Data Science and R Programming

Rob Hayward

June 12, 2014

Introduction

This is an introduction to data science and R programming. There are two Coursera courses. The main clasess can be found Data science toolkit and R Programming.

0.1 Git

Some rules for adding files

- git add . adds all new files
- git add -u updates tracking for files that were changed or deleted
- git add -A all new files and all changes.

If you try to mix different classes in a vector, R will coerce to the lowest common denominator. For example, if you mix numeric and character, you will get two characters; if you try to mix logical and numeric, you will get numeric, if you try to mix logical and character, you will get character.

A list is a special object. It can change class.

```
x <- list(1, "a", TRUE, 1 + 4)
x

## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
```

```
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 5
```

The elements are indexed by double brackets.

0.2 Factors

Factors are integer vectors with a label. The level is determined by alphabetical order. Table can be used on factors. "Unclass" will strip the class and take the factor down to an integer. The levels can be set using "level" argument. Elements of an object can usually be named with "name". Lists and matrices can have names.

0.3 Subsetting

There are a number of ways to sub-set

- [always returns an object of the same class as the original. Can select more than one element.
- [[is used to extract elements of a list or data frame. Can only extract a single element and the class may not be an element or a dataframe.
- $ used to extract named elements of a list or data frame. Same attributes as above.

Subsetting a matrix will usually return a vector. However, a matrix can be returned by setting drop = FALSE. This can be important if taking a column of a matrix. You will get a vector rather than a matrix unless you set drop = FALSE

```
x <- list(foo = 1:4, bar = 0.6)
x[1]

## $foo
## [1] 1 2 3 4

x[[1]]
```

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

x$bar

## [1] 0.6

x[["bar"]]

## [1] 0.6
```

The [[]] can be used with computed indices.

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]]

## [1] 1 2 3 4

x$name

## NULL

x$foo

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

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.18))
x[[c(1, 2)]]

## [1] 12

x[[1]][[3]]

## [1] 14

x[[c(2, 1)]]

## [1] 3.14
```

0.4 reading data

One way to speed up reading of large files is to read a small amount and to then create a vector with the column classes to input into the colClasses section.

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- readtable("datatable.txt", colClasses = classes)
```

Interfaces with the outside world

- file connection to a file
- gzfile connection to a gzip compression
- bzfile connection to a bzip2 compression
- url connection to a web page

0.5 Control Structures

- if, else: testing a condition
- for: executing a loop a fixed number of times
- while: executing a loop *while* a condition is in place
- repeat execute an infinite loop
- breat: break the execution of a loop
- next: skip an iteration of a loop
- return: exit function

Nexted loops are inside each other. These are usually going to be used with things like matrices.

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    print(x[i,j])
  }
}
```

As another example that will keep flipping a coin until the score goes above or below a number.

```
z <- 5
while(z >= 3 && z <=10) {
  print(z)
  coin <- rbinm(1, 1, 0.5)

  if(coin ==1) { ## randomm walk
    z <- z + 1
  } else {
    z <- z -1
  }
}
```

Repeat could be used for running a loop until a certain tolerance is reached. For example,

```
x0 <- 1
tol <- 1e-8

repeat {
    x1 computeEstimate()

    if(abs(x1 - x0) < tol) {
          break
    } else {
          x0 <- x1
    }

}
```

This is used in optimistation algorithms. Usually good to add a limit on the number of repeats. For example a loop.

1 Functions

Create an object of a class function *Lazy Evaluation* means that the arguments of the function are only evaluated as they are needed. If an artgument is not used or evaluated, there is no problem. However, if the function tries to perform some activity with an argument that does not exist (print for example), it will throw an error.

The ... argument is used to repeated already existing argument defaults. For example, to extend a copy of the plot function,

```
myplot <- function(x, y, type = 'l', ...) {
  plot(x, y, type = type, ...)
}
```

They can also be used with generic functions to pass additional arguments to the function. All elements that are named after the ... have to be named explicitly.

1.1 Scoping

How does R assignm values to symbols?

For example,

```
lm <- function{x * x}
lm
```

There is already a function called lm in R. R will search for each of the packages in a list. This can be found using the search function.

```
search()
```

```
##  [1] ".GlobalEnv"        "package:knitr"     "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

This will search the environments. The first search is the Global Environment is the workspace. If you have something in the Global Environment, it will take that first. In this case, it will find the recently defined lm function in the Global Environment and will use that insteady of the one in stats package. The base package is always the last on the list.

When a package is loaded with the library function, that package gets put in the second place. R has separate namespaces for functions and other objects. However, in the global environment, it is only possible to have one object for each name.

Scoping rules deal with the way that values are assigned to variables. Remember that there are variables that are the arguments of the function and other variables. R uses *lexical* or *static* scoping rather than *dynamic* scoping. This is particularly useful for statistical calculations.

For example,

```
f <- function(x, y) {
  x^2 + y/z
}
```

There are two formal arguments x and y but it is not clear where z comes from. This is a free variable that was not defined. Scoping rules define how values are assigned to these *free variables*.

Scoping rules

- The values of the free variables are searched for in the environment in which the variable was defined.
- An environment is a collection of (symbol, value) pairs. For example, x is a symbol and 3.1 may be the value.
- Every environment has a parent. It is possible for an environment to have multiple children.
- The only environment without a parent is the *empty environment*
- A function and an environment creates a closure.
- Therefore, if the function is defined in the global environment, this will be teh first place that a value for the free variable will be sought.
- If this does not work, the next stop is the parent environment. This is the next environment on the search list. If there is no value in the package, the search continues down until the empty environment is reached.
- If there is no value in all the environments, an error is returned.

This becomes important when functions are defined within other functions. Therefore, a fucntion can return a function. In this case the function is defined within another fucntion (not the global environment).

For example,

```
make.power <- function(n) {
    pow <- function(x) {
        x^n
    }
    pow
}
cube <- make.power(3)
```

```
square <- make.power(2)
cube(3)

## [1] 27

square(3)

## [1] 9
```

Here n is a free variable as it is not defined in the pow function. However, it is defined in the make.power function.

To explain the difference between lexical and dynamic scoping, look at the following.

```
y <- 10
f <- function(x) {
    y <- 2
    y^2 + g(x)
}
g(x) <- function(x) {
}

## Error:  could not find function "g<-"

f(3)

## Error:  could not find function "g"
```

In Lexical scoping, the value of y in the function g is looked up in the environment in which the function was defined, in this case the global environment, so the value of y is 10.

With dynamic scoping, the value of y is looked up in the envoronment from which the function was called (sometimes referred to as the *calling environment.*). In that case the value of y would be 2.

if a function is called from the global environment, the global environment and the calling environment are the same so that Lexical scoping can appear to be dynamic scoping.

The consquences of Lexical scoping

- All objects in R must be stored in memory.
- All functions must carry a pointer to their respective defining environment.

1.2 Vectorisation

This can be a way to avoid looping. Vectroising makes things simple. It applies the fuction to each element of a vector. For example,

```
x <- 1:4
y <- 6:9
x + y

## [1]  7  9 11 13

x > 9

## [1] FALSE FALSE FALSE FALSE

y == 8

## [1] FALSE FALSE  TRUE FALSE

x * y

## [1]  6 14 24 36

x/y

## [1] 0.1667 0.2857 0.3750 0.4444
```

For matrix operation

```
x <- 1:4
y <- 6:9
x * y  # element wise multiplication

## [1]  6 14 24 36

x/y  # element wise multiplication

## [1] 0.1667 0.2857 0.3750 0.4444

# True matrix multiplication
x %*% y

##      [,1]
## [1,]   80
```

1.3 Dates and times

Times have two formats

- POSIXct large integer with the date and time
- POSIXlt a list with information on the day of the week, year etc

Generic functions weekday, months, quarters give the day, month name or quarter when applied to a date, POSIXct or POSIXlt function. Things can be moved back an forward between the POSIXct and POSIXlt functions.

```
x <- Sys.time()
x

## [1] "2014-06-12 17:58:48 BST"

p <- as.POSIXlt(x)
names(unclass(p))

## [1] "sec"   "min"   "hour"  "mday"  "mon"   "year"  "wday"  "yday"  "isdst"

p$sec

## [1] 48.7
```

The strptime function will convert dates in character format into date or time objects.

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 09:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x

## [1] "2012-01-10 10:40:00" "2011-12-09 09:10:00"

class(x)

## [1] "POSIXlt" "POSIXt"
```

It is possible to use regular operations on dates and times. However, different classes cannot be mixed. For example,

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34.21", "%d %b %Y %H:%S")
x - y

## Warning:  Incompatible methods ("-.Date", "-.POSIXt") for "-"
## Error:  non-numeric argument to binary operator

x <- as.POSIXlt(x)
x - y

## Time difference of 356.5 days
```

These operators keep track of very difficult things like leap years, leap seconds daylight savings and time zones. For example,

```
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
x - y

## Time difference of 2 days

x <- as.POSIXct(x)
y <- as.POSIXct(y, tz = "GMT")
y - x

## Time difference of -2 days
```

The first shows the leap year and the second shows the difference in timezones.

1.4 Loop functions

There are a number of loop functions in R

- lapply Look over list and evaluate a function for each element.
- sapply Same but simplify the result
- apply Margins of an array
- {tapply Apply over a subset of a vector
- mapply Mulgtivariate version of tapply

Also split is useful.

These make a lot of use of *annonomous functions*. These are functions that do not have names. For example, write a function to extract the first column.

```
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
lapply(x, function(elt) elt[, 1])

## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

sapply will simplify. For example, it may return a vector if there is a list of single objects or it may return a matrix.