R for Biologists - An Introduction to R (Lecture 3)

Selecting and Modifying subsets of data

Subsets of the elements of a vector may be selected by appending to the name of the vector an index vector in square brackets. More generally any expression that evaluates to a vector may have subsets of its elements similarly selected by appending an index vector in square brackets immediately after the expression.

Such index vectors can be any of four distinct types.

A logical vector. In this case the index vector must be of the same length as the vector from which elements are to be selected. Values corresponding to TRUE in the index vector are selected and those corresponding to FALSE are omitted. For example

```
x <- c(-2,0,3,4,NA)
y <- x[!is.na(x)]
```

creates (or re-creates) an object y which will contain the non-missing values of x, in the same order. Note that if x has missing values, y will be shorter than x. Also

```
z <- (x+1)[(!is.na(x)) & x>0]
```

creates an object z and places in it the values of the vector x+1 for which the corresponding value in x was both non-missing and positive.

A vector of positive integral quantities. In this case the values in the index vector must lie in the set $\{1, 2, \ldots, \text{length(x)}\}$. The corresponding elements of the vector are selected and concatenated, in that order, in the result. The index vector can be of any length and the result is of the same length as the index vector. For example `x[6]` is the sixth component of x and

```
x[1:10]
```

```
##  [1] -2  0  3  4 NA NA NA NA NA NA
```

selects the first 10 elements of x (assuming length(x) is not less than 10). Also

```
c("x","y")[rep(c(1,2,2,1), times=4)]
```

```
##  [1] "x" "y" "y" "x" "x" "y" "y" "x" "x" "y" "y" "x" "x" "y" "y" "x"
```

(an admittedly unlikely thing to do) produces a character vector of length 16 consisting of "x", "y", "y", "x" repeated four times.

A vector of negative integral quantities. Such an index vector specifies the values to be excluded rather than included. Thus

```
y <- x[-(1:5)]
```

gives y all but the first five elements of x.

A vector of character strings. This possibility only applies where an object has a names attribute to identify its components. In this case a sub-vector of the names vector may be used in the same way as the positive integral labels in item 2 further above.

```
fruit <- c(5, 10, 1, 20)
names(fruit) <- c("orange", "banana", "apple", "peach")
lunch <- fruit[c("apple","orange")]
```

The advantage is that alphanumeric names are often easier to remember than numeric indices. This option is particularly useful in connection with data frames, as we shall see later.

An indexed expression can also appear on the receiving end of an assignment, in which case the assignment operation is performed only on those elements of the vector. The expression must be of the form vector[index_vector] as having an arbitrary expression in place of the vector name does not make much sense here.

The vector assigned must match the length of the index vector, and in the case of a logical index vector it must again be the same length as the vector it is indexing.

For example

```
x[is.na(x)] <- 0
```

replaces any missing values in x by zeros and

```
y[y < 0] <- -y[y < 0]
```

has the same effect as

```
y <- abs(y)
```

Vectors are the most important type of object in R, but there are several others which we will meet more formally in later sections.

Objects, their modes and attributes

Intrinsic attributes

The entities R operates on are technically known as objects. Examples are vectors of numeric (real) or complex values, vectors of logical values and vectors of character strings. These are known as "atomic" structures since their components are all of the same type, or *mode*, namely **numeric, complex, logical, character and raw**.

Vectors must have their values all of the same mode. Thus any given vector must be unambiguously either logical, numeric, complex, character or raw. (The only apparent exception to this rule is the special *"value"*" listed as `NA` for quantities not available, but in fact there are several types of NA). Note that a vector can be empty and still have a mode. For example the empty character string vector is listed as character(0) and the empty numeric vector as numeric(0).

R also operates on objects called lists, which are of mode `list`. These are ordered sequences of objects which individually can be of any mode. lists are known as "recursive" rather than atomic structures since their components can themselves be lists in their own right.

The other recursive structures are those of mode `function` and `expression`. Functions are the objects that form part of the R system along with similar user written functions, which we discuss in some detail later. Expressions as objects form an advanced part of R which will not be discussed in this class.

By the mode of an object we mean the basic type of its fundamental constituents. This is a special case of a "property" of an object. Another property of every object is its length. The functions mode(object) and length(object) can be used to find out the mode and length of any defined structure.

Further properties of an object are usually provided by attributes(object), see Getting and setting attributes. Because of this, mode and length are also called "intrinsic attributes" of an object. For example, if z is a complex vector of length 100, then in an expression mode(z) is the character string "complex" and length(z) is 100.

R caters for changes of mode almost anywhere it could be considered sensible to do so, (and a few where it might not be). For example with

```
z <- 0:9
```

we could put

```
digits <- as.character(z)
digits
```

```
##  [1] "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
```

A further coercion, or change of mode, reconstructs the numerical vector again:

```
d <- as.integer(digits)
```

Now d and z are the same. There is a large collection of functions of the form as.something() for either coercion from one mode to another, or for investing an object with some other attribute it may not already possess. The reader should consult the different help files to become familiar with them.

Changing the length of an object

An "empty" object may still have a mode. For example

```
e <- numeric()
e
```

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

makes e an empty vector structure of mode numeric. Similarly `character()` is a empty character vector, and so on. Once an object of any size has been created, new components may be added to it simply by giving it an index value outside its previous range. Thus

```
e[3] <- 17
e
```

```
## [1] NA NA 17
```

now makes e a vector of length 3, (the first two components of which are at this point both NA). This applies to any structure at all, provided the mode of the additional component(s) agrees with the mode of the object in the first place. This automatic adjustment of lengths of an object is used often.

Conversely to truncate the size of an object requires only an assignment to do so. Hence if alpha is an object of length 10, then

```
alpha <- 1:10
alpha <- alpha[2 * 1:5]
```

makes it an object of length 5 consisting of just the former components with even index. We can then retain just the first three values by

```
length(alpha) <- 3
```

and vectors can be extended (by missing values) in the same way.

The Class of an Object

All objects in R have a class, reported by the function class. For simple vectors this is just the mode, for example "numeric", "logical", "character" or "list", but "matrix", "array", "factor" and "data.frame" are other possible values.

```
m <- matrix(1:10,ncol=2)
class(m)
```

```
## [1] "matrix"
```

```
mode(m)
```

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

A special attribute known as the class of the object is used to allow for an object-oriented style of programming in R. For example if an object has class "data.frame", it will be printed in a certain way, the plot() function will display it graphically in a certain way, and other so-called generic functions such as summary() will react to it as an argument in a way sensitive to its class. More on object-oriented programming and Classes later.

To remove temporarily the effects of class, use the function unclass(). For example if winter has the class "data.frame" then

```
winter <- data.frame(numbers=1:26,characters=letters)
winter
```

```
##    numbers characters
## 1        1          a
## 2        2          b
## 3        3          c
## 4        4          d
## 5        5          e
## 6        6          f
## 7        7          g
## 8        8          h
## 9        9          i
## 10      10          j
## 11      11          k
## 12      12          l
## 13      13          m
## 14      14          n
```

```
## 15      15          o
## 16      16          p
## 17      17          q
## 18      18          r
## 19      19          s
## 20      20          t
## 21      21          u
## 22      22          v
## 23      23          w
## 24      24          x
## 25      25          y
## 26      26          z
```

will print it in data frame form, which is rather like a matrix, whereas

```
unclass(winter)
```

```
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26
##
## $characters
##  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
## Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
##
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26
```

will print it as an ordinary list. Only in rather special situations do you need to use this facility, but one is when you are learning to come to terms with the idea of class and generic functions.

Data Types

We've gone over the basic type of vector, but there are many kinds of usefull object in R

Ordered and Unordered Factors

A factor is a vector object used to specify a discrete classification (grouping) of the components of other vectors of the same length. R provides both ordered and unordered factors. While the "real" application of factors is with model formulae (see Contrasts), we here look at a specific example.

Suppose, for example, we have a sample of 30 tax accountants from all the states and territories of Australia and their individual state of origin is specified by a character vector of state mnemonics as

```
state <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
                  "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
                  "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
                  "sa",  "act", "nsw", "vic", "vic", "act")
state
```

```
##  [1] "tas" "sa"  "qld" "nsw" "nsw" "nt"  "wa"  "wa"  "qld" "vic" "nsw"
## [12] "vic" "qld" "qld" "sa"  "tas" "sa"  "nt"  "wa"  "vic" "qld" "nsw"
## [23] "nsw" "wa"  "sa"  "act" "nsw" "vic" "vic" "act"
```

Notice that in the case of a character vector, "sorted" means sorted in alphabetical order.

A factor is similarly created using the factor() function:

```
statef <- factor(state)
```

The `print()` function handles factors slightly differently from other objects:

```
statef
```

```
##  [1] tas sa  qld nsw nsw nt  wa  wa  qld vic nsw vic qld qld sa  tas sa
## [18] nt  wa  vic qld nsw nsw wa  sa  act nsw vic vic act
## Levels: act nsw nt qld sa tas vic wa
```

To find out the levels of a factor the function levels() can be used.

```
levels(statef)
```

```
## [1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

R will first convert the vector to a character string, using `as.character`

```
unclass(statef)
```

```
##  [1] 6 5 4 2 2 3 8 8 4 7 2 7 4 4 5 6 5 3 8 7 4 2 2 8 5 1 2 7 7 1
## attr(,"levels")
## [1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

Ordered Factors

The levels of factors are stored in alphabetical order, or in the order they were specified to factor if they were specified explicitly with the `levels` parameter.

Sometimes the levels will have a natural ordering that we want to record and want our statistical analysis to make use of. The ordered() function creates such ordered factors but is otherwise identical to factor. For most purposes the only difference between ordered and unordered factors is that the former are printed showing the ordering of the levels, but the contrasts generated for them in fitting linear models are different.

The function tapply() and ragged arrays

To continue the previous example, suppose we have the incomes of the same tax accountants in another vector (in suitably large units of money)

```
incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
                  61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
                  59, 46, 58, 43)
```

To calculate the sample mean income for each state we can now use the special function tapply():

```
incmeans <- tapply(incomes, statef, mean)
incmeans
```

```
##      act      nsw       nt      qld       sa      tas      vic       wa
## 44.50000 57.33333 55.50000 53.60000 55.00000 60.50000 56.00000 52.25000
```

giving a means vector with the components labelled by the levels

The function tapply() is used to apply a function, here mean(), to each group of components of the first argument, here incomes, defined by the levels of the second component, here statef, as if they were separate vector structures. The result is a structure of the same length as the levels attribute of the factor containing the results.

Suppose further we needed to calculate the standard deviation of the state income means.

```
incsd <- tapply(incomes, statef, sd)
incsd
```

```
##        act        nsw         nt        qld         sa        tas
##  2.1213203 10.5577775  6.3639610  9.1815031  5.4772256  0.7071068
##        vic         wa
## 11.7260394  5.3150729
```

In this instance what happens can be thought of as follows. The values in the vector are collected into groups corresponding to the distinct entries in the factor. The function is then applied to each of these groups individually. The value is a vector of function results, labelled by the levels attribute of the factor.

The combination of a vector and a labelling factor is an example of what is sometimes called a ragged array, since the subclass sizes are possibly irregular. When the subclass sizes are all the same the indexing may be done implicitly and much more efficiently, as we see in the next section.

Packages

All R functions and datasets are stored in packages. Only when a package is loaded are its contents available. This is done both for efficiency (the full list would take more memory and would take longer to search than a subset), and to aid package developers, who are protected from name clashes with other code.

To see which packages are installed, issue the command

```
library()
```

with no arguments. To load a particular package (e.g., the boot package containing functions from Davison & Hinkley (1997)), use a command like

```
library(boot)
```

Users connected to the Internet can use the install.packages(), update.packages() functions and RStudio tab Packages to install and update packages.

To see which packages are currently loaded, use

```
search()
```

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

to display the search list. Some packages may be loaded but not available on the search list: these will be included in the list given by

```
loadedNamespaces()
```

```
##  [1] "magrittr"  "graphics"  "tools"     "htmltools" "utils"
##  [6] "yaml"      "grDevices" "stats"     "datasets"  "stringi"
## [11] "rmarkdown" "knitr"     "methods"   "stringr"   "digest"
## [16] "boot"      "base"      "evaluate"
```

Standard (Base) Packages

The standard (or base) packages are considered part of the R source code. They contain the basic functions that allow R to work, and the datasets and standard statistical and graphical functions. They should be automatically available in any R installation.

Contributed Packages

There are thousands of contributed packages for R, written by many different authors. Some of these packages implement specialized statistical methods, others give access to data or hardware, and others are designed to complement textbooks. Some (the recommended packages) are distributed with every binary distribution of R. Most are available for download from CRAN and its mirrors) and other repositories such as Bioconductor and Omegahat

Session Info

```
sessionInfo()
```

```
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.11.3 (El Capitan)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] boot_1.3-17
##
```

```
## loaded via a namespace (and not attached):
##  [1] magrittr_1.5    tools_3.2.3     htmltools_0.3   yaml_2.1.13
##  [5] stringi_1.0-1   rmarkdown_0.9.5 knitr_1.12.3    stringr_1.0.0
##  [9] digest_0.6.9    evaluate_0.8
```