# Data Science and R Programming

Rob Hayward

June 11, 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.