



Statistics with R

Objects, classes and functions

Deepayan Sarkar

University of Wisconsin – Madison

Summer Institute for Training in Biostatistics (2005)

Objects

Objects in R are anything that can be assigned to a variable. They could be

- Constants: `2`, `13.005`, `"January"`
- Special symbols: `NA`, `TRUE`, `FALSE`, `NULL`, `NaN`
- Things already defined in R: `seq`, `c` (functions), `month.name`, `letters` (character), `pi` (numeric)
- New objects we can create using existing objects (this is done by evaluating **expressions** — e.g., `1 / sin(seq(0, pi, length = 50))`)

Different types of Objects

R objects come in a variety of types. Given an object, the functions `mode` and `class` tell us about its type.

- `mode(object)` has to do with how the object is stored
- `class(object)` gives the `class` of an object. The main purpose of the class is so that generic functions (like `print` and `plot`) know what to do with it.
- Often, objects of a particular class can be created by a function with the same name as the class.

```
> obj <- numeric(5)
> obj
[1] 0 0 0 0 0
> mode(obj)
[1] "numeric"
> class(obj)
[1] "numeric"
```

Mode and class

The mode of an object tells us how it is stored. Two objects may be stored in the same manner but have different class. How the object will be printed will be determined by its class, not its mode.

```
> x <- 1:12
> y <- matrix(1:12, 2, 6)
> class(x)
[1] "integer"
> mode(x)
[1] "numeric"
> class(y)
[1] "matrix"
> mode(y)
[1] "numeric"
> print(x)
[1] 1 2 3 4 5 6 7 8 9 10 11 12
> print(y)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
```

Classes and generic functions

This is achieved by the following mechanism:

- When `print(y)` is called, the `print` function determines that `class(y)` is `"matrix"`, so it looks for a function named `print.matrix`. Such a function exists, so the actual printing is done by `print.matrix(y)`
- When `print(x)` is called, the `print` function determines that `class(x)` is `"integer"`, so it looks for a function named `print.integer`. There is no such function, so instead the fallback is used, and the actual printing is done by `print.default`
- This happens only for **generic functions** (those that call `UseMethod`)

This is actually a simplified version of what really happens, but it's close enough for us.

Functions

- Functions in R are simply objects of a particular type (of mode "function")
- Like other objects, they can be assigned to variables. Most of the time, they actually are assigned to a variable, and we refer to the function by the name of the variable it is assigned to
- All standard functions (like `print`, `plot`, `c`) are actually variables, whose value is an R object of mode "function". When we refer to the `seq` function, we actually mean the value of the variable `seq`

```
> class(seq)
[1] "function"
> mode(seq)
[1] "function"
> print(seq)
function (...)
UseMethod("seq")
<environment: namespace:base>
```

Calling functions

- To call a function, the function object is followed by a list of arguments in parentheses: `fun.obj(arglist)`
- Every function has a list of formal arguments (displayed by `args(fun.obj)`)
- Arguments can be matched by
 - position: e.g., `plot(v1, v2)`
 - name: e.g., `plot(x = v1, y = v2)`
 - defaults: many arguments have default values which are used when the arguments are not specified. For example, `plot` can be given arguments called `col`, `pch`, `lty`, `lwd`. Since they have not been specified in the calls above, the defaults are used.

This is best understood by looking at examples

Functions are objects

Functions are regular R objects, just like other objects like vectors, data frames and lists. So,

- variables can be assigned values which are functions
- functions can be passed as arguments to other functions. We have already seen examples of this — namely `lapply` and `sapply` — where one of the arguments is a function object.

What happens when a function is called ?

- Most functions return a new R object, which is typically assigned to a variable or used as an argument to another function
- Some functions are more interesting for their **side-effects**, e.g., `plot` produces a graphical plot, `write.table` writes some data to a file on your computer

Expressions

Before we talk more about functions, we need to know a bit about **expressions**.

Expressions are statements which are evaluated to create an object. They consist of operators (+, *, ^) and other objects (variables and constants).

e.g., `2 + 2`, `sin(x)^2`

```
> a <- 2 + 2
```

```
> print(a)
```

```
[1] 4
```

```
> b <- sin(a)^2
```

```
> print(b)
```

```
[1] 0.57275
```

Expressions

- R has **expression blocks** which consist of multiple expressions
- All individual expressions inside this composite block are evaluated one by one. All variable assignments are done, and any `print()` or `plot()` call have the appropriate side-effect.
- But the most important feature is that this whole composite block **can be treated as a single expression** whose value when evaluated will be the value of the **last expression evaluated inside the block**

```
> a <- {  
+   tmp <- 1:50  
+   log.factorial <- sum(log(tmp))  
+   sum.all <- sum(tmp)  
+   log.factorial  
+ }  
> print(a)  
[1] 148.4778  
> print(log.factorial)  
[1] 148.4778  
> print(sum.all)  
[1] 1275
```

Defining a function

A new function is defined / created by a construct of the form `fun.name <- function(arglist) expr` where

- `fun.name` is a variable where we store the function. This variable will be used to call the function later
- `arglist` is a list of formal arguments. This list
 - can be empty (in which case the function takes no arguments)
 - can have just some names (in which case these names become variables inside the function, whose values have to be supplied when the function is called)
 - can have some arguments in `name = value` form, in which case the names are variables available inside the function, and the values are their default values
- `expr` is an expression (typically an **expression block**) (which can make use of the variables defined in the argument list). This part is also referred to as the **body** of a function, and can be extracted by `body(fun.obj)`
- Inside functions, there can be a special `return(val)` call which exits the function and returns the value `val`

Variables and Scope

When an expression involves a variable, how is the value of that variable determined ?

- When inside a function, the variable is first searched for **inside** the function. This includes
 - Variables defined as arguments of that function
 - Variables defined inside the function

Variables that are defined inside a function remain in effect only inside the function. When the function complete, these variables can no longer be accessed

- If a variable is not found inside a function, it is searched for **outside** the function. The exact details of this is complicated, but all you need to know is that
 - if you define a variable outside the function, it can be accessed inside the function as well. If two variables of the same name are defined both outside and inside the function, the **one inside will be used**
- If no variable with that name is found, an error is generated

Variables and Scope

```
> myvar <- 1
> myfun1 <- function() {
+   myvar <- 5
+   print(myvar)
+ }
> myfun1()
[1] 5
> myvar
[1] 1
> myfun2 <- function() {
+   print(myvar)
+ }
> myfun2()
[1] 1
```

`args` and `body` give the two components of a function:

```
> args(myfun1)
function ()
NULL
> body(myfun1)
{
  myvar <- 5
  print(myvar)
}
```

What happens to objects ?

An object is created whenever an expression is evaluated. Unless the result is assigned to some variable (or used as part of another expression), this object is lost. In the following example, all the i^2 values are calculated, but nothing is stored and nothing is printed.

```
> for (i in 1:10) i^2
```

Actually, the earlier rule about expressions hold, and this expression produces the value of the last expression evaluated inside it:

```
> a <- for (i in 1:10) i^2
> print(a)
[1] 100
```

But this is obviously not the sort of use the `for` construct is designed for.

What happens to objects ?

What to do with an object after creating it depends on what the purpose is.

- If the only intent is to see the value, the object can be printed using the `print` function

```
> for (i in 1:5) print(i^2)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

- If the intent is to use these values for later, they should be assigned to some variable

```
> a <- numeric(10)
> for (i in 1:10) a[i] <- i^2
> print(a)
[1] 1 4 9 16 25 36 49 64 81 100
```

Automatic printing

Although we haven't made it explicit, it should be clear by now that objects are usually printed automatically when they are evaluated on the command prompt without assigning them to a variable.

- This is equivalent to calling `print()` on the object
- This does not always happen (as in the for example above). R has a mechanism for suppressing this printing, which is used in such cases.
- The automatic printing **never** happens inside a function. So remember to deal with whatever objects you evaluate inside your functions.
- The suppression of automatic printing can sometimes lead to unexpected behaviour. To be safe, use calls to `print` explicitly.