Manuel J. A. Eugster

Programming R

Chapter 3: Navigate through R

Last modification on May 8, 2012

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

3 Navigate through R

First steps in finding, reading, understanding, etc. R objects and evaluation.

The main literature for this section is:

- *Software for Data Analysis: Programming with R* by Chambers (2008)
- *R Language Definition* by R Core Team (2012)
- *R Help Desk: Accessing the Sources* by Ligges (2006)
- *How R Searches and Finds Stuff* by Gupta (2012)

3.1 Global environment

Everything starts in the **global environment**. It is the root of the user workspace and an assignment operation from the command line will cause the relevant object to belong to the global environment.

```
> x <- 1
> ls()

[1] "x"
```

Now, R uses different concepts—packages, namespaces, search paths, generic methods, etc.—to make R usable for the users. However, these concepts make it harder to navigate through R—one has to have a basic understanding of the concepts to orient oneself.

For example, the `identity()` function is not listed in the objects listing above, nevertheless it is available and usable:

```
> identity(1)

[1] 1
```

The `identity()` function is defined in the package:

```
> find("identity")

[1] "package:base"
```

It is available to the user because R maintains a **search path** of attached packages

```
> search()

 [1] ".GlobalEnv"            "package:HSAUR2"
 [3] "package:scatterplot3d" "package:MASS"
 [5] "package:lattice"       "package:knitr"
 [7] "package:stats"         "package:graphics"
 [9] "package:grDevices"     "package:utils"
[11] "package:datasets"      "package:methods"
[13] "Autoloads"             "package:base"
```

and looks up the function by name first in the global environment and then successively in the parent environments. Attaching or detaching a package obviously changes the search path.

The global environment object is accessible via the `globalenv()` function.

3.2 Search Path

Be it that one has defined an object `identity` at the global environment:

```
> identity <- 17
```

Then, obviously, two objects are found:

```
> find("identity")

[1] ".GlobalEnv"   "package:base"
```

One defined in the global environment, and one in the `base` package. Because of the chain of environments, the object in the global environment is found first:

```
> identity

[1] 17
```

In order to use the function in the `base` package, the **double colon operator** `::` has to be used:

```
> base::identity(10)


[1] 10
```

In consequence, removing the `identity` object, removes the object in the global environment:

```
> rm(identity)
> ls()


[1] "txt" "x"
```

Note that removing an object within a package (e.g., `rm(base::identity)`) is not allowed.

3.3 Packages

Similarly to the collection of the user-defined objects in the global environment, the objects defined by a package are collected in an environment as well. This becomes visible when printing the source code of a function:

```
> identity


function (x)
x
<bytecode: 0x02234d4c>
<environment: namespace:base>
```

The last line prints the environment in which the function is defined.

3.4 Namespaces

In contrast to the global environment, not all objects of a package have to be visible to the user. The **namespace management system** for packages allows the package author to specify, among other things, which objects in the package should be **exported** to make them available to package users.

```
> library("HSAUR2")
```

In order to see the exported one has to get the namespace environment of a package via the `getNamespace()` function,

```
> ns <- getNamespace("HSAUR2")
```

and can then receive a list of all exported objects via the `getNamespaceExports()` function:

```
> getNamespaceExports(ns)

[1] "HSAURtable"
```

This package has only one object exported. We can access the exported objects of a loaded and attached package as customary by entering its name

```
> HSAURtable

function (object, ...)
UseMethod("HSAURtable")
<environment: namespace:HSAUR2>
```

or, more precisely, with the double colon operator `HSAUR2::HSAURtable`.

As a namespace is also an environment, we can simply get a list of all defined objects:

```
> ls(envir = ns)

 [1] "caption"                "chkS"
 [3] "cpRoutsave"             "exename"
 [5] "extRact"                "extractBibtex"
 [7] "gattach"                "HSAURcite"
 [9] "HSAURtable"             "HSAURtable.data.frame"
[11] "HSAURtable.table"       "isep"
[13] "isi2bibtex"             "pkgs"
[15] "pkgversions"            "pkgyears"
[17] "prettyS"                "readBibtex"
[19] "Rwelcome"               "toBibtex.HSAURcitation"
[21] "toBibtex.txtBibtex"     "toLatex.dftab"
[23] "toLatex.tabtab"
```

Non-exported objects, like `exename`, can not be accessed with the double colon operator

```
> HSAUR2::exename

Error: 'exename' is not an exported object from 'namespace:HSAUR2'
```

but with the **triple colon operator** `:::`

```
> HSAUR2:::exename

function ()
{
    tversion <- paste(version$major, "0", substr(version$minor,
        1, 1), substr(version$minor, 3, 3), sep = "")
    return(paste("rw", tversion, ".exe", sep = ""))
}
<environment: namespace:HSAUR2>
```

However, notice that there is a reason why objects are not exported. One can not be sure that the object behaves equally in the next version of the package (or even exists anymore). Furthermore, documentation is only required for exported objects.

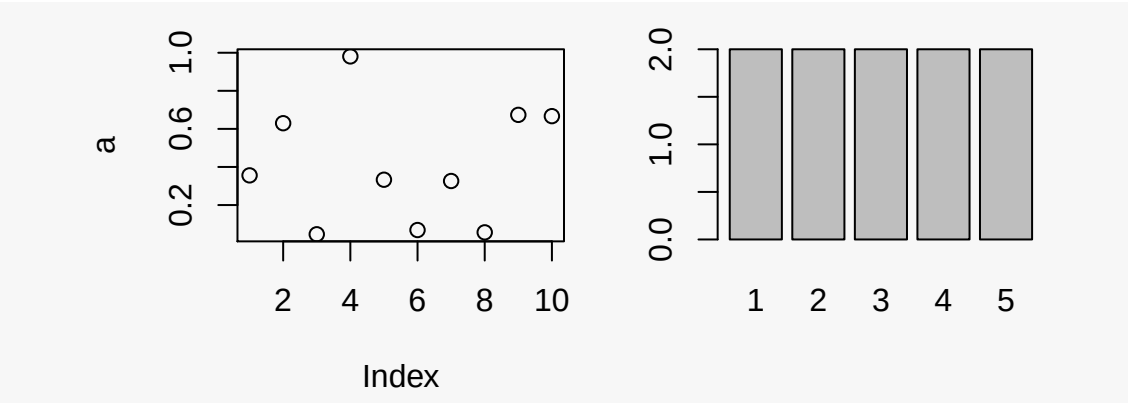
3.5 Generic functions (S3)

Given are two vectors, one of storage type `numeric` and one of storage type `factor`:

```
> a <- runif(10)
> b <- gl(5, 2)
```

If we plot the two vectors, the same function calls results in different results—one scatterplot and one barplot:

```
> par(mfrow = c(1, 2), mar = c(4, 4, 1, 0))
> plot(a)
> plot(b)
```



Calling the documentation `?plot` also only gives very general information.

The function `plot()` is, in fact, a **generic function** in the **S3** object system. This is a very simple mechanism for an object-oriented style of programming. The concrete executing function—now called a **method**—is determined by the class of the first argument.

Now, the classes of the two exemplar vectors are:

```
> class(a)

[1] "numeric"

> class(b)

[1] "factor"
```

Note that in this example the classes are equal to the type of the storage mode (`typeof()`) but this has not to be the case in general.

Generic functions are identifiable by the code line `UseMethod("...")` in their function body:

```
> plot

function (x, y, ...)
UseMethod("plot")
<bytecode: 0x0255257c>
<environment: namespace:graphics>
```

As one can see, no real computation is done in this generic function. Here, the `UseMethod()` function just determines the method to be **dispatched**.

All available plotting methods are determined by the function `methods()`:

```
> methods("plot")

 [1] plot.acf*             plot.correspondence* plot.data.frame*
 [4] plot.decomposed.ts*   plot.default         plot.dendrogram*
 [7] plot.density          plot.ecdf            plot.factor*
[10] plot.formula*         plot.function        plot.hclust*
[13] plot.histogram*       plot.HoltWinters*    plot.isoreg*
[16] plot.lda*             plot.lm              plot.mca*
[19] plot.medpolish*       plot.mlm             plot.ppr*
[22] plot.prcomp*          plot.princomp*       plot.profile*
[25] plot.profile.nls*     plot.ridgelm*        plot.shingle*
[28] plot.spec             plot.stepfun         plot.stl*
[31] plot.table*           plot.trellis*        plot.ts
[34] plot.tskernel*        plot.TukeyHSD

   Non-visible functions are asterisked
```

The naming scheme of S3 methods is `function.class`, where `class` is the class of the first argument. There are several plotting methods available; e.g., for the classes `data.frame`, `lm`, and also `factor`. However, there is not method for the class `numeric`. In such a case, the default method `plot.default` is executed.

Knowing this, the appropriate documentation can be opened:

```
> ?plot.factor
> ?plot.default
```

To see the source code of the methods, the functions `getS3method()` and `getAnywhere()` are available:

```
> getS3method("plot", "factor")
> getAnywhere("plot.factor")
```

If we know the package, we can also use the double (if the method is visible) or the triple (if the function is non-visible) colon operators; e.g.:

```
> graphics:::plot.factor
```

3.6 Print Objects (S3)

This mechanism is also used to show meaningful representations of objects. When an object is required

```
> x


[1] 1
```

implicitly the appropriate `print()`-method based on the class of the object is executed

```
> print(x)


[1] 1
```

Execute `methods("print")` to see the available printing methods; and see `?print` and `?print.default` for possible additional adjustments for printing objects.

Note that most `summary()` methods use this possibility in a very elegant way. Look, for example, at the `summary.lm()` and the `print.summary.lm()` methods.

3.7 Generic functions (S4)

The `methods` package provides a second object system. It is more formal—however, at first we can think of it as an extension of the S3 system where more arguments than only the first argument can be used for method dispatching.

The `stats4` package introduces some basic statistical functions based on the S4 system.

```
> library("stats4")
```

The definition of generic functions in the `S4` system looks slightly different; enter `coef` or

```
> show(coef)


standardGeneric for "coef" defined from package "stats"

function (object, ...)
standardGeneric("coef")
<environment: 0x02a6cb60>
Methods may be defined for arguments: object
Use  showMethods("coef")  for currently available ones.
```

Here, the indicating line is `standardGeneric("...")`. As denoted in the output, S4 `coef()` methods are listed using the `showMethods()` function:

```
> showMethods("coef")


Function: coef (package stats)
object="ANY"
object="mle"
object="summary.mle"
```

Currently, there is a `coef()` method available for objects of the classes `mle`, `summary.mle`, and the default `ANY`.

Use `getMethod()` to see the source of the methods; the first argument is the method name, the second argument the signature, i.e., the classes of the relevent arguments.

```
> getMethod("coef", "mle")
```

3.8 Function Browsing

To understand a function it is often useful to execute its expressions interactively. The `debug()` function arranges that we can execute a function step-by-step and investigate the created objects.

For example, we want to investigate (i.e., **debug**) the function `jitter()` which adds noise to numbers.

```
> debug(jitter)
```

Whenever now the function is called,

```
> jitter(1:10)
```

its execution is interrupted, a browser is opened and one can enter commands or R expressions, followed by a newline. The commands are

n (or just an empty line, by default). Advance to the next step.

c continue to the end of the current context: e.g. to the end of the loop if within a loop or to the end of the function.

cont synonym for c.

where print a stack trace of all active function calls.

Q exit the browser and the current evaluation and return to the top-level prompt.

In order to **undebug** the `jitter()` function, execute:

```
> undebug(jitter)
```

This also works for S3 methods (`debug("function.class")`) but not for S4 methods (see `?trace` and the chapter about debugging).

3.9 Browsing after an Error

Often we only want want to specify a debugging action when an error (or warning) occurs. The action is specified by the value of the global `error` option:

```
> options(error = recover)
```

Whenever an error occurs, the execution is interrupted, the list of current calls is printed, and the user can select one of them. The standard R browser is then invoked from the corresponding environment; and one can enter commands or R expressions (see above).

For example,

```
> jitter(letters[1:3])
```

throws an error because only numerical values are allowed; the browser is invoked and one can investigate the environment of the function.

In order to disable debugging in the case of an error, set the global `error` option to `NULL`.

```
> options(error = NULL)
```

Another important function is `traceback()`. If an error occurred with an unidentifiable error message, this functions shows the sequence of calls that lead to the error.

Bibliography

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

Suraj Gupta. How R searches and finds stuff. Blog entry, accessed on 2012-05-08, 2012. URL http://obeautifulcode.com/R/How-R-Searches-And-Finds-Stuff/.

Uwe Ligges. R Help Desk: Accessing the sources. *R News*, 6(4):43–45, 2006.

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