MTH 2520 R Notes 9

8 R Programming Structures (Cont'd)

8.3 Looping

8.3.1 Introduction

- ***Loops*** are used to ***iterate*** (repeat) the execution of one or more R statements. They're implemented in any of three ways:

```
for()       # Repeat a set of statements a specified number of times
while()     # Repeat a set of statements as long as a specified con-
            # dition is met
repeat      # Repeat a set of statements until a break command is
            # encountered
```

- Two other commands, `break` and `next`, are used, respectively, to terminate a loop's iterations and to skip ahead to the next iteration:

```
break       # Terminate a loop's iterations
next        # Skip ahead to the next iteration
```

8.3.2 `for()` Loops

- `for()` loops are used when we know in advance how many iterations the loop should perform.
- For example, the following commands print the numbers $1^2, 2^2, ..., 5^2$ to the R console (output not shown):

```
> print(1^2)
> print(2^2)
> print(3^2)
> print(4^2)
> print(5^2)
```

We can achieve the same task much more succinctly using a `for()` loop by typing:

```
> for(i in 1:5) {          # i takes the values 1, 2, 3, 4, 5 in succession.
+   print(i^2)              # The print() statement is executed 5 times,
+ }                         # once for each value of i.

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

- The general form of a `for()` loop is:

```
> for(var in seq) {
    statement1
    statement2
        .
        .
        .
    statementq
  }
```

  where `seq` is a vector (or list), `var` (whose name you're free to change) takes values `seq[1]`, `seq[2]`, ..., `seq[length(seq)]` sequentially, each time triggering another iteration of the loop, during which `statements 1` through `q` are executed.

  The `statements` usually involve the variable `var`.

- For single-statement loops, we can skip the curly brackets `{ }` as long as we put the entire `for()` loop on one line, like this:

```
> for(var in seq) statement1
```

8.3.3 `while()` Loops

- `while()` loops are used to iterate a set of statements when we don't know in advance how many iterations the loop should perform, but we want it to continue performing them as long as some condition is met, and to terminate once it's no longer met.
- Here, a `while()` loop is used to print $1^2, 2^2, ..., 5^2$ to the console:

```
> i <- 1
> while(i <= 5) {              # As long as i <= 5, print
+   print(i^2)                 # the value of i^2.
+   i <- i + 1                 # Increment the value of i.
+ }

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Above, *prior* to each iteration, R checks whether `i <= 5` is `TRUE`, and only proceeds if it is.

- The general form of a `while()` loop is:

```
> while(cond) {
    statement1
    statement2
        .
        .
        .
    statementq
  }
```

where `cond` is a `"logical"` expression (i.e. it evaluates to `TRUE` or `FALSE`) and involves a variable *whose value changes during the iterations.*

Prior to each iteration, R checks whether `cond` is `TRUE`. If it is, R proceeds with the iteration, otherwise the process is terminated.

8.3.4 `repeat` Loops

- A `repeat` loop iterates a set of statements until a `break` statement is encountered.
- Below, a `repeat` loop is used to print $1^2, 2^2, ..., 5^2$ to the screen:

```
> i <- 1                    # Initialize the value of i.
> repeat {
+   print(i^2)
+   i <- i + 1              # Increment the value of i.
+   if (i > 5) break        # Check whether to terminate the iterations.
+ }

[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

- The general form is of a `repeat` loop is:

```
> repeat {
    statement1
    statement2
          .
          .
          .
    statementq
  }
```

At least one of the `statements` should be of the form:

```
if (cond) break
```

where `cond` is a `"logical"` (`TRUE` or `FALSE`) expression which *may be updated during the loop's iterations.*

8.3.5 Advancing to the Next Iteration Using `next`

- A `next` statement is used to advance to the next iteration. For example, to skip 3^2 when printing the numbers $1^2, 2^2, 3^2, 4^2, 5^2$ to the screen, we could type:

```
> for(i in 1:5) {
+   if (i == 3) next        # Advances to the start of the next iteration, with-
+   print(i^2)              # out executing the print(i^2) statement, if i == 3
+ }
```

```
[1] 1
[1] 4
[1] 16
[1] 25
```

Above, R checks whether `i == 3` is `TRUE` at the start of each iteration, and if so, skips ahead to the next iteration (without executing the `print()` statement first).

8.3.6 Terminating an "Endless" Loop

- Once in a while we (mistakenly) write a loop that has no way of stopping, for example:

```
> i <- 1
> while(i <= 5) {
+   print("I can't stop because i is always 1")
+ }
```

To terminate the iterations, hit the **Escape** key or select **Stop Current Computation** in R's **Misc** pulldown menu.

8.3.7 Nested Loops

- Loops can be ***nested***, one inside the other.

The inner loop' cycles through its entire set of iterations *once for each iteration of the outer loop*:

```
> for(i in 1:4) {
+     for(j in 1:3) {
+         print(c(i, j))
+     }
+ }
```

```
[1] 1 1
[1] 1 2
[1] 1 3
[1] 2 1
[1] 2 2
[1] 2 3
[1] 3 1
[1] 3 2
[1] 3 3
[1] 4 1
[1] 4 2
[1] 4 3
```

- Nested loops are useful for looping over rows and columns of a matrix (or dimensions of an array). For example, suppose we want to create this matrix `x`:

```
    j=1 j=2 j=3 j=4 j=5 j=6
i=1   1   2   3   4   5   6
i=2   1   2   3   4   5   6
i=3   1   2   3   4   5   6
i=4   1   2   3   4   5   6
i=5   1   2   3   4   5   6
i=6   1   2   3   4   5   6
```

We can create `x`, using nested `for()` loops, by starting with a matrix of 0's:

```
> x <- matrix(0, nrow = 6, ncol = 6)       # Start with a matrix of all 0's
> rownames(x) <- c("i=1", "i=2", "i=3", "i=4", "i=5", "i=6")
> colnames(x) <- c("j=1", "j=2", "j=3", "j=4", "j=5", "j=6")
> x
```

```
    j=1 j=2 j=3 j=4 j=5 j=6
i=1   0   0   0   0   0   0
i=2   0   0   0   0   0   0
i=3   0   0   0   0   0   0
i=4   0   0   0   0   0   0
i=5   0   0   0   0   0   0
i=6   0   0   0   0   0   0
```

Then we overwrite the 0's by desired values using nested loops. The outer loop iterates over rows, and for each row, the inner loop over columns within that row:

```
> for(i in 1:6) {                  # i is the row number, ranging from 1 to 6.
+     for(j in 1:6) {              # j is the column number, ranging from 1 to 6.
+         x[i, j] <- j             # Insert column numbers into the ith row of x.
+     }
+ }
```

The result is:

```
> x

    j=1 j=2 j=3 j=4 j=5 j=6
i=1   1   2   3   4   5   6
i=2   1   2   3   4   5   6
i=3   1   2   3   4   5   6
i=4   1   2   3   4   5   6
i=5   1   2   3   4   5   6
i=6   1   2   3   4   5   6
```

- Now suppose we want to create this matrix `x`:

```
    j=1 j=2 j=3 j=4 j=5 j=6
i=1   1   0   0   0   0   0
i=2   1   2   0   0   0   0
i=3   1   2   3   0   0   0
i=4   1   2   3   4   0   0
i=5   1   2   3   4   5   0
i=6   1   2   3   4   5   6
```

To create `x` using nested `for()` loops, we start with a matrix of 0's again:

```
> x

    j=1 j=2 j=3 j=4 j=5 j=6
i=1   0   0   0   0   0   0
i=2   0   0   0   0   0   0
i=3   0   0   0   0   0   0
i=4   0   0   0   0   0   0
i=5   0   0   0   0   0   0
i=6   0   0   0   0   0   0
```

But now, we stop the inner loop (`j`) before reaching the 6th column, and the stopping point for the inner loop is the current value of `i` (the row number):

```
> for(i in 1:6) {               # i is the row number, ranging from 1 to 6.
+     for(j in 1:i) {           # j is the column number, ranging from 1 to i.
+         x[i, j] <- j          # Insert column numbers into the ith row of x.
+     }
+ }
```

The result is:

```
> x

    j=1 j=2 j=3 j=4 j=5 j=6
i=1   1   0   0   0   0   0
i=2   1   2   0   0   0   0
i=3   1   2   3   0   0   0
i=4   1   2   3   4   0   0
i=5   1   2   3   4   5   0
i=6   1   2   3   4   5   6
```

8.3.8 Looping Over List Elements

- We can loop over the elements of a *list*. For example, here's a list:

```
> my.list <- list(
+     w = c(4, 4, 5, 5, 6, 6),
+     x = c("a", "b", "c"),
+     y = c(5, 10, 15),
+     z = c("r", "s", "t", "u", "v")
+ )
```

To loop over the elements of `my.list`, printing the list element at each iteration, we type:

```
> for(i in my.list) {
+   print(i)                          # i is a vector-valued list element.
+ }
```

```
[1] 4 4 5 5 6 6
[1] "a" "b" "c"
[1]  5 10 15
[1] "r" "s" "t" "u" "v"
```

The above command is equivalent to:

```
> for(i in 1:4) {
+   print(my.list[[i]])
+ }
```

```
[1] 4 4 5 5 6 6
[1] "a" "b" "c"
[1]  5 10 15
[1] "r" "s" "t" "u" "v"
```

Section 8.3 Exercises

Exercise 1 Guess how many times `"Good Sport"` will be printed to the screen in each of the following sets of commands. Then check your answers.

a)
```
> for(i in 1:10) {
    print("Good Sport")
  }
```

b)
```
> x <- c(13, 11, 17, 9)
> for(i in x) {
    print("Good Sport")
  }
```

Exercise 2 *Prior* to each iteration of a `while()` loop, R checks whether the `condition` is met, and only proceeds if it is met. Guess how many times `"Frisbee Sailing"` will be printed to the screen in each of the following sets of commands. Then check your answers.

a)
```
> i <- 3
> while(i < 1) {
    print("Frisbee Sailing")
    i <- i + 1
  }
```

b)
```
> i <- 0
> while(i < 3) {
    print("Frisbee Sailing")
  }
```

c)
```
> i <- 0
> while(i < 3) {
    print("Frisbee Sailing")
    i <- i + 1
  }

> i <- 1
> while(i < 3) {
    print("Frisbee Sailing")
    i <- i + 1
  }
```

Exercise 3 Guess how many times `"Masked Marvel"` will be printed to the screen in each of the following sets of commands. Then check your answers.

a)
```
> i <- 1
> repeat {
    if (i > 3) break
    print("Masked Marvel")
    i <- i + 1
  }
```

b)
```
> i <- 0
> repeat {
    i <- i + 1
    if (i > 3) break
    if (i == 2) next
    print("Masked Marvel")
  }
```

Exercise 4 How does the positioning of the statement `i <- i + 1` affect a `while()` loop? Compare the following two loops. Will they perform the same number of iterations? Will their outputs be the same? Make sure to check your answers.

```
> i <- 1
> while(i <= 3) {
    print(i^2)
    i <- i + 1
  }

> i <- 1
> while(i <= 3) {
   i <- i + 1
   print(i^2)
  }
```

Exercise 5 How does the positioning of the `break` statement affect a `repeat` loop? Compare the following three loops. Will they perform the same number of iterations? Will their outputs be the same? Make sure to check your answers.

```
> i <- 1
> repeat {
    print(i^2)
    i <- i + 1
    if (i > 3) break
  }

> i <- 1
> repeat {
    if (i > 3) break
    print(i^2)
    i <- i + 1
  }

> i <- 1
> repeat {
    print(i^2)
    if (i > 3) break
    i <- i + 1
  }
```

Exercise 6 The sum of squares

$$\sum_{i=1}^{10} i^2 \quad = \quad 1^2 + 2^2 + \cdots + 10^2$$

can be computed using a `for()` loop by typing:

```
> sum.sq <- 0
> for(i in 1:10) {
    sum.sq <- sum.sq + i^2
  }
```

Why is it necessary to make the assignment `sum.sq <- 0` *before* entering the loop? What would happen if `sum.sq <- 0` wasn't there? Try it (but type `rm(sum.sq)` first to remove it from your Workspace if it's currently there).

Exercise 7 To create a vector `num.sq` containing the values $1^2, 2^2, \ldots, 10^2$, we can use the following `for()` loop:

```
> num.sq <- NULL
> for(i in 1:10) {
    num.sq <- c(num.sq, i^2)
  }
```

Why is it necessary to make the assignment `num.sq <- NULL` *before* entering the loop? What would happen if `num.sq <- NULL` wasn't there?

Exercise 8 Write commands that use *nested* `for()` loops to create the following matrix `x`:

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    0    2    3    4    5    6
[3,]    0    0    3    4    5    6
[4,]    0    0    0    4    5    6
[5,]    0    0    0    0    5    6
[6,]    0    0    0    0    0    6
```

8.4 User-Defined Functions

- We can create a ***user-defined function*** using:

```
function()          # Used to create user-defined functions.
return()            # Used within a function definition to terminate
                    # the function call and return a value.
```

- Here's a simple (mathematical) function $f(x)$:

$$f(x) = x^2$$

In R, we can represent this as user-defined function that takes an argument `x` and returns its square:

```
> my.f <- function(x) {
+   return(x^2)
+ }
```

We use user-defined functions just as we would built-in functions:

```
> my.f(x = 2)

[1] 4
```

We can look at the function definition by typing its name on the command line:

```
> my.f

function(x) {
  return(x^2)
}
```

- The general format for a *user-defined function* is:

```
> my.fun <- function(arg1, arg2, ..., argk) {
     statement1
     statement2
          .
          .
          .
     statementq
     return(value)              # The value to be returned.  We could
   }                            # also just write value (without return()).
```

Above,

  - `arg1`, `arg2`, ..., `argk` are argument names (that we're free to choose) for k (formal) arguments.
  - `statement1`, `statement2`, ..., `statementq` are a set of q statements (which may involve `arg1`, `arg2`, ..., `argk`).
  - `value` is a value (or expression that gives a value) to be returned by the function.

- If the body of the function consists of just a `value` (or expression), we can omit the curly brackets `{ }` as long as we write the entire function definition on a single line:

```
> my.fun <- function(arg1, arg2, ..., argk) value
```

- As another example, here's a (mathematical) function that computes the average absolute value of two numbers x and y:

$$f(x, y) = \frac{|x| + |y|}{2},$$

We can write this function in R as:

```
> AvgAbsVal <- function(x, y) {
+   avg <- (abs(x) + abs(y)) / 2
+   return(avg)
+ }
```

We can now call `AvgAbsVal()`, passing it values via the arguments `x` and `y`:

```
> AvgAbsVal(x = -4, y = 2)

[1] 3
```

- In fact, we don't need to use `return()`. In the absence of a `return()` statement, R returns the value of any expression that appears by itself as the last line of the function definition:

```
> AvgAbsVal <- function(x, y) {
+   avg <- (abs(x) + abs(y)) / 2
+   avg
+ }
```

8.4.1 Formal Arguments and Actual Arguments

- A function's arguments are sometimes referred to as ***formal*** arguments. Values passed to the function are referred to as ***actual*** arguments.
- For example, below, `x` is a *formal* argument, and `z` is an *actual* argument:

```
> g <- function(x) {
+   return(x + 1)
+ }

> z <- 5
> g(x = z)              # x is a formal argument, z is an actual argument

[1] 6
```

8.4.2 Specifying Default Values for a Function's Arguments

- We can specify default values for one or more of a function's arguments by specifying `arg = expr` in the function definition:

```
> my.fun <- function(arg1, arg2 = expr2, ..., argk = exprk) {
    statement1
    statement2
        .
        .
        .
    statementq
    return(value)
  }
```

- For example, below we define `AvgAbsVal()` so that the default value for `y` is `NULL`.

```
> AvgAbsVal <- function(x, y = NULL) {
+   if (is.null(y)) {
+     return(abs(x))
+   } else {
+     avg <- (abs(x) + abs(y)) / 2
+     return(avg)
+   }
+ }
```

Now, if a value for `y` isn't passed to `AvgAbsVal()`, it uses `NULL`:

```
> AvgAbsVal(x = -120)

[1] 120
```

8.4.3 Variable Number of Arguments Using "..."

- Functions can be written to take a *variable number* of arguments. The argument name `...` in the function definition will match any number of arguments.

  Within the body of the function, we can refer to `...` as if it was the name of a variable.

- For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

  ```
  > mean.of.all <- function(...) {
  +     return(mean(c(...)))
  + }
  ```

  If `us.sales`, `europe.sales`, and `other.sales` were vectors, the command

  ```
  > mean.of.all(us.sales, europe.sales, other.sales)
  ```

  would combine them and take the mean of the combined data. The effect of `c(...)` is as if `c()` were called with the same three arguments, `us.sales`, `europe.sales`, and `other.sales`, that were passed to `mean.of.all()`.

- Many of R's built-in functions take a variable number of arguments. For example look at the help files for `list()` and `c()` by typing:

  ```
  > ? list
  > ? c
  ```

8.4.4 Printing Warning or Error Messages Using `warning()` or `stop()`

- The following functions are useful for terminating a function call or just printing a warning message:

```
stop()          # Terminate a function call and print an error message.
warning()       # Print a warning message (without terminating the
                # function call).
```

- `stop()` and `warning()` are usually used in `if()` statements within functions.
- `stop()` terminates a function call (without returning a value) and prints an error message. Here's an example:

```
> my.ratio <- function(x, y) {
+     if (y == 0) stop("Cannot divide by 0")
+     return(x/y)
+ }
```

An attempt to pass the value `0` for `y` now results in the following:

```
> my.ratio(x = 3, y = 0)

Error in my.ratio(x = 3, y = 0) : Cannot divide by 0
```

(Note that the last line, `return(x/y)`, was never encountered during the call to `my.ratio()`.)

- `warning()` just prints a warning message to the screen without terminating the function call. Here's an example:

```
> my.ratio <- function(x, y) {
+     if (y == 0) warning("Attempt made to divide by 0")
+     return(x/y)
+ }
```

Now when we pass the value `0` for `y`, the function call isn't terminated (`Inf` is returned), but we get a warning message:

```
> my.ratio(x = 3, y = 0)

[1] Inf
Warning message:
In my.ratio(x = 3, y = 0) : Attempt made to divide by 0
```

Section 8.4 Exercises

Exercise 9 Here's a function:

```
> f1 <- function(x) {
    y <- x + 1
    return(y)
  }
```

If we replace `return(y)` by just `y`:

```
> f2 <- function(x) {
    y <- x + 1
    y
  }
```

does the function do the same thing?

Exercise 10 In the code below, which argument (`x` or `z`) is the *formal* argument and which is the *actual* argument?

```
> g <- function(x) {
+   return(x^2 - 1)
+ }


> z <- 2
> g(x = z)


[1] 3
```

Exercise 11

a) Write a function that takes two arguments, `x` and `y`, and returns their *relative difference*, defined as

$$f(x, y) = \left| \frac{x - y}{y} \right|,$$

where $|\cdot|$ is the absolute value. Test your functions by passing it a few different values for `x` and `y`.

b) What happens when you pass it the value `y = 0`? What about when you pass it `x = 0` and `y = 0`.

c) Rewrite your function so that it specifies a default value of 1 for `y`.

Exercise 12 Write a function that takes a vector argument `x` and returns a *list* containing the mean, median, standard deviation, and range of `x`. Use `mean()`, `median()`, `sd()`, and `range()`.

Exercise 13 Look at the help file for `list()` by typing

```
> ? list
```

What do the `...`'s mean when it says `list(...)` under Usage?

Exercise 14

a) Write a function that takes two vectors `x` and `y` and returns the maximum value in the two vectors combined. **Hint**: Use `max()` with `c(x, y)`.

b) Modify your function so that it uses `...` to take a *variable number* of vectors as arguments, and returns the maximum value in all the vectors combined.

Exercise 15

a) Write a function that takes an argument `x` and returns `sqrt(x) + 1`.

b) Modify your function so that it first checks whether `x < 0`, and if so, terminates the function call and prints an appropriate error message (using `stop()`).

c) Modify your function again so that, after checking whether `x < 0`, it then goes on to check whether `x == 0`, and if so, prints an appropriate warning message but continues with the function call (using `warning()`).

8.5 Scope

- ***Scope*** refers to the way R looks up values of variables when it needs them.

8.5.1 Local and Global Variables

- When the function:

```
> f1 <- function() {
+   x <- 3
+   y <- 4
+   x + y
+ }
```

is called, R evaluates `x + y` by looking for(and finding) the values of `x` and `y` within the confines of `f1()`, and `f1()` returns 7:

```
> f1()

[1] 7
```

- Below, when `f2()` is called, R looks for the values of `x` and `y` within `f2()`, but only finds `y`. It then looks for `x` in the Workspace, where it finds it, and `f2()` returns 7:

```
> x <- 3                          # x is a global variable

> f2 <- function() {
+   y <- 4                        # y is a local variable
+   x + y
+ }
```

```
> f2()

[1] 7
```

- A variable created within the body of a function, like `y` above, is called a ***local*** variable for that function. A function's formal arguments, if any, to which values have been passed are also considered *local* variables.

  A variable created on the command line and stored in the Workspace, like `x` above, is called a ***global*** variable to the function.

- When R needs to look up the value of a variable while executing a function call, it looks first among the *local* variables. If if doesn't find it there, it then looks among the *global* variables.

  Thus, if a global variable and a local variable share the same name, the local variable is used:

```
> x <- 1                    # This x is a global variable

> f1 <- function() {
+   x <- 3                  # This x is a local variable, and is the one
+   y <- 4                  # used within f1()
+   x + y
+ }

> f1()

[1] 7
```

- Global variables are visible from within a function, but local variables aren't visible from outside the function. In fact, local variables are *temporary*, and disappear when the function call is completed:

```
> y                           # y doesn't exist after the call to f1()
                              # is completed

Error: object 'y' not found
```

- Assigning a value to a local variable *won't* overwrite a pre-existing global of the same name:

```
> x <- 1                    # This x is a global variable

> f1 <- function() {
+   x <- 3                  # This x is a local variable.  Assigning it
+   y <- 4                  # a value here doesn't affect the global x.
+   x + y
+ }
```

```
> f1()

[1] 7

> x                              # The global x didn't change

[1] 1
```

Section 8.5 Exercises

Exercise 16 When R needs the value of a variable, it looks first among the local variables. If it doesn't find it there, it then looks among the global variables. Guess what value will be returned by `my.fun()`, then check your answer.

```
> x <- 5


> my.fun <- function() {
+   x <- 3
+   return(x)
+ }


> my.fun()
```

Exercise 17 When R needs the value of a variable, it looks first among the local variables. If it doesn't find it there, it then looks among the global variables. Guess what value will be returned by `my.fun()`, then check your answer.

```
> x <- 5


> my.fun <- function() {
    return(x)
  }


> my.fun()
```

Exercise 18 A function's formal arguments are local variables. Guess what value will be returned by `my.fun()`, then check your answer.

```
> x <- 5


> my.fun <- function(x) {
+   return(x)
+ }
```

```
> my.fun(x = 3)
```

Exercise 19 How does changing the value variable in a function affect a global variable that has the same name? Guess what value of `x` will be printed after the last command below. Then check your answer.

```
> x <- 5


> my.fun <- function() {
    x <- 3
    return(x)
  }


my.fun()


> x
```

Exercise 20 What happens to local variables after the function call is completed? Guess what value of `x` will be printed after the last command below. Then check your answer.

```
> rm(x)


> my.fun <- function() {
+   x <- 3
+   return(x)
+ }


> my.fun()


[1] 3


> x
```

8.6 Environments

- Each function has an associated ***environment***, which can be thought of as a "container" that holds all of the objects where the function is created. A function's environment contains all the variables that are *global* to that function *plus* the function itself.
- We can look at the environment of a function using `environment()`, and we can list the objects in its environment using `ls()`:

```
environment()                # Returns the environment of a function.
ls()                         # Lists the objects in the current
                             # environment.
```

8.6.1 The Top-Level (or Global) Environment

- When a function is created on the *command line*, it's environment is the so-called ***global environment***, which is another name for the Workspace:

```
> w <- 2                                    # w is global to f()

> f <- function(y) {
+   d <- 3                                  # d and y are local to f()
+   return(d * (w + y))
+ }

> environment(f)

<environment: R_GlobalEnv>
```

  (Note that R refers to the global environment, or Workspace, as `R_GlobalEnv`.)

- We can list the objects in `f()`'s environment using `ls()`:

```
> ls()

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

  Note that `f()`'s environment contains all the variables in the Workspace (just `w` at this point) *plus* `f()` itself.

8.6.2 Nested Functions and the Scope Hierarchy

- When a function is created *inside another function*, it's said to be ***nested*** in that function.
- If a function is nested inside another one, its *global* variables are the local variables of the outer function *plus* the outer function's global variables. These *two* sets of variables comprise the inner function's *environment* (which also contains the inner function itself).
- For example:

```
> w <- 2                          # w is global to f() and therefore also to h()
```

```
> f <- function(y) {
+   d <- 3                    # y and d are local to f() but global to h()
+   h <- function() {
+     b <- 5                  # b is local to h()
+     return(d * (w + y))
+   }
+   return(h())
+ }
```

Above,

- `w` is global to `f()` and therefore also to `h()`.
- `y` and `d` are local to `f()`, but global to `h()`.
- `b` is local to `h()`.

In terms of *environments*, each function's environment consists of its global variables plus itself. Therefore:

- `f()`s environment contains `w` and `f()` itself.
- `h()`s environment contains `y`, `d`, `w`, and `h()` itself (i.e. it contains `f()`s environment *plus* all the variables that are local to `f()`).

- When R needs the value of a variable for a computation inside a function, it first looks for it among the local variables. If it doesn't find it there, it proceeds up the "scope hierarchy", first looking among the local variables of the outer function, then among the variables in the *global environment* (Workspace).
- This ***scope hierarchy*** (nesting of environments) continues when there are more levels of *nesting*, e.g. one function that was created in another that was created in another.
- We can use a `print(ls())` statement to see which objects are local to `f()`:

```
> w <- 2                    # w is global to f() and therefore also to h()

> f <- function(y) {
+   d <- 3                    # y and d are local to f() but global to h()
+   h <- function() {
+     b <- 5                  # b is local to h()
+     return(d * (w + y))
+   }
+   print(ls())
+   return(h())
+ }

> f(2)

[1] "d" "h" "y"
[1] 12
```

- Likewise we can use a `print(environment())` statement to view the environment of `h()`:

```
> w <- 2                        # w is global to f() and therefore also to h()

> f <- function(y) {
+   d <- 3                      # y and d are local to f() but global to h()
+   h <- function() {
+     b <- 5                    # b is local to h()
+     return(d * (w + y))
+   }
+   print(environment(h))
+   return(h())
+ }

> f(2)

<environment: 0x049ed674>
[1] 12
```

In the output above, the environment of `h()` is referred to by its memory location.

8.6.3 Writing "Upstairs" in the Scope Hierarchy

- To assign a value to a variable in the global environment (Workspace) from within a function, or more generally to assign a value higher up the scope hierarchy from within a function, we use:

```
<<-           # Assign a value to a variable in the global environ-
              # ment (Workspace).
assign()      # Assign a value to a variable in an environment higher
              # up the scope hierarchy.
```

- Here's an example using the so-called ***superassignment operator*** `<<-`:

```
> d <- 4                        # This value of d will be overwritten from
                                # within f().

> f <- function(y) {
+   d <<- 3                     # This overwrites the global d
+   return(d * y)
+ }

> f(2)

[1] 6
```

```
> d

[1] 3
```

Above, the command `d <<- 3` assigns `3` to `d` in the global environment, or Workspace, and overwrites the previous value of `d`.

- Here's how to accomplish the same thing using `assign()`:

```
> d <- 4                          # This value of d will be overwritten from
                                  # within f().

> f <- function(y) {
+     assign("d", 3, envir = .GlobalEnv)     # This overwrites the global d
+     return(d * y)
+ }

> f(2)

[1] 6

> d

[1] 3
```

Note that when `assign()` is used:

  - `d` is written as a character string (i.e. in quotes as `"d"`)
  - The global environment is written as `.GlobalEnv`
  - We could have used `pos = -1` instead of `envir = .GlobalEnv` to indicate that the assignment should take place one level higher in the scope hierarchy.

- Be aware that assignment to the variable `d` using `<<-` actually results in a search up the environment hierarchy, stopping at the first level at which the name `d` is encountered. If it's not encountered, then assignment is done in the global environment. For example:

```
> d <- 4                          # This value of d won't be overwritten

> f <- function(y) {
+     d <- 5                      # This value of d will be overwritten from
+                                 # within h()
+     h <- function() {
+         d <<- 3                 # This overwrites the d that's local to f()
+         return(d * y)
+     }
+     return(h())
+ }
```

```
> f(2)

[1] 6

> d                                  # This global d didn't change

[1] 4
```

8.6.4 When Should You Use Global Variables?

- Here are suggestions about using global variables. They're especially important when your code will be shared with other R users:
  - Assignment to the global environment using `<<-` or `assign()` is **not recommended**, and should be used only when necessary, because it can accidentally overwrite existing variables.
  - The use of a global variable can be justified when that variable needs to be accessed by several different functions (that aren't nested).
  - It's generally preferable to pass variables as arguments to functions rather than accessing them from the global environment.

Section 8.6 Exercises

Exercise 21 For each of the following sets of commands, guess what value will be returned by the call to `f()` on the last line. Then check your answer.

a)
```
> w <- 5

> f <- function(y) {
    return(w + y)
  }

> f(2)
```

b)
```
> w <- 5

> f <- function(y) {
    w <- 4
    return(w + y)
  }

> f(2)
```

Exercise 22 Among the variables `w`, `d`, and `y`, which are *global* to `h()` and which are *local* to `h()`?

```
> w <- 2
```

```
> f <- function(y) {
    h <- function() {
      d <- 3
      return(w + y + d)
    }
    return(h())
  }
```

Exercise 23 The call to `f()` on the last line below will give an error message. Try to figure out why, then check your answer by running the code?

```
> rm(d)

> f <- function() {
    y <- 2
    h <- function() {
      d <- 3
      return(y)
    }
    return(d * h())
  }

> f()
```

Exercise 24 When R needs the value of a variable for a computation inside a function, it first looks for it among the local variables. If it doesn't find it there, it proceeds up the "scope hierarchy", first looking among the local variables of the outer function, then among the variables in the *global environment* (Workspace).

a) Guess what value will be returned by the call to `f()` in the last line below, then check your answer:

```
> x <- 1

> f <- function() {
    x <- 2
    h <- function() {
      x <- 3
      return(x)
    }
    return(h())
  }

> f()
```

b) Guess what value will be returned by the call to `f()` in the last line below, then check your answer:

```
> x <- 1

> f <- function() {
    x <- 2
    h <- function() {
      return(x)
    }
    return(h())
  }

> f()
```

c) Guess what value will be returned by the call to `f()` in the last line below, then check your answer:

```
> x <- 1

> f <- function(x) {
    h <- function() {
      return(x)
    }
    return(h())
  }

> f(2)
```

d) Guess what value will be returned by the call to `f()` in the last line below, then check your answer:

```
> x <- 1

> f <- function() {
    h <- function() {
      return(x)
    }
    return(h())
  }

> f()
```

Exercise 25

a) Variables in the *global environment* (Workspace) are visible from within the function `my.fun1()` below, but `a` and `b`, which are local to `my.fun2()`, aren't visible from outside `my.fun2()`. The call to `my.fun2()` on the last line below produces an error message. See if you can figure out what the error message will be, then check your answer by running the code:

```
> rm(a, b)

> my.fun1 <- function() {
    return(a + b)
  }

> my.fun2 <- function() {
    a <- 2
    b <- 3
    return(my.fun1())
  }

> my.fun2()
```

b) Now see if you can guess what value will be returned by the call to `my.fun2()` on the last line below. Then check your answer:

```
> a <- 0
> b <- 1

> my.fun1 <- function() {
    return(a + b)
  }

> my.fun2 <- function() {
    a <- 2
    b <- 3
    return(my.fun1())
  }

> my.fun2()
```

Exercise 26 The *superassignment* operator `<<-` assigns values to variables in the *global environment* (Workspace), even if it's used within a function. What value for `w` will be printed in the last line below:

```
> w <- 1

> f <- function() {
    w <<- 3
    return(NULL)
  }


> f()

> w
```

8.7 Vectorized User-Defined Functions and `sapply()`

- A user-defined function will be *vectorized* as long as the built-in operators and functions it uses are vectorized.
- For example, below, since both `sqrt()` and `abs()` are vectorized, `sqrt.abs()` will be vectorized too:

```
> sqrt.abs <- function(x) {                # x will be a vector
+     z <- sqrt(abs(x))
+     z
+ }

> sqrt.abs(x = c(-4, 9, -16, 25))

[1] 2 3 4 5
```

- When a user-defined function *isn't* vectorized, we can use `sapply()` (for "simplified apply"):

```
sapply()              # Apply a function to each element of a vector.
```

- For example, suppose we've written a function `my.fun()` that doesn't operate on vectors one element at a time (i.e it's not vectorized), but we want to apply it to each element of a vector `x`. We'd type:

```
> sapply(x, FUN = my.fun)
```

8.8 Replacement Functions

- Some of R's built-in functions can be used both to *return* a value and to *replace* a value. Such functions are called ***replacement functions***.
- For example `names()` is a *replacement function* because it *returns* the names of a data frame and *replaces* them:

```
> x

  var1 var2   var3
1    1   19  small
2    2   20 medium
3    3   16  large

> names(x)                                  # This returns the names of x.

[1] "var1" "var2" "var3"
```

```
> names(x) <- c("ID", "Weight", "Size")     # This replaces the names of x.
> names(x)

[1] "ID"     "Weight" "Size"
```

- It's possible to create user-defined replacement functions. See the textbook.

8.9 Recursion

- ***Recursion*** is a programming technique in which a function calls itself.
- Here's an example in which the function `f()` takes a non-negative integer `x` and returns the factorial of `x`, denoted $x!$ and defined as

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)(x-2)\cdots(2)(1) & \text{if } x > 0 \end{cases}$$

- Notice that we can write

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x(x-1)! & \text{if } x > 0 \end{cases}$$

  a form that lends itself well to *recursion*. Now we can write $x!$ as a function $f(x)$ as:

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ xf(x-1) & \text{if } x > 0 \end{cases}$$

  To implement the factorial function in R using recursion, we type:

```
> f <- function(x) {
+     if (x == 0) {
+         return(1)
+     } else {
+         return(x * f(x - 1))              # f() calls itself here
+     }
+ }

> f(0)

[1] 1

> f(1)

[1] 1

> f(2)

[1] 2
```

```
> f(10)

[1] 3628800
```

- In general, to solve a problem of type X by writing a recursive function `f()`:
  1. Break the original problem of type X into one or more smaller problems of type X.
  2. Within `f()`, call `f()` on each of the smaller problems.
  3. Within `f()`, piece together the results of Step 2 to solve the original problem.

8.10 Infix Functions

- Most functions are "prefix" functions – the name of the function comes *before* the arguments.
- An ***infix*** function is one where the name of the function comes *in between* the arguments.
- For example, `+`, `-`, `*`, `/`, `^`, `%%`, and `%*%` are all infix functions:

```
> 3 + 2

[1] 5
```

- Note that infix functions are *functions*, and can be used like a function when put in quotes:

```
> '+'(3, 2)

[1] 5
```

- You can define your own infix function, but its name has to start and end with `%`, and when you define it, you write it's name in quotes. For example:

```
> '%+%' <- function(a, b) {
+   paste(a, b)                          # paste() combines "character"s
+ }

> "home" %+% "run"

[1] "home run"
```

We could also write:

```
> '%+%'("home", "run")

[1] "home run"
```

8.11 Internal Functions

- Whereas some of Rs built-in functions are written in R, and typing their name prints out their source, others (the ***internal*** ones) are written in C, and typing their name gives either `.Primitive()` or `.Internal()`
- For example, `apply()` is written in R, and typing

  ```
  > apply
  ```

  prints out its source code.

  But `sqrt()` is written in C, and typing its name gives:

  ```
  > sqrt
  ```

  ```
  function (x)  .Primitive("sqrt")
  ```

  `.Primitive()` and `.Internal()` are the R functions that are used to execute the internal C commands.

8.12 Tools for Composing Function Code

8.12.1 Text Editors and the `edit()` Function

- When writing long functions or code sequences, it's not convenient to edit directly on the R command line.
- Instead, it's easier to edit in a *text editor* such as Notepad and then either copy/paste into R or read the code from the text file. Another option is to use R's built-in editor. Here are some useful functions:

  ```
  source()     # Read R commands from a text file, and execute them in R.
  edit()       # Edit a function or data set using R's built-in editor.
  ```

- For example, suppose we have the following commands saved in a text file '**C:\myRcode.txt**':

  ```
  my.fun <- function(message) {
      print(message)
  }

  my.fun("Hello World")
  ```

  We can execute those commands using `source()` by typing:

  ```
  > source('C:/myRcode.txt')
  ```

  ```
  [1] "Hello World"
  ```

- To use `edit()` to edit a function (`my.fun()`, say), type:

```
> my.fun <- edit(my.fun)
```

- We can also edit a data set (`my.data`, say) using `edit()`:

```
> edit(my.data)
```

  This opens up a spreadsheet-like window. Note that if we want our edits to be saved, we need to type:

```
> my.data <- edit(my.data)
```

8.12.2 Integrated Development Environments (IDEs)

- ***Integrated development environments*** are editors designed specifically for writing computer program code. Several have been created specifically for use with R. Among them are **RStudio**, **Tinn-R**, and **Eclipse** (with the **StatET** plug in).