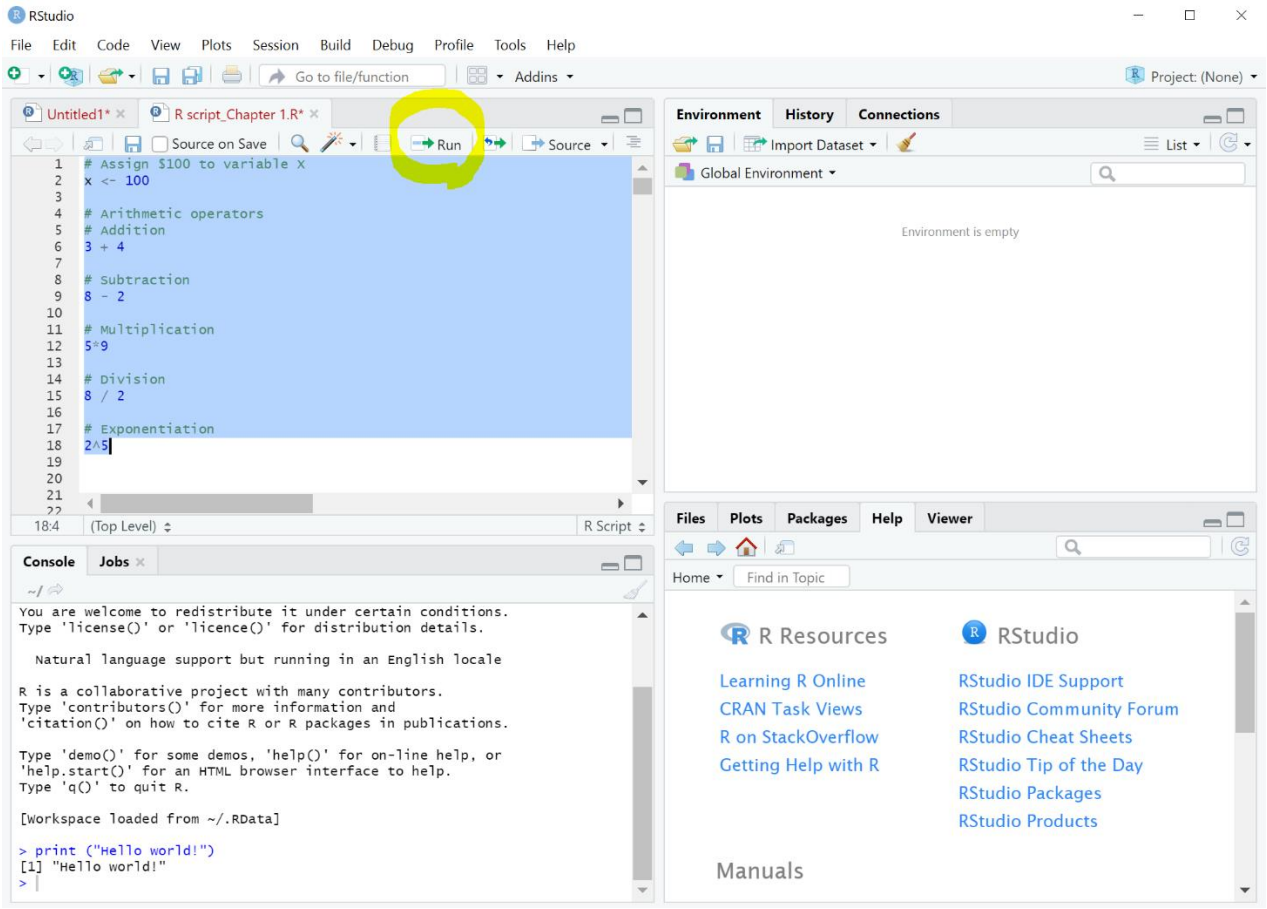


Remarks: As shown above, in R Studio you can use either way. Your codes can be written on Console or on Source. The main difference between these two ways is that you can use your codes again later on by saving the script on your computer drive when writing on Source. You can save the script you wrote by clicking File – Save (or Save as) in toolbar on the top of the screen. One of the advantages to use Source is that you can run the multiple lines of your code at once without running the code line by line. For example, if your code is as follows (this code is from ‘R script_Chapter 1.R’ posted on Moodle) :

```
# Assign $100 to variable X
x <- 100

# Arithmetic operators
# Addition
3 + 4

# Subtraction
8 - 2

# Multiplication
5*9

# Division
```

```
8 / 2

# Exponentiation
2^5
```

You can type the codes line by line on Console by obtaining the results whenever you press Enter. However, it will be more convenient that you directly type the codes on Source, select the block of codes you want to run, and then click "Run".



4. Assignment and Variables

The assignment operator used in **R** to create new variable is <-. A variable allows you to store a value or an object in **R**. Then, you can later use this variable's name to easily access the value or the object that is stored within this variable (Note that " # " means "Do not run this line.")

```
# Assign $100 to variable x
x <- 100
```

5. Operators in R

You can use binary operators on vectors and matrices as well as scalars.

1) Arithmetic operators

- Addition: +
- Subtraction: –
- Multiplication: *
- Division: /
- Exponentiation: ^ or **
- Modulo: %%

The ^ operator raises the number to its left to the power of the number to its right. For example, 2^4 is 2^4 or 16. The modulo returns the remainder of the division of the number to the left by the number on the right, for example 5 modulo 4 or 7 %% 3 is 1.

```
# Check out these arithmetic operators.
# Addition
3 + 4

# Subtraction
8 - 2

# Multiplication
5*9

# Division
8 / 2

# Exponentiation
2^5

# Modulo
9 %% 2
```

Suppose you have $100 in variable x and $150 in variable y. You want to know how much money the two variables have together.

```
# Assign $100 in variable x
x <- 100

# Assign $150 in variable y
y <- 150

# Add x and y
x + y
```

```
# Add x and y, save the result to new variable z
z <- x + y
```

2) Logical operators

- < : less than
- <= : less than or equal to
- > : greater than
- >= : greater than or equal to
- == (two equal signs): exactly equal to
- != : not equal to
- !x : not x
- x | y : x OR y
- x & y : x AND y
- isTRUE(x) : test if x is TRUE

3) Vectorized arithmetic

The combine **c(…)** is used to define vectors.

```
# Create variable x with a vector (1, 2, 3, 4, 5)
x <- c(1, 2, 3, 4, 5)       or        x <- c(1:5)
x
[1] 1 2 3 4 5
```

You can do calculations with vectors just like ordinary numbers as long as they are of the same length.

```
# Create variable y with a vector (6, 7, 8, 9, 10)
y <- c(6:10)

# Save the result of multiplying x and y to z
z <- x * y
z
[1] 6 14 24 36 50
```

Recycling Rule: When two vectors are not of the same length, **R** processes the vector element in pairs, starting at the first elements of both vectors. At a certain point, the shorter vector is exhausted while the longer vector still has unprocessed elements. **R** returns to the beginning of the shorter vector, "recycling" its elements; continues taking elements from the longer vector; and completes the operation. It will recycle the shorter-vector elements as often as necessary until the operation is complete (Please try !!!).

4) Others

Besides the arithmetic operators we have used so far, there are some more that we often use: `log, exp,` and, `sqrt`. We can also use the well-known constant π by simply typing `pi`, instead of its value 3.141593…..

6. Financial Returns

Suppose you have $100 on your saving account. During January, you earn a 3% return on that account. How much do you have at the end of January? You multiply your original $100 by 1.03 to get the amount at the end of January.

```
# Create a variable for starting_mysaving (So, you assign 100 to a variable starting_mysaving)
starting_mysaving <- 100


# How much do you have at the end of January?
jan_mysaving <- starting_mysaving*1.03
jan_mysaving
[1] 103
```

The result from the calculation is assigned to a variable jan_mysaving.

If you earn another 5% on your saving account in February, how would you calculate the total amount at the end of February?

```
# How much do you have at the end of February?
feb_mysaving <- starting_mysaving*1.03*1.05
feb_mysaving
[1] 108.15
```

Or

```
feb_mysaving <- jan_mysaving*1.05
feb_mysaving
[1] 108.15
```

7. Basic Data Types

- Numeric: Decimal numbers (i.e. 5.2). Integer (a numeric without a decimal piece) must be specified like 5L.
- Logical: Boolean values, TRUE and FALSE (should be **capital letters**; true and false are not valid.)
- Character: Text values, i.e. “Hello, world.”

Should use “ ” or ‘ ’.

```
# Assign the numeric 120 to samsung_stock
samsung_stock <- 120
```

```
# Assign the character "AA" to credit_rating
credit_rating <- "AA"

# Assign a Boolean value FALSE to answer
answer <- FALSE

# Print answer
answer
[1] FALSE
```

If you want to check what data type a variable is,

```
# Check data type
class(samsung_stock)
[1] "numeric"

class(credit_rating)
[1] "character"

class(answer)
[1] "logical"
```

8. Vectors and Matrices

1) Combine c(…)

You have seen that combine function **c(…)** is used to create a vector (you created a numeric vector above). Note that you can also create a character vector and a logical vector.

```
# Create a character vector of bond credit ratings and assign it to a variable credit_rating
credit_rating <- c("AAA", "AA", "BBB", "BB", "B")
credit_rating
[1] "AAA" "AA" "BBB" "BB" "B"

# Create a logical vector and assign it to a variable logic
logic <- c(TRUE, TRUE, FALSE)
logic
[1] TRUE TRUE FALSE
```

Remarks: It is important to remember that only one data type can be constructed in a vector, meaning that you cannot have both a numeric and a character in the same vector. If you create a vector containing both data types, the lower ranking type will be coerced into the higher ranking type. The hierarchy for coercion is as follows: character > numeric > integer > logical

For example, c(2.3, "AAA") results in c("2.3", "AAA") because the numeric 2.3 coerce into the character data type.

However, logical data type is coerced differently depending on what the higher data type is.

For example, c(TRUE, 2.3) will result in c(1, 2.3) because TRUE (or FALSE) is coerced into the numeric 1 (or 0), but c(TRUE, "AAA") is converted to c("TRUE", "AAA").

2) Vector names(…)

Suppose that you earn a 5% return in January and 3% return in February. You can put these returns into a vector by doing this.

```
# Put returns into a vector
ret <- c(5, 2)
```

Now, you want to add names to each return in your vector. You can do this using **names(…)**.

```
# Add names on vector ret
names(ret) <- c("Jan", "Feb")
ret
Jan  Feb
  5    2
```

3) Weighted average

The weighted average is a way to calculate portfolio return over a given time period. Consider the following example:

Your portfolio contains 30% of stock A and 70% of stock B. Assume that stock A earned a 3% return and stock B earned 5% return in January. What is your total portfolio return in January?

To obtain total portfolio return, you can take the following steps:
First, take the return of each stock in your portfolio
Second, multiply the return by the weight of the stock
Third, sum up all of the results you have.

```
# Weights and returns
ret_A <- 3
ret_B <- 5
weight_A <- 0.3
weight_B <- 0.7

# Portfolio return
ret_portfolio <- weight_A*ret_A + weight_B*ret_B
ret_portfolio
[1] 4.4
```

Alternatively, you can use vectors. Take a look at the following code.

```
ret <- c(3, 5)
weight <- c(0.3, 0.7)

mul_ret_weight <- ret*weight

ret_portfolio <- sum(mul_ret_weight)
ret_portfolio
[1] 4.4
```

As you can see, **sum(…)** is used to sum up each element in the vector.

What if you have equal weight to Stock A and Stock B? That is, you have 50% of Stock A and 50% of Stock B in your portfolio. You can use the code above by switching vector weight to c(0.5, 0.5). Also, you can do this.

```
# Equally weighted average
weight <- .5
mul_ret_weight <- ret*weight
ret_portfolio <- sum(mul_ret_weight)
ret_portfolio
[1] 4
```

The vector ret is a vector of length 2, but weight is a vector of length 1. **R** reuses the .5 in vector weight twice to make it the same length of ret.

4) Vector subsetting

What if you want to use a piece of your vectors? For example, if you want to use the first month of return out of 6 months of returns, you can subset the vector using […].

Suppose you have the vector of 6 months of returns:

```
# 6 months of returns
ret <- c(5, 4, 6, 2, 1, 4)
```

Now, you select the first month return:

```
# Select the first month return
first_ret <- ret[1]
first_ret
[1] 5
```

If you want to use the first three months, you can use `ret[1:3]` or `ret[c(1, 2, 3)]`.

You can also select a piece of your vectors by name. First of all, we add names on vector ret as you have seen on vector **names(…)** above

```
# Selecting by name
# Add names on vector ret
names(ret) <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun")
ret
Jan Feb Mar Apr May Jun
  5   4   6   2   1   4

# Select the first month by name
ret["Jan"]
Jan
  5
```

5) Creating a matrix

Matrices are 2-dimension vectors (row and column). You can create a 2x2 matrix using **matrix(…)**.

```
# Creating 2x2 matrix
matrix(data = c(1, 2, 3, 4), nrow = 2, ncol = 2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4

# Creating 2x2 matrix using a vector
mat_vector <- c(1, 2, 3, 4)
matrix(data = mat_vector, nrow = 2, ncol = 2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

6) Creating a matrix using bind functions

Alternative way to create a matrix is to combine multiple vectors together by using bind functions such as **cbind(…)** and **rbind(…)**, representing column bind and row bind, respectively. Let us combine two vectors of Samsung and Apple stock prices.

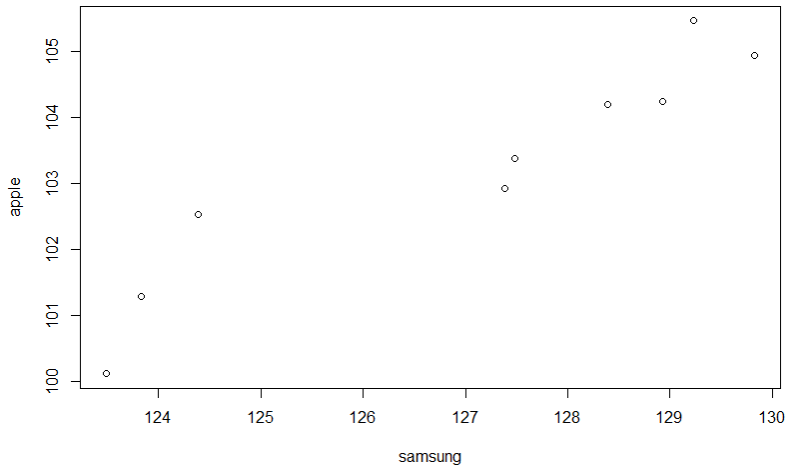
```
# Creating vectors of stock prices
samsung <- c(123.49, 124.39, 123.83, 127.39, 128.39, 127.48, 129.83, 128.93, 129.23)
apple <- c(100.12, 102.53, 101.29, 102.93, 104.21, 103.39, 104.95, 104.25, 105.47)

# cbind (combining by column)
cbind(samsung, apple)
     samsung   apple
[1,]  123.49  100.12
[2,]  124.39  102.53
[3,]  123.83  101.29
[4,]  127.39  102.93
[5,]  128.39  104.21
[6,]  127.48  103.39
[7,]  129.83  104.95
```

```
[8,]     128.93    104.25
[9,]     129.23    105.47


# rbind (combining by row)
rbind(samsung, apple)
             [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]   [,9]
samsung    123.49 124.39 123.83 127.39 128.39 127.48 129.83 128.93 129.23
apple      100.12 102.53 101.29 102.93 104.21 103.39 104.95 104.25 105.47
```

7) Visualization of a matrix

Visualization makes it possible to gain some insights about the relationships in the data. Using the stock price data you created above, you can plot the matrix to see the relationship between two companies' stock prices.

```
# Plot the matrix for stock prices of Samsung and Apple.
samsung_apple_matrix <- cbind(samsung, apple)
plot(samsung_apple_matrix)
```



8) Correlation

From the plot you created above, you can see there exists positive relationship between two stocks. It seems that Apple's stock moves up when Samsung's stock does. As you have learned in statistics course, correlation coefficient is a measure of relationship between two variables, Apple's stock and Samsung's stock prices in this case, and is represented by a number from -1 to +1. When correlation coefficient is +1 (or – 1), it is said that two variables represent perfect positive (or negative) correlation. Zero correlation coefficient means that the variables (stock prices in this case) move independently of each other.

The **cor(…)** function can be used for calculation of correlation coefficient between two vectors (variables) or for creating a correlation matrix, given a matrix.

```
# Calculation of correlation coefficient
cor(samsung, apple)
[1] 0.9418553

# Creating a correlation matrix
cor(samsung_apple_matrix)

           samsung     apple
samsung  1.0000000 0.9418553
apple    0.9418553 1.0000000
```

9) Matrix subsetting

As in vector subsetting, you can select a part of matrix you want to use using […]. The basic structure for this is: matrix name[row, col]. For example, in order to select the first row and the second column in samung_apple_matrix, you can do the following:

```
# Select the first row and the second column
samsung_apple_matrix[1, 2]
 apple
100.12
```

Also, you can select multiple rows or columns as follows:

```
# Select the first three rows
samsung_apple_matrix[1:3, ]
     samsung  apple
[1,]  123.49 100.12
[2,]  124.39 102.53
[3,]  123.83 101.29

# Select the entire first column
samsung_apple_matrix[, 1]
[1] 123.49 124.39 123.83 127.39 128.39 127.48 129.83 128.93 129.23

# Select the entire second column by name
samsung_apple_matrix[, "apple"]
[1] 100.12 102.53 101.29 102.93 104.21 103.39 104.95 104.25 105.47
```

9. Data Frames

1) Creating data frame

Data frame is similar to matrix in that it has rows and columns, but they are different in that multiple data types are contained in a single data frame. That is, data frame can hold a different type of data in

each column. You can use **data.frame(…)** function to create a data frame of several companies which includes the following variables:

- company: The company listed on S&P500
- stock_price: Current stock price of a company
- num_shares: number of shares outstanding of a company

Now, let us create data frame including those variables above.

```
# Create data frame
stock_dataframe <- data.frame(company = c("Samsung", "Apple", "IBM"), stock_price = c(110, 104, 100), num_shares = c(1000, 800, 1200))

stock_dataframe
  company stock_price num_shares
1 Samsung         110       1000
2   Apple         104        800
3     IBM         100       1200
```

Or, you can also create vectors separately and combine them as follows:

```
# Alternative way to create data frame
company <- c("Samsung", "Apple", "IBM")
stock_price <- c(110, 104, 100)
num_shares <- c(1000, 800, 1200)

data.frame(company, stock_price, num_shares)
```

2) head(…), tail(…), and str(…)

The **head(…)** function returns the first 6 rows of a data frame by default. To change the number of rows, use **head(…, n = …)**.

The **tail(…)** function returns the last 6 rows of a data frame by default. To change the number of rows, use **tail(…, n = …)**.

The **str(…)** function checks the structure of an object. This function returns the data type of the object and list each column variable with its data type.

```
# Print out the first 2 rows
head(stock_dataframe, n = 2)
  company stock_price num_shares
1 Samsung         110       1000
2   Apple         104        800

# Print out the last 2 rows
```

```
tail(stock_dataframe, n = 2)
  company stock_price num_shares
1   Apple         104        800
2     IBM         100       1200

# Check out the structure of data frame, stock_dataframe
str(stock_dataframe)
'data.frame':  3 obs. of  3 variables:
 $ company    : Factor w/ 3 levels "Apple","IBM",..: 3 1 2
 $ stock_price: num  110 104 100
 $ num_shares : num  1000 800 1200
```

Note that the class of company is a Factor, not a character. We will talk about Factor later. You can just skip what it is at this point.

3) Naming columns and rows in data frame

You can change the column names using **colnames(…)** just as you have used **names(…)** in vectors.

```
# Change the column names to "comp_name", "stockprice", and "numshares", respectively
colnames(stock_dataframe) <- c("comp_name", "stockprice", "numshares")

# Print out the column names of stock_dataframe
colnames(stock_dataframe)
[1] "comp_name"  "stockprice" "numshares"
```

Similarly, you can also change the row names using **rownames(…)**.

4) Data frame subsetting

Suppose that you want to subset data frame. To do that, you can use **[…]** just as in matrices.

```
# Select the first row
stock_dataframe[1, ]
comp_name stockprice numshares
1   Samsung        110      1000

# Select the second column
stock_dataframe[ , 2]
[1] 110 104 100

# Select the first column by name
stock_dataframe[ , "comp_name"]
[1] Samsung Apple   IBM
Levels: Apple IBM Samsung
```

Alternative way to select a specific column in data frame is to use the **$**.

```
# Select the first column
stock_dataframe$comp_name
```

```
[1] Samsung Apple   IBM
Levels: Apple IBM Samsung
```

In our example, the data frame has only three columns (company, stock price, and number of shares). What if you want to select a certain company's stock price when the data frame has hundreds of columns? Take a look!

```
# Select the stock prices of Apple
subset(stock_dataframe, comp_name == "Apple")
  comp_name stockprice numshares
2     Apple        104       800
```

5) Adding new columns

Now you want to add new column, sales, to stock_dataframe.

```
# Add new column, sales, to data frame
stock_dataframe$sales <- c(1000, 1200, 1400)
stock_dataframe
  comp_name stockprice numshares sales
1   Samsung        110      1000  1000
2     Apple        104       800  1200
3       IBM        100      1200  1400
```

10. Factors

Factors are used to represent categorical data. Factors are stored as integers and have labels associated with these unique integers. Note that they are integers, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set values, known as *levels*. By default, **R** always sorts *levels* in alphabetical order.

The **factor(…)** function is used to create and modify factors in **R**. Suppose you create a vector of credit ratings: AAA, AA, A, BBB, BB, B.

```
# Create a vector of credit ratings
credit_rating <- c("AAA", "AA", "A", "BBB", "AA", "A", "BB", "A", "BBB", "B")
```

R will assign 1 to the level "A", 2 to the level "AA", 3 to the level "AAA", 4 to the level "B", 5 to the level "BB", 6 to the level "BBB" (in alphabetical order).

To create a factor from the vector credit_rating, you can do the following:

```
# Create a factor of credit ratings
credit_factor <- factor(credit_rating)
credit_factor
[1] AAA AA  A   BBB AA  A   BB  A   BBB B
Levels: A AA AAA B BB BBB
```

1) levels(…)

Using the **levels(…)**, you can access the unique levels of the factor.

```
# Access the levels of the factor
levels(credit_factor)
[1] "A"   "AA"  "AAA" "B"   "BB"  "BBB"
```

Also, you can use this to change the name of the factor levels.

```
# Change the name of the factor levels
levels(credit_factor) <- c("Single_A", "Double_A", "Triple_A", "Single_B", "Double_B",
"Triple_B")
credit_factor
[1] Triple_A Double_A Single_A Triple_B Double_A Single_A Double_B Single_A Triple_B
[10] Single_B
Levels: Single_A Double_A Triple_A Single_B Double_B Triple_B
```

2) summary(…)

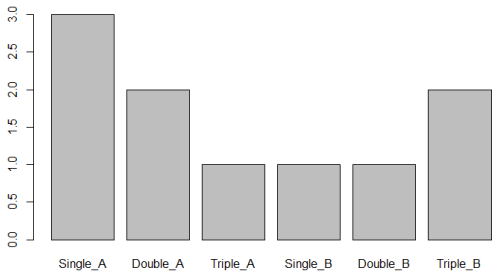
summary(…) function gives you the number of each factor level.

```
# Summarize factor levels
summary(credit_factor)
Single_A Double_A Triple_A Single_B Double_B Triple_B
       3        2        1        1        1        2
```

3) Visualization of factor

You can create the plot (bar chart) using a factor.

```
# Create plot of factor levels
plot(credit_factor)
```



4) cut(…)

Suppose x is a numeric vector which is to be converted to a factor by cutting. The **cut(x, …)** divides the range of x into intervals and codes the values in x according to which interval they fall.

This is an example to create a factor from a numeric vector. Suppose you have the following numeric vector:

```
# Create a numeric vector including 1 to 100 ranking
AA_ranking <- c(3, 45, 23, 18, 95, 38, 58, 67, 73, 83, 74, 27, 4, 48, 38, 28, 21, 13, 98, 73)

# Create a factor with 4 evenly spaced groups
AA_factor <- cut(AA_ranking, breaks = c(0, 25, 50, 75, 100))
AA_factor
[1] (0,25]   (25,50]  (0,25]   (0,25]   (75,100] (25,50]  (50,75]  (50,75]  (50,75]  (75,100]
(50,75]  (25,50]
[13] (0,25]   (25,50]  (25,50]  (25,50]  (0,25]   (0,25]   (75,100] (50,75]
Levels: (0,25] (25,50] (50,75] (75,100]
```

The **breaks =** in **cut(x, …)** function is used for specifying the thresholds you want to establish. Note that (0, 25] in the factor levels means 0 is not included in the group, and 25 is included in the group.

Now let us put names the levels.

```
# Rename the levels
levels(AA_factor) <- c("Low", "Medium", "High", "Very High")
AA_factor
[1] Low       Medium    Low       Low       Very High Medium    High      High      High
Very High High
[12] Medium    Low       Medium    Medium    Medium    Low       Low       Very High High
Levels: Low Medium High Very High
```

5) Creating an ordered factor

As mentioned above, the factor levels are created in alphabetical order. However, suppose you want to make the order of credit rating from least risky to most risky: that is, AAA, AA, A, BBB, BB, B. To do that, you can specify **ordered = TRUE** when creating a factor.

```
# Create an ordered factor
credit_factor_ordered <- factor(credit_rating, ordered = TRUE, levels = c("AAA", "AA", "A",
"BBB", "BB", "B"))
credit_factor_ordered
[1] AAA AA  A   BBB AA  A   BB  A   BBB B
Levels: AAA < AA < A < BBB < BB < B
```

6) Factor subsetting

You can subset factors just as you have used **[…]** in subsetting vectors. What if you want to remove the Triple_A from **credit_factor** you have created?

```
# Print credit_factor
credit_factor
[1] Triple_A Double_A Single_A Triple_B Double_A Single_A Double_B Single_A Triple_B Single_B
Levels: Single_A Double_A Triple_A Single_B Double_B Triple_B

# Remove the Triple_A from credit_factor
credit_factor[-1]
[1] Double_A Single_A Triple_B Double_A Single_A Double_B Single_A Triple_B Single_B
Levels: Single_A Double_A Triple_A Single_B Double_B Triple_B
```

As you can see, **[-1]** removed the first factor, Triple_A. However, note that it left the Triple A level behind. In order to remove both factor and level, you have to add **drop = TRUE**:

```
credit_factor[-1, drop = TRUE]
[1] Double_A Single_A Triple_B Double_A Single_A Double_B Single_A Triple_B Single_B
Levels: Single_A Double_A Single_B Double_B Triple_B
```

7) stringsAsFactors

We have seen the **str(…)** function to check out the structure of an object in data frame. In stock_dataframe, the class of company is a Factor, not a character. As such, **R** converts all characters into factors when creating data frames by default. Very often times you may face the situations where character columns do not work properly after converting your data into data frame in **R**. So, to turn off this default, you need to add **stringsAsFactors = FALSE** when you create data frame:

```
# Create data frame (stringsAsFactors = FALSE)
stock_dataframe <- data.frame(company = c("Samsung", "Apple", "IBM"), stock_price = c(110, 104,
100), num_shares = c(1000, 800, 1200), stringsAsFactors =  FALSE)
str(stock_dataframe)
'data.frame':  3 obs. of  3 variables:
 $ company    : chr  "Samsung" "Apple" "IBM"
 $ stock_price: num  110 104 100
 $ num_shares : num  1000 800 1200
```

11. List

A list is an **R** structure containing different data types, including other lists. A lot of the modeling functions produce lists as their return values, but we can also construct one ourselves using the **list(…)** function. From the following example, we create i) list components and then ii) a list as well as see (or print) the list.

```
# Create list components
name <- "Correlation Matrix: Samsung and Apple Stock"
stockprice_samsung <- c(100.0, 101.3, 103.4, 100.9, 99.2)
```

```
stockprice_apple <- c(110.3, 112.8, 114.1, 111.2, 109.8)
cor_matrix <- cor(cbind(stockprice_samsung, stockprice_apple))

# Create a list
my_portfolio <- list(name, stockprice_samsung, stockprice_apple, cor_matrix)

# See the list
my_portfolio
[[1]]
[1] "Correlation Matrix: Samsung and Apple Stock"

[[2]]
[1] 100.0 101.3 103.4 100.9  99.2

[[3]]
[1] 110.3 112.8 114.1 111.2 109.8

[[4]]
                  stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000
```

1) Name lists

You can name the components as you create the list.

```
# Create a list with names of components
my_portfolio_new <- list(Portfolio_Name = name, Samsung = stockprice_samsung, Apple =
stockprice_apple, Correlation = cor_matrix)
my_portfolio_new
$`Portfolio_Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

$Samsung
[1] 100.0 101.3 103.4 100.9  99.2

$Apple
[1] 110.3 112.8 114.1 111.2 109.8

$Correlation
                  stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000
```

Or, you can use **names(…)** especially when the list has been already created.

```
# Create a list with names of components using names() function
names(my_portfolio) = c("Portfolio Name", "Samsung", "Apple", "Correlation")
my_portfolio
$`Portfolio Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

$Samsung
[1] 100.0 101.3 103.4 100.9  99.2
```

```
$Apple
[1] 110.3 112.8 114.1 111.2 109.8

$Correlation
                   stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000
```

2) Subsetting a list

To extract an item from a list, you can use the single bracket **[…]**.

```
# Subsetting a list
my_portfolio[1]
$`Portfolio Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

my_portfolio[c(1, 2)]
$`Portfolio Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

$Samsung
[1] 100.0 101.3 103.4 100.9  99.2
```

Also, as we already name the list, we can use the **$** operator.

```
# Subsetting a list using $ operator
my_portfolio$Correlation
                   stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000
```

3) Adding to a list

You can simply add new components to the list using **$** operator.

```
# Adding new component, weight
my_portfolio$Weight <- c(Samsung = 0.4, Apple = 0.6)
my_portfolio
$`Portfolio Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

$Samsung
[1] 100.0 101.3 103.4 100.9  99.2

$Apple
[1] 110.3 112.8 114.1 111.2 109.8

$Correlation
                   stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000

$Weight
```

```
Samsung   Apple
    0.4     0.6
```

4) Removing from a list

Suppose you add IBM stock to your list by mistake as follows:

```
# Adding IBM stock prices in the list, my_portfolio
my_portfolio$IBM <- c(150.6, 153.2, 145.6, 147.8, 143.5)
my_portfolio
$`Portfolio Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

$Samsung
[1] 100.0 101.3 103.4 100.9  99.2

$Apple
[1] 110.3 112.8 114.1 111.2 109.8

$Correlation
                   stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000

$Weight
Samsung   Apple
    0.4     0.6

$IBM
[1] 150.6 153.2 145.6 147.8 143.5
```

Now, you decide to remove the component, IBM, mistakenly added to the list.

```
# Remove IBM stock prices from the list, my_portfolio
my_portfolio$IBM <- NULL
my_portfolio
$`Portfolio Name`
[1] "Correlation Matrix: Samsung and Apple Stock"

$Samsung
[1] 100.0 101.3 103.4 100.9  99.2

$Apple
[1] 110.3 112.8 114.1 111.2 109.8

$Correlation
                   stockprice_samsung stockprice_apple
stockprice_samsung          1.0000000        0.9643101
stockprice_apple            0.9643101        1.0000000

$Weight
Samsung   Apple
    0.4     0.6
```

5) split(…)

split(…) makes it possible to split up one data frame into multiple separated data frames. For example, suppose you want to split stock_dataframe you have used in Data Frames section into data frames divided by company (in this case, since there are three different companies (Samsung, Apple, and IBM) in stock_dataframe, you will have three different data frames after using the **split(…)**).

```
# Split
group <- stock_dataframe$company
split_stock_dataframe <- split(stock_dataframe, group)
split_stock_dataframe
$`Apple`
  company stock_price num_shares
2   Apple         104        800

$IBM
  company stock_price num_shares
3     IBM         100       1200

$Samsung
  company stock_price num_shares
1 Samsung         110       1000
```

6) Recombining

You can also recombine separated data frames back into one data frame.

```
# Unsplit
unsplit_stock_dataframe <- unsplit(split_stock_dataframe, group)
unsplit_stock_dataframe
  company stock_price num_shares
1 Samsung         110       1000
2   Apple         104        800
3     IBM         100       1200
```

This is typically used when you have to apply some transformations to each group. In that case, what you would do is that you split your data frame by group, transform something on each group, and then recombine separated data frames back into one data frame. Suppose you want to multiply the number of shares in Samsung by two, in Apple by three, and in IBM by four. So, to do that, we use data frame we already split up above.

```
# Access each company's number of shares column and apply transfroms
split_stock_dataframe$Samsung$num_shares <- split_stock_dataframe$Samsung$num_shares*2
split_stock_dataframe$Apple$num_shares <- split_stock_dataframe$Apple$num_shares*3
split_stock_dataframe$IBM$num_shares <- split_stock_dataframe$IBM$num_shares*4
split_stock_dataframe
$`Apple`
  company stock_price num_shares
2   Apple         104       2400
```

```
$IBM
  company stock_price num_shares
3     IBM         100       4800

$Samsung
  company stock_price num_shares
1 Samsung         110       2000
```

```
# Recombine separated data frames back into one data frame
new_stock_dataframe <- unsplit(split_stock_dataframe, group)
new_stock_dataframe
  company stock_price num_shares
1 Samsung         110       2000
2   Apple         104       2400
3     IBM         100       4800
```

7) attributes(…)

attributes(…) function accesses an object's attributes, such as column names, row names, names, dimensions, class, comment, and so on. Also, **attr(…)** function is used to access a particular attribute.

```
# See attributes of stock_dataframe
attributes(stock_dataframe)
$`names`
[1] "comp_name"  "stockprice" "numshares"

$class
[1] "data.frame"

$row.names
[1] 1 2 3
```

```
# Access an attribute, names
attr(stock_dataframe, which = "names")
[1] "comp_name"  "stockprice" "numshares"
```