Chapter 2. Intermediate R for Finance

1. Dates

R offers several options to deal with dates and times. The Date handles dates (without times), and the POSIXct classes allow for dates and times with control for time zones. The Date is used for calendar date objects which look like “2019-01-01”. The POSIXct is used for datetime objects like “2019-01-01 12:20:30 CST”. The general rule for date and time data in **R** is to use the simplest technique possible.

To get the current date, the **Sys.Date()** function will return a Date object which can be converted to a different class if necessary. On the other hand, the **Sys.time()** function will return the current date and time.

```
# Current date
Sys.Date()
[1] "2019-03-04"

# Current date and time
Sys.time()
[1] "2019-03-04 12:16:33 CST"
```

1) Creating dates

Often time you will have to create dates for yourself from character strings. The **as.Date(…)** function allows a variety of input formats.

```
# The collapse of Lehman Brothers
lb_bankruptcy <- as.Date("2008-09-15")

# Print the lb_bankruptcy
lb_bankruptcy
[1] "2008-09-15"
```

Keep in mind that dates are stored as the number of days since January 1, 1970, and datetimes are stored as the number of seconds since then. You can check these out as follows.

```
# lb_bankruptcy as a numeric
as.numeric(lb_bankruptcy)
[1] 14137

# Current time as a numeric
as.numeric(Sys.time())
[1] 1545331078
```

By using vectors, you can create a large number of dates as follows.

```
# Create a vector of dates as character
dates <- c("2019-01-01", "2019-01-02", "2019-01-03", "2019-01-04", "2019-01-05)

# Create dates
as.Dates(dates)
[1] "2019-01-01" "2019-01-02" "2019-01-03" "2019-01-04" "2019-01-05"


# Assign the days of the week
names(dates) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
dates
      Monday      Tuesday    Wednesday     Thursday       Friday 
"2019-01-01" "2019-01-02" "2019-01-03" "2019-01-04" "2019-01-05"

# Subset the dates to return only the date for Tuesday
dates["Tuesday"]
     Tuesday 
"2019-01-02"
```

2) Date formats

The following symbols can be used with the **format(…)** function to print dates.

Symbol	Meaning	Example
%d	Day as a number (0 - 31)	01-31
%a %A	Abbreviated weekday Unabbreviated weekday	Mon Monday
%m	Month (00 – 12)	00-12
%b %B	Abbreviated month Unabbreviated month	Jan January
%y %Y	2-digit year 4-digit year	07 2007

Here is an example.

```
# Assign today’s date to a variable today
today <- Sys.Date()

# Change the format of the variable today to “Unabbreviated month Day as a number 4-digit year”
format(today, format = "%B %d %Y")
[1] "December 21 2018"
```

However, the variable `‘today’` is still

```
today
[1] "2020-09-06"
```

You just printed out the `'today'` with the format you wanted. The variable `'today'` is not of the format you wanted. How can you change the variable `'today'` to that with the format you want? This will be your homework.

You can also define format when you use **as.Date(…)** function.

```
# Convert date info in format 'mm/dd/yyyy'
dates <- c("01/01/2019", "01/02/2019", "01/03/2019", "01/04/2019", "01/05/2019")
as.Date(dates, format = "%m / %d / %Y")
[1] "2019-01-01" "2019-01-02" "2019-01-03" "2019-01-04" "2019-01-05"
```

You can convert dates to character data using the **as.Character(…)** function.

```
# Convert dates to character data
strDates <- as.character(dates)
strDates
[1] "01/01/2019" "01/02/2019" "01/03/2019" "01/04/2019" "01/05/2019"
```

3) Subtraction of dates

If two dates (using any of the date or date/time classes) are subtracted, **R** will return the results in the form of a date difference. For example, S&P 500 index experienced a crash on July 13, 2008, and another on November 14, 2008. To calculate the date interval between the two crashes, we can simply subtract the two dates, using any of the classes that have been introduced.

```
# Subtraction between two dates
first_crash <- as.Date("2008-07-13")
second_crash <- as.Date("2008-11-14")
second_crash - first_crash
Time difference of 124 days
```

You can also use the **difftime(…)** function to figure out the date or time interval.

```
# Alternative way to find date interval
difftime(second_crash, first_crash)
Time difference of 124 days
```

2. Operators

1) Vectorized operations

Many operations in **R** are vectorized. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.

```
# Comparing two vectors
samsung <- c(223.28, 226.23, 234.58, 228.29, 230.75)
apple <- c(231.24, 224.95, 235.42, 226.27, 229.26)
samsung > apple
[1] FALSE  TRUE FALSE  TRUE  TRUE

samsung > 230
[1] FALSE FALSE  TRUE FALSE  TRUE
```

2) And / Or

In **R**, the operator " | " and " & " indicate the logical operations OR and AND, respectively. For example, to test both conditions hold you, use &, and to test only one condition holds you, use | as follows:

```
# Testing both conditions hold
(samsung > 220) & (samsung < 230)
[1]  TRUE  TRUE FALSE  TRUE FALSE

# Testing only one condition holds
(samsung < 227) | (samsung > 230)
[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

3) Negation

" ! " is called Logical NOT operator. It takes each element of the vector and gives the opposite logical value.

```
# Negate False
!FALSE
[1] TRUE

# Negate a condition
!(samsung < 128)
[1] TRUE TRUE TRUE TRUE TRUE
```

3. If statements

The syntax of if statement is:

```
if (test expression) {
  statement
}
```

If the *test expression* is TRUE, the *statement* gets executed. But, if it is FALSE, nothing happens. The *test expression* can be a logical or numeric vector, but only the first element is taken into consideration. In the case of numeric vector, zero is taken as FALSE, rest as TRUE.

```
# If statement
IBM <- 120
if (IBM > 110) {
  print ("IBM is greater than 110")
}
[1] "IBM is greater than 110"
```

1) If….else statement

The syntax of if….else statement is:

```
if (test expression) {
  statement 1
} else {
  statement 2
}
```

The else part is optional and is only evaluated if *test expression* is FALSE. It is important to note that else must be in the same line as the closing braces of the if statement.

```
# If ... else statement
IBM <- 120
if (IBM < 110) {
  print ("IBM is less than 110")
} else {
  print ("IBM is greater than 110")
}
[1] "IBM is greater than 110"
```

2) If….else ladder

The if …. else ladder (if …. else …. if) statement allows you execute a block of code among more than 2 alternatives. The syntax of it is:

```
if (test expression 1) {
  statement 1
} else if (test expression 2) {
  statement 2
} else if (test expression 3) {
  statement 3
} else (test expression 4) {
  statement 4
}
```

```
# If ... else ladder
IBM <- 120
if (IBM < 100) {
  print ("Buy!")
} else if (IBM >= 100 & IBM < 110) {
  print ("Keep watching!")
} else {
  print ("Sell!")
}
[1] "Sell!"
```

3) Multiple if statements (if statement inside an if statement)

In some cases, you will need to make more multiple choices in **R**. The most intuitive way to solve more complicated conditions is just to chain the choices. The following is the structure of nested if statements.

```
# Nested if statement
IBM <- 120
IBMshares <- 50
if (IBM < 100) {
  print ("Buy!")
} else if (IBM >= 100 & IBM < 110) {
  print ("Keep watching!")
} else {
  if (IBMshares >= 50) {
    print ("Sell!")
  } else {
    print ("You do not have enough shares to sell.")
  }
}
[1] "Sell!"
```

4) ifelse(…) function

ifelse(…) function is a shorthand function to the tradition if … else statement. The syntax of **ifelse(…)** function is:

```
ifelse (test expression, x, y)
```

The *test expression* should be a logical vector. The return value is a vector with the same length as *test expression*. This returned vector has element from x if the corresponding value of *test expression* is

TRUE or from y if the corresponding value of *test expression* is FALSE. Suppose you have a vector of stock prices, and you want to return "Buy!" whenever IBM < 100, and "Keep watching", otherwise. You will not be able to do this using a simple if statement. **ifelse(…)** function makes it possible to do that.

```
# ifelse function
IBM <- c(110.24, 109.95, 99.42, 111.27, 98.26)
ifelse (IBM < 100, yes = "Buy!", no = "Keep watching!")
[1] "Keep watching!" "Keep watching!" "Buy!" "Keep watching!" "Buy!"
```

4. Loops

A loop is a way to repeat a sequence of instructions under certain conditions. It allows you to automate parts of your code that are in need of repetition. Now let us take a look at **repeat** loop first.

1) repeat loop

A **repeat** loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop. The syntax of **repeat** loop is:

```
repeat {
  statement
  if (test expression) {
    break
}
```

You have to put a condition explicitly inside the body of the loop and use the **break** statement to exit the loop. Otherwise, your code will result into an infinite loop.

```
# repeat loop
IBM <- 120
repeat {
  print (IBM)
  IBM = IBM + 1
  if (IBM == 130) {
    break
  }
}
[1] 120
[1] 121
[1] 122
[1] 123
[1] 124
[1] 125
[1] 126
[1] 127
```

```
[1] 128
[1] 129
```

In the example above, the code inside the loop is first executed before checking the condition. On the other hand, you can also check the condition first and then run the *statement* as follows:

```
# repeat loop: checking condition first
repeat {
  if (IBM == 130) {
    break
  }
  print (IBM)
  IBM = IBM + 1
}
[1] 120
[1] 121
[1] 122
[1] 123
[1] 124
[1] 125
[1] 126
[1] 127
[1] 128
[1] 129
```

2) while loop

A **while** loop is used to loop until a specific condition is met. The syntax of **while** loop is:

```
while (test expression) {
  statement
}
```

The *test expression* is evaluated and the body of the loop is entered if the result is TRUE. The *statement* is executed, and the flow returns to evaluate *test expression* again. This process is repeated until *test expression* evaluates to FALSE. No need for a break statement because the condition is checked at each iteration. For example, suppose you have 1,000 shares of IBM stock that you have to sell. Every day you sell 200 shares until you have nothing. You will use a loop to model the process of selling off the stocks every day, where each iteration you decrease your total share and print out the new total.

```
# while loop
shares <- 1000
while (shares > 0) {
  shares <- shares - 200
  print (paste("Shares remaining:", shares))
}
[1] "Shares remaining: 800"
[1] "Shares remaining: 600"
[1] "Shares remaining: 400"
```

```
[1] "Shares remaining: 200"
[1] "Shares remaining: 0"
```

A **paste** function used in the example above is used to concatenate vectors by converting them into character.

What if you want to see graphically the decrease of shares remaining? You can still use a **while** loop but will need to append the shares remaining at each iteration to a plot.

```
# Initial setting for visualization
shares <- 1000              # initial shares
i <- 0                      # x axis counter
x_axis <- i                 # x axis
y_axis <- shares            # y axis
plot (x_axis, y_axis, xlim = c(0, 5), ylim = c(0, 1000))      # initial plot
```

The description of the initial setting above is as follows:

shares: your current shares

i: incremented each time your shares reduced. The next point on the x axis.

x_axis: a vector of i's. The x axis for the plots.

y_axis: a vector of shares. The y axis for the plots.

Then, you create the initial plot. Just single point of current shares.

```
# while loop for visualization
while (shares > 0) {
  shares <- shares - 200
  i <- i + 1
  x_axis <- c(x_axis, i)
  y_axis <- c(y_axis, shares)

  plot (x_axis, y_axis, xlim = c(0, 5), ylim = c(0, 1000))
  }
```

Now, you created a while loop. While you still have shares remaining:

shares decrease by 200,

i is incremented,

x_axis is extended by 1 more point,

y_axis is extended by the next shares point,

and the plot is created from the updated data.

3) For loop

A **for** loop is used to iterate over a vector in **R**. The syntax of **for** loop is:

```
for (val in sequence) {
  statement
}
```

Here, the *sequence* is a vector and *val* takes on each of its value during the loop. In each iteration, *statement* is evaluated.

Let us take a look at simple example. The following example is to count the number of even numbers in a vector.

```
# Count the number of even numbers in a vector
x <- c(1, 2, 3, 6, 8, 9, 12, 15, 18, 20)
count <- 0
for (val in x) {
  if (val %% 2 == 0) {
    count = count + 1
  }
}
print(count)
```