Basic Course on R: Object-Oriented Programming

Elizabeth Ribble*

20-24 May 2019

Contents

*emcclel3@msudenver.edu

1 Object Oriented Programming

1.1 Objects and Classes

1.1.1 Objects

- In R, every "thing" is an ***object***. The vectors `x` and `logicVec` and the matrix `X` are examples of objects:

```
x

##  [1]  2  4  2  7  9 11  9  4  2  6  7  9  9  4  3
```

```
logicVec

## [1]  TRUE FALSE  TRUE
```

```
X

##      [,1] [,2] [,3]
## [1,]    2   11    7
## [2,]    4    9    9
## [3,]    2    4    9
## [4,]    7    2    4
## [5,]    9    6    3
```

- Objects can be passed as arguments to functions which perform operations on them and then return other objects.

1.1.2 Classes of Objects

- Each object belongs to a certain ***class*** of objects. We can determine what class an object belongs to using `class()` or `is.cname()`:

```
class()     # Determines the class of an object.  Can also be
            # used to assign a class to an object.
is.cname()  # Returns TRUE if an object belongs to the class
            # "cname" (e.g. is.numeric(), is.data.frame(), etc.)
            # and FALSE otherwise.
```

- For example, the object `x` belongs to the *numeric vector* class, `logic.vec` belongs to the *logical vector* class, and `X` belongs to the *matrix* class:

```
class(x)

## [1] "numeric"

class(logicVec)

## [1] "logical"

class(X)

## [1] "matrix"
```

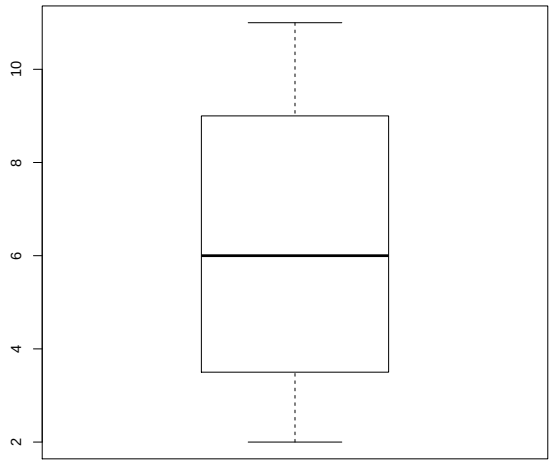
- There are two types of classes in R, **S3** classes and **S4** classes. Nearly all of R's built-in classes are the S3 type, including numeric, character, and logical vectors, matrices and arrays, lists, and data frames.

1.2 Generic Functions and Methods

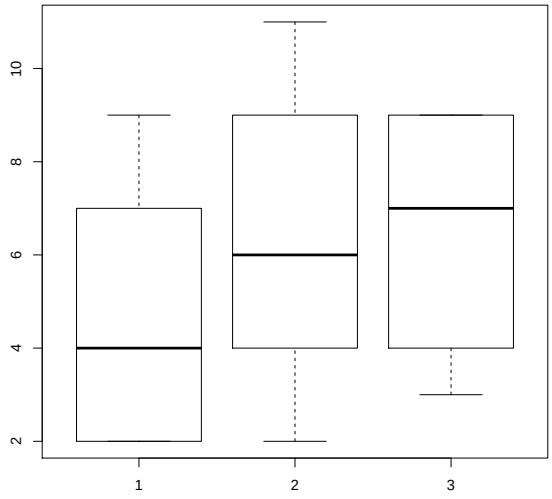
1.2.1 Generic Functions

- Some R functions only accept arguments from a single class. Others accept objects from more than one class.
- Functions that accept objects from more than one class are called ***generic functions***.
- For example, `boxplot()` is a generic function because it can be passed a numeric vector *or* a matrix (among other classes of objects). Below it's first passed a vector and then a matrix:

```
boxplot(x)                # x is a vector.
```




```
boxplot(X)                    # X is a matrix with 3 columns.
```



Notice that `boxplot()` does different things depending on what class of object is passed to it – when passed a vector it produces a single boxplot, but when passed a matrix, it produces side-by-side boxplots of the matrix's columns.

1.2.2 Methods

- There are actually different "versions" of the `boxplot()` function that are referred to as its ***methods***.
- Each `boxplot()` *method* accepts a specific class of arguments. One method accepts vectors and another accepts matrices.
- When we pass an object to a generic function, the function determines the class of the object and then "dispatches" it to the appropriate method.
- If we look at a generic function's definition (by typing its name on the command line), we see that it calls another function, `UseMethod()`, which dispatches the object to the appropriate method. For example:

```
boxplot


## function (x, ...)
## UseMethod("boxplot")
## <bytecode: 0x2df2d00>
## <environment: namespace:graphics>
```

- We can view the methods associated with a given generic function using:

```
methods()       # Determine the S3 methods that are associated
                # with a given generic function
showMethods()   # Determine the S4 methods that are associated
                # with a given generic function
```

- For example, to see the methods for `boxplot()` type:

```
methods(boxplot)


## [1] boxplot.default  boxplot.formula* boxplot.matrix
## see '?methods' for accessing help and source code
```

- The method `boxplot.default()` is the "version" of `boxplot()` that accepts vectors. The `boxplot.matrix()` method is the one that accepts matrices. So typing

```
boxplot(x)
```

is equivalent to typing

```
boxplot.default(x)
```

and typing

```
boxplot(X)
```

is equivalent to typing

```
boxplot.matrix(X)
```

(Actually, in addition to vectors, `boxplot.default()` also accepts several other classes of objects.)

- In general, methods have names of the form `fname.cname()`, where `fname()` is name of the generic function and `cname` is the class of objects that the method accepts.
- "Non-visible" methods (such as `boxplot.formula`) are ones that are "not found" when you type their name on the command line, but R can find them when it needs them. (If you need to view a "Non-visible" method's definition you can do so using `getAnywhere()`, for example `getAnywhere(boxplot.formula)`.)
- For the other ("visible") methods, we can view their definitions by typing their name on the command line, e.g.

```
boxplot.matrix;

## function (x, use.cols = TRUE, ...)
## {
##     groups <- if (use.cols) {
##         split(c(x), rep.int(1L:ncol(x), rep.int(nrow(x), ncol(x))))
##     }
##     else split(c(x), seq(nrow(x)))
##     if (length(nam <- dimnames(x)[[1 + use.cols]]))
##         names(groups) <- nam
##     invisible(boxplot(groups, ...))
## }
## <bytecode: 0x43191a0>
## <environment: namespace:graphics>
```

2 Performance Enhancement: Speed and Memory

2.1 Writing Faster R Code

- Computationally intensive tasks, e.g. those involving very large data sets, can require excessive amounts of computing time and/or memory usage.
- Some ways of speeding up your R code are:
  - Optimizing your code by using *vectorization* (and avoiding loops), *bytecode compilation*, and other techniques.
  - Writing the computationally intensive chunks code in C or C^{++} and then calling them from R.
  - Using parallel R computing.
- The amount of time required to run your code will depend on:
  - The computer it's being run on.
  - The number and types of other processes (applications) that are running while your R code is being executed, such as web browsers, word processors, music players, etc.
- To test the speed of a chunk of R code, we use:

```
system.time()      # Returns the computation time required to
                   # execute a chunk of R code (in seconds)
```

- `system.time()` takes as its argument one or more R commands (enclosed by curly brackets if there are more than one), and returns time spent executing them.
- Here's an example in which we time how long it takes to add two vectors together using a loop:

```
x <- runif(100000)
y <- runif(100000)
z <- NULL
system.time(
    for(i in 1:100000) {
       z[i] <- x[i] + y[i]
    }
)


##    user  system elapsed
##   0.040   0.000   0.039
```

- The return value of `system.time()` contains three components, all reported in seconds:
  - ***User time***: The CPU time spent by the *current process* (i.e. the current R session).[1]
  - ***System time***: The CPU time spent by the *operating system* on behalf of the current process (R session) carrying out tasks that must also be carried out on behalf of other processes (applications), e.g. input/output tasks such as accepting keyboard input, printing to the screen, opening files for reading or writing, etc.[2]
  - ***Elapsed time***. This is the time you would get using a stopwatch, and includes time spent on processes (applications) unrelated to the R session (e.g. open web browsers, music players, word processors, etc.).

[1]The ***CPU*** (central processing unit) is the computer's hardware that carries out instructions of processes (applications) and the operating system, such as performing arithmetic and logical operations and input/output tasks such as receiving messages from the keyboard, printing to the screen, and writing to a file.

[2]The ***operating system*** is the computer's software that manages processes (applications), making sure they don't interfere with each other, and performs common services for those applications such as directing input/output, e.g. from the keyboard to the screen, to or from a file, or a to printer.

- Usually we don't care too much whether the CPU is being used by R or by the operating system (on behalf of R). We just want to know how much total CPU time was used. **The sum of the user and system times gives the total CPU time spent executing the R code**.

2.2 Removing Unnecessary Computations from Loops

- Here are two examples, both of which add $\sqrt{2}$ to each of 100,000 numbers. The first is *inefficient*. The second improves upon the first by removing the computation of $\sqrt{2}$ from the loop because it doesn't belong there:

  1. In this code we unnecessarily compute $\sqrt{2}$ during each of 100,000 iterations of the loop:

```
x <- runif(100000)
system.time(
    for(i in 1:100000) {
      y <- sqrt(2)    # This can be moved outside the loop
      x[i] <- x[i] + y
    }
)
```

```
##    user  system elapsed
##   0.012   0.000   0.013
```

2. The code can be made more efficient by moving the command `y <- sqrt(2)` outside the loop:

```
system.time({
    y <- sqrt(2)            # This was pulled from the loop
    for(i in 1:100000) {
        x[i] <- x[i] + y
    }
})
```

```
##    user  system elapsed
##   0.008   0.000   0.010
```

- It turns out that function calls (like `sqrt(2)` above) are time consuming in R, in part because they involve setting up environments, sometimes several nested inside each other, to store local variables. The second loop above only calls `sqrt()` once (as opposed to 100,000 times), so it's faster. (The task can be performed even faster using vectorization).

2.3 Vector Preallocation

- ***Vector preallocation*** refers to creating a vector *prior* to executing a loop, and then replacing its elements during the loop's iterations. Preallocation can speed up R code.

- Here are three examples in which we use a loop to add two vectors `x` and `y` together, storing the result in another vector `z`. The first two don't use preallocation, the third does.

1. Here's an *inefficient* set of code that uses `c()` to *recreate* `z` each iteration of the loop:

```
x <- runif(100000)
y <- runif(100000)
z <- NULL                   # do not preallocate the elements of z
system.time(
    for (i in 1:100000) {
        z <- c(z, x[i] + y[i])  # R has to recreate z entirely
                                # each iteration
    }
```

```
)

##    user  system elapsed
##  22.784   0.636  23.418
```

2. An alternative approach is to *redimension* `z` by increasing its length one element each iteration, but this too is *inefficient* because it turns out that redimensioning a vector takes as much time as recreating it altogether:

```
z <- NULL                    # do not preallocate the elements of z
system.time(
    for (i in 1:100000) {
        z[i] <- x[i] + y[i] # R has to redimension z (change
    }                        # its length) each iteration
)

##    user  system elapsed
##   0.036   0.004   0.038
```

3. Now watch how much faster the code is when we *preallocate* space for the elements of `z` before entering the loop:

```
z <- rep(NA, 100000)       # Now preallocate 100,000 elements of z,
                           # assigning each the value NA
system.time(
    for (i in 1:100000) {
        z[i] <- x[i] + y[i] # R only has to assign to a single
    }                        # element of z, with no redimensioning
)

##    user  system elapsed
##   0.012   0.000   0.014
```

We see that preallocation speeds the code up considerably.

2.4 Using Vectorization and Avoiding Loops

- Recall that many of R's built-in functions (`sqrt()`, `abs()`, etc.) and operators ('`+`', '`-`', '`*`', '`/`', '`^`', etc.) are *vectorized*.
- It's usually *much* faster to use *vectorization* than to perform the same task using a loop.

- For example, below we time two sets of code that perform the same task, adding the elements of two vectors `x` and `y` together. The first uses a loop and the second the vectorized property of the '+' operator:

```
x <- runif(100000)
y <- runif(100000)
z <- rep(NA, 100000)
```

1. Here we use a loop, which is *inefficient*:

```
system.time(
    for(i in 1:length(x)) {
        z[i] <- x[i] + y[i]
    }
)

##    user  system elapsed
##   0.012   0.000   0.013
```

2. Now we use vectorization to speed up the code:

```
system.time(
    z <- x + y
)

##    user  system elapsed
##   0.004   0.000   0.000
```

   Note that the vectorized code was *much* faster than the loop.

- The reason why vectorization is (usually) faster than looping is that vectorized computations are carried out behind the scenes in the C language, which is much faster than R. (Recall that underlying R is a suite of C functions that R invokes when it needs them).

  More precisely, vectorization usually involves fewer R function calls, which as noted can be slow. For example, although it may not look like it, in the loop above, the '+' operation actually involves making 100,000 calls to a function `"+"()`:

```
"+"(2, 3)

## [1] 5
```

Using vectorization, the `"+"()` function is only called once, at which point all further computational tasks are passed to the C language.

2.5 Bytecode Compilation

- A ***bytecode compiler*** translates a so-called ***high-level*** language like R, which is closer to human spoken language and therefore easier to understand but relatively slow, into a ***low-level*** language called ***bytecode***, which is closer to the computer's ***machine instruction language*** and therefore harder to understand but much faster.

- R comes equipped with its own bytecode compiler, in the built-in package `compile`, which we can use to try to speed up our code. We'll look at one function from the `compile` package:

```
cmpfun() # Translate (compile) a function from R code to bytecode
```

- Here's an example of how `cmpfun()` is used to speed up the first example of Subsection 2.4:

```
x <- runif(100000)
y <- runif(100000)
z <- rep(NA, 100000)
library(compiler)     # The 'compiler' package comes built-in with R
f <- function() {
    for(i in 1:length(x)) {
        z[i] <<- x[i] + y[i]    # Note the use of '<<-'
    }
}
```

```
cf <- cmpfun(f)
```

```
system.time(cf())
```

```
##    user  system elapsed
##    0.02    0.00    0.02
```

We see by comparison that the compiled code runs faster than the uncompiled code in the first example of Subsection 2.4. (Note, though, that using bytecode still isn't as fast as using vectorization.)