Functions

Marcus Wurzer

In the preceding units, we already used many functions and got to know some details about their mode of operation, e.g., function calls and function arguments. And we discussed that all functions in R are objects. Now, we will extend this basic knowledge.

Function calls

In notebook 1, we got to know that a function is called by its name, followed by brackets containing (zero, one, multiple) function arguments that can be specified. The function arguments are seperated by commas and may or may not have default values.

R is a functional programming language. This means that even mathematical operations like `*`, `+`, `-` etc. or the assignment operator `<-` are functions - they are just used in a special short form for convenience. In long form, we would have to specify the code like that:

```
"+"(1, 5) # = 1 + 5
```

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

```
"<-"(x, 3) # = x <- 3
x
```

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

Creating functions

We already defined a simple function in notebook 4 (Conditions and loops):

```
square <- function(x) x * x
square
```

```
## function(x) x * x
```

```
square(x = 2)
```

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

The general form of a function is given by:

```
f <- function(arguments) {
  code
}
```

When the code is very simple and can be represented by one line, we can pass on using curly braces (see above).

R functions can be edited using the `fix()` function:

```
fix(square)
```

Function components

We can get information about the components of the function using the following commands. The `body()` is just the the code inside the function.

```
body(square)
```

```
## x * x
```

Using `formals()`, we get a list of arguments used to control a function (here: `x`):

```
formals(square)
```

```
## $x
```

Finally, the environment shows us "where we are", i.e., where the function's variables are located (here: in the global environment):

```
environment(square)
```

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

In conjunction with the assignment operator, we can modify each of these three parts of a function separately.

Primitive functions

Some functions do not contain R code, but call C code directly. They are only found in the base package and are called *primitive functions*. When trying to show the function components of primitive functions, they are all `NULL`, e.g., for `sum()`:

```
body(sum)
```

```
## NULL
```

```
formals(sum)
```

```
## NULL
```

```
environment(sum)
```

```
## NULL
```

Calling `sum()`, we see that it is a primitive function:

```
sum
```

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

Return values

Per default, objects created in the last line of a function are returned:

```
add_sqrt <- function(x) {
  y <- x + x
  z <- sqrt(y)
  y
  z
}

add_sqrt(3)
```

```
## [1] 2.44949
```

If we also want to return `y`, we could use:

```
add_sqrt <- function(x) {
  y <- x + x
  z <- sqrt(y)
  y
  c(y, z)
}

add_sqrt(3)
```

```
## [1] 6.00000 2.44949
```

An explicit return in list form can be forced using `return()`:

```
add_sqrt <- function(x) {
  y <- x + x
  z <- sqrt(y)
  return(list(y = y, z = z))
}

add_sqrt(3)
```

```
## $y
## [1] 6
##
## $z
## [1] 2.44949
```

`return()` may also be used in other parts of the code, not necessarily in the final line.

Lexical scoping

When talking about functions, one question concerns the visibility/existence of objects: Are they present in the workspace (usually the case when working directly in the console) or only within the scope of a function? This is of importance for a variety of reasons:

a. Objects created within the scope of a function should not overwrite objects of the same name existing in the workspace.
b. Objects created within the scope a function should be used for calculations instead objects of the same name existing in the workspace.
c. Objects that are only needed temporarily should not be created in the workspace for clarity and memory-saving reasons.

Because of that, it makes sense to evaluate functions in specific environments. Scoping rules govern how R looks up the value of an object, e.g., that the value returned for `y` is 10:

```
y <- 10
y
```

```
## [1] 10
```

We only treat *lexical scoping* here, not *dynamic scoping*, the other type of scoping. Lexical scoping has four basic principles: *Name masking*, *Functions vs. variables*, *A fresh start*, and *Dynamic lookup*:

Name masking

```
rm(list = c("x", "y")) # remove objects x and y

f <- function() {
  x <- 1
  y <- 2
  c(x, y)
}

f()
```

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

```
x
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

```
y
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

```
rm(f)
```

Here, `x` and `y` were defined within the function. If a name isn't defined inside a function, R will look one level up:

```
x <- 2
g <- function() {
  y <- 1
  c(x, y)
}

x
```

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

```
y
```

```
## Error in eval(expr, envir, enclos): object 'y' not found
```

```
g()
```

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

```
rm(x, g)
```

We already discussed that `=` may also be used as assignment operator. Comparing `=` and `<-`, we can now see how the behavior differs when used within a function:

```
sum(z = 1:10)
```

```
## [1] 55
```

```
z  # assignment was not made since the scope was within 'sum'
```

```
## Error in eval(expr, envir, enclos): object 'z' not found
```

```
sum(z <- 1:10)
```

```
## [1] 55
```

```
z
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

If R also doesn't find the object in the global environment (like, e.g., `y` above), it will continue with the loaded packages. We can have a look at the search path for R objects using `search()`:

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

The first element of the search path, `.GlobalEnv`, is placed at "position 0" (= "in the middle of the search path"), the last element of the search path (here: "position -9") is the `base` package. When a function is called, a new environment is created and placed before the above search path (i.e., at "position 1"). A function that would be called from within this function would again create a new environment and added to the search path ("position 2", just before "position 1" etc.). By starting from the highest position in the search path, down to the last position (`base` package), always the newest version of an object will be found, because it is the last one that was created.

Functions vs. variables

Finding functions works the same way as finding variables.

A fresh start

This means that every time you run a function, this current run is completely independent from all the previous ones:

```
j <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

j()
```

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

```
j()
```

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

```
j()
```

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

```
rm(j)
```

Reason for this behavior: An new enviroment is created to host execution *every time* the function is called.

Dynamic lookup

```
f <- function() x
x <- 15
f()
```

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

returns the value of x when `f()` is run. When `x` is changed (outside the function environment!), also the function output changes:

```
x <- 20
f()
```

```
## [1] 20
```

Problem: We didn't explicitely pass x to function `f`. We see that the function is not self-contained, which is usually to be avoided.

Lazy evaluation

R function arguments are lazy, i.e., they are evaluated with delay (only when they are actually used). We define a function called `lazy.func()`:

```
lazy.func <- function(x, calculate = TRUE) {
  if (calculate) x <- x + 1
  print(a)
}
```

Now we call the function as follows:

```
lazy.func(a <- 3, calculate = FALSE)
```

```
## Error in lazy.func(a <- 3, calculate = FALSE): object 'a' not found
```

Why is `a` not found? Because `calculate = FALSE`, `x <- x + 1` is not evaluated. Since `x` was not evaluated, also assignment operation `a <- 3` was not evaluated, leading to an error for `print(a)`, because `a` is not known.

Now, we use `calculate = TRUE`:

```
lazy.func(a <- 3)
```

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

This worked because `x <- x + 1` was evaluated. Here, `x` was directly specified by `a <- 3` and thus, `a` was created as well and is known when `print(a)` is called. We see that it makes little sense to use assignment operators in arguments because of lazy evaluation, but the principle of lazy evaluation makes sense in the following example:

```
call.f <- function(x) return(list(Call.f = substitute(x), value = x))
call.f(1 + 2)
```

```
## $Call.f
## 1 + 2
##
## $value
## [1] 3
```

`substitute` returns the parse tree for the (unevaluated) expression expr (first argument), substituting any variables bound in env (second argument, defaults to the current evaluation environment), e.g.:

```
x <- pi / 4
substitute(sin(x), list(x = x))
```

```
## sin(0.785398163397448)
```

Function `call.f` only contains a `return()` command returning a list of two elements: Since `x` is only evaluated when called, `substitute(x)` can still extract the initial expression `1 + 2`.

`force()` can be used to force argument evaluation, i.e., to override lazy evaluation.