Objects, functions and concepts for efficient R programming

Lex Comber

May 2016

Introduction

If you have done a little bit of R then undoubtedly you will have encountered different data structures and you will have used functions, even if you were aware of it or not. As your R expertise progresses, you will probably try to solve more complex problems using, manipulating different data structures. This chapter covers a lot of ground – it will:

- Introduce data types and classes (vectors, lists, matrices, S4, data frames)
- Describe how to test for and manipulate data types
- Describe how to read, write, load and save different data types

Part 1: Basic Ingredients

I am sure you are familiar with assigning values to variables and possibly even to different types of variables. Lets have a look at the `cars` dataset.

```r
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00
```

It is also possible to examine the actual data values within these variables:

```r
class(cars); typeof(cars); class(unlist(cars[[1]])); class(cars[,1])
```

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

## [1] "list"

## [1] "numeric"

## [1] "numeric"
```

So there are a number of different data types and data classes and you should use the R `help` and explore how commonly used classes of variables are defined and created `character`, `vector`, `matrix`, `array`, `list` and `data.frame` class of variables

```
?matrix
```

In the same way there are a number **test** and **conversion** functions related to these data classes that can be used to test for particular data classes and critically to **coerce** data into different classes. The test functions start with `is.` and the conversion functions with `as.`

```
my.var <- cars[,1]
class(my.var)
```

```
## [1] "numeric"
```

```
is.numeric(my.var)
```

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

```
is.character(my.var)
```

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

```
is.logical(my.var)
```

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

```
is.logical(is.vector(my.var))
```

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

The basic or core data **types** and associated tests and conversions are shown in the table below.

Table 1. Core data types, associated tests and conversions

type	test	conversion
character	is.character	as.character
complex	is.complex	as.complex
double	is.double	as.double
expression	is.expression	as.expression
integer	is.integer	as.integer
list	is.list	as.list
logical	is.logical	as.logical
numeric	is.numeric	as.numeric
single	is.single	as.single
raw	is.raw	as.raw

In the same way it is possible to test and coerce data **classes**:

Table 2. Core data classes, associated tests and conversions

class	test	conversion
character	is.character	as.character
logical	is.logical	as.logical
numeric	is.numeric	as.numeric
vector	is.vector	as.vector
matrix	is.matrix	as.matrix
data.frame	is.data.frame	as.data.frame
array	is.array	as.array
factor	is.factor	as.factor

Consider the code below. This creates a vector and then coerces it to a matrix

```
my.var <- c(2000, 1243, 543, 1243, 212, 545, 654, 168, 109)
my.var
```

```
## [1] 2000 1243  543 1243  212  545  654  168  109
```

Now we can apply some tests to `my.var`:

```
# note the use of the ; -  poor practice but saves page space
class(my.var); typeof(my.var); is.vector(my.var); is.numeric(my.var); is.matrix(my.var)
```

```
## [1] "numeric"

## [1] "double"

## [1] TRUE

## [1] TRUE

## [1] FALSE
```

The vector `my.var` can be converted or *coerced* to other formats. The code below creates a matrix of flows between 3 regions from `my.var`.

```
flow <- matrix(my.var, ncol = 3, nrow = 3, byrow=TRUE)
flow
```

```
##      [,1] [,2] [,3]
## [1,] 2000 1243  543
## [2,] 1243  212  545
## [3,]  654  168  109
```

And rows and columns can have names, not just 1,2,3,...

```
colnames(flow) <- c("Leeds", "Liverpool", "Leicester")
rownames(flow) <- c("Leeds", "Liverpool", "Leicester")
flow
```

```
##           Leeds Liverpool Leicester
## Leeds      2000      1243       543
## Liverpool  1243       212       545
## Leicester   654       168       109
```

If you are not familiar with the

Some more specific examples

Factors

The function `factor` creates a vector with specific categories, defined in the levels parameter. The ordering of factor variables can be specified and an ordering function also exists. The functions `as.factor` and `as.ordered` are the coercion functions. The test `is.factor` returns TRUE or FALSE depending on whether their arguments is of factor type or not and `is.ordered` returns `TRUE` when its argument is an ordered factor and `FALSE` otherwise. First, let us examine factors:

```
# a vector assignment
house.type <- c("Bungalow", "Flat", "Flat", "Detached", "Flat", "Terrace", "Terrace")
# a factor assignment
house.type <- factor(c("Bungalow", "Flat", "Flat", "Detached",
  "Flat", "Terrace", "Terrace"),
  levels=c("Bungalow","Flat","Detached","Semi","Terrace"))
house.type
```

```
## [1] Bungalow Flat     Flat     Detached Flat     Terrace  Terrace
## Levels: Bungalow Flat Detached Semi Terrace
```

The function `table` can be used to summarise

```
table(house.type)
```

```
## house.type
## Bungalow     Flat Detached     Semi  Terrace
##        1        3        1        0        2
```

The function `levels` can be used to control what can be assigned

```
house.type <- factor(c("People Carrier", "Flat","Flat", "Hatchback",
  "Flat", "Terrace", "Terrace"),
  levels=c("Bungalow","Flat","Detached","Semi","Terrace"))
house.type
```

```
## [1] <NA>    Flat    Flat    <NA>    Flat    Terrace Terrace
## Levels: Bungalow Flat Detached Semi Terrace
```

Factors are useful for categorical or classified data – that is, data values that must fall into one of a number of predefined classes. It is obvious how this might be relevant to geographical analysis, where many features represented in spatial data are labelled using one of a set of discrete classes. Ordering allows inferences about preference or hierarchy to be made (lower–higher, better–worse, etc.) and this can be used in data selection or indexing (as above) or in the interpretation of derived analyses.

Ordering

There is no concept of ordering in factors. However, this can be imposed by using the ordered function.

```
income <-factor(c("High", "High", "Low", "Low", "Low", "Medium",
  "Low", "Medium"), levels=c("Low", "Medium", "High"))
income > "Low"
```

```
## Warning in Ops.factor(income, "Low"): '>' not meaningful for factors

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

By ordering the factors using `ordered` then logical operations can be performed

```
income <-ordered (c("High", "High", "Low", "Low", "Low", "Medium",
  "Low", "Medium"),levels=c("Low", "Medium", "High"))
income > "Low"
```

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

Thus we can see that ordering is implicit in the way that the levels are specified and allows other, ordering related functions to be applied to the data. The functions `sort` and `table` are helpful functions. In the above code relating to factors, `table` was used to generate a tabulation of the data in `house.type`. It provides a count of the occurrence of each level in `house.type`. The command `sort` orders a vector or factor. You should use the help in R to explore how these functions work and try them with your own variables. For example:

```
sort(income)
```

Lists

The `character`, `numeric` and `logical` data types and the associated data classes described above all contain elements that must all be of the same basic type. Lists do not have this requirement. Lists have slots for different elements and can be considered as *an ordered collection of elements*. A `list` allows you to gather a variety of different data types together in a single data structure and the *nth* element of a 'lista is denoted by double square brackets.

```
tmp.list <- list("Lex Comber",c(2005, 2016),
  "Lecturer", matrix(c(60,40,01,23155789), c(2,2)))
tmp.list
```

```
## [[1]]
## [1] "Lex Comber"
##
## [[2]]
## [1] 2005 2016
##
## [[3]]
## [1] "Lecturer"
##
## [[4]]
##      [,1]     [,2]
## [1,]   60        1
## [2,]   40 23155789
```

```r
# elements of the list can be selected
tmp.list[[4]]
```

```
##      [,1]     [,2]
## [1,]   60        1
## [2,]   40 23155789
```

From the above it is evident that the function `list` returns a list structure composed of its arguments. Each value can be tagged depending on how the argument was specified. The conversion function `as.list` attempts to coerce its argument to a list. It turns a `factor` into a list of one-element factors and drops attributes that are not specified. The test `is.list` returns `TRUE` if and only if its argument is a list. These are best explored through some examples; note that list items can be given names.

```r
employee <- list(name="Lex Comber", start.year = 2015,
  position="Professor")
employee
```

```
## $name
## [1] "Lex Comber"
##
## $start.year
## [1] 2015
##
## $position
## [1] "Professor"
```

Lists can be joined together with `append`, and `lapply` applies a function to each element of a list.

```r
append(tmp.list, list(c(7,6,9,1)))
```

```
## [[1]]
## [1] "Lex Comber"
##
## [[2]]
## [1] 2005 2016
##
## [[3]]
## [1] "Lecturer"
##
## [[4]]
##      [,1]     [,2]
## [1,]   60        1
## [2,]   40 23155789
##
## [[5]]
## [1] 7 6 9 1
```

```r
lapply(tmp.list[[2]], is.numeric)
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] TRUE
```

```
lapply(tmp.list, length)
```

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

Note that the length of a matrix, even when held in a list, is the total number of elements.

Defining your own classes

In R it is possible to define your own data type and to associate it with specific behaviours, such as its own way of printing and plotting / drawing. For example, later the `plot` function is used to draw maps for spatial data objects. This is because a variant of the function `plot` has been defined for the `SpatialPolygon/Point/Line` classes of objects and R applies this function when these objects are passed to it.

To illustrate this, suppose we create a list containing some employee information.

```
employee <- list(name="Lex Comber", start.year = 2015, position="Professor")
```

This can be assigned to a new class, called `staff` in this case (it could be any name but meaningful ones help).

```
class(employee) <- "staff"
```

Then we can define how R treats that class in the form `<existing function>.<class>`. This can be used to change, for example, how objects of this class are printed. Note how the existing function for printing is modified by the new class definition:

```
print.staff <- function(x) {
  cat("Name: ",x$name, "\n")
  cat("Start Year: ",x$start.year, "\n")
  cat("Job Title: ",x$position, "\n")}
```

We can see this print class in action:

```
print(employee)
```

```
## Name:  Lex Comber
## Start Year:  2015
## Job Title:  Professor
```

You can see that R knows to use a different print function if the argument is a variable of class `staff`. You could modify how your R environment treats existing classes in the same way, but do this with caution. You can also 'undo' the class assigned by using `unclass` and the `print.staff` function can be removed permanently by using `rm(print.staff)`:

```
print(unclass(employee))
```

```
## $name
## [1] "Lex Comber"
##
## $start.year
## [1] 2015
##
## $position
## [1] "Professor"
```

Very important point

The ability to define different classes of variable in R is a very important property: many packages or libraries define specific classes. A key example is the package `sp`. This defines a number of class of variables that relate to commonly used spatial data formats. You may have to install this package if you have not used it before: `install.packages("sp", dep = TRUE)`. Some of the more commonly used classes defined in `sp` are listed in the table below.

Table 3 The `sp` spatial data classes

NonAttributed	Attributed	ArcGIS version
SpatialPoints	SpatialPointsDataFrame	Point shapefiles
SpatialLines	SpatialLinesDataFrame	Line shapefiles
SpatialPoints	SpatialPolygonsDataFrame	Polygon shapefiles
SpatialPixels	SpatialPixelsDataFrame	Raster
SpatialGrid	SpatialGridDataFrame	Grid

The `sp` Spatial Data format

You should load the `newhaven` dataset that comes as part of the `GISTools` package and explore the data types that are loaded using the `ls` function and then examine their geographic properties and attributes. You could even plot them!

```
library(GISTools)
data(newhaven)
ls()
```

The variable `blocks` is a `SpatialPolygonsDataFrame` - it has attributes.

```
class(blocks)
head(data.frame(blocks))
head(blocks@data)
summary(blocks)
summary(blocks@data)
```

Whereas `breach` is a `SpatialPoints` object (in this case recording breaches of the peace) without any attributes.

```
summary(breach)
plot(blocks)
plot(breach, add = T, pch = 1, col = "red")
```

Very often we have data that is in a particular format such as `shapefile` format. R has the ability to read and write many different spatial data formats. Consider the `blocks` dataset that was loaded earlier. This can be written out as a shapefile in the following way:

```
writePolyShape(blocks, "blocks.shp", )
```

You will see that a shapefile has been written into your current working directory, with its associated supporting files (`.dbf`, etc) that can be recognised by other applications (QGIS etc). Similarly this can be read into R and assigned to a variable, provided using the `readShapePoly` function in the `maptools` package (loaded with `GISTools`):

```
new.blcoks <- readShapePoly("blocks.shp")
```

You should examine the `readShapeLines`, `readShapePoints`, `readShapePoly` functions and their associated `write` functions. You should also note that R is able to read and write other proprietary spatial data formats, which you should be able to find through a search of the R help system or via an internet search engine.

Self-Test Questions Part 1

Below are a number of self-test questions. In contrast to the previous sections where the code is provided in the text for you to work through (i.e. you to enter and run yourself), the Self-Test Questions are tasks for you to complete, mostly requiring you to write R code. Answers to them are provided later in this document. The self-test questions relate to the main data types that have been introduced: `factors`, `matrices`, `lists` (named and unnamed) and `classes`.

Factors

Recall from the descriptions above that factors are used to represent categorical data - where a small number of categories are used to represent some characteristic in a variable. For example the colour of a particular model of car sold by a showroom in a week can be represented using factors:

```
colours <- factor(c("red","blue","red","white",
    "silver","red","white","silver",
    "red","red","white","silver","silver"),
    levels=c("red","blue","white","silver","black"))
```

Since the only colours this car comes in are red, blue, white, silver and black, these are the only levels in the factor. **Self-Test Question 1** Suppose you were to enter:

```
colours[4] <- "orange"
colours
```

What would you expect to happen? Why?

Next, use the `table` function to see how many of each colour were sold. First re-assign the colours (as you may have altered this variable in the previous self-test question):

```
colours <- factor(c("red","blue","red","white",
    "silver","red","white","silver",
    "red","red","white","silver","silver"),
    levels=c("red","blue","white","silver","black"))
table(colours)
```

```
## colours
##    red   blue  white silver  black
##      5      1      3      4      0
```

Note that the result of the `table` function is just a standard vector, but that each of its elements are named - the names in this case are the levels in the factor. Now suppose you had simply recorded the colours as a character variable, in `colours2` as in the below - and then computed the table:

```
colours2 <-c("red","blue","red","white",
    "silver","red","white","silver",
    "red","red","white","silver")
# Now, make the table
table(colours2)
```

Self-Test Question 2: What two differences do you notice between the results of the two `table` expressions above?

Now suppose we also record the type of car - it comes in saloon, convertible and hatchback. This can be specified by another factor variable called `car.type`:

```
car.type <- factor(c("saloon","saloon","hatchback",
    "saloon","convertible","hatchback","convertible",
    "saloon", "hatchback","saloon", "saloon",
    "saloon","hatchback"),
    levels=c("saloon","hatchback","convertible"))
```

The `table` function can also work with two arguments:

```
table(car.type, colours)
```

This gives a two-way table of counts - that is, counts of red hatchbacks, silver saloons and so on. Note that the output this time is a `matrix`. For now enter:

```
crosstab <- table(car.type,colours)
```

to save the table into a variable called `crosstab` to be used later on.

Self-Test Question 3: What is the difference between `table(car.type,colours)` and `table(colours,car.type)`

Finally in this section, ordered factors will be considered. Suppose a third variable about the cars is the engine size, and that the three sizes are 1.1 litres, 1.3 litres and 1.6 litres. Again, this is stored in a variable, but this time the sizes are ordered. Enter:

```
engine <- ordered(c("1.1litre","1.3litre","1.1litre",
    "1.3litre","1.6litre","1.3litre","1.6litre",
    "1.1litre","1.3litre","1.1litre", "1.1litre",
    "1.3litre","1.3litre"),
    levels=c("1.1litre","1.3litre","1.6litre"))
```

Recall that with `ordered` variables, it is possible to use comparison operators `>`, `<`, `>=`, `==` and `<=`. For example:

```
engine > "1.1litre"
```

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

Self-Test Question 4: Using the `engine`, `car.type` and `colours` variables, write expressions to give the following: - The colours of all cars with engines with capacity greater than 1.1 litres. - The counts of types (i.e. hatchback etc) of all cars with capacity below 1.6 litres. - The counts of colours of all hatchbacks with capacity greater than or equal to 1.3 litres.

Matrices

In the last section that you created a matrix called `crosstab` - and that this was a matrix. In earlier sections a number of functions were shown that could be applied to matrices:

```
dim(crosstab) # Matrix dimensions
rowSums(crosstab) # Row sums
colnames(crosstab) # Column names
```

Another important tool for matrices is the `apply` function. This applies a function to either the rows or columns of a matrix giving a single-dimensional list as a result. A simple example finds the largest value in each row:

```
apply(crosstab,1,max)
```

In this case, the function `max` is applied to each row of `crosstab`. The `1` as the second argument specifies that the function will be applied row by row. If it was `2` then the function would be column by column:

```
apply(crosstab,2,max)
```

A useful function is `which`. Given a list of numbers, it returns the index of the those that meet the test For example:

```
example <- c(1.4,2.6,1.1,1.5,1.2)
which(example== max(example))
```

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

```
which(example > 1.4)
```

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

so that in this case, the second element is the largest. But I hope that you can see how `which` can be used with attributes of spatial data (find the census areas with a deprivation index > 10).

Self-Test Question 5: What happens if there is more than one number taking the largest value in a list? Either use the help facility or experimentation to find out.

Self-Test Question 6: `which.max` can be used in conjunction with `apply`. Write an expression to find the index of the largest value in each row of `crosstab`

There will be some more questions relating to Lists and other classes of variable after we have covered some ground on writing functions.

Answers to self-test questions Part 1

Q1: 'orange' isn't one of the factor's levels, so the result is a `NA`.

```
colours <- factor(c("red","blue","red","white","
    silver","red","white","silver",
    "red","red","white","silver"),
 levels=c("red","blue","white","silver","black"))
colours[4] <- "orange"
```

```
## Warning in `[<-.factor`(`*tmp*`, 4, value = "orange"): invalid factor
## level, NA generated
```

```
colours
```

```
##  [1] red    blue   red    <NA>   <NA>   red    white  silver red    red
## [11] white  silver
## Levels: red blue white silver black
```

Q2: There is no count for 'black' in the character version - `table` doesn't know that this value exits, since there is no 'levels' information. Also the order of colours is alphabetical in the character version. In the factor version, it is based on that specified in the `factor` function.

Q3: The first variable is tabulated along the rows, the second along the columns.

Q4: Colours of all cars with engines with capacity greater than 1.1 litres:

```
# Undo the colour[4] <- 'orange' line used above
colours <- factor(c("red","blue","red","white","
    silver","red","white","silver",
    "red","red","white","silver"),
 levels=c("red","blue","white","silver","black"))
colours[engine > "1.1litre"]
```

```
## [1] blue   white  <NA>   red    white  red    silver <NA>  
## Levels: red blue white silver black
```

Counts of types of all cars with capacity below 1.6 litres:

```
table(car.type[engine < "1.6litre"])
```

```
## 
##      saloon   hatchback convertible 
##           7           4           0
```

Counts of colours of all hatchbacks with capacity greater than or equal to 1.3 litres:

```
table(colours[(engine >= "1.3litre") & (car.type == "hatchback")])
```

```
## 
##    red   blue  white silver  black 
##      2      0      0      0      0
```

Q5: The index returned corresponds to the **first** number taking the largest value.

Q6: An expression to find the index of the largest value in each row of `crosstab` using `which.max` and `apply`.

```
apply(crosstab,1,which.max)
```

```
##      saloon   hatchback convertible 
##           1           1           3
```

Part 2: Functions

So far much of what you have done has been line by line. However, there are occasions when either you want to do things a set number of times, or you want do do something reputedly or you may want to do something until some condition is met. The point is that you will **not** want to write repeated lines of code, that are nearly the same, expect for a small change. This is where functions can help.

The aim of 2nd part of this worksheet is to introduce some basic programming principles and routines that will allow you to do many things repeatedly in single block of code. This is the basics of writing computer programmes. We will:

- Examine how to combine commands into loops
- Control loops in different ways using if, else, repeat, logical operators, etc
- Create functions, test them and to make them universal

Condition Statements

Consider the variable below:

```
student.heights <- c(1.38,1.61,1.67,1.52,1.32, 1.88, 1.94)
```

We may wish to identify whether the fist element has a value *less than 1.6*: this is a *conditional command* as in this case the operation to print something is carried out conditionally (i.e. if the condition is met).

```
student.heights
```

```
## [1] 1.38 1.61 1.67 1.52 1.32 1.88 1.94
```

```
if (student.heights[1] < 1.6) { cat('Student is short\n') } else
  { cat('Student is tall\n')}
```

```
## Student is short
```

Alternatively we may wish to examine all of the elements in the variable `student.heights` and, depending on whether each individual value meets the condition, perform the same operation. We can carry out operations repeatedly using a *loop* structure as below. Notice the construction of the *for* loop in the form `for(variable in sequence) R expression`.

```
for (i in 1:3) {
    if (student.heights[i] < 1.6) { cat('Student',i,' is short\n') }
    else { cat('Student',i,' is tall\n')} }
```

```
## Student 1  is short
## Student 2  is tall
## Student 3  is tall
```

A third situation is where we wish to perform the same set of operations, group of conditional or looped commands over and over again, perhaps to different data. We can do this by grouping code and defining our own *functions*.

```
assess.student.height <- function(student.list, thresh)
  { for (i in 1:length(student.list))
    { if(student.list[i] < thresh) {cat('Student',i, ' is short\n')}
      else { cat('Student',i,' is tall\n')}
    }
  }
assess.student.height(student.heights, 1.6)
```

```
## Student 1  is short
## Student 2  is tall
## Student 3  is tall
## Student 4  is short
## Student 5  is short
## Student 6  is tall
## Student 7  is tall
```

```
student.heights2 <- c(1.8,1.45,1.67,1.24)
assess.student.height(student.heights2, 1.5)
```

```
## Student 1  is tall
## Student 2  is short
## Student 3  is tall
## Student 4  is short
```

Notice how the code in the function `assess.student.height` above modifies the original loop: rather than `for(i in 1:3)` it now uses the length of the variable `1:length(student.list)` to determine how many times to loop through the data. Also a variable `thresh` was used for whatever threshold the user wishes to specify.

Building blocks for Programs

In the examples above a number of programming concepts were introduced. Before we start to develop these more formally into functions it is important to explain these *ingredients* in a bit more detail.

Conditional Statements

Conditional Statements test to see whether some *condition* is `TRUE` or `FALSE` and if the answer is `TRUE` some specific actions are undertaken. Conditional Statements are composed of :

- **if**
- **else**

The `if` statement is followed by a `condition`, an expression that is evaluated, and then a `consequent` to be executed if the condition is `TRUE`. The format of an `if` statement is: `If-Condition-Consequent` Actually this could be read as 'If the condition is true then the consequent is…'. The components of a `conditional statement` are:

- the `Condition`, an R expression that is either `TRUE` or `FALSE`
- the `Consequent`, any valid R statement which is only executed if the `Condition` is `TRUE`

For example, consider the simple case below where the value of `x` is changed and the same condition is applied. The results are different (in the first case a statement is printed to the console, in the second it is not), because of the different values assigned to `x`.

```r
x <- -7
if (x < 0) cat("x is negative")
```

```
## x is negative
```

```r
x <- 8
if (x < 0) cat("x is negative")
```

Frequently `if` statements also have an *Alternative* consequent that is executed when the condition is `FALSE`. Thus the format of the `conditional statement` is expanded to: `If-Condition-Consequent-Else-Alternative` Again, this could be read as *If the condition is true then do the consequent… Or, if the condition is not true then do the alternative.* The components of a `conditional statement` that includes an `alternative` are :

- the `Condition`, an R expression that is either `TRUE` or `FALSE`

- the `Consequent` and `Alternative`, which can be any valid R statements
- the `Consequent` is executed if the `Condition` is `TRUE`
- the `Alternative` is executed if the `Condition` is `FALSE`

The example is expanded below to accommodate the alternative:

```r
x <- -7
if (x < 0) cat("x is negative") else cat("x is positive")
```

```
## x is negative
```

```r
x <- 8
if (x < 0) cat("x is negative") else cat("x is positive")
```

```
## x is positive
```

The `Condition` statement is composed of one or more `Logical operators` and in R these are defined as follows:

Table 4 Logical operators

Logical Operator	Description
==	Equal
! =	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
!	Not (goes in front of other expressions)
&	And (combines expressions)
|	OR (combines expressions)

There are quite a few more `is`-type functions (i.e. `logical` evaluation functions) that return `TRUE` or `FALSE` statements that can be used to develop conditional tests. To explore these enter have a look at the functions beginning with I in the help of the R base package.

The examples below illustrate how the logical tests `all` and `any` may be incorporated into conditional statements:

```r
x <- c(1,3,6,8,9,5)
if (all(x > 0)) cat("All numbers are positive")
```

```
## All numbers are positive
```

```r
x <- c(1,3,6,-8,9,5)
if (any(x > 0)) cat("Some numbers are positive")
```

```
## Some numbers are positive
```

```r
any(x==0)
```

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

Code Blocks: { and }

Frequently we wish to execute a group of `Consequent` statements together if some `Condition` is `True`. Groups of statements are called `code blocks` and in R are contained by `{` and `}` The examples below show how `code blocks` can be used if a to execute `Consequent` statements and can be expanded to execute `Alternative` statements if the `Condition` is `False`.

```r
x <- c(1,3,6,8,9,5)
if (all(x > 0)) {
  cat("All numbers are positive\n")
  total <- sum(x)
  cat("Their sum is ",total) }
```

```
## All numbers are positive
## Their sum is  32
```

The curly brackets are used to group the consequent statements: that is, they contain all of the actions to be performed if the condition is met is `TRUE` and all of the alternative actions if the condition is not met (i.e. is `FALSE`): `if condition { consequents } else { alternatives }`. These are illustrated in the code below:

```r
x <- c(1,3,6,8,9,-5)
if (all(x > 0)) {
  cat("All numbers are positive\n")
  total <- sum(x)
  cat("Their sum is ",total) }  else {
  cat("Not all numbers are positive\n")
  cat("This is probably an error\n")
  cat("as numbers are rainfall levels") }
```

```
## Not all numbers are positive
## This is probably an error
## as numbers are rainfall levels
```

Functions

The introductory section above included a function called `assess.student.height`. The format of a function is: `function name <- function(argument list) {R expression}` The R expression is usually a code block and in R the code is contained by curly brackets or braces: { and }. Wrapping the code into a `function` allows it to be used without having to retype the code each time you wish to use it. Instead, once the function has been defined and compiled, it can be called repeatedly and can be called with different arguments or parameters. Notice in the function below that there are a number offsets of containing brackets { } that are variously related to the `Function`, the `Consequent` and the `Alternative`.

```r
mean.rainfall <- function(rf)
{ if (all(rf> 0))                   #open Function
  { mean.value <- mean(rf)          #open Consequent
    cat("The mean is ",mean.value)
  } else                            #close Consequent
    { cat("Warning: Not all values are positive\n")  #open Alternative
    }                               #close Alternative
  }                                 #close Function
mean.rainfall(c(8.5,9.3,6.5,9.3,9.4))
```

```
## The mean is  8.6
```

More commonly functions are defined that do something to the input specified in the `argument list` and return the result, either to a variable or to the console window, rather than just printing something out. This is done using `return()` within the function. Its format is `return( R expression )`. Essentially what this does if it is used in a function is to make `R expression` the value of the function. In the below the `mean.rainfall` function now returns the mean of the data passed to it and this can be assigned to another variable:

```r
mean.rainfall2 <- function(rf) {
if (all(rf> 0)) {
  return( mean(rf))} else {
  return(NA)}
  }
mr <- mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))
mr
```

```
## [1] 8.6
```

Loops and repetition

Very often, we would like to run a code block a certain number of times, for example for each record in a `data.frame` or a `SpatialPolygonDataFrame`. This is done using `for` loops. The format of a loop is: `{ for( 'loop variable' in 'list of values' ) do R expression}` Again, typically code blocks are used as in the example of a `for` loop:

```r
for (i in 1:5) {
    i.cubed <- i * i * i
    cat("The cube of",i,"is ",i.cubed,"\n")}
```

```
## The cube of 1 is  1
## The cube of 2 is  8
## The cube of 3 is  27
## The cube of 4 is  64
## The cube of 5 is  125
```

When working with a `data.frame` and other tabular like data structures, it common to want to perform a series of R expressions on each row, on each column or on each data element. In a `for` loop the `'list of values'` can be a simple sequence of 1 to n (`1:n`) where n is related to the number of rows or columns in a dataset of the data or the length of the input variable as in the `assess.student.height` function above.

However, there are many other situations when a different `'list of values'` is required. The function `seq` is a very useful helper function that generates number sequences. It has the following format: `seq(from, to, by = step value)` or `seq(from, to, length = sequence length)`. In the example below, it is used to generate a sequence of 0 to 1 in steps of 0.25:

```
for (val in seq(0,1,by=0.25)) {
    val.squared <- val * val
    cat("The square of",val,"is ",val.squared,"\n")}
```

```
## The square of 0 is  0
## The square of 0.25 is  0.0625
## The square of 0.5 is  0.25
## The square of 0.75 is  0.5625
## The square of 1 is  1
```

`Conditional` loops are very useful when you wish to run a code block until a certain condition is met. In R these are specified using the `repeat` and `break` functions:

```
i <- 1
n <- 654
repeat{
  i.squared <- i * i
  if (i.squared > n) break
  i <- i + 1 }
  cat("The first square number exceeding",n, "is ",i.squared,"\n")
```

```
## The first square number exceeding 654 is  676
```

Debugging

As you develop your code and compile it into functions, especially initially, you will probably encounter a few teething problems: hardly any reasonably sized function works first time! There are two general kinds of problem :

- The function crashes (i.e. it throws up an error)
- The function doesn't crash, but returns the wrong answer

Usually the second kind of error is the worst. `Debugging` is the process of finding the problems in the function. A typical approach to debugging is to 'Step' through the function line by line and in so doing find out where a crash occurs, if one does. You should then check the values of variables to see if they have the values they are supposed to. R has tools to help with this.

To debug a function:

- Enter `debug(<<Function Name>>)`
- Then, call the function

For example, enter:

Then just use the function you are trying to debug and R goes into 'debug mode':

```
mean.rainfall2(c(8.5,9.3,6.5,9.3,9.4))
```

```
## debugging in: mean.rainfall2(c(8.5, 9.3, 6.5, 9.3, 9.4))
## debug at <text>#1: {
##     if (all(rf > 0)) {
##         return(mean(rf))
##     }
##     else {
##         return(NA)
##     }
## }
## debug at <text>#2: if (all(rf > 0)) {
##     return(mean(rf))
## } else {
##     return(NA)
## }
## debug at <text>#3: return(mean(rf))
## exiting from: mean.rainfall2(c(8.5, 9.3, 6.5, 9.3, 9.4))
```

```
## [1] 8.6
```

You will notice that the prompt becomes `Browse>` and the line of the function about to be executed is listed. You should note a number of features associated with `debug`:

- entering a return executes it, and debug goes to next line
- typing in a variable lists the value of that variable
- R can ‘see’ variables that are specific to the function
- typing in any other command executes that command

When you enter `c` the return runs to the end of a loop/function/block. Typing in `Q` exits the function.

A final comment is that learning to write functions and programming is a bit like learning to drive - you may ‘pass the test’ but you will become a good driver by spending time behind the wheel. Similarly, the best way to learn to write functions is to **practice** and the more you practice the better you will get at programming. You should try to set yourself various function writing tasks and examine the functions that are introduced throughout this book. Additionally most of the commands that you use in R are functions that can themselves be examined: entering them without any brackets afterwards will reveal the blocks of code they use. Have a look at the `ifelse` function by entering at the R prompt:

```
ifelse
```

```
## function (test, yes, no)
## {
##     if (is.atomic(test)) {
##         if (typeof(test) != "logical")
##             storage.mode(test) <- "logical"
##         if (length(test) == 1 && is.null(attributes(test))) {
##             if (is.na(test))
##                 return(NA)
##             else if (test) {
##                 if (length(yes) == 1 && is.null(attributes(yes)))
##                   return(yes)
##             }
##             else if (length(no) == 1 && is.null(attributes(no)))
##                 return(no)
##         }
##     }
##     else test <- if (isS4(test))
##         methods::as(test, "logical")
##     else as.logical(test)
##     ans <- test
##     ok <- !(nas <- is.na(test))
##     if (any(test[ok]))
##         ans[test & ok] <- rep(yes, length.out = length(ans))[test &
##             ok]
##     if (any(!test[ok]))
##         ans[!test & ok] <- rep(no, length.out = length(ans))[!test &
##             ok]
##     ans[nas] <- NA
##     ans
## }
## <bytecode: 0x7f9e7a16f1f0>
## <environment: namespace:base>
```

This allows you to examine the code blocks and the control etc in existing functions.

Self Test Questions Part 2

These practical questions are perhaps more self-directed than earlier ones. You will download some R code for 2 functions that do the following:

1. retrieves the geo-location of addresses that are passed to it from the Google API
2. returns the crimes in the UK for a specific period around a location

Your overall task is to examine the code carefully. Then you will create a further R function to which if passed an address or some kind of geographic reference and a crime type, maps the crime around that location.

Materials you will need

Firstly, download the R file `getdata.R` from the GitHub folder (also included as an appendix in this document), and place it in your working directory. Secondly, make sure you have all of the 'helper' packages installed: 'GISTools,rjson,RCurlandXML' installed. If you do not have these, start up R and enter:

```
install.packages(c("GISTOOls", rjson","RCurl","XML"),depend=TRUE)
```

Once these are installed you are ready to proceed.

Tasks

Firstly, make sure `getdata.R` is in your working directory. Then enter:

```r
source("getdata.R")
```

This doesn't run any code, but loads the functions `update.police`, `geocode.i` and `geocode`. You should open 'getdata.R'. The functions defined do the following:

- *map.crime* provides a `SpatialPointsDataFrame` variable of all crimes currently recorded at the `http://data.police.uk/api` website, for a specific date (month)
- *geocode.i* provides a georeference for an individual address via Google's service (limit of 2500 per day)
- *geocode* provides a `SpatialPointsDataFrame` of geo-references for a list of addresses

Task 1 Using `debug` step through each of these functions to see how they work.

Task 2 Write a new function called `map.my.local.crime.type` that takes an address, a month and a crime type and returns a map of the crime in that area (hint you may wish to think about how you deal with unlisted crime types - perhaps `factor` and specified `levels` will help?)

Task 3 Finally, make sure that your code is commented, including information explaining how to use your function, as it is in `getdata.R`.

Some functions that may help

You could explore the code below that generates a Google map around a location. First download a map:

```r
require(RgoogleMaps)
```

```
## Loading required package: RgoogleMaps
```

```r
MyMap <- GetMap(c(53.80848, -1.552792),zoom=14)
```

Then plot the map

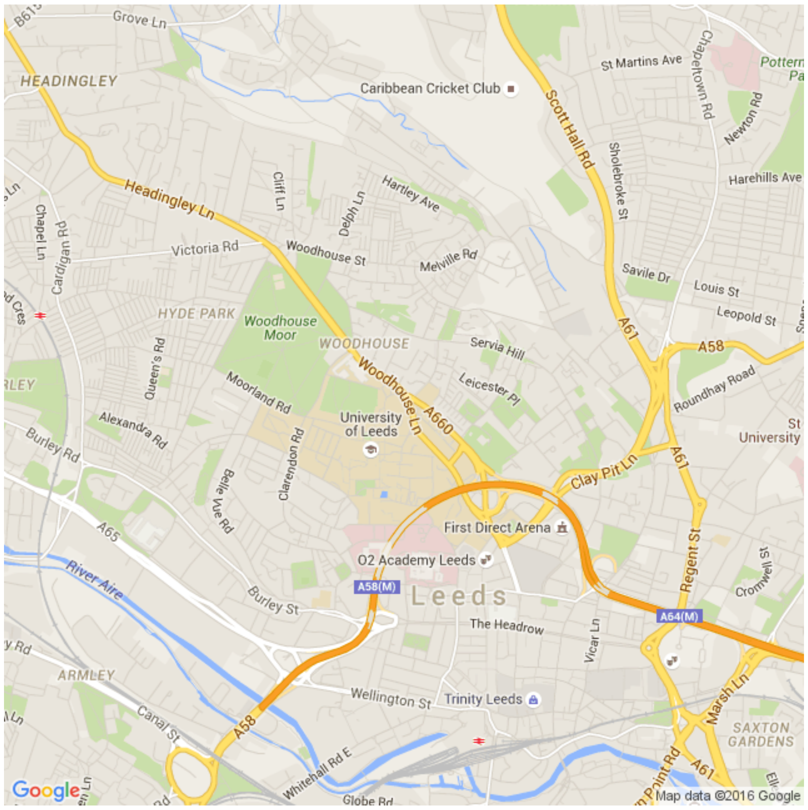
```r
PlotOnStaticMap(MyMap)
```

You can plot points on the map as well

```r
PlotOnStaticMap(MyMap, lat = 53.80848, lon = -1.552792,
   pch = 19, col = "red", asp = 1)
```

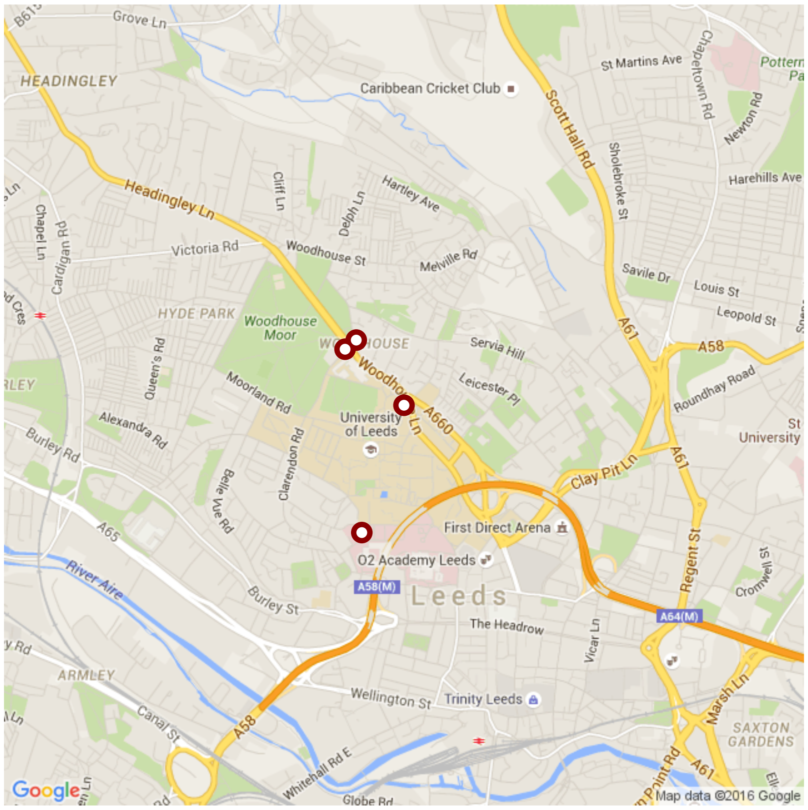
The function below plots the Google map in a nice window

```r
backdrop <- function(googlemap) {
    # Set x and y plot limits
    lim.x <- c(-320,320)
    lim.y <- c(-320,320)
    # Set map fill the entire window
    par(mar=c(0,0,0,0))
    # Create an empty plot
    plot(lim.x,lim.y,type= 'n',asp=1,
        xlab= '',ylab= '',xaxt= 'n',
        yaxt= 'n',bty= 'n')
    # Put  a box around it
    box()
    # Now add the map as a raster
    rasterImage(googlemap$myTile,-320,-320,320,320)
}
backdrop(MyMap)
```



And the code below plots a series of points on the map

```
backdrop(MyMap)
lat.list <- c(53.80848, 53.80332, 53.81112, 53.81075)
lon.list <- c(-1.552792, -1.555689, -1.556071, -1.556838)
pts.XY <- LatLon2XY.centered(MyMap, lat.list,lon.list)
points(pts.XY$newX,pts.XY$newY,pch=16,col= 'darkred',cex=1.5)
points(pts.XY$newX,pts.XY$newY,pch=16,col= 'white',cex=0.75)
```



END

Appendix: getdata.R

```
# getdata.R
# Lex Comber
# May 2016

require(GISTools)
require(RCurl)
require(rjson)

# Part 1
# map.crime()
# function to collect police.uk data and convert it to an SPDF
#
# Use:     crime.pts <- map.crime()
#          crime.pts <- map.crime(52.96827, -1.160437, "2016-01")
#
# Value: a SpatialPointsDataFrame of crimes
# for 1.5km (1 mile) radius around the location
#
```

```r
#
# The data frame of the returned SPDF has following columns:
#
#   coordinates - the lat and lon of the crime
#   category    - one of "Anti-social behaviour","Burglary","Robbery",
#                  "Vehicle crime","Violent crime","Other crime"
#   street      - the approximate address of the crime
#                   or crime location e.g. "On or near Scrogg Road"
#

# WARNING: the function accesses data from police UK at the 'street' level:
# Note that this means data references points at the centre of nearest street
# to exact crime location and is no more precise than this -
# do not infer geographical associations when overlaying other data
# having greater precision.

# First define 2 helper functions that return the Lat Lon and the attributes
# These were designed to match the data that are returned from the police website
# note that the 'return()' is implicit within these functions
getLonLat <- function(x) as.numeric(c(x$location$longitude, x$location$latitude))
getAttr <- function(x) c( x$category, x$location$street$name, x$location_type)

map.crime <- function(lat = 53.80366, lng = -1.553957, date = "2015-08") {
    # The function has defaults specifiied for lat, lng and date
    # This uses the getForm function in the Rcurl package to get crime data
    crimes.buf <- getForm( 'http://data.police.uk/api/crimes-street/all-crime',
        lat=lat,
        lng=lng,
        date=date)
    # The crimes.buf data are converted to an R object using fromJSON
    crimes <- fromJSON(crimes.buf)
    # The helper functions extract location and attributes
    crimes.loc <- t(sapply(crimes,getLonLat))
    crimes.attr <- as.data.frame(t(sapply(crimes,getAttr)))
    colnames(crimes.attr) <- c("category", "street", "location_type")
    # These are converted to a SPDF
    crimes.pts <- SpatialPointsDataFrame(crimes.loc,crimes.attr)
    # Specify the projection - in this case just geographical coordinates
    proj4string(crimes.pts) <- CRS("+proj=longlat")
    return(crimes.pts)
}

# Example of use
# crime.pts <- map.crime()
# crime.pts <- map.crime(52.96827, -1.160437, "2016-01")
# plot(crime.pts,pch= 1,col="red")

# Note that ‘head’ doesn’t work on SpatialPointsDataFrames
# crimes.pts[1:6,]
# Note that types of crimes can be selected for
# asb.pts <- crimes.pts[crimes.pts$category== "anti-social-behaviour",]
# cda.pts <- crimes.pts[crimes.pts$category== "criminal-damage-arson",]
# Part 2
# geocode - provides geocoding for addresses via Google
#
# Use: Value <- geocode("NG7 6LH")
#
# addresses:  a character vector - each element is an address
#
#
# Value: a 4-column data frame
#     acc:  Description of accuracy.
#    addr:  Address supplied
#     lat:  Latitude if Google got a single match
#     lng:  Longitude if Google got a single match
#
# Limitation - only 2,500 look-ups per ip address per day

# Geocoding example,  using the Google api

# The following function works for a SINGLE address
# This is really just a 'helper' function
geocode.i <- function(addr) {
    # gets the raw data from the Google API
    urlData <- getForm("http://maps.googleapis.com/maps/api/geocode/json",
               address=addr, sensor="false",binary=F)
    # The urlData data are converted to an R object using fromJSON
    urlData <- fromJSON(urlData)
    # defaul latitude, longitude and accuracy values
    # in case the address cannt be located by Google
    lat <- -999
    lng <- -999
    acc <- "UNRESOLVED"
    # Check to see that the return from Google is valid
    if (urlData$status == "OK") {
        geoResults <- urlData$results
        # check to see that the geoResults has a value
        if (length(geoResults) == 1) {
            geoResults <- geoResults[[1]]
            lat <- geoResults$geometry$location$lat
            lng <- geoResults$geometry$location$lng
            acc <- geoResults$geometry$location_type }}
    # returns the values to a data frame
    return(data.frame(acc=acc,addr=addr,lat=lat,lng=lng))}

# The wrapper function that takes a list of addresses
geocode <- function(addr.list) {
    # creates an empty result variable
    result <- NULL
    # loops through each of the addresses in sequence
    for (addr in addr.list) {
        # adds the result to the result variable
        result <- rbind(result,geocode.i(addr))}
    return(result) }
```

```r
# Example of use
# geocode("Worsley Building, Leeds")
# add.list <- c("ls2 9JT", "Worsley Building, Leeds", "Leeds United, Elland Rd, Leeds")
# geocode(add.list)
# add.list <- c("ls2 9JT", "Worsley Building, Leeds", "Nafees Restaurant,
#  69A Raglan Rd, Leeds", "Akmal's Tandoori, 235 Woodhouse Ln, Leeds, Leeds")
# geocode(add.list)
```