# Chapter 3: Functions and Packages

*Kevin Kam Fung Yuen*
*PhD, Senior Lecturer, School of Business, Singapore University of Social Sciences*
*kfyuen@suss.edu.sg, kevinkf.yuen@gmail.com*

*May 27, 2019*

## Contents



## 1 Relational operators

Relational operators are used to determine a TRUE/FALSE value. The operators are listed as blow.

| Notation | Explanation |
|---|---|
| < | Less than |
| > | Greater than |
| == | Equal to |
| != | Not equal to |
| >= | Greater than or equal to |
| <= | Less than or equal to |

There is an important note that we should not confuse the assignment symbol (=) with equality symbol (==).

```
50>=80
50<=100
50 != 80
50!=50
80 >80
80>=80
# what happen to this
50=80
```

If there are more than one conditions, we can combine conditions with and (&&) and or (||).

```
FALSE && FALSE
TRUE && TRUE
TRUE && FALSE
FALSE || FALSE
TRUE || TRUE
TRUE || FALSE
50>=80
50<=100
50>=80 && 50<=100
```

## 2 Conditional Statements

We can use if statements to perform different groups of commands with respect to different conditions. Command(s) can be grouped by braces (‘{ }’ ). The if statements can be used in several ways.

```
*   if (condition 1)  {commands 1}
*   if (condition 1)  {commands 1}  else {commands 2}
*   if (condition 1)  {commands 1}  else if (condition 2)  {commands 2} ,  and more else if statements,
```

If the condition is TRUE, a brace of commands under such condition will be executed. We can use relational operators to determinate the conditions.

```
a=50
if(a>=80 && a<=100)
{
  print("A class")
} else if (a<80 && a>=60)
{
```

```
  print("B class")
} else if (a<60 && a>=40)
{
  print("c class")
} else if (a<40 && a>=0)
{
  print("D class")
} else
{
  print("your input should be between 0 and 100")
}
```

# 3 Repetitive Statements

## 3.1 for Loop

Quite often we have to repeat the same operations for a number of times. We can use *for* loops statement to handle this. The structure of for loop is as below.

```
for (a in A) {
a set of command(s);
}
```

For the statements above, we repeat the same set of commands for each value a in the vector A. Let's take a look for the example below.

```
for (x in 1:10)
{
  print(x)
}
```

In the example above, the function/command *print()* is repeated to call for the values set from 1 to 10. This result is the same as what we do manually as below.

```
print(1)
print(2)
  ...
print(10)
```

Supposed that we need to find a total value of adding the numbers from 1 to 10, the *for* loop function can be used in this way.

```
total = 0;
for (x in 1:10)
{
  total = total + x;
}
total
```

## 3.2 while loop

When we have no information about the number of iterations, we can use while loops statement.

```
while (condition) {
 commands;
}
```

Condition of while loop is identical to if statement. When condition is TRUE, Commands with bracket are repeated. Otherwise, the loop will stop. A very important note is that if the condition is always TRUE, the programme will have unlimited loops.

```
# what is the value from 20 adding to, such that the sum of the closest result is to just over 200.

total = a =20;
while (total  <= 200)
{
  a=a+1;
  total=total+a
}
a
total

# to verify the result of a
# if the value is a-1, what is the sum from 20 to a-1
sum(20:(a-1))
# if the value is a-1, what is the sum from 20 to a
sum(20:a)
# What is the conclusion
```

## 3.3 apply, lapply, and sapply

The *apply()* function returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

```
A=matrix(c(1:12), nrow=3, ncol=4)
apply(A,2,min) # min of each columns
apply(A,1,max) # max of each row
```

The *lapply()* function returns a list of results by applying FUN to the corresponding element of list or vector. The *sapply()* function is a wrapper of *lapply* by default returning a vector or matrix.

```
# input data
xx <- list(a = 2:5, b = exp(-5:6), b = c(TRUE, TRUE,FALSE,FALSE,TRUE))
xx
# use lapply
lapply(xx,sum)
# use sapply
sapply(xx,sum)
```

The apply functions above mainly deal with the return value is independent of the other return value. if the function can perform the data in parallel, apply functions are recommended due to more efficient computation.

# 4 Calling and Creating Functions

## 4.1 Calling Functions

Previously, we have gone through uses of some functions which are the built-in functions. More functions are as below.

```
x = c(4.5,5.6,6.3,2.9,3.0,5.23)
(y =matrix(x,3,2))
length(x)  #number of elements
dim(y)  #number of elements in each dimension in data
abs(x)   #absolute value
sqrt(x)  #square root
ceiling(x) #ceiling value
floor(x)   #floor value
trunc(x)   #truncate value
round(x, digits=4)
signif(x, digits=4)
cos(x)  # cosine
sin(x)  # sine
tan(x)  # tangent
log(x) #natural logarithm
log10(x) # logarithm x of base 10
exp(x)   #e^x
```

## 4.2 Creating Functions

When the built-in function does not satisfy our requirements, we can create our own functions for our own purpose. An example of function format is defined as below.

```
functionName <- function( argument1, argument1 = value2) {
        statement1;
        statement2;
        statement3;
        ....
        statementn;
        return(...)
  }
```

We may set default value for argument. We can use ?log to find the R documentation of log (Figure 1). We will find that *log(x, base = exp(1))* where the *base* is set to *exp(1)* by default.

```
?log
```

In the R document, we can find log with bases 2 and 10, i.e. log2 and log10. If we would like to compute log with base 8, we need to define the default value as 8.

```
x=c(1,8,64,100)
log(x, base=8)
```

Alternatively, we can create a function *log8*. So we can use log8 in the future without specifying the base value.

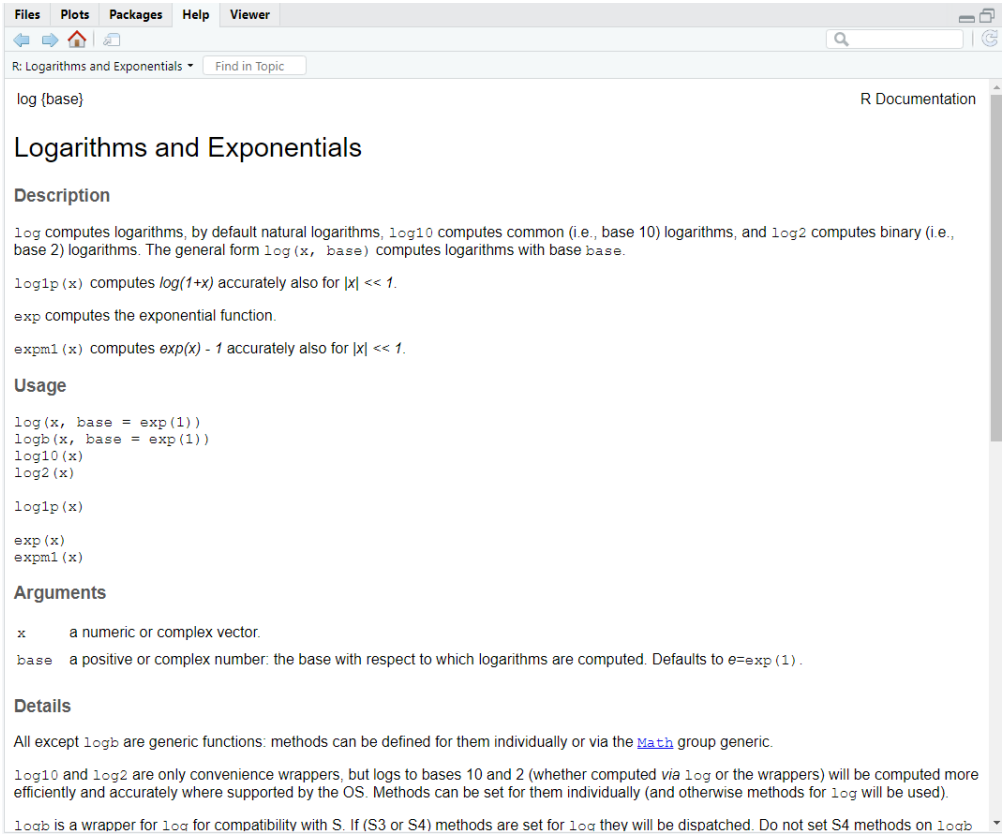


Figure 1: R documentation for log function

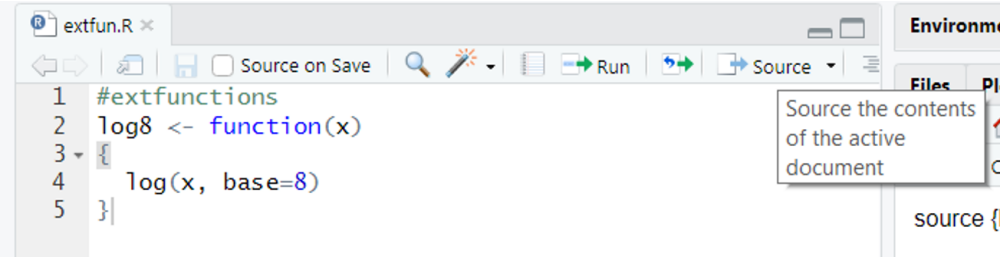


Figure 2: Source a file

```
log8 <- function(x)
{
  log(x, base=8)
}
x  #recall x
log2(x)
log10(x)
log8(x) # call the log8 created by ourselves
```

The function will *return* the value of the last statement within curly brackets *{}*. *return()* function is optional.

### 4.3 Sourcing Functions

Quite often, a separated R file is used to store functions whilst another file is used to execute the functions by referring this R function file with *source* function. for example, if we created a R file called *extfun.R*, which contains user defined functions, we can use the commend as below.

```
source('extfun.R')
```

Alternative, we can click the *Source* button in RStudio UI (Figure 2) to generate and execute the code in the file.

## 5 Installing and Loading R Packages

R functions and datasets can be stored as packages. When a package is loaded, the functions and datasets in the package can be used. We use *library()* to explore what packages are installed in the current R environment.

```
library()
```

If we need to use the package called *MASS* in R, we just use library(datasets) to load a datasets package.

```
library(MASS)
```

Alternatively, we can click the *packages* tab in RStudio, and the packages installed in R library are displayed. When we click the package to be used, the system automatically loads *library(package_name)* to load the package selected. If the packages are not used, we should not click the packages. Loading unnecessary packages consumes unnecessary memory.

We can use *search()* to see which packages are currently loaded.

```
search()
```

We use the double-colon operator :: to call a function in a package, or namespace. For example, the sin() function can be referred as base::sin(). As it is defined in the base package, which is automatically loaded when R environment starts, we do not need to include the full namespace, but just a function name.

```
sin(20)
base::sin(20)
```

Quite often, as R is the open language, same function names are in several packages. Suppose that one function is in two packages. When two packages are loaded one by one, the latter one will replace the former one.

Packages are often inter-dependent, and loading one may cause others to be automatically loaded. The colon operators described above will also cause automatic loading of the associated package. When packages with namespaces are loaded automatically, they are not added to the search list. We can use loadedNamespaces() to find the load list.

```
loadedNamespaces()
```

We can find that "compiler" package is displayed by in the results of loadedNamespaces(), but not in the results of search(). We can use help.start() to start the HTML help system for the available help topics of the packages installed.

Quite often we need to install a new package. If the package is in the repository of CRAN, we can use the *install.packages("newPackageName")* to install the new function.

```
install.packages("e1071")
```

The package may be required to be updated after certain period of time. We can use *update.packages("PackageName")* function.

```
update.packages("e1071")
```

Alternatively, we use RStudio to manage the package installation and update ( by the methods below. • Tools -> Install Packages • Tools -> Check for Updates

We use *sessionInfo()* to display the version information about R, the OS and attached or loaded packages.

```
sessionInfo()
```

# 6 Exercises

## 6.1 Exercise 1

Use *for* loop to replace the R code as below.

```
total=0
(total=total-1)
(total=total-2)
(total=total-3)
(total=total-4)
(total=total-5)
total
```

## 6.2 Exercise 2

Create a summation function by specifying the a low bound value and an upper bound value.

## 6.3 Exercise 3

create a summation function taking a vector of numeric data. Comparing your result with sum.

## 6.4 Exercise 4

Create a product function. Test your results for 2*3*4*5*6...19*20.

## 6.5 Exercise 5

Referring the codes in examples of this chapter, create a function, *score2grade()* to convert a score (0-100) to a grade ("A" to "D") for a test. Test your result as well.

## 6.6 Exercise 6

Use *For* loop to process a vector of scores,{50,40,30,-10,34,90,80,65,100,150}. into a vector of grades by calling self-defined function *score2grad()*.

## 6.7 Exercise 7

Create a transpose function, which works the same as t() in R

## 6.8 Exercise 8

Create a function similar to *apply()* function for sum, which works the same as *apply(...)* taking *sum(...)* in R.

## 6.9 Exercise 9

1. We have R code as below

```
a=c(2,3);b=c(4,5);
```

Which statement(s) below will return a single value of TRUE?

a. a>b || a<b
b. a>b && a<b
c. a>b & a<b
d. a>b | a<b

2. Which options have no syntax error?

```
a.
a=1
if(a==10)
{
  print("a=10")
}

b.
a=1
if(a==10)
{
  print("a=10")
}  else {
  print("a !=10")
}

c.
a=1
if(a==10)
{
  print("a=10")
}  elseif {
  print("a !=10")
}

d.
a=1
if(a=10)
{
  print("a=10")
}
```

3. What is the output for T

```
T=0
for (x in 20:30)
{
  T = T+ x;
}
T
```

a. 30
b. 20
c. 275
d. 0

4. What is result for T?

```
T = 0;
while (T <= 10)
{
  T=T+5
}
T
```

a. 0
b. 10
c. 15
d. 5

5. What will be the result of n in the code below?

```
y =matrix(1:6,3,2)
n=length(y)
n
```

a. n=6
b. n=2
c. n=3
d. n=8

6. According to Newton's Law of Universal Gravitation, there is an equation below F = mg F is The force of gravity on Earth. The acceleration due to gravity, g , is a constant of 9.807. If the mass m is 5, which option(s) below can produce the correct answer to calculate the value of F?

```
a.
F <- function(m)
{
  9.807*m
}
F(5)

b.
F <- function(m)
{
  return(9.807*m)
}
F(5)

c.
F = function(m)
{
  return(9.807*m)
}
F(5)

d.
F = function(a)
{
  a=9.807*m
  return(a)
}
F(5)
```