Introduction to R

Sunthud Pornprasertmanit Kimberly Gibson Scott Drotar Terrence Jorgensen

University of Kansas

R Console

R is a syntax-based program for statistical analysis. All data, statistical results, and variables are saved in objects. For example, we would like to make a constant, `x`, and save it as an object. We can do it by,

```
x <- 10
```

You can view the constant again by just typing it in the program.

```
x
```

The arrow, `<-`, means "is assigned as." You can use equal sign instead.

```
y = 5
```

We can evaluate lines of code from an R editor window by highlighting the line or lines and typing 'Ctrl-R' (in Windows).

Even though we can assign values to variables in either way, I recommend using the arrow sign. The equal sign is not intuitive and, sometimes, we can confuse with double equal sign, ==, which is covered in the later section.

We may save a variable as text. We need to use the double quotation marks to cover the text, such as

```
z <- "Hello World!"
```

We may also use single quotes for strings, such as

```
zz <- 'Hello World!'
```

If we look at both `z` and `zz`, they both contain the save value.

```
z
zz
```

R is a mix between statistics packages and programming languages, which is its advantage. Thus, some characteristics are inherited from programming languages, such as escape character, `\`. Let's try

```
m <- "c:\windows"        # It's gonna be a warning. It will just leave out the '\'
```

Latest Updated: February 9, 2013

```
n <- "c:/windows"        # It's gonna be okay.
```

R will recognize "\w" as a command, which does not exist. So, it shows a warning. The common escape character commands are

\\	Typing backslash as text
\"	Typing double quotation as text
\n	New Line
\t	Tab

The new line and tab will not show in R GUI. It will show in the character.

```
cat("turn now")
cat("\turn \now")
```

Don't worry about it right now. The basic idea is just to be careful about the escape character.

The number sign is used for comments. R will skip every character after the # number sign until it finds new line command. Thus, you can copy and paste this example including comment lines without any errors.

Summarize Data and Missing Values

R provides some default data for us. We will use some of them. We will use the `airquality` data for illustrating some functions. We can see the details of this data by using the "?" function

```
?airquality
```

We can summarize the data by typing

```
summary(airquality)
```

The result shows the descriptive statistics of each variable, including

Min.	=	Minimum
1st Qu.	=	The first quantile (25th percentile)
Median	=	Median
Mean	=	Mean, Average
3rd Qu.	=	The third quantile (75th percentile)
Max.	=	Maximum
NA's	=	Number of missing values (which is not shown when the variable does not have any missing values)

You can save the result as another object to use later

```
sum.airquality <- summary(airquality)
```

See the summary again by typing

```
sum.airquality
```

We can see the raw data by typing

```
airquality
```

You can see a missing value in some cells as `NA`, such as the `Ozone` variable of the fifth case. We can see dimensions of the data by

```
dim(airquality)
```

We can save the number of rows (cases) and number of columns (variables) as objects by

```
n <- dim(airquality)[1]
p <- dim(airquality)[2]
```

Alternatively, we can use

```
n <- nrow(airquality)
p <- ncol(airquality)
cat("\"airquality\" is a", n, "by", p, "matrix")
```

To see the first several cases or the last cases, we can use the `head` or `tail` function, respectively:

```
head(airquality)
tail(airquality)
```

We can see list of objects by

```
ls()
```

If we do not use some objects, we can remove the object from the workspace by `rm()`,

```
rm(zz)
```

R will delete the `zz` object from the workspace. We can clear all objects from the workspace by

```
rm(list = ls())
```

Let's run a regression analysis using wind speed predicting temperature

```
result <- lm(Temp ~ Wind, data = airquality)
summary(result)
```

Descriptive Statistics

The variable in the data frame cannot be manipulated directly, such as

```
Ozone          # It is going to be error
```

We can select only one variable from the data by using the dollar sign with a variable name after the dataset:

```
airquality$Ozone
```

We may save the extracted vector by assigning it to the new object:

```
air.ozone <- airquality$Ozone
```

We can find mean and sum of each variable in the `airquality` dataset by the `colMeans` and `colSums` functions. We can find the standard deviation of each variable by using `apply` to apply the `sd` function to each column in the dataset:

```
colMeans(airquality)
colSums(airquality)
apply(airquality, MARGIN = 2, FUN = sd)
```

You will see that the provided outputs are `NA`, which means that there are missing values in the variable. To ignore the missing data, we can set the attribute `na.rm`:

```
colMeans(airquality, na.rm = TRUE)
apply(airquality, 2, sd, na.rm = TRUE)
```

For `apply`, the first argument is any target dataset. The second argument is 2 meaning that applies the specified function on columns (1 = rows, 2 = columns). The third argument is a desired function. Then, you may add other arguments relating to the desired function, such as `na.rm = TRUE`. We can see the details in the help pages:

```
?mean
?sd
?sum
?apply
```

We do not need to save the variable to an object and then run the descriptive statistics. We can use the functions on the dataset with the dollar sign for selecting a variable directly:

```
meantemp <- mean(airquality$Temp, na.rm = TRUE)
```

Exercise 1

Use `attitude` data. Find the summary, number of rows, number of columns, and run a regression analysis predicting overall rating, `rating`, by attitude toward handling employee complaints, `complaints`. Extract the `rating` variable and find the mean and standard deviation of this variable. Find the means and standard deviations of each variable in the dataset.

Operations: Mathematics, Boolean

We can use basic arithmetic operations for numbers or variables: Add, Subtract, Multiply, Division

```
50 + 70
(55 - 75) * 2
(airquality$Temp - 32) / 9
```

Even power or square root

```
2^3
8^(1/3)
sqrt(airquality$Temp)
```

We can use the result of any operation to replace the existing object, even if the object is in the right hand side of the operation.

```
x <- 7 + 6
x <- x / 2          # The old x (13) was replaced by (13/2 or 6.5)
```

R provides some mathematical constants, such as

```
pi
exp(1) # "e"
```

Sometimes, we would like to find a remainder of division function, by `%%`

```
5%%2
```

Boolean operations relates to operations of `TRUE` and `FALSE` values. To compare two values, the possible commands are

==	equal
!=	not equal
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

The = can be used for <- in assigning values. However, the = is not recommended because some people interpret = as equal not assign

```
20 > 5
x <- 7
x != 7
y <- 5:15
z <- 15:5
y <= z
```

As you can see, the possible results are `TRUE` (`T`) and `FALSE` (`F`). We can manipulate the Boolean result by

`&`	and	(T & T = T;	T & F = F;	F & T = F;	F & F = F)
`|`	or	(T | T = T;	T | F = T;	F | T = T;	F | F = F)
!	not	(!T = F;	!F = T)		

For example,

```
y <- 5:15
(y > 7) & (y < 12)
(airquality$Temp > 90) | (airquality$Wind > 15)
```

We can use the Boolean result to select some desired cases.

```
select <- (airquality$Temp > 90) | (airquality$Wind > 15)
airquality[select,]
```

Creating Vectors and Data Management

As you have seen, we can use `:` to create a sequence vector

```
1:5
8:-2
```

The `seq` function can create a sequence when the interval is greater than 1: `seq(first value, destination, step)`

```
seq(1, 20, 5)
seq(100, 1, -7)
```

We can create a vector with the repeat scores: `rep(value, repeated times)`

```
rep(20, 7)
rep(NA, 4)
```

Sometimes, we would like to create a vector manually. We can combine value to be a vector by `c(,)`

```
c(3, 5)
x <- c(5, 2, 7, -9)
y <- c(rep(1,5), rep(2,5))
```

For the data frame, sometimes we want to add a new case; we can use the `rbind` function

```
new <- c(18, 150, 10.2, 70, 10, 1)     # New airquality case
airquality2 <- rbind(airquality, new)
```

To add a new variable, there are two possible ways: `cbind`, `data.frame`

```
group <- c(rep(1, 100), rep(2, 53))
airquality3 <- cbind(airquality, group)
airquality4 <- data.frame(airquality, group)
```

Getting Help in R

R has thousands of functions spread out over hundreds of different packages, and it would be impossible to memorize them all. So, what do we do when we don't know what a function does?

Help from within R: "`?`" and "`??`"

```
??ttest
```

Bring up an information window that lists any function from one of your packages that has any association with a *t*-test. They are listed as "`package::function`" followed by a very brief description.

```
?t.test
```

Pull up an internet window with everything about the function "`t.test`" nicely laid out. All help files have the same basic format. Describes structure of the function's syntax, lists what the output is, gives related functions, and often provides example code.

What if we want to know every function from a given package?

www.r-project.org is a gold mine of information. Click 'CRAN' and then choose a mirror site. Click 'packages' to bring up a list of every "approved" package. Every package has a .pdf help file that tells you everything about the functions that the package gives you.

What if all we know is that we want to compute an odds ratio?

Google is your best friend when using R. Search "R odds ratio" and you'll get tons of good stuff. There are always numerous ways to accomplish a given task. Since R is open-source there are lots of very active forums where you can find answers (or pose new questions).

Exercise 2

1. Save a number 5 in the object `a`. Create a new object `b` that is equal to $1 - \frac{a^2}{5}$.
2. Check whether the object `a` is in between 2 and 4 (include 2 and 4 too).
3. Select `attitude` data that have `rating` over 50.
4. Select `attitude` data that have `rating` and `complaints` over their means.
5. Select `attitude` data that the number of rows is the multiplicity of 3.
6. Add a new variable in the attitude data indicating whether rating score was greater than 50 or not.
7. Create a vector `vec1` containing values 3, 6, 8, and 9. Create a vector `vec2` containing values 1 to 4. Then, add them together.
8. Let's check `?attitude`, `?seq`, and `?sum`.
9. Search "Centering R" in any search engines and try the example from the help page of the function you found. (?scale)

Importing Data

R can import many types of data files. Some types are supported within the base packages, while others require an additional package, such as '`foreign`'. This template will demonstrate the import of data in comma-separated values (CSV) format, tab-delimited format, and SPSS data format.

The most-used function to import data into R is `read.table()`. First, let's pull up the help file for read.table()

```
?read.table
```

Notice that `read.table` has some special versions `read.csv` (for comma-separated values) and `read.delim` (for tab-delimited). We can also just use `read.table` and specify the appropriate separator (`sep = ","` or `sep = "\t"`).

Before we can import data, we have to tell R where to look for the data file. You can either specify a full path name, or change the working directory. Let's change the working directory:

```
setwd("C:/RSem")
```

We can also use the **File → Change dir...** menu item in Windows. In Mac OS X, it is **Miscellaneous → Change working directory**. Now, if our data files are in the working directory, we are ready to import our data.

```
getwd()          # show us the working directory
list.files()     # show us what files are in the current working directory
```

For a comma-separated values file, with a top row of column names and the value of -999 for missing data values:

```
data.csv <- read.table(file = "RSem.csv", header = TRUE, sep = ",",
      na.strings = -999)
```

For a tab-delimited file, with no column names and `-999` for the missing data value

```
data.txt <- read.table("RSem.txt", sep = "\t", na.strings = -999)
```

Now to import SPSS data, we must first install and load a package called 'foreign'.

The easiest way to install and load packages is to use the Packages menu.

1) **Packages → Install package(s)...**
2) Select a mirror site close to you
3) Scroll to package '`foreign`' and click **OK**

To load the package, you can follow a similar procedure:

1) **Packages → Load package...**
2) Scroll to package '`foreign`' and click '**OK**'

or you can use `install.packages()` and `library()`

```
install.packages("foreign")
library(foreign)
```

Now we are ready to read in an SPSS data file. First, let's look at the help file

```
?read.spss
```

Import SPSS data

```
data.spss <- read.spss(file = "RSem.sav", to.data.frame = TRUE)
```

Ignore the warning message now. It will be okay. The better way to import the data from SPSS is to save the data by the SPSS program as a `*.csv` file and use `read.csv` to import the data to R.

So now that we have loaded data, how can we tell if loaded correctly? First, check dimensions (number of rows and columns)

```
dim(data.csv)
```

Next, check column names (if `header=TRUE`)

```
colnames(data.csv)
```

If it's a large matrix or dataframe, we could view a few rows or columns

```
data.csv[1:10,]          # show the first 10 rows
```

```
data.csv[,1:5]          # show the first 5 columns
```

Exporting Data

Similar to `read.table()`, the most-used export function is `write.table()`. It has similar options. Let's look at the help page:

```
?write.table
```

Now, let's export our data into different formats than it was originally imported

First, export the data from the csv file into a tab-delimited file without row names

```
write.table(data.csv, file = "RSem.out.txt", sep = "\t", row.names =
            FALSE, col.names = TRUE)
```

Next, let's export the data from the tab-delimited file into a csv file with `NA` values written as -999

```
write.table(data.txt, file = "RSem.out.csv", sep = ",", na = "-999")
```

Exercise 3

Export the `airquality` data to file "`RSem2.csv`" using comma separated value and set `NA` equal to 999. Then, read the file back in and save to an object, `RSem`.

Exploring Data

To begin with, we can see the distribution of a variable by creating a histogram

```
hist(airquality$Temp)
```

Edit number of bins

```
hist(airquality$Temp, breaks = 15)
```

We can see the distribution of a single variable to normal distribution by q-q plot

```
qqnorm(airquality$Temp)
```

We can put the diagonal line to see the discrepancy between observed and expected quantiles.

```
qqline(airquality$Temp)
```

Modifying graphs

`col`	colors
`main`	Title of the figure

`sub` Subtitle of the figure
`xlab` X-axis label
`ylab` Y-axis label

For example,

```
hist(airquality$Temp, ylab = "Frequency", xlab = "Air Temperature",
     main = "New York Air Quality in 1973", sub = "Figure 1", col =
     "yellow")
```

We may see the distribution of a variable by boxplot, both including all data or separating by groups

```
boxplot(airquality$Temp)
boxplot(airquality$Temp ~ airquality$Month)
```

The scatterplot can be created by plot function

```
plot(airquality$Ozone, airquality$Wind)
```

To put the regression line on the plot, we need to create the result of linear model analysis as an object. Then, create the regression line from the object.

```
fit <- lm(airquality$Wind ~ airquality$Ozone)  # Create linear model object (Y ~ X)
abline(fit, col = "red")                       # Put the fit line on the plot
```

We can see the equation by

```
fit
```

The regression line: predicted wind = 12.61 - 0.065(Ozone)

Group Statistics

We use the '`aggregate`' function to create group statistics. For example, mean and *SD* of temperature in each month

```
aggregate(Temp ~ Month, data = airquality, mean)
aggregate(Temp ~ Month, data = airquality, sd)
```

Let create a new variable to see whether the case is the first half of the month.

```
FirstHalf <- airquality$Day <= 15
airquality2 <- cbind(airquality, FirstHalf)
```

Now, let's find the mean and *SD* of temperature divided by Month and FirstHalf

```
aggregate(Temp ~ Month + FirstHalf, data = airquality2, mean)
```

```
aggregate(Temp ~ Month + FirstHalf, data = airquality2, sd)
```

We can see the descriptive statistics of multiple variables at once.

```
aggregate(cbind(Temp, Ozone, Wind) ~ Month, data = airquality, mean)
```

Exercise 4

1. Find the result of the analyzed regression that we used to create the regression line.
2. Create a histogram of `rating` variable in the `attitude` data.
3. Create a scatterplot of `complaints` (on X axis) and `rating` (on Y axis) variables and impose the regression line in the scatterplot.
4. Create a new variable representing a median split of `complaints` (check `?median`). Then, create a boxplot of rating by two groups (high vs. low rating on handling employee complaints).
5. Check the new dataset, `Orange`. Find group means and standard deviations of `circumference` by `Tree`.

Data Analysis

One-sample *t*-test

We will use the `t.test` function for all *t*-test for comparing means. The function attributes are different across types of analysis. First, a one-sample *t* test

```
t.test(airquality$Temp, mu = 70)
```

This analysis would be used to see whether the population mean of the temperature is different from 70. The result showed that the average temperature is 77.88, $t(152) = 10.30$, $p < .001$ (in the result $p = 2.2 \times 10^{-16}$).

Independent *t*-test

For the independent-samples *t* test, I would like to create the "`summer`" variable in the `airquality` data. May, June, and July will be true in the summer variable. August and September will be false in the variable. Next, compare the temperature between true and false in the summer variable, by independent *t*-test

```
summer <- airquality$Month <= 7
t.test(airquality$Temp ~ summer)
```

CAUTION: the default independent *t*-test assumes that the variances are not equal across groups. Thus, R uses the corrected *t*-test. To use the traditional independent *t* test, the `var.equal` attribute should be specified as `TRUE`

```
t.test(airquality$Temp ~ summer, var.equal = TRUE)
```

Assuming equal variances, the means of the true and false group are 76.15 and 80.49, respectively. The difference is significant, $t(151) = 2.84$, $p = .005$.

Paired-sample *t*-test

I will introduce new data, `attitude`

```
?attitude
```

In the `attitude` data, I would like to compare the average attitudes toward opportunity to learn and advancement. The paired-samples *t* test will be used by specifying the paired argument as `TRUE`.

```
t.test(attitude$learning, attitude$advance, paired = TRUE)
```

The average difference is 13.43. The difference is significant, $t(29) = 6.85$, $p < .001$.

Correlation, Covariance

We can analyze correlation test by the `cor.test` function.

```
cor.test(attitude$learning, attitude$advance)
```

The attitudes toward opportunity to learn and advancement are significantly positively correlated, $r(28) = .53$, $p = .003$.

Sometimes, we do not want to test the correlation. We only want to create a correlation matrix for further analysis. We can use the `cor` function and apply for a data frame.

```
cor(attitude)
# just the lower diagonal
as.dist(cor(attitude))
```

We can do the same thing for variance–covariance matrix by

```
cov(attitude)
```

Try to find the correlation matrix for the `airquality` data. You will see some `NA`s in the correlation matrix. Use help function to find out how to deal with `NA` in the correlation function.

Chi-square: Goodness-of-Fit

For the chi-square test, we may analyze for the Goodness-of-Fit and contingency table. For the Goodness-of-Fit test, we have a variable and would like to see the observed frequencies in each cell in the population are different from expected frequencies. For example, we obtained 90 males and 83 females in our sample. We want to test the bias in the sampling process, assuming that the population proportions of each sex are equal.

```
chisq.test(c(90, 83))
```

The result found that the sampling process is not biased, $\chi^2(1, N = 173) = 0.28$, $p = .59$. We may change the expected proportion

```
chisq.test(c(90, 83), p = c(0.6, 0.4))
```

Sometimes, we do not have frequencies. We have raw data instead, such as Month in the `airquality` data. We need to transform the raw data to frequencies before using the Goodness-of-fit test, by the `table` function.

```
chisq.test(table(airquality$Month))
```

Chi-square: Contingency table

For the contingency table, I will introduce a new dataset, `HairEyeColor` data.

```
?HairEyeColor
```

This data is arranged in three-dimension crosstabs. The first dimension is on row, `Hair`. The second dimension is on column, `Eye`. The third dimension is on layer, `Sex`. I would like to ignore row dimension, hair. Let's say that I am interested in people who have black hair only. So, I make the new data by

```
black <- HairEyeColor[1, , ]
```

Then, the chi-square test is used to see whether, in the sample with black hair, the proportions of eye color are different across sex.

```
chisq.test(black)
```

The result found that the proportions of eye color is not significantly different across sex, $\chi^2(3, N = 108) = 2.16$, $p = .54$. Note that some cells will have expected values less than 5. The approximate chi-squared may be not correct. We may use the Fisher's Exact test by

```
fisher.test(black)
```

Sometimes, we have raw data. We need to make the contingency table before running the chi-squared test. The table function is still helpful. For example, I would like to see whether the missing value of the `Ozone` variable in the `airquality` data is systematically related to `Month`. First, we create a new variable called `Ozone.NA` to be true when the `Ozone` is missing in a particular case

```
Ozone.NA <- is.na(airquality$Ozone)
```

where `is.na` is the function to see whether the value is missing

Next, make the contingency tables of Month and the missing pattern of `Ozone`

```
MonthOzoneNA <- table(airquality$Month, Ozone.NA)
```

Finally, analyze by the chi-square test

```
chisq.test(MonthOzoneNA)
```

The result found that the missing pattern of the `Ozone` variable and `Month` when the data obtained are significantly related, $\chi^2(4, N = 153) = 44.75, p < .001$.

Exercise 5

1. Check the `cars` dataset. Find the correlation between both variables.
2. Check the `Seatbelts` dataset. Find whether the averages of car driver killed are different between law effective months. Note that this dataset is not saved in an appropriate format. Use

```
Seatbelts2 <- as.data.frame(Seatbelts)
```

   to transform the data to an appropriate format (in the `data.frame` format in R). Then, use the appropriate statistic for the new data.
3. Check `occupationalStatus` dataset. Use chi-square to detect a relationship between both variables.

Basic Programming

In this part, we will introduce '`for`' and '`if`,' which is usually used in programming.

'`for`' is used for doing something repeatedly for a number of times. The structure of '`for`' is

```
for(i in 1:100) {
        statements
}
```

'`i`' is a counter variable. '`i`' will start from 1 and do the statement and then change to 2 and do the statement and so on. Finally, the '`i`' will be 100, do the statement, and stop the loop. For example,

```
x <- 0
for(i in 1:5) {
      x <- x + i
}
```

`x` start at 0.

The first loop,	`x <- 0 + 1,`	which is 1.
The second loop,	`x <- 1 + 2,`	which is 3.
The third loop,	`x <- 3 + 3,`	which is 6.
The fourth loop,	`x <- 6 + 4,`	which is 10.
The fifth loop,	`x <- 10 + 5,`	which is 15.

`x` will be 15.

As another example,

```
x <- rep(NA, 10)
x[1] <- 1
x[2] <- 1
for(i in 3:10) {
      x[i] <- x[i - 1] + x[i - 2]
}
```

`x` is a vector. The first and second elements are 1. The `i` will start at 3.

The first loop,	`x[3] <- x[2] + x[1],`	which is 1 + 1 = 2
The second loop,	`x[4] <- x[3] + x[2],`	which is 2 + 1 = 3
The third loop,	`x[5] <- x[4] + x[3],`	which is 3 + 2 = 5
The fourth loop,	`x[6] <- x[5] + x[4],`	which is 5 + 3 = 8
The fifth loop,	`x[7] <- x[6] + x[5],`	which is 8 + 5 = 13
The sixth loop,	`x[8] <- x[7] + x[6],`	which is 13 + 8 = 21
The seventh loop,	`x[9] <- x[8] + x[7],`	which is 21 + 13 = 34
The eighth loop,	`x[10] <- x[9] + x[8],`	which is 34 + 21 = 55

`x` will contain the two starting values and eight result values.

As shown above, the '`for`' loop can go through each element in a vector. It can go through each element in a data frame. If it use with the '`if`' function, it is very powerful.

'`if`' is used for doing something when a condition is satisfied. The structure of '`if`' is

```
if(condition) statement
```

```
x <- 100
if(x > 50) cat("x is greater than 50.")
```

where the `cat` function is used for printing text in the screen.

In this structure, we can put only one line of code. Another structure is

```
if(condition) {
      statements
}
```

```
y <- 101
if(y %% 2 == 1) {
      cat("y is odd.\n")
      z <- (y - 1) / 2
      cat(c("The quotient is", z, ".\n"))
}
```

'`if ... else`' is doing one thing if a condition is satisfied and doing another if the condition is not satisfied. The structure is

```
if(condition) {
        statements
} else {
        statements
}
```

```
x <- 100
y <- 100
if(y > x) {
      cat("y is more than x.")
} else {
      cat("y is not more than x.")
}
```

We can combine both '`if`' and '`for`' together. For example, we would like to create a new variable that codes '`1`' when the value below a median of a variable and '`0`' when the value is above the median of the variable.

```
med <- median(airquality$Wind)
n <- dim(airquality)[1]
group <- rep(NA, n)
for(i in 1:n) {
   if(airquality$Wind[i] > med) {
      group[i] <- 1
   } else {
      group[i] <- 2
   }
}
```

'`if ...else if … else`' is used if we have more than two conditions. The structure is

```
if(condition) {
        statements
} else if(condition) {
        statements
} else {
        statements
}
```

```
x <- 100
y <- 100
if(y > x) {
      cat("y is more than x.")
} else if(y == x) {
      cat("y is equal to x.")
} else {
```

```
      cat("y is less than x.")
}
```

Creating Functions

Sometimes, some commands are used repeatedly, such as the median split shown above. It would be nice to create a function that take some inputs, computes something, and creates outputs for us. The structure is

```
Name <- function(inputs)
{
        operations
        return(outputs)
}
```

For example, if we would like to create a function that transforms temperature in Fahrenheit to Celsius, like typing `F_to_C(50)` and the program answers 10. The function can be created by

```
F_to_C <- function(f)              # F_to_C is the name of function, f is input variable
{
   result <- ((f - 32) / 9) * 5    # Create a new variable inside the function
   return(result)                  # The function will use the result variable as output.
}
F_to_C(50)
```

As another example, we can create a function for median split

```
MedSplit <- function(x)            # New function that take a vector x.
{
   med <- median(x)
   n <- length(x)
   group <- rep(NA, n)             # Create a new variable as a result
   for(i in 1:n) {
      if(x[i] > med) {
         group[i] <- 1
      } else {
         group[i] <- 2
      }
   }
   return(group)
}
```

R Script

Sometimes, we need to run the same codes later. The R commands should be saved and be ready for later use. R script is designed for this task. All R script has the extension .R. We can also use the **File Open Script…** menu item in Windows. Please open the **RSem.R** file from your file directory.

The script can be run by highlighting a part of code and clicking '**Run line or selection**' or using `Ctrl+R`.

We may make a new script file by **File → New Script**. This script can be saved for later use.

Exercise 6

1. Save all codes you write for the following items into an R script.
2. Use `for` loop to subtract 1000 by 1, 2, 3, …, and 20. The result should be 790.
3. Use `if ... else` statement to check whether `x <- 22` is odd or even.
4. Create a new variable called `ratingclass` to represent the degree of the `rating` variable in the `attitude` dataset. Use `for` loop and `if ... else` function to classify as 1 for the rating score lower than 50, 2 for the rating score from 50 to70, and 3 for the rating score greater than 70.
5. Write a function to return `TRUE` if an input number is odd and return `FALSE` if the input number is even.

Alternative Programs

RStudio

This program integrates R script, R console, graphs, and the list of objects in a workspace in a single window. This program is highly recommended for R beginners. See http://rstudio.org/ for further details.

Notepad++ and NppToR

The `Notepad++` is an improved Notepad in Windows. This program provides a lot of nice features for programming, including color-coded and a point-and-click command for commenting lines. This program is applicable for most computer language, such as `C`, `JavaScript`, `HTML`, `R`, or `Tex`. The add-on `NppToR` can be used to run R from `Notepad++`. See http://notepad-plus-plus.org/ and http://sourceforge.net/projects/npptor/ for further details.

Emacs

This program is also designed for programming. The connection to R is built in the program. `Emacs` is quite hard to use but very handy. See http://www.gnu.org/software/emacs/ for further details.

Give Us Feedback

This article is used for R Statistical Software Seminar, Center for Research Methods and Data Analysis, University of Kansas. If you find any errors or give suggestions, please let us know at

Satisfaction Survey

https://redcap.ittc.ku.edu/surveys/?s=jNyWou

Sunthud Pornprasertmanit
Center for Research Methods and Data Analysis
University of Kansas
Email: psunthud@ku.edu