CONTROL FLOWS AND PROGRAMMING

Tobias Niedermaier

FUNCTIONS

FUNCTIONS SO FAR ...

So far, we called functions to do things for us. E.g.

```
x <- 1:5
sin(x)
```

```
[1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243
```

```
log(x, base=2)
```

```
[1] 0.000000 1.000000 1.584963 2.000000 2.321928
```

```
sum(x)
```

```
[1] 15
```

We also used functions to create data frames, inspect objects or load/save data. E.g.

```
data(mtcars, package = "datasets")

str(mtcars)
```

```
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
```

```
$ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num  16.5 17 18.6 19.4 17 ...
$ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
$ am  : num  1 1 1 0 0 0 0 0 0 0 ...
$ gear: num  4 4 4 3 3 3 3 4 4 4 ...
$ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

DEFINING OWN FUNCTIONS

But we can also write our own functions. In mathematical terms, this is obvious:

Consider a function $f(x) = x^2 + cos(x) + 2$.

We can automate the evaluation using our own defined function.

```
1 our_function <- function(x){
2   y <- x^2 + cos(x*3)*2 + 2
3   return(y)
4 }
```

Note the `return(...)` statement at the end of the function.

We can now use the function to calculate the result for given values.

```
1 x <- seq(-2,2, length.out = 10)
2 y <- our_function(x)
3 y
```

```
[1] 7.920341 4.328340 1.271220 1.612151 3.621157 3.621157 1.612151 1.271220
[9] 4.328340 7.920341
```

DEFINING OWN FUNCTIONS CONT'D

We can generalize this concept to arbitrary inputs (not only numerical). Here are two examples:

```
 1  # Combine three arguments and returns a list with all combinations concatenated
 2  function1 <- function(x, y, z){
 3    element1 <- c(x,y)
 4    element2 <- c(x,z)
 5    element3 <- c(y, z)
 6    element4 <- c(x, y, z)
 7
 8    return(list(element1, element2, element3, element4))
 9  }
10
11  function1(1,2,3)
```

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

[[2]]
[1] 1 3

[[3]]
[1] 2 3

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

```r
function1("a", "b", "c")
```

```
[[1]]
[1] "a" "b"

[[2]]
[1] "a" "c"

[[3]]
[1] "b" "c"

[[4]]
[1] "a" "b" "c"
```

```r
# A function that sum up the columns and rows of a matrix with additional info
function2 <- function(m){
  print("Dimension of input matrix:")
  print(dim(m))

  rs <- rowSums(m)
  cs <- colSums(m)
  s <- sum(m)

  return(list(RowSums = rs, ColSums = cs, FullSum = s))
}

m1 <- matrix(data=1:9, nrow=3, ncol=3, byrow=FALSE)
m2 <- matrix(-100:100, 100, 2)

function2(m1)
```

```
[1] "Dimension of input matrix:"
[1] 3 3
```

```
$RowSums
[1] 12 15 18

$ColSums
[1]  6 15 24

$FullSum
[1] 45
```

```
1 function2(m2)
```

```
[1] "Dimension of input matrix:"
[1] 100   2

$RowSums
  [1] -100  -98  -96  -94  -92  -90  -88  -86  -84  -82  -80  -78  -76  -74  -72
 [16]  -70  -68  -66  -64  -62  -60  -58  -56  -54  -52  -50  -48  -46  -44  -42
 [31]  -40  -38  -36  -34  -32  -30  -28  -26  -24  -22  -20  -18  -16  -14  -12
 [46]  -10   -8   -6   -4   -2    0    2    4    6    8   10   12   14   16   18
 [61]   20   22   24   26   28   30   32   34   36   38   40   42   44   46   48
 [76]   50   52   54   56   58   60   62   64   66   68   70   72   74   76   78
 [91]   80   82   84   86   88   90   92   94   96   98

$ColSums
[1] -5050  4950

$FullSum
[1] -100
```

EXERCISES 2 TASKS 1

CONDITIONS

IF-ELSE STATEMENT

Consider a function that should do something. However, it depends on the input type.

```
 1  # function should sum up the values. If it is of type character, it should just paste everything to
 2  typed_sum <- function(x){
 3    if (class(x) == "character") {
 4      ret <- paste(x, collapse = " ")
 5    } else {
 6      ret <- sum(x)
 7    }
 8    return(ret)
 9  }
10
11  typed_sum(1:5)
```

```
[1] 15
```

```
 1  typed_sum(c("This", "will", "be", "one", "sentence"))
```

```
[1] "This will be one sentence"
```

```
 1  typed_sum(c("I", "am", 18+17, "years old"))
```

```
[1] "I am 35 years old"
```

- The `else {...}` is optional.
- If more conditions are required, one can use `else if {...}`

LOOPS

FOR LOOPS

So far, we can automate code (avoid repetitive code) now using functions. But we can automate even more using a loop!

An short example:

```
1 x <- c("a", "b", "c", "d")
2 for (i in x) {
3   print(i)  # print each element of a vector on after another
4 }
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

A more complex example:

Let's calculate the Fibonacci sequence until 10.

```
1 a <- rep(0, 10) # this is a container where we will store the solution
2 a[2] <- 1
3
4 # here we need a for loop because we must access the two arguments calculated in the steps before
5 for (i in 3:10) {
6   a[i] <- a[i-2] + a[i-1]
```

```
 7  }
 8  a
```

```
[1]  0  1  1  2  3  5  8 13 21 34
```

WHILE LOOPS

We can also repeat operations until a defined condition is met.

In this example, we sum the elements in a vector until they exceed 100. We also print the number of used elements.

```r
x <- c(11, 20, 1, 44, 99, 2000, 100)

dynamic_sum <- 0
i <- 1
while (dynamic_sum < 100) {
  i <- i + 1
  dynamic_sum <- sum(x[1:i])
}
print(paste("Used elements of the vector:", i))
```

```
[1] "Used elements of the vector: 5"
```

```r
print(paste("Sum is:", dynamic_sum))
```

```
[1] "Sum is: 175"
```

Note that you can use loops in functions as well

EXERCISES 2 TASKS 2

APPLY-FAMILY

LAPPLY

- Consider an operation, that you want to apply to each element of a list. You have 3 options: Write code for each list element
- Iterate over all list elements and call a function to with each element , i.e. in each iteration
- Apply the function to each element directly

LAPPLY EXAMPLES

Easy example:

```
1  l <- list(1:5, 1:100, 1:1000)
2  lapply(l, sum)  # calculate the sum of each element
```

```
[[1]]
[1] 15

[[2]]
[1] 5050

[[3]]
[1] 500500
```

Data frames are just lists! So we can use this fact here. We may calculate the maximum value of each column.

```
1  str(iris) # iris data set has a factor. max() is not meaningful on factors.
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
1  lapply(iris[, 1:4], max)
```

```
$Sepal.Length
[1] 7.9

$Sepal.Width
[1] 4.4

$Petal.Length
[1] 6.9

$Petal.Width
[1] 2.5
```

SAPPLY

`sapply` is basically the same as lapply, but tries to simplify the result. In our last example, this makes sense: Each element is just a number.

```
1  sapply(iris[, 1:4], max)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
         7.9          4.4          6.9          2.5
```

APPLY

There is a basic `apply` function. It is intended to apply a function on an *array*. We have to specify the *margin*. This defines, on which axis, the function should be applied.

```
1  (m <- matrix(1:6, 3, 2))
```

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

```
1  apply(m, MARGIN = 1, FUN = sum)  # rowsums
```

```
[1] 5 7 9
```

```
1  apply(m, MARGIN = 2, FUN = sum)   # colsums
```

```
[1]  6 15
```

```
1  apply(m, MARGIN = 1:2, FUN = sum)  # sum on each element
```

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

Warning

Apply on data frames will cast a data frame into a matrix with (as.matrix/array!)

OTHER APPLY FUNCTIONS

There are a lot of other apply functions. To name some of them:

- `mapply` (apply a function to multiple vectors/lists)
- `tapply` (apply over ragged vectors)
- `pbapply` (adds a progress bar, package: `pbapply`)
- `mclapply` (parallel version of lapply, package: `parallel`)

EXERCISES 2 TASKS 3