Programmer's Introduction to R (V1.1)

Stephyn G. W. Butcher

October 5, 2014

This is a programmer oriented quick introduction to R. There are a few things that programmers want to know about a new language:

1. Basic Data Types
2. Data Structures
3. Loops and Conditionals
4. Functions
5. Libraries
6. Data Frames
7. Importing Data

Once we have discussed these, we will get into the more statistics oriented aspects of R.

Basic Data Types

From a programming point of view, there are really only four basic data types in R: double and character:

```
typeof(1)
```

```
## [1] "double"
```

```
typeof("a")
```

```
## [1] "character"
```

```
typeof(FALSE)
```

```
## [1] "logical"
```

Variables and Assignment

R is a dynamically typed language. Assignment is accomplished using the <- operator. The = will also work but not in all cases. The common practice is to use <-:

```
a <- 1
a
```

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

```
b <- "Hello"
b
```

```
## [1] "Hello"
```

```
a <- TRUE
a
```

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

Variables are named using the usual naming conventions except for one that may throw you for a (infinite) loop:

```
my.odd <- 3
my.odd
```

```
## [1] 3
```

```
my.even <- 4
my.even
```

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

`.` is not an operator in R... it is just a character that may legally appear in a variable name. This means that "my" is *not* an object and it is not a namespace. It's just a name and `my.odd` and `my.even` have nothing in common with each other except whatever semantics you wish to add. A good example of this is the set of functions for reading files that all start with `read.`. Good times.

But there is something a bit more mysterious going on. Why is the output each time preceded by "[1]"? The answer takes us directly into data structures:

Vectors

Although 1 and "a" *look* like scalars, they are, in fact, vectors. (Almost) everything is a vector in R with indices starting at 1 (not 0):

```
x <- c(1, 2, 3)
x[1]
```

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

```
x[0]
```

```
## numeric(0)
```

We note several things here starting with the last output. Accessing a vector using index 0 is not an error but it is wrong. Additionally, note that even accessing the first element of a vector `x[1]` returns an vector. Weird. So what *is* `c()`? `c()` is the concatenate function.

This might now make a bit more sense. `1`, `2` and `3` are themselves vectors and when you want to make a larger vector of out of them, you concatenate them. What will the following do?

```
x <- c(c(1, 2, 3), c(4, 5, 6))
x
```

```
## [1] 1 2 3 4 5 6
```

The display of larger vectors makes purpose of the numbers in square brackets even more apparent:

```
seq(0, 100, 1)
```

```
##   [1]   0   1   2   3   4   5   6   7   8   9
##  [11]  10  11  12  13  14  15  16  17  18  19
##  [21]  20  21  22  23  24  25  26  27  28  29
##  [31]  30  31  32  33  34  35  36  37  38  39
##  [41]  40  41  42  43  44  45  46  47  48  49
##  [51]  50  51  52  53  54  55  56  57  58  59
##  [61]  60  61  62  63  64  65  66  67  68  69
##  [71]  70  71  72  73  74  75  76  77  78  79
##  [81]  80  81  82  83  84  85  86  87  88  89
##  [91]  90  91  92  93  94  95  96  97  98  99
## [101] 100
```

The number in square brackets indicates the index of the initial element displayed on that row. Sweet.

Unlike many other languages, what we usually call `String` is not a vector of `Characters`:

```
c("Hello", "there", "data", "science")
```

```
## [1] "Hello"   "there"   "data"    "science"
```

```
"Hello"[1]
```

```
## [1] "Hello"
```

`"Hello"` is just a `Character` type. The concatenation of strings (or appending) uses the varg function `paste()`:

```
paste("hello", "there")
```

```
## [1] "hello there"
```

Matrices

Matrices are multi-dimensional vectors which can be created from vectors using two functions: `rbind` which is short for *row bind* and `cbind` which is short for *column bind*. Note the new notation in display. This is also how you access rows, columns and elements:

```
m1 <- rbind(c(1, 2, 3), c(4, 5, 6))
m1
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
m1[1, ]
```

```
## [1] 1 2 3
```

```
m1[, 3]
```

```
## [1] 3 6
```

```
m2 <- cbind(c(1, 2, 3), c(4, 5, 6))
m2
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
m2[, 1]
```

```
## [1] 1 2 3
```

```
m2[3, ]
```

```
## [1] 3 6
```

Lists

Strangely, what is called a Hash, Hashmap, Struct or Dictionary in other languages is called a List in R. That's just wrong. Elements in a list are almost always accessed using the `$` as opposed to square brackets:

```
x <- list(title = "Data Science", students = 16)
x$title
```

```
## [1] "Data Science"
```

```
x$students
```

```
## [1] 16
```

```
x["students"]
```

```
## $students
## [1] 16
```

Note that you **can** use square brackets and this is often necessary in the case of programmatic access or if the key generate from some external data source such that it might included embedded spaces.

Loops and Conditionals

For any language to be Turing complete, it must have repetition and conditionals. R is Turing complete. The main looping structure is `for` and `if` is the conditional expression. As we saw before, the boolean literals are `TRUE` and `FALSE`. We can write a function that adds the elements of two vectors if the element from the first vector is less then the element of the second vector but multiply them otherwise.

```
x <- c(seq(1, 5), seq(15, 20))
x
```

```
##  [1]  1  2  3  4  5 15 16 17 18 19 20
```

```
y <- seq(5, 15)
y
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

```
z <- c()
for (i in 1:length(x)) {
    if (x[i] < y[i]) {
        z[i] <- x[i] + y[i]
    } else {
        z[i] <- x[i] * y[i]
    }
}
z
```

```
##  [1]   6   8  10  12  14 150 176 204 234 266
## [11] 300
```

R has Lisp underpinnings so everything is an expression (this is true of Ruby as well) so we can write this as:

```
x <- c(seq(1, 5), seq(15, 20))
x
```

```
##  [1]  1  2  3  4  5 15 16 17 18 19 20
```

```
y <- seq(5, 15)
y
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

```
z <- c()
for (i in 1:length(x)) {
    z[i] <- if (x[i] < y[i]) {
```

```
        x[i] + y[i]
    } else {
        x[i] * y[i]
    }
}
z
```

```
##  [1]   6   8  10  12  14 150 176 204 234 266
## [11] 300
```

Note, however, that loops are used far less often in R than in other programming languages precisely because the language is vector-based. For example, compare the following:

```
x <- seq(1, 10)
for (i in 1:length(x)) {
    x[i] <- x[i] + 1
}
x
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

which uses an explicit loop to:

```
x <- seq(1, 10)
x <- x + 1
x
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

compared to:

```
x <- seq(1, 11)
y <- seq(5, 15)
z <- x + y
z
```

```
##  [1]  6  8 10 12 14 16 18 20 22 24 26
```

These automatic vector operations come at a price because if the lengths are differ, R will automatically recycle the shorter vector. This is what it does in the `x <- x + 1` case. Remember that `1` is a vector of length 1 and that can be cycled through `x`.

Suppose instead we have:

```
x <- seq(1, 10)
y <- c(10, 20)
z <- x + y
z
```

```
##  [1] 11 22 13 24 15 26 17 28 19 30
```

This is the same as adding `1 + 10` then `2 + 20` then `3 + 10` then `4 + 20`, etc. If the length of the larger is not a multiple of the shorter sometimes a warning is given and sometimes it works without an error:

```
x <- seq(1, 10)
y <- seq(10, 20, 30)
z <- x + y
z
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

Functions

Well organized code or even code that can be reused requires that we be able to define functions. We define functions as follows:

```
my.addition <- function(x, y) {
    return(x + y)
}
my.addition(1, 2)
```

```
## [1] 3
```

Note that unlike some expression based languages, the `return` expression is required. Multiple return values can be affected by using a List (remember that List is R's Dictionary/Hash). You may also describe default values:

```
my.and <- function(x, y = TRUE) {
    return(x & y)
}
my.and(TRUE)
```

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

```
my.and(TRUE, FALSE)
```

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

In many cases, your function can be applied to vectors for free:

```
my.and(c(TRUE, TRUE, FALSE))
```

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

Libraries

Libraries are loaded using their name:

```
library(datasets)
```

Strangely, quotes are not required here. To find out what packages are installed on your system, use the `library()` function:

```
library()
```

Finally, to install a library that you want to use, use `install.packages()`:

```
# install.packages( 'ggplot2')
```

in which case, parentheses are required.

Data Frames

R is a statistical programming language. Although we discussed the basic, raw data structures, the the central data structure for all of the statistical packages is usually the Data Frame.

Data Frames

Althought Data Frame is just a List really (remember that a List in R is a Hash table), the object oriented features of R allow a list that specifies a class name entry to define functions that treat this list is special ways. This is essentially the way all OOP works.

We can create a Data Frame from regulary Vectors. The constructor function will return a Data Frame form those vectors.

```
n = c(13.4, 23.1, 33.9, 23.7, 33.1, 41.7, 56.2)
o = c("low", "middle", "high", "middle", "high",
    "low", "low")
s = c("35+", "<35", "35+", "35+", "<35", "35+",
    "35+")
b = c(TRUE, FALSE, TRUE, FALSE, FALSE, FALSE,
    TRUE)

df = data.frame(n, o, s, b)

df
```

```
##      n      o   s     b
## 1 13.4    low 35+  TRUE
## 2 23.1 middle <35 FALSE
```

```
## 3 33.9   high 35+  TRUE
## 4 23.7 middle 35+ FALSE
## 5 33.1   high <35 FALSE
## 6 41.7    low 35+ FALSE
## 7 56.2    low 35+  TRUE
```

It is usually the case that we do not want to print out the entire data frame. Instead we can use `head` and `tail`:

```
head(df)
```

```
##      n      o   s     b
## 1 13.4    low 35+  TRUE
## 2 23.1 middle <35 FALSE
## 3 33.9   high 35+  TRUE
## 4 23.7 middle 35+ FALSE
## 5 33.1   high <35 FALSE
## 6 41.7    low 35+ FALSE
```

```
head(df, 3)
```

```
##      n      o   s     b
## 1 13.4    low 35+  TRUE
## 2 23.1 middle <35 FALSE
## 3 33.9   high 35+  TRUE
```

The names of the columns will be the names of the variables:

```
colnames(df)
```

```
## [1] "n" "o" "s" "b"
```

But this may not be what we want or if we load/import data (described later), there may be no indication of what the variables should be called. In that case, we can change the names of the variables. The syntax looks odd, though…it doesn't seem like assigning a value to the return of a function should work. It's actually a different function `colnames()<-` instead of `colnames()`:

```
colnames(df) <- c("Earnings", "Satisfaction",
    "Employment", "Male")
df
```

```
##   Earnings Satisfaction Employment  Male
## 1     13.4          low        35+  TRUE
## 2     23.1       middle        <35 FALSE
## 3     33.9         high        35+  TRUE
## 4     23.7       middle        35+ FALSE
```

```
## 5     33.1         high        <35 FALSE
## 6     41.7          low        35+ FALSE
## 7     56.2          low        35+  TRUE
```

Because R is polymoprhic and functional, many functions and operations that work for Lists, Vectors and Matrices, also work for Data Frames. For example, selecting a row is the same as for Matrices:

```
df[1, ]
```

```
##   Earnings Satisfaction Employment Male
## 1     13.4          low        35+ TRUE
```

as is selecting a column:

```
df[, 1]
```

```
## [1] 13.4 23.1 33.9 23.7 33.1 41.7 56.2
```

However, because selecting a column is so common, it can be done without the comma:

```
df[1]
```

```
##   Earnings
## 1     13.4
## 2     23.1
## 3     33.9
## 4     23.7
## 5     33.1
## 6     41.7
## 7     56.2
```

You can also select columns by names:

```
df["Earnings"]
```

```
##   Earnings
## 1     13.4
## 2     23.1
## 3     33.9
## 4     23.7
## 5     33.1
## 6     41.7
## 7     56.2
```

or

```
df$Earnings
```

```
## [1] 13.4 23.1 33.9 23.7 33.1 41.7 56.2
```

Notice, however, that they print out differently. This last type of access is often easiest and a compelling reason to *not* name your variables anything that contain an interior space.

As with Matrics and Vectors, we can use a Vector of Indices to subset a Data Frame, either its columns:

```
df[c("Earnings", "Male")]
```

```
##   Earnings  Male
## 1     13.4  TRUE
## 2     23.1 FALSE
## 3     33.9  TRUE
## 4     23.7 FALSE
## 5     33.1 FALSE
## 6     41.7 FALSE
## 7     56.2  TRUE
```

or rows:

```
df_odd <- df[c(1, 3, 5, 7), ]
df_odd
```

```
##   Earnings Satisfaction Employment  Male
## 1     13.4          low        35+  TRUE
## 3     33.9         high        35+  TRUE
## 5     33.1         high        <35 FALSE
## 7     56.2          low        35+  TRUE
```

Note that the comma is required for rows. However, it's unlikely that we want to work with only the odd rows. We might want to look at only the low satisfaction rows. We need to 1) create an indexing vector and 2) use it on the data frame:

```
low_index <- df$Satisfaction == "low"
low_index
```

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

```
df[low_index, ]
```

```
##   Earnings Satisfaction Employment  Male
## 1     13.4          low        35+  TRUE
## 6     41.7          low        35+ FALSE
## 7     56.2          low        35+  TRUE
```

Apply Functions to Data Frames

While Vector and Matrix operations are naturally defined in R, operations on Data Frames are not. If you want to apply a function to the rows or columns of a Data Frame, you'll need to use one of the special functions, `Xapply` where X can be "l", "s", "v", "m" or "t" depending on exactly what you want. These are not actually special to Data Frames, you can use them for other Lists and Vectors as well.

For example, we can find out what *type* each column of our Data Frame is by using `sapply` and `typeof`:

```
sapply(df, typeof)
```

```
##     Earnings Satisfaction   Employment
##     "double"    "integer"    "integer"
##         Male
##    "logical"
```

Whoa... that's a bit surprising. The variable *Satisfaction* is listed as an *integer*. We'll return to that in a minute.

```
sapply(df, mean)
```

```
## Warning: argument is not numeric or logical: returning NA
## Warning: argument is not numeric or logical: returning NA
```

```
##     Earnings Satisfaction   Employment
##      32.1571           NA           NA
##         Male
##       0.4286
```

That gives us some warnings, NA's and weirdness as well. We can take the mean of earnings but apparently not the variables Satisfaction or Employment. Oddly, we *can* take the mean of the boolean variable, *Male*. Wacky.

Classes

What is that "integer" business all about?

We talked about types in R before but there are different ways of looking at the idea of types in R. There are *types*, there are *modes* and there are *classes*. The types and modes are like things we've discussed before and we can use `sapply` to see what the types and modes of our data are:

```
sapply(df, typeof)
```

```
##     Earnings Satisfaction   Employment
##     "double"    "integer"    "integer"
##         Male
##    "logical"
```

```
sapply(df, mode)
```

```
##     Earnings Satisfaction   Employment
##    "numeric"    "numeric"    "numeric"
##         Male
##    "logical"
```

As we saw, not all operations can be performed on each type or mode. There is however, a more abstract taxonomy of variables used in R with respect to statistical analyses and that is *class*:

```
sapply(df, class)
```

```
##     Earnings Satisfaction   Employment
##    "numeric"     "factor"     "factor"
##         Male
##    "logical"
```

There are three classes in R: numeric, factor and logical. Numeric and logical are the same as before but *factor* is different. Factor is an old statistical name for categorical variables and each value of a factor is a *level*. In fact, if we print out those two variables using the `$` notation, it will not only show the *apparent* values but the possibilities, levels, as well:

```
df$Satisfaction
```

```
## [1] low    middle high   middle high   low
## [7] low
## Levels: high low middle
```

```
df$Employment
```

```
## [1] 35+ <35 35+ 35+ <35 35+ 35+
## Levels: <35 35+
```

If we have a huge data set, this might not be feasible. We can ask for the levels of a variable directly:

```
levels(df$Satisfaction)
```

```
## [1] "high"   "low"    "middle"
```

Sometimes the level *names* are not what we want. For example, suppose that we want to change the names of the Employment levels to "part" and "full"-time:

```
levels(df$Employment)
```

```
## [1] "<35" "35+"
```

```
levels(df$Employment) <- c("part", "full")
df$Employment
```

```
## [1] full part full full part full full
## Levels: part full
```

Note that because these are treated as factors, the levels have no intrinsic order... unless you tell R that they do. For example,

```
levels(df$Satisfaction)
```

```
## [1] "high"   "low"    "middle"
```

Note that the ordering is "high", "middle", "low". If we were to graph anything using these factors, this is the order they would appear in the graphs. That's not particularly helpful. R does have the notion of *ordered* factors. We just need to tell R that Satisfaction is ordered. We can accomplish this using the `factor` function:

```
df$Satisfaction <- factor(df$Satisfaction, levels = c("low",
    "middle", "high"), ordered = TRUE)
df$Satisfaction
```

```
## [1] low    middle high   middle high   low
## [7] low
## Levels: low < middle < high
```

Notice that the when the levels are described there are "<" signs to indicate the intended order for the factors. If we ask for the class of Satisfaction, we get:

```
class(df$Satisfaction)
```

```
## [1] "ordered" "factor"
```

"ordered factor" to indicate that the factor is indeed ordered.

Converting Classes

R will not always do the right thing when loading/importing data. Sometimes what you intend to be numeric will be interpreted as a factor (because it is all integers) or vice versa and so it is handy to know how to convert these.

Suppose we have some kind of Likert Scaled variable and it is interpreted as a factor. We may really want to interpret this as a number. We can use `as.numeric`. If we want, we can re-assign the numeric version into the same or a different variable in the Data Frame.

```
f <- factor(c(1, 2, 3, 4, 5, 5, 3, 4, 1))
mean(f)
```

```
## Warning: argument is not numeric or logical:
## returning NA
```

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

```
mean(as.numeric(f))
```

```
## [1] 3.111
```

We can also go the other way:

```
f <- c(1, 2, 3, 4, 5, 5, 3, 4, 1)
mean(f)
```

```
## [1] 3.111
```

```
as.factor(f)
```

```
## [1] 1 2 3 4 5 5 3 4 1
## Levels: 1 2 3 4 5
```

```
mean(as.factor(f))
```

```
## Warning: argument is not numeric or logical:
## returning NA
```

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

It can get a lot more complicated, however, and you should consult the internet for your particular problem.

Importing Data

Creating a data frame from scratch is a pain. What we really want to do import data. We will examine important the basic data imports.

Importing CSV/TSV

There's really nothing to importing a CSV or TSV. For the moment, we are assuming that the file or files are on the local storage system but you can easily acquire data using HTTP or other means using R.

You need to make sure the data file is available to your program by either specifying an appropriate absolute path or a relative path to your working directory or set the current working directory with `setwd`.

```
df <- read.csv("data1.csv")
df
```

```
##   Earnings Satisfaction Employment  Male
## 1     13.4          low       full  TRUE
## 2     23.1       middle       part FALSE
## 3     33.9         high       full  TRUE
## 4     23.7       middle       full FALSE
## 5     33.1         high       part FALSE
## 6     41.7          low       full FALSE
## 7     56.2          low       full  TRUE
```

```
sapply(df, class)
```

```
##     Earnings Satisfaction   Employment
##    "numeric"     "factor"     "factor"
##         Male
##    "logical"
```

Our file has headers. What if the file doesn't have headers?

```
df <- read.csv("data2.csv")
df
```

```
##   X13.4    low full TRUE.
## 1  23.1 middle part FALSE
## 2  33.9   high full  TRUE
## 3  23.7 middle full FALSE
## 4  33.1   high part FALSE
## 5  41.7    low full FALSE
## 6  56.2    low full  TRUE
```

```
sapply(df, class)
```

```
##     X13.4       low      full     TRUE.
## "numeric"  "factor"  "factor" "logical"
```

That's bad. We need to tell `read.csv` that our file has no headers:

```
df <- read.csv("data2.csv", header = FALSE)
df
```

```
##     V1     V2   V3    V4
## 1 13.4    low full  TRUE
## 2 23.1 middle part FALSE
## 3 33.9   high full  TRUE
## 4 23.7 middle full FALSE
## 5 33.1   high part FALSE
## 6 41.7    low full FALSE
## 7 56.2    low full  TRUE
```

```
sapply(df, class)
```

```
##        V1        V2        V3        V4
## "numeric"  "factor"  "factor" "logical"
```

Better but the variables (columns) will have generic names. You can either set these ahead of time in read.csv or you can assignment them later as described previously.

It is worth looking the documentation for `read.csv` because the function has many options that can be set that will take care of most of your data import needs. For example, suppose your data set uses "?" to indicate "NA"... you can tell `read.csv` that by setting `na.strings`. Is your file tab separated? use `sep` to indicate that the separate is a tab instead of a comma.

Use `?read.csv` to find the help for `read.csv`

More Complicated Data

There are ways to make R read from databases (MySQL, SQLite, PostegreSQL, etc), from Hadoop, from Excel. These data sources are well behind this short introduction but knowing that something can be done is half the battle to googling it (the other half is finding the right combination of search terms for the particular domain/application).