Writing R Function

1. Basics

Arguments

There are no strict requirements of the types of arguments you put in the function. Arguments can be numerical, factor, character; it can be vector, matrix, data.frame, list, or even another function.

Example1:

```
fun1 <- function(x) {
    y <- 2 * x + 1
    return(y)
}
out <- fun1(1)
out
```

```
## [1] 3
```

```

out <- fun1(c(1:20))
out
```

```
##  [1]  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35
37 39 41
```

Example2:

```
fun2 <- function(x) {
    a <- paste(x, "is easy!", sep = " ")
    return(a)
}
x <- "Writing function"
fun2(x)
```

```
## [1] "Writing function is easy!"
```

```
x2 <- c("Writing function", "Using R", "Learning stat")
fun2(x2)
```

```
## [1] "Writing function is easy!" "Using R is easy!"
## [3] "Learning stat is easy!"
```

By assigning the function output to another variable, the result of running the function is saved in that variable. It can be used as a global variable in the future, such as the “out” in example 1. But when running example 2, I didn't save the function output in any variable, so there is no way to call the results directly in the future.

Namespace

You can name the function with whatever name you like. But keep in mind that if you happen to give two functions the same name, the first function will be covered by the second one.

If you happen to use a name same as a native R function, R will remember your function by that name, and you will have trouble calling the native R function.

Example:

```
rm(list = ls())
x <- rnorm(10)
y <- runif(10)

lm(y ~ x)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)            x
##        0.72        -0.21
```

```
lm <- function(x, y) {
    return(sum(x + y))
}
lm(x, y)
```

```
## [1] 10.2
```

Function output

You can output anything you like, vector, matrix, data.frame, lists. Whatever you didn't output, and is created within the R function will not be saved. That is one of the reasons that we prefer functions most of the time. By default, R only output the last variable in the function.

```
est <- function(x) {
    mean <- mean(x)
    var <- var(x)
}

vector <- seq(0, 100, by = 2)
out <- est(vector)
out
```

```
## [1] 884
```

It is safe to put a line 'return()' at the end of the function. If we need it to output more stuff, such as, both the mean and the variance. We can do

- item Save the vector of c(mean, variance).

  ```
  est <- function(x) {
    mean <- mean(x)
    var <- var(x)
    return(c(mean, var))
  }

  vector <- seq(0, 100, by = 2)
  out <- est(vector)
  out
  ```

  ```
  ## [1]  50 884
  ```

  But I don't quite recommend this lazy way, 'cause you might forget whether the first element is the mean or the variance before long.

- Save it as a data frame, then we can call the output elements with a dollar sign.

```
est <- function(x) {
  mean <- mean(x)
  var <- var(x)
  return(data.frame(mean.X = mean, var.X = var))
}

vector <- seq(0, 100, by = 2)
out <- est(vector)
out
```

```
##   mean.X var.X
## 1     50   884
```

```
out$mean.X
```

```
## [1] 50
```

```
out$var.X
```

```
## [1] 884
```

Most of the time, we want to output more complex things, for example we want one function to output vector, matrix, character string at the same time. We'd better save them to a list.

- Save as a list.

```
est <- function(x) {
  mean <- mean(x)
  var <- var(x)
  y <- x^2 + 1
  z <- outer(x, x)
  out.list <- list(mean.X = mean, var.x = var, Y = y, Z
= z)
  return(out.list)
}

vector <- seq(0, 100, by = 2)
out <- est(vector)
out$Y
```

```
##  [1]     1     5    17    37    65   101   145   197
257   325   401
## [12]   485   577   677   785   901  1025  1157  1297
1445  1601  1765
## [23]  1937  2117  2305  2501  2705  2917  3137  3365
3601  3845  4097
## [34]  4357  4625  4901  5185  5477  5777  6085  6401
6725  7057  7397
## [45]  7745  8101  8465  8837  9217  9605 10001
```

Saving function

Save everything in a .R file, and use “source()” function to call it when you need it. The source() function is actually running that .R file line by line. It is the same as you open the .R file yourself, and highlight the code, and run it.

```
source("TutorOct2/environment.R")
```

```
## Warning: cannot open file 'TutorOct2/environment.R': No
such file or
## directory
```

```
## Error: cannot open the connection
```

This is not always convenient, because it's very possible that some part of the .R file is not necessary to be called. You just put it there when you write that code the first time, but you only want to keep a specific function in that .R file. We can save the function by itself. Save the function as an R object (.RData file).

```
save(est, file = "TutorOct2/function.RData")
```

```
## Warning: cannot open compressed file
'TutorOct2/function.RData', probable
## reason 'No such file or directory'
```

```
## Error: cannot open the connection
```

When using it,

```
attach("TutorOct2/function.RData")
```

```
## Error: file 'TutorOct2/function.RData' not found
```

```
est
```

```
## function(x) {
##     mean <- mean(x)
##     var <- var(x)
##     y <- x^2 + 1
##     z <- outer(x, x)
##     out.list <- list(mean.X = mean, var.x = var, Y = y, Z
= z)
##     return(out.list)
## }
```

2. Function Environments Scope

We always running the program in certain environment. In R, there is the most common global environment.

```
environment()
```

```
## <environment: R_GlobalEnv>
```

Environments are created implicitly by function calls. In this case the environment contains the variables local to the function (including the arguments), and its enclosure is the environment of the currently called function.

Environments may also be created directly by “new.env()”. The “parent.env()” function may be used to access the enclosure of an environment.

Simple example

```
fun <- function(x){
    print(environment())
    print(paste("x=", x, sep=""))
    z <- 3
    print(paste("z=", z, sep=""))
    return(z)
}

out.z <- fun(1)
# <environment: 0x000000003ff65b20>
# [1] "x=1"
# [1] "z=3"

z
# Error: object 'z' not found

x
# Error: object 'x' not found

out.z
# [1] 3
```

Another good and simple example

Taking global environment variables when the function can not find it within the function environment.

```
rm(list=ls())

insidefun <- function(a,b){
    print(environment())
    out <- a+b+z
    print(paste("out=",out, sep=""))
    return(out)
}

out1 <- insidefun(a=1, b=2)
# <environment: 0x00000000404f8250>
# Error in insidefun(a = 1, b = 2) : object 'z' not found

z <- 1
out1 <- insidefun(a=1, b=2)
# <environment: 0x00000000407f9950>

out1
# [1] 4
```

This is a bad habit of coding. It's better to define all the variables in the function, instead of letting R use whatever it can find.

3. Function within function

When defined and called inside another function, a function will try to find the missing variable in the upper level environment.

```
rm(list=ls())

z
# Error: object 'z' not found

fun2 <- function(x, a, b){
    print("In fun2:")
    print(environment())
    print(paste("x=", x, sep=""))
    z <- 3
    print(paste("z=", z, sep=""))

    insidefun <- function(a,b){
        print("In insidefun")
        print(environment())
        out <- a+b+z
        print(paste("out=",out, sep=""))
        return(out)
    }

    print("In fun2")
    print(environment())
    return(insidefun(a, b))
}


out2 <- fun2(1, 2, 3)
# [1] "In fun2:"
# <environment: 0x000000003af29848>
# [1] "x=1"
# [1] "z=3"
# [1] "In fun2"
# <environment: 0x000000003af29848>
# [1] "In insidefun"
# <environment: 0x00000000003f67a0>
# [1] "out=8"


out2
# [1] 8
```

A bad example

If we separate the two functions in the last example:

```
rm(list=ls())

insidefun <- function(a,b){
    print(environment())
    out <- a+b+z
    print(paste("out=",out, sep=""))
    return(out)
}

fun3 <- function(x, a, b){
    print("In fun2:")
    print(environment())
    print(paste("x=", x, sep=""))
    z <- 3
    print(paste("z=", z, sep=""))

    print("In fun2")
    print(environment())
    return(insidefun(a, b))
}
```

Then, when we are calling function “insidefun” within “fun2”, “insidefun” is searching variables in the global environment instead of the environment within “fun2”.

```
out3 <-  fun3(1, 2, 3)
# [1] "In fun2:"
# <environment: 0x0000000040b4dcc0>
# [1] "x=1"
# [1] "z=3"
# [1] "In fun2"
# <environment: 0x0000000040b4dcc0>
# <environment: 0x0000000040b734b0>
# Error in insidefun(a, b) : object 'z' not found

out3
# Error: object 'out3' not found
```

It would work if we have variable z in our global environment.

```
z <- 1
out3 <-  fun3(1, 2, 3)
# [1] "In fun2:"
# <environment: 0x0000000040a50860>
# [1] "x=1"
# [1] "z=3"
# [1] "In fun2"
# <environment: 0x0000000040a50860>
# <environment: 0x0000000040a58028>
# [1] "out=6"

out3
# [1] 6
```

Other tips for writting functions

Default variables

```
def.fun <- function(x=1){
    print(environment())
    print(paste("x=", x, sep=""))
    z <- 3
    print(paste("z=", z, sep=""))
    return(z)
}

def.fun()
# <environment: 0x0000000040d56ff8>
# [1] "x=1"
# [1] "z=3"
# [1] 3
```

Extra variables

```
rm(list=ls())

fun4 <- function(x, ...){

    list <-list(...)

    print("In fun2:")
    print(environment())
    print(paste("x=", x, sep=""))


    print("In fun2")
    print(environment())

    insidefun <- function(a,b, z){
        print(environment())
        out <- a+b+z
        print(paste("out=",out, sep=""))
        return(out)
    }

    return(insidefun(a=list[[1]], b=list[[2]], c=list[[3]]))
}

out4 <- fun4(x=1, a=2, b=3, z=1)
# [1] "In fun2:"
# <environment: 0x000000003fdaab50>
# [1] "x=1"
# [1] "In fun2"
# <environment: 0x000000003fdaab50>
# <environment: 0x000000003fd8f048>
# [1] "out=6"

out4
# [1] 6
```

```
%##============================================================##

%## un-necessary eg
%##============================================================##

%library(lattice)
%my.df <- data.frame(y=c(100, 0.2, 0.3, 1.4, 1.3), x=c(1000,
1, 2, 1, 2))
%
%
%breakme <- function(df, fo=y ~ x) {
%   # I know that first row is garbage
%   modified.df <- df[-1,]
%   # I also want to compute some things here to label my
data
%   modified.df$big <- modified.df$y > 1
%   xyplot(fo, modified.df, groups=modified.df$big,
auto.key=T)
%}
%
%breakme(my.df)
%
%xyplot.formula
%breakme(my.df, fo=x~y)
%
%methods(xyplot)
%# [1] xyplot.formula* xyplot.ts*
%#
%#    Non-visible functions are asterisked
%getAnywhere(xyplot.formula)
```