Solutions to Chapter 6

Manuel Steiner

Contents

6.1.2 Exercise

> **Question 1**: What function allows you to tell if an object is a function?
> What function allows you to tell if a function is a primitive function?

The function `is.function(x)` checks if `x` is a function, primitive or any other type.

```
testfunc <- function(a) 1 + a
typeof(testfunc); is.function(testfunc)
```

```
## [1] "closure"
```

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

```
# For a primitive function:
typeof(sum); is.function(sum)
```

```
## [1] "builtin"
```

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

The function `is.primitive(x)` returns `TRUE` if and only if `x` is a primitive function (one of "builtin" or "special"). It returns `FALSE` otherwise

```
typeof(sum) # a primitive
```

```
## [1] "builtin"
```

```
is.primitive(sum)
```

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

```
typeof(testfunc) # not of type builtin or special
```

```
## [1] "closure"
```

```
is.primitive(testfunc) # not a primitive
```

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

Question 2: This code makes a list of all functions in the base package

```
objs <- mget(ls("package:base"), inherits = TRUE)
funs <- Filter(is.function, objs)
```

Use it to answer the following questions:
a. Which base function has the most arguments?
b. How many base functions have no arguments? What's special about those functions?
c. How could you adapt the code to find all primitive functions?

Lets first understand what the code does. The function `ls()` with no argument return a list of all objects currently stored in the global environment. If given a specific environment or the name of an element in the search path (accesible with `search()`) the function returns all objects found in that specifc environment or search path element. Using `search()` gives the following elements in the search path

```
search()
```

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

Hence, `ls("package:base")` return a vector of names of all the objects in the base package.

```
head(ls("package:base"))
```

```
## [1] "-"         "-.Date"    "-.POSIXt"  "!"         "!.hexmode" "!.octmode"
```

The functions `get(x, ... )` and `mget(x, ...)` search the current environment of the call to these functions for the object given by `as.character(x)` returning its **value** if found. The differences between `get()` and `mget()` are:

1. For `mget()` the argument `x` can be a character vector of object names, while `get()` accepts only one object. Note: if given more than one object name `get()` returns the first object in the vector ignoring the rest.

2. The default behaviour of `get()` is to search through all enclosing environments, while `mget()` does not. This may be changed by adding `inherits = TRUE` as an argument.

3. If no object is found `get()` and `mget()` return an error. For `mget()` this may be changed by providing a list of values to to be used if the obeject is not found.

```r
x <- 1 + 3
y <- 2 + 3
get("x") # only one argument allowed
```

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

```r
mget("x") # return the same but as a list
```

```
## $x
## [1] 4
```

```r
### The following returns only x
get(c("x", "y"))
```

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

```r
### This causes an error
mget(c("x", "z"))
```

```
## Error: value for 'z' not found
```

```r
get("z")
```

```
## Error in get("z"): Objekt 'z' nicht gefunden
```

```r
get(c("z", "x"))
```

```
## Error in get(c("z", "x")): Objekt 'z' nicht gefunden
```

```r
### This works, however
get(c("x", "z")) # z is ignored
```

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

```r
### Changing the default behviour for objects not found:
mget(c("y", "x", "z"), ifnotfound = list("This is not good"))
```

```
## $y
## [1] 5
##
## $x
## [1] 4
##
## $z
## [1] "This is not good"
```

If the objects are not found in the environment from which `mget()` was called the argument `inherits = TRUE` causes the function to search for the objects in the enclosing environments too. Therefore the first line of code list all the objects found in the base package and returns their value. It is important to set `inhertis = TRUE` since the function `mget()` is not called from the same environment as the objects in the base package:

```
## Returns an error
mget(ls("package:base"))
```

```
## Error: value for '-' not found
```

```
objs <- mget(ls("package:base"), inherits = TRUE) # correct
head(objs, 2)
```

```
## $`-`
## function (e1, e2)  .Primitive("-")
##
## $`-.Date`
## function (e1, e2)
## {
##     coerceTimeUnit <- function(x) as.vector(round(switch(attr(x,
##         "units"), secs = x/86400, mins = x/1440, hours = x/24,
##         days = x, weeks = 7 * x)))
##     if (!inherits(e1, "Date"))
##         stop("can only subtract from \"Date\" objects")
##     if (nargs() == 1)
##         stop("unary - is not defined for \"Date\" objects")
##     if (inherits(e2, "Date"))
##         return(difftime(e1, e2, units = "days"))
##     if (inherits(e2, "difftime"))
##         e2 <- coerceTimeUnit(e2)
##     if (!is.null(attr(e2, "class")))
##         stop("can only subtract numbers from \"Date\" objects")
##     structure(unclass(as.Date(e1)) - e2, class = "Date")
## }
## <bytecode: 0x00000000099d6de8>
## <environment: namespace:base>
```

The function `Filter(f, x, ...)` extracts the elements of a vector `x` for which the logical function `f` return `TRUE`. Therefore the following code chunk checks each element of `objs` whether it is a function returning its value if the condition is `TRUE`.

```
funs <- Filter(is.function, objs)
```

If we compute the length of `objs` (the complete list of all objects in the base package) and `funs` (the list of all objects in the base package that are not functions) we see that `length(objs) - length(funs)`$= 10$ are not functions. To exctract these elements use `Negate(is.function)`.

```
Filter(Negate(is.function), objs)
```

Now we can answer the questions

a. Which function has the most arguments?

We use `sapply()` to compute the number of arguments per function using `formals()` to access the formal function arguments.

```
leng <- sapply(funs, function(x) length(formals(x)))
slength <- sort(leng, decreasing = TRUE)
slength[1:4]
```

```
##             scan   format.default          formatC merge.data.frame
##               22               16               14               12
```

We see that with 22 arguments the base function `scan()` has the most arguments, followed by `format.default()` with 16.

b. How many base function have no arguments? What's special about those functions?

```
zerolength <- funs[leng == 0]
```

There are `length(zerolength)`= 225 functions without any argument. Most of the functions are primitive function. To be precise: there are `sum(sapply(zerolength, is.primitive))`= 183 primitive functions. The rest calls `.Internal()` pretty quickly. To see the non-primitive function we can type

```
Filter(Negate(is.primitive), zerolength)
```

c. How could you adapt the code to find all primitive functions?

This is easily done by replacing `is.function` by `is.primitive`

```
funs_prim <- Filter(is.primitive, objs)
table(sapply(funs_prim, is.primitive)) # only TRUE's
```

```
##
## TRUE
##  183
```

Question 3: What are the three important components of a function?

The three components are: the (formal) arguments accessed by `formals()`, the `body()` and the function environment. The latter is the "map" of the location of the variables used within the function.

Question 4: When does printing a function not show what environment it was created in?

If the function was created in the global environment only the body and the formals are shown

```
mean ## from the base package
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000008dc75d0>
## <environment: namespace:base>
```

```
testfunc ## created in the global environment
```

```
## function(a) 1 + a
```

6.2.5 Exercise

Question 1: What does the following code return? Why? What does each of the three `c`'s mean?

```
c <- 10
c(c = c)
```

```
##  c
## 10
```

The code returns a named vector with 10 as its only entry. The `c`'s have three different meanings here. The first part assigns the value 10 to an object named "`c`". So `c` is the *object name* in this case. The second `c` is short for "concatenate"" and is the name of a (primitive) function that initiates a vector. Hence, the second `c` is a *function name*. The last `c` within the vector is an attribute of an entry in a that vector (in this case a *name attribute*).

Question 2: What are the four principles that govern how R looks for values

There are four principles:

Name masking If a name that is used by the function isn't found within the function body, R starts to search through all the enclosing environments. If the name cannot be found anywhere an error is printed.

Functions vs. variables Function names and variable names are treated the same way unless the name is clearly associated with a function. This is mainly the case for functions such as `c()` or any name followed by `()` as this indicates a function name. In general, one should avoid using names that resemble function names (e.g. avoid calling your variable mean, sum, c etc.).

A fresh start Each time a function is run it creates a temporary environment in which all values created during execution are stored. If the function is executed a second time a new environment is created. Hence each run is completly independent of the previous. All values created in the previous run a effectively discarded.

Dynamic Lookout A function starts looking for values when it is run! This means that the result of a call to a function may differ depening on how and which external values are defined. This is generally hard to avoid making it even more necessary to write function as self-contained as possible.

Question 3: What does the following function return? Make a prediction before running the code yourself

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x^2
    }
    f(x) + 1
  }
  f(x)*2
}
f(10)
```

It should return 202.

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x^2
    }
    f(x) + 1
  }
  f(x)*2
}
f(10)
```

```
## [1] 202
```

Indeed correct.

6.4.6 Exercise

Question 1: Clarify the following list of odd function calls:

```
x <- sample(replace = TRUE, 20, x = c(1:10, NA))
y <- runif(min = 0, max = 1, 20)
cor(m = "k", y = y, u = "p", x = x)
```

When writting function arguments one should stick to the order in which the arguments are read (accesible via `formals("function")` or simply via the help file). Wheather the function arguments should be named or not depends on the context. Although R does partial name matching one should avoid writting partial names, as this only causes confusion and does not really save time (there is auto-completion! nowadays).

Generally, one should write the code in a way that it is as readable as possible when comming back to it one year later or when sending it to someone else, which means: its better to be more verbal than not. Thus, the better approach would have been

```
x <- sample(x = c(1:10, NA), size = 20, replace = TRUE)
y <- runif(20, min = 0, max = 1) # here the x = 20 is probably really to much.
cor(x, y, use = "pairwise", method = "kendall")
```

Question 2: What does this function return? Why? Which principle does it illustrate?

```
f1 <- function(x = {y <- 1; 2}, y = 0){
x + y
}
```

Remark: the curly braces can be used to define multiple values at the time:

```
x <- {z <- 1; y <- z + 3; 5}
x
```

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

This defines `x`, `z` and `y`.

```
mget(ls())
```

```
## $x
## [1] 5
##
## $y
## [1] 4
##
## $z
## [1] 1
```

The value of `x` is 5 since the object defined (here `x`) takes on the last value in the curly brace expression. This is true even if only assigments are made

```
x <- {z <- 1; y <- z + 3}
x
```

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

The value of `x` is now 4 since `z`= 1 and `y`= 1+ 3.

As to the question: the function `f1()` returns 3 because R reads the arguments from left to right. Hence, given that we didnt enter any function arguments, the value of `x` is taken to be the default value 2 (the last value in the curly braces). But since calling `x` causes the default value to be evaluated `y` is assigned a value of 1. Therefore, the default value of y (here: 0) is no longer needed and remains unevaluated. This illustrates several concepts. First it shows that evaluation is strictly left to right. Second, it illustrates lazy evaluation. If we were to enter `x` explicitly we would get

```
f1(x = 1)
```

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

as now the expression `{y <- 1; 2}` is replaced by `1` thus leaving `y` undefined causing the default value to jump in.

Question 2: What does this function return? Why? Which principle does it illustrate?

```
f2 <- function(x = z){
z <- 100
x
}
```

```
f2 <- function(x = z){
z <- 100
x
}
f2()
```

```
## [1] 100
```

The function returns 100. This is again an example of lazy evaluation. The default argument `x = z` is only evaluated once it gets called by the function. As `z` is defined prior to evaluating `x`, this works. But note the order. The following causes an error since `x` (via `z`) has not been defined yet.

```
rm(list = ls()) # to remove previouly defined x and z.
f3 <- function(x = z){
  x
  z <- 100
}
f3()
```

```
## Error in f3(): Objekt 'z' nicht gefunden
```

6.5.3 Exercise

Question 1: Create a list of all replacement functions found in the base package. Which ones are primitive functions?

funs <- mget(ls("package:base"), inherits = TRUE) repl_funs <- funs[grepl("^(.*?)<-$", names(funs))] ?grepl names(funs)

funs grepl("<-$", names(funs)) head(funs)

require(pryr) find_funs("package:base", fun_body, pattern = "as.data.frame", fixed = TRUE) mem_used() ?cor