Learning R

Gavin L. Simpson

February, 2017

Introduction to R

Why R?

- R was designed from the ground up as a language for data analysis
- It is free and open source, and available on all the major OSes
- Huge package ecosystem (> 10,000) covering bewildering array of statistical methods, data visualisations, data import and manipulation
- Cutting edge; R is used by thousands of statisticians & R code or a package often accompanies papers developing new methods
- For the most part, a great community providing help, blog posts etc

What is R?

- The S statistical language was started at Bell Labs on May 5, 1976
- A system for general data analysis jobs that could replace the *ad hoc* creation of Fortran applications
- The S language was licensed by Insightful Corporation for use in their *S-PLUS* software
- In 2004 Insightful bought the S language from Lucent (formerly AT&T and before that Bell Labs)
- Robert Gentleman and Ross Ihaka designed a language that was compatible with S but which worked in a different way internally
- They called this language R
- There was a lot of interest in R and eventually it was made Open Source under the GNU GPL-2
- R has drawn around it a group of dedicated stewards of the R software — R Core

R on the web

- The R homepage is located at: www.r-project.org
- The download site is called CRAN — the **Comprehensive R Archive Network**
- CRAN is a series of mirrored web servers to spread the load of thousands of users downloading R and associated packages
- The CRAN master is at: cran.r-project.org

Starting R and other preliminaries

- You start R in a variety of ways depending on your OS
- R starts in a **working directory** where it looks for files and saves objects
- Best to run R in a new directory for each project or analysis task
- `getwd()` and `setwd()` get and set the working directory
- To exit R, the function `q()` is used
- You will be asked if you want to save your workspace; invariably you should answer `n` to this

Getting help

- R comes with a lot of documentation
- To get help on functions or concepts within R, use the `"?"` operator
- For help on the `getwd()` function use: `?getwd`
- Function `help.search("foo")` will search through all packages installed for help pages with `"foo"` in them
- How the help is displayed is system dependent
- Google is your friend
- StackOverflow's R tag: stackoverflow.com/questions/tagged/r

Working with R & entering commands

- Type commands at prompt `">"` and these are evaluated when you hit RETURN
- If a line is not syntactically complete, the prompt is changed to `"+"`
- If returned object not assigned, it is printed to console
- Assigning the results of a function call achieved by the assignment operator `"<-"`
- Whatever is on the right of `"<-"` is assigned to the object named on the left of `"<-"`
- Enter the name of an object and hit RETURN to print the contents
- `ls()` returns a list of objects currently in your workspace

Working with R & entering commands

```
> 5 * 3
```

```
[1] 15
```

```
> radius <- 5
> pi * radius^2
```

```
[1] 78.53982
```

```
> ans <- 5 * 3
> ans
```

```
[1] 15
```

```
> ans2 <- ans + 20
> ans2
```

```
[1] 35
```

```
> ls()
```

```
[1] "ans"    "ans2"   "radius"
```

Data structures

The basic data structures

Following Hadley Wickham's *Advanced R*, the basic data structures in R can be classified according to

1. whether their contents are all the same (homogeneous) or not,
2. the dimensionality of the object

Homo	geneous Hete	rogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	NA

The best way to understand which data structure an R object is, is to use `str()`

Vectors

Vectors are the basic type of data object in R and there are two main types

1. *atomic* vectors
2. lists

Each with three common properties

- What type of vector the object is: `typeof()`
- Its length: `length()`
- Attributes, which are extra information or metadata: `attributes()`, `attr()`

Atomic vectors

Atomic vectors differ fundamentally from lists because atomic vectors can only contain elements that are *all of the same type*

The four main types of atomic vector are

1. logical
2. integer
3. double
4. character

Two other types are less often encountered; raw and complex vectors

Atomic vectors

We create atomic vector with `c()`, the *concatenate* or *combine* function

```
> dbl <- c(1, 2, 3)
> int <- c(1L, 2L, 3L)
> logi <- c(TRUE, FALSE, TRUE)            # Avoid using c(T, F, T)
> chr <- c("Hello", "World")
```

If an element (observation) is missing, use `NA`

Atomic vectors — coercion

All elements of atomic vectors must be of the same type. If you mix types or attempt to combine atomic vectors of different types, they will be *coerced* towards the most general type.
The ordering is (from least to most general)

1. logical
2. integer
3. double
4. character

```
> str(c("a", 1))
 chr [1:2] "a" "1"
> str(c(TRUE, 10))
 num [1:2] 1 10
> str(c(10L, 2))
 num [1:2] 10 2
```

Atomic vectors — coercion

One handy coercion is the conversion of logical vectors to numeric (integer or double).

A `TRUE` is `1`, whilst a `FALSE` is `0`, which allows them to be used in numeric mathematical operations

Coercion happens atomically, but you can control this by explicitly coercing to the required type with

- `as.characer()`
- `as.double()`
- `as.integer()`
- `as.logical()`

```
> x <- c(FALSE, FALSE, TRUE)
> as.numeric(x)

[1] 0 0 1

> sum(x)                                    # count up the TRUEs

[1] 1

> mean(x)                                   # proportion of TRUE

[1] 0.3333333
```

Lists

Lists are exceedingly common in R — it's often how fitted statistical models are stored

Lists are *general* vectors because they elements they contain can be of any type — lists can even contain lists

```
> x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
> str(x)
```

```
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
```

```
> x <- list(list(list(list)))
> str(x)
```

```
List of 1
 $ :List of 1
  ..$ :List of 1
  .. ..$ :function (...)
```

Factors

Factors are useful for storing data where observations can take on one of a finite set of values

Factors combine integer vectors with attributes to create a new data structure

They have `class()` `"factor"` and a `levels()` attribute which lists the set of values the elements can take

```
> x <- factor(c("x","y","z","z","x","z","y"))
> x
```

```
[1] x y z z x z y
Levels: x y z
```

```
> class(x)
```

```
[1] "factor"
```

```
> levels(x)
```

```
[1] "x" "y" "z"
```

```
> x[2] <- "c"                             # throws a warning
```

```
Warning in `[<-.factor`(`*tmp*`, 2, value = "c"): invalid factor level, NA
generated
```

```
> x
```

```
[1] x    <NA> z    z    x    z    y
Levels: x y z
```

```
> x <- factor(c("x","y","z","z"), levels = c("z","y","x")) # specify levels
```

Matrices & arrays

What separates atomic vectors from arrays is the presence of a `dim()` attribute

As arrays are really atomic vectors with this extra attribute, they can only contain elements of a single type

Matrices are a special case of a multi-dimensional array, where there are only two dimensions

```
> m <- matrix(1:6, ncol = 3, nrow = 2)
> m
```

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

```
> m <- matrix(1:6, c(3,2))
>
> m <- 1:6
> dim(m) <- c(3,2)
> dim(m) <- c(2,3)
> m
```

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

Data frames

Data frames are a bit like Excel worksheets for R; they are like 2-d matrices with the exception that the columns can store different types of objects

Internally, data frames are lists, with the extra restriction that each component (column) of the data frame has to be of the same length

Data frame can be created using `data.frame()`

```
> df <- data.frame(x = 1:3, y = c("a", "b", "c"))
> str(df)
```

```
'data.frame':   3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

```
> df <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)
> is.data.frame(df)
```

```
[1] TRUE
```

```
> nrow(df)
```

```
[1] 3
```

```
> class(df)
```

```
[1] "data.frame"
```

Data frames II

Additional columns and rows can be added to a data frame using `cbind()` and `rbind()` respectively

```
> cbind(df, data.frame(z = 3:1))
```

```
  x y z
1 1 a 3
2 2 b 2
3 3 c 1
```

```
> rbind(df, data.frame(x = 10, y = "z"))
```

```
   x y
1  1 a
2  2 b
3  3 c
4 10 z
```

Subsetting

Subsetting is an incredibly useful and powerful way of manipulating data objects — but it is hard to learn initially as there is a lot of detail to remember

Subsetting atomic vectors

The main subsetting function is [— it takes one of several types of arguments which determines how the vector is subset

- *positive integers* return the specified elements
- *negative integers* omit the specified elements from the result
- *logical vectors* return elements where the logical vector is TRUE
- *nothing* returns the original vector
- *zero* returns a zero-length vector

Subsetting a list works the same way with [

```
> x <- c(1.3, 4.5, 2.3, 4.2, 5.4)
> x[c(3, 1)]

[1] 2.3 1.3

> x[-c(3, 1)]

[1] 4.5 4.2 5.4

> x[x > 3]

[1] 4.5 4.2 5.4

> x[c(TRUE, FALSE)]

[1] 1.3 2.3 5.4

> x[]

[1] 1.3 4.5 2.3 4.2 5.4

> x[0]

numeric(0)

> (y <- setNames(x, letters[1:5]))

  a   b   c   d   e
1.3 4.5 2.3 4.2 5.4

> y[c("a", "c")]

  a   c
1.3 2.3
```

Subsetting matrices and arrays

The main subsetting function is [— it takes one of several types of arguments which determines how the array is subset

- a pair of 1-d vectors, one for each dimension
- a single 1-d vector
- a matrix

```
> a <- matrix(1:9, nrow = 3)
> colnames(a) <- LETTERS[1:3]
> a[1:2, ]
```

```
     A B C
[1,] 1 4 7
[2,] 2 5 8
```

```
> a[, 2]
```

```
[1] 4 5 6
```

Subsetting data frames

Again, the main subsetting function is `[` — you can use it to subset a data frame as if it were

- a matrix, with an indexing vector for the rows and the columns
- a list

```
> df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
> df[df$x == 2, ]
```

```
  x y z
2 2 2 b
```

```
> df[c("x", "y")]
```

```
  x y
1 1 3
2 2 2
3 3 1
```

```
> df[, c("x", "y")]
```

```
  x y
1 1 3
2 2 2
3 3 1
```

```
> str(df["x"])
```

```
'data.frame':   3 obs. of  1 variable:
 $ x: int  1 2 3
```

```
> str(df[, "x"])                           # drops the enpty dimension
```

```
 int [1:3] 1 2 3
```

Re-use