Lecture 8. R Programming Structures

R and Data Visualization

BIG2006, Hanyang University, Fall 2022

R Programming Structures

R is a block-structured language in the ALGOL-descendant family, such as C, C++, Python, Perl, and so on.

We cover the basic structures of R as a programming language.

Some more details on loops and functions will be covered.

Control Statements

Loops

- There will be one iteration of the loop for each component of vector `x`, with `n` taking on the values of those components.
- In the first iteration, $\mathtt{n} = \mathtt{x}[1]$; in the second iteration, $\mathtt{n} = \mathtt{x}[2]$; and so on.

```
x <- c(5,12,13)
for (n in x) print(n^2)
```

```
## [1] 25
## [1] 144
## [1] 169
```

- C-style looping with `while` and `repeat` is also available.

```r
i <- 1
while (TRUE) {
  i <- i+4;  if (i > 10) break
}
# while (i <= 10) i <- i+4 # similar loop to above
```

```r
i <- 1
repeat {
  i <- i+4;  if (i > 10) break
}  # repeat has no Boolean exit condition
i
```

```
## [1] 13
```

Looping Over Nonvector Sets

- R does not support iteration over nonvector sets, but there are some indirect ways to do it.
- get(): takes as an argument a character string representing the name of some object and returns the object of that name.

```
u <- matrix(c(1,2,3,1,2,4),nrow=3,ncol=2)
v <- matrix(c(8,12,20,15,10,2),nrow=3,ncol=2)
for (m in c("u","v")){
  z <- get(m)
  print(lm(z[,2] ~ z[,1]))
}
# Here, m was first set to u.
# Then assign the matrix u to z,
# which allows the call to lm() on u.
```

```
##
## Call:
## lm(formula = z[, 2] ~ z[, 1])
##
## Coefficients:
## (Intercept)       z[, 1]
##     -0.6667       1.5000
##
##
## Call:
## lm(formula = z[, 2] ~ z[, 1])
##
## Coefficients:
## (Intercept)       z[, 1]
##      23.286       -1.071
```

if-else

```r
if (r == 4) {
  x <- 1
} else {
  x <- 3; y <- 4
}
```

- Although the `if` section consists of just a single statement, the braces are needed.
- The right brace before `else` is used by the R parser to deduce that this is an `if` – `else` rather than just an `if`.

- An `if` – `else` statement works as a function call and it returns the last value assigned.

```
x <- 2
y <- if(x == 2) x else x+1
y
```

```
## [1] 2
```

```
x <- 3
if(x == 2) y <- x else y <- x+1
y
```

```
## [1] 4
```

Arithmetic and Boolean Operators and Value

Table 7-1: Basic R Operators

Operation	Description
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x ^ y`	Exponentiation
`x %% y`	Modular arithmetic
`x %/% y`	Integer division
`x == y`	Test for equality
`x <= y`	Test for less than or equal to
`x >= y`	Test for greater than or equal to
`x && y`	Boolean AND for scalars
`x || y`	Boolean OR for scalars
`x & y`	Boolean AND for vectors (vector x,y,result)
`x | y`	Boolean OR for vectors (vector x,y,result)
`!x`	Boolean negation

```r
x <- as.logical(c("TRUE","FALSE","TRUE"))
y <- as.logical(c("TRUE","TRUE","FALSE"))
x & y
```

```
## [1]  TRUE FALSE FALSE
```

```r
x && y # looks at just the first elements of each vector
```

```
## [1] TRUE
```

```r
if (x[1] && y[1]) print("both TRUE")
```

```
## [1] "both TRUE"
```

```r
# if (x & y) print("both TRUE")
```

Note: In evaluation an `if`, we need a single Boolean, not a vector of Boolean.

The Boolean values `TRUE` and `FALSE` can be abbreviated as `T` and `F`. These values change to 1 and 0 in arithmetic expression:

```
1 < 2
```

```
## [1] TRUE
```

```
(1 < 2) * (3 < 4) * (5 < 1)
```

```
## [1] 0
```

```
(1 < 2) == TRUE
```

```
## [1] TRUE
```

Return Values

The return value of a function can be any R object, e.g., a list and another function. Without calling `return()`, the value of the last executed statement will be returned by default.

```
oddcount <- function(x) {
  k <- 0
  for (n in x) {
    if (n %% 2 == 1) k <- k+1
  }
  k # or return(k).
}
oddcount(1:11)
```

```
## [1] 6
```

```r
oddcount <- function(x) {
  k <- 0
  for (n in x) {
    if (n %% 2 == 1) k <- k+1
  }
  # k
}
oddcount(1:11)
```

Note: The above function wouldn't work. The last executed statement is the call to `for`(), which returns the value NULL.

`k` or `return`(`k`)? Calling `return`() in the function lengthens execution time, however, unless the function is very short, the time saved is negligible.

Returning Complex Objects

- You can return complex objects.

```
g <- function(){
  t <- function(x) return(x^2)
  return(t)
}
g()
```

```
## function(x) return(x^2)
## <environment: 0x0000000012f76088>
```

Note: If your function has multiple return values, place them in a list or other container.

Functions Are Objects

R functions are *first-class objects*, meaning that they can be used for the most part just like other objects.

```
g <- function(x) return(x+1)
formals(g)
```

```
## $x
```

```
body(g)
```

```
## return(x + 1)
```

Note: Some of R's most fundamental built-in functions are written directly in C, and thus they are not viewable.

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

- You can also assign them, use them as arguments to other functions, and so on.

```
f1 <- function(a, b) return(a+b)
f2 <- function(a, b) return(a-b)
f <- f1
f(3,2)
```

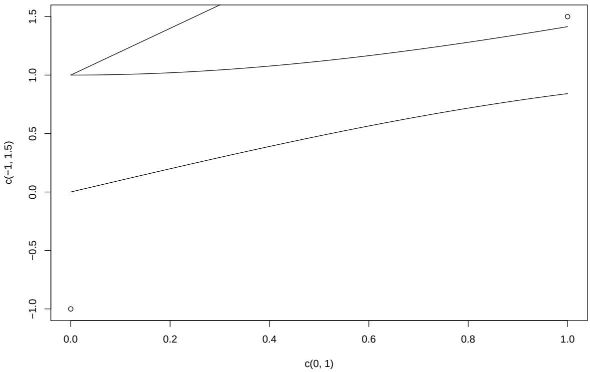
```
## [1] 5
```

```
g <- function(h,a,b) h(a,b)
g(f2,3,2)
```

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

- You can loop through a list consisting of several functions.

```r
g1 <- function(x) return(sin(x))
g2 <- function(x) return(sqrt(x^2+1))
g3 <- function(x) return(2*x+1)
plot(c(0,1),c(-1,1.5)) # prepare the graph, specifying X and Y ranges
for (f in c(g1,g2,g3)) plot(f,0,1,add=T) # add plot to existing graph
```



- You can replace functions using `formals()` and `body()`.

```r
g <- function(h,a,b) h(a,b)
body(g) <- quote(2*x + 3)
g
```

```
## function (h, a, b)
## 2 * x + 3
```

```r
g(3)
```

```
## [1] 5 3 5
```

Environment and Scope Issues

A function consists not only of its arguments and body but of its *environment*.

The latter is made up of the collection of objects present at the time the function is created.

An understanding of how environments work in R is essential for writing effective R functions.

The Top-Level Environment and The Scope Hierarchy

```
rm(list=ls())
w <- 12
f <- function(y) {
  d <- 8
  h <- function() {
    return(d*(w+y))
  }
  return(h())
}
environment(f)
```

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

Note: The function `f()` is created at the *top level*, i.e., at the interpreter command prompt, and thus has the top-level environment, which in R output is referred to as `R_Global` but which confusingly you refer to in R code as `.GlobalEnv`.

Note: The hierarchical nature of scope in `f()` implies that since `w` is global to `f()`, it is global to `h()` as well. Indeed, we do use `w` within `h()`.

Try the following code:

```
> f(2)
[1] 112
> d
Error: object 'd' not found
> h
Error: object 'h' not found
```

`h()` is local to `f()` and invisible at the top level.

ls()

- ls() lists the objects of an environment.
- ls.str() provides a bit more information.

```r
ls()
```

```
## [1] "f" "w"
```

```r
ls.str()
```

```
## f : function (y)
## w :  num 12
```

- With the `envir` argument, it prints the names of the locals of any frame in the call chain.

```
f <- function(y){ d <- 8; return(h(d,y)) }
h <- function(dee,yyy){
  print(ls())
  print(ls(envir=parent.frame(n=1)))
  return(dee*(w+yyy))
} # the argument n specifies how many frames
  # to go up in the call chain.
f(2)
```

```
## [1] "dee" "yyy"
## [1] "d" "y"

## [1] 112
```

Functions Have (Almost) No Side Effects

- Functions do not change nonlocal variables, i.e., there are no side effects.
- The code in a function has read access to its nonlocal variables, but it does not have write access to them.

```
w <- 12
f <- function(y) {
  d <- 8; w <- w + 1; y <- y - 2
  print(w)
  h <- function() return(d*(w+y))
  return(h())
}
```

```r
t <- 4
f(t) # w -> w+1 & d*(w+y) -> d*((w+1)+(y-2))
```

```
## [1] 13
```

```
## [1] 120
```

```r
c(w, t)
```

```
## [1] 12  4
```

Note: `w` at the top level did not change, even though it appeared to change within `f()`. Only a local *copy* of `w`, within `f()`, changed.

Recursion

A *recursive* function calls itself. A classic example is Quicksort, an algorithm used to sort a vector of numbers from smallest to largest.

```
qs <- function(x) { # sort() (written in C) is much faster
  if (length(x) <= 1) return(x) # termination condition
  pivot <- x[1]; therest <- x[-1]
  sv1 <- therest[therest < pivot]
  sv2 <- therest[therest >= pivot]
  sv1 <- qs(sv1); sv2 <- qs(sv2)
  return(c(sv1,pivot,sv2))
}
qs(c(5,4,12,13,3,8,88))
```

```
## [1]  3  4  5  8 12 13 88
```

Replacement Functions

```r
x <- c(1,2,4)
names(x) <- c("a","b","ab")
x
```

```
##  a  b ab
##  1  2  4
```

```r
y <- "names<-"(x, value=c("a","b","ab"))
y
```

```
##  a  b ab
##  1  2  4
```

Note: The call here is indeed to a function named `names<-()`.

What's Considered a Replacement Function?

- Any assignment statement in which the left side is not just an identifier (meaning a variable name) is considered a replacement function.

When encounting this:

```
g(u) <- v
```

R will try to execute this:

```
u <- "g<-"(u, value=v)
```

```r
x <- c(8, 88, 5, 12, 13)
x[3]
```

```
## [1] 5
```

```r
"["(x,3)
```

```
## [1] 5
```

```r
y <- "[<-"(x,2:3,value=99:100)
y
```

```
## [1]   8  99 100  12  13
```

Tools for Composing Function Code

Text Editors and Integrated Development Environments

- You can use a text editor to write your code in a file and then read it into R from the file.
- `source` function can be used in R.

```
source("your_code.R")
```

Note: If you don't have much code, you can cut and paste from your editor window to your R window.

The `edit()` Function

- For a small, quick change, the `edit()` function can be handy.
- It opens the default editor on the code for `f` below, which we could then edit and assign back to `f`.

```
f <- edit(f)
```

Writing Your Own Binary Operations

- You can invent your own operations.
- Write a function whose name begins and ends with %, with two arguments of a certain type, and a return value of that type.

```
"%a2b%" <- function(a,b) return(a+2*b)
3 %a2b% 5
```

```
## [1] 13
```

Anonymous Functions

```
inc <- function(x) return(x+1)
```

- The functions without the last step (the assignment) are called *anonymous*, since they have no name.
- Anonymous functions can be convenient if they are short one-liners and are called by another function.

```r
z <- matrix(1:6, nrow=3,ncol=2)
f <- function(x) x/c(2,8)
y <- apply(z,1,f)
y
```

```
##      [,1]  [,2] [,3]
## [1,]  0.5 1.000 1.50
## [2,]  0.5 0.625 0.75
```

```r
y <- apply(z,1,function(x) x/c(2,8))
y
```

```
##      [,1]  [,2] [,3]
## [1,]  0.5 1.000 1.50
## [2,]  0.5 0.625 0.75
```

Reference

- Matloff, N. The Art of R Programming: A Tour of Statistical Software Design. No Starch Press. Chapter 7.