Fourth R meeting

Marco Smolla

November 20, 2013

The `if` statement

The **`if`** statement is a simple way to combine a logical question (TRUE or FALSE as result) with additional commands how to proceed in the one or the other case. In the simplest case the **`if`** statement is followed by a command which is executed when the statement is true. Additionally there can be an **`else`** that tells R what to do if the statement is false. A similar result is gained with the function **`ifelse()`**.

```
x <- 0.5
y <- runif(n = 1, min = 0, max = 1)

if (x > y) print("x is bigger than y")
if (x < y) res <- 3 * x else res <- 2 * y
ifelse(x < y, yes = 3 * x, no = 2 * y)

if (y <= 0.5) {
    res <- 2 * x + y
    print(res)
} else {
    res <- x + 2 * y
    print(res)  # Note: Here you can avoid code replication. How?
}
```

`for` and `while` loops

Use loops when the result of the former calculation is necessary for the next calculation (see example below). If the values are independently calculable it is good to think about using apply functions (see below).
There are two basic loops available, the **`for`** loop that repeats the same calculation 'for a given number of steps', and the **`while`** loop that calculates 'while/until a given expression is met.'

```r
for (i in 1:10) {
    print(i * i)
}
```

```r
t <- 0
z <- 5
while (t < z) {
    # Note: what would happen if we'd use t!=z ?
    print(t)
    t <- t + 2
}
```

More complex `for` loops

In a more complex way we can use the loop indicator (here we use `i`) to address a specific element of our data within the loop (a column, row, or list element).

```r
data <- list(runif(5), runif(10), runif(25))
data2 <- list()
for (i in 1:length(data)) {
    data2[[i]] <- mean(data[[i]])
}

data3 <- data.frame(value1 = runif(25, 1, 100), value2 = runif(25, 1, 100),
    value3 = runif(25, 1, 100))
data4 <- c()
for (i in 1:ncol(data3)) {
    data4 <- c(data4, sum(data3[, i]))
}
```

`apply` and `lapply` function

Apply functions are a fast way to apply a single (or more complex) functions to a data.frame (`apply()`) or a list (`lapply()`). These apply functions are in general faster than loops. This is due to the fact that they are written in C and that every value can be calculated independently. This is especially powerful for long vectors, lists, or data.frames.

```r
lapply(data, mean)
apply(data3, MARGIN = 1, mean)
apply(data3, MARGIN = 2, sum, na.rm = T)
apply(data3, MARGIN = 2, function(o) return(o + 1))  # equal to data3+1
```

Simple S3 functions

S3 functions go back to the third version of the S language, the predecessor of R. It is a simple way to combine several data handling steps. Therefore, an object is created similar to any other variable and then the individual handling steps are defined. Let us create a function that calculates the mean and the sum of a given vector and returns these two values as a named data.frame.

```
vector <- runif(25, 1, 100)

calculate <- function(x) {
    mean <- mean(x)
    sum <- sum(x)
    df <- data.frame(MEAN = mean, SUM = sum)
    return(df)
}

calculate(vector)

##    MEAN  SUM
## 1 57.03 1426
```

More complex S4 functions

Other than S3 functions S4 functions are stricter and thus help to keep track of what is happening within the function. At first we need to define our function as a generic function so that R knows that 'Calculate' is a defined function with the variable 'x.' In the next step we define the actual 'Calculate' function, declare the class for each variable, and define the variables that the function will use. The benefits of S4 functions are (1) that you can use the same function name for different object classes (see below) and (2) that you can define default values using the `setGeneric()` function.

```
setGeneric("Calculate", function(x, print = T) standardGeneric("Calculate"))
setMethod(f = "Calculate", signature = c(x = "numeric"), definition = function(x,
    print) {
    mean <- mean(x)
    sum <- sum(x)
    if (print == TRUE)
        print("DONE")
    return(data.frame(MEAN = mean, SUM = sum))
})

Calculate(vector)

setMethod(f = "Calculate", signature = c(x = "list"), definition = function(x) {
```

```
    mean <- lapply(x, mean)
    sum <- lapply(x, sum)
    return(list(MEAN = mean, SUM = sum))
})

Calculate(list(vector, vector))
```

And of course you can use the apply functions in combination with your own functions.

```
lapply(data, calculate)
```