Manuel J. A. Eugster

Programming R

Chapter 4: Writing your own functions

Last modification on May 15, 2012

Draft of the R programming book I always wanted to read
http://mjaeugster.github.com/progr

4 Writing your own functions

Extend R with own ideas.

The main literature for this section is:

- *Software for Data Analysis: Programming with R* by Chambers (2008)
- *An Introduction to R* by R Core Team (2012a)
- *R Language Definition* by R Core Team (2012b)

4.1 Function definition

Functions are the workhorses of R. A function is a group of statements (i.e. a grouped expression) that takes some input, does some computations and returns a result.

A function is defined by an assignment of the form

```
> name <- function(arg_1, arg_2, ...) {
+   statements
+ }
```

`arg_1`, `arg_2`, `...` are called the **formal arguments**. They are usually used in the statements, i.e., the function **body**, to compute the **return value** (the result of last statement).

A call to the function then takes the form:

```
> name(expr_1, expr_2)
```

Functions split your source code into smaller, more manageable, parts. The advantages of this include (cf. Wikipedia, 2012):

- decomposition of a complex programming task into simpler steps;
- reducing the duplication of code within a program;
- enabling the reuse of code across multiple programs;

- improves debugging and testing;

A simple example is a function which finds the most deviant observation of a given input vector:

```
> which_maxdev <- function(x) {
+   mdn <- median(x)
+   devs <- abs(x - mdn)
+   which.max(devs)
+ }
```

The function has the name `which_maxdev`, takes the input vector `x` and returns the index of the element with the most deviation within `x`. Note that the statement in the last line is the return value of the function (one can also use the `return()` for returning a value not at the end of a function).

The function is called as follows:

```
> data("Forbes2000", package = "HSAUR2")
>
> which_maxdev(Forbes2000$sales)

[1] 10

> which_maxdev(Forbes2000$marketvalue)

[1] 2
```

4.2 The ... argument

Computing the index of the element with the most deviation for the `profits` variable of the `Forbes2000` `data.frame` results in

```
> which_maxdev(Forbes2000$profits)

integer(0)
```

This is the case because `Forbes2000$profits` contains `NA` values and therefor the median of the vector is `NA` as well (debug the function to investigate this problem). Now, the `median()` functions has an argument `na.rm`

```
> args(median)

function (x, na.rm = FALSE)
NULL
```

to indicate whether `NA` values should be stripped and the median of the remaining elements should be computed.

We can easily extend our function `which_maxdev()` with the same functionality by introducing the `...` argument:

```
> which_maxdev <- function(x, ...) {
+   mdn <- median(x, ...)
+   devs <- abs(x - mdn)
+   which.max(devs)
+ }
```

Arguments which are not listed in the function definition of `which_maxdev` are now collected and passed on to the underlying function `median()`. We can now call the function which the additional `na.rm` argument:

```
> which_maxdev(Forbes2000$profits, na.rm = TRUE)

[1] 364
```

The dot-dot-dot argument is used by many graphic functions to pass on graphical parameters to `par()` to control the graphical output. Take, for example, a look at the code of `plot.default`.

4.3 Named arguments and defaults

If we want to make the removing of the `NA` values the default behavior, we have to define an argument `na.rm` with the default value `TRUE` and pass on this argument to the `median()` function.

```
> which_maxdev <- function(x, na.rm = TRUE) {
+   mdn <- median(x, na.rm = na.rm)
+   devs <- abs(x - mdn)
+   which.max(devs)
+ }
```

The computation now works for `Forbes2000$profits` without any further argument specification:

```
> which_maxdev(Forbes2000$profits)


[1] 364
```

Note that if arguments to called functions are given in the `name = object` form, they may be given in any order. The `which_maxdev()` function may be invoked in several ways, for example

```
> which_maxdev(Forbes2000$sales)
> which_maxdev(x = Forbes2000$sales, na.rm = FALSE)
> which_maxdev(na.rm = FALSE, x = Forbes2000$sales)
> which_maxdev(na.rm = FALSE, Forbes2000$sales)
```

are all equivalent.

4.4 Argument matching

The first thing that occurs in a function evaluation is the matching of formal to the actual or supplied arguments. This is done by a three-pass process:

Exact matching on tags. For each named supplied argument the list of formal arguments is searched for an item whose name matches exactly. It is an error to have the same formal argument match several actuals or vice versa.

```
> which_maxdev(Forbes2000$sales, na.rm = FALSE)
```

Partial matching on tags. Each remaining named supplied argument is compared to the remaining formal arguments using partial matching. If the name of the supplied argument matches exactly with the first part of a formal argument then the two arguments are considered to be matched. It is an error to have multiple partial matches.

```
> which_maxdev(Forbes2000$sales, n = FALSE)
```

Positional matching. Any unmatched formal arguments are bound to unnamed supplied arguments, in order. If there is a `...` argument, it will take up the remaining arguments, tagged or not.

```
> which_maxdev(n = FALSE, Forbes2000$sales)
```

If any arguments remain unmatched an error is declared.

```
> which_maxdev(Forbes2000$sales, foo = 1)

Error: unused argument(s) (foo = 1)
```

4.5 Argument checking

As programmers of a function we have to ensure that the function does what it aims to do. This also means that we have to catch wrong input arguments and provide a meaningful error message.

The function `which_maxdev()` for example does not work correctly for the following inputs:

```
> which_maxdev(Forbes2000$name)

Warning message: argument is not numeric or logical: returning NA

Error: non-numeric argument to binary operator

> which_maxdev(as.matrix(Forbes2000[, c("sales", "marketvalue")]))

[1] 2002

> which_maxdev(Forbes2000$sales, na.rm = "yes")

Error: argument is not interpretable as logical
```

Test can be incorporated most easily if they are **assertion tests**; that is, expressions that are asserted to evaluate to the logical value `TRUE`. R provides the function `stopifnot()` to run assertion test.

We extend the function `which_maxdev()` to catch wrong input arguments; i.e., `x` has to be a numeric vector, and `na.rm` has to be a logical value:

```
> which_maxdev <- function(x, na.rm = TRUE) {
+   stopifnot(is.numeric(x) & is.vector(x))
+   stopifnot(is.logical(na.rm))
```

```
+
+   mdn <- median(x, na.rm = na.rm)
+   devs <- abs(x - mdn)
+   which.max(devs)
+ }
```

The computation of the observation with the maximum deviation of a matrix now raises an error:

```
> which_maxdev(as.matrix(Forbes2000[, c("sales", "marketvalue")]))

Error: is.numeric(x) & is.vector(x) is not TRUE
```

Note: Use `apply()` to properly compute this function for a matrix or data.frame:

```
> apply(as.matrix(Forbes2000[, c("sales", "marketvalue")]), 2,
+       which_maxdev)

      sales marketvalue
         10           2
```

4.6 Return value

The return value of a function can be any R object. Per definition the return value of a function is the value of the last expression of the function. In case of the `which_maxdev()` function its return value is the value of the `which.max()` function. Complex return values can be created by using (nested) lists.

One can use `return()` to explicitly indicate the return value; this is useful to exit a function "at any time", e.g., within a `if` statement. Note that `invisible()` allows function return values which can be assigned, but which do not print when they are not assigned.

4.7 Function scope

Scope or the scoping rules are simply the set of rules used by the evaluator to find a value for a symbol.

The symbols which occur in the body of a function can be divided into three classes:

Formal parameters of a function are those occurring in the argument list of the function. Their values are determined by the process of binding the actual function arguments to the formal parameters.

Local variables: are those whose values are determined by the evaluation of expressions in the body of the functions. Local variables must first be defined, this is typically done by having them on the left-hand side of an assignment.

Free variables: are those which are not formal parameters or local variables. Their value is determined by searching in the environment of the function, then its enclosure and so on until the global environment is reached. This is called **lexical scope**.

Formal parameters and local variables are called **bound variables**—they are bound to the environment of the function and do not exist outside of it. Free variables are called **unbound variables**—R has to search for their value in the enclosing environments.

Consider the following function definition:

```
> f <- function(x) {
+   y <- 2 * x
+   cat("x =", x, "\n")
+   cat("y =", y, "\n")
+   cat("z =", z, "\n")
+ }
```

In this function, `x` is a formal parameter, `y` is a local variable and `z` is a free variable.

Executing `f()` throws an error, because the free variable `z` is not defined in any enclosing environment of the function:

```
> f(2)


x = 2
y = 4


Error: object 'z' not found
```

If we now define an object `z` in the global environment

```
> z <- 42
```

R finds this `z` because of the lexical scoping rule and the function is executed without an error:

```
> f(2)


x = 2
y = 4
z = 42
```

Note that using free variables in a function is very error-prone and should not be used without a very (very!) good reason. Generally all variables needed within a function should be passed via the formal arguments.

Bibliography

John Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008. ISBN 9780387759357.

R Core Team. *An Introduction to R*, 2012a. URL http://cran.r-project.org/doc/manuals/R-intro.html.

R Core Team. *The R language definition*, 2012b. URL http://cran.r-project.org/doc/manuals/R-lang.html.

Wikipedia. Subroutine. Wikipedia entry, accessed on 2012-05-14, 2012. URL http://en.wikipedia.org/wiki/Function_%28computer_science%29.