Working With Data

Laurence Kell
ICCAT

Abstract

R has a rich variety of data types such as scalars, vectors, matrices, arrays, data frames and lists. Before you can take advantage of R and the many contributed packages you have to be able to import your data that may have been stored in a variety of text files, spreadsheets, databases or even binary files. Once you have got your data into R it will be necessary to convert them from one type to another (i.e. coercion), summarise, subset and plot them and to transform objects often by writing R functions. In this document we describe the main R data types, show how to imported data into R and to manipulate them.

Keywords: R, data types, import, export.

---

1. Introduction

R has a rich variety of data types such as scalars, vectors, matrices, arrays, data frames and lists, as well as data in S4 classes. However, before you can take advantage of R and contributed packages to work on your own data you have to be able to import it into R from a variety of files, e.g. text, spreadsheets, databases or binary files. Then once you have your data in R it may be necessary to convert them from one type to another (i.e. coercion), plot, summarise and subset them and to transform objects often by writing R functions. In this document we describe the main R data types, then show how to manipulate them and imported data into R.

R data types including scalars, vectors, matrices, data frames and lists, which may contain variables of different types, e.g. numerical, character, logical. See the R language definition for full documentation and the R reference card for a summary of methods and functions to use with them.

2. Help

R has extensive documentation and the main ways of finding help are

```
?lm                  #  specific help - if you know the name of the function
args(lm)             #  gives the specific arguments to the function
example(log)         #  Example code
help.start()         #  Start the browser
help.search("plot")  #  if you only know the subject area
apropos('lm')        #  returns a character string of all potential matches
??plot               #  returns the package and all available methods
help.start()         #  launches the online html help.
example(array)       #  returns the example code provided in the help doc
find('par')          #  will tell you what package something is in
search()             #  returns a list of attached packages
demo(graphics)       #  demos of what a package does
```

There are also extensive online resources, i.e. websites such as CRAN, R FAQ, Windows R FAQ and R seek; mailing lists for users where you can find examples & join discussions; the R Journal, blogs and reference guides such as Quick-R.

An important feature of R are the many packages that have been contributed by users for a variety of tasks. These often have there own guides and mailing lists and guides, for example the graphics package ggplot2.

When working in an R Session you will want to occasionally save code as you are working on. You can recall the code, i.e. the commands, that you have been running in your session by typing history(). By default only 25 lines are shown, you can also save your entire history to a file and even run previous commands. To find out more use the help i.e.

```
> ?history
```

3. Data

Vectors are continuous cells containing data of a single type. Types can be character (i.e. string), integer, double, logical, raw (or bytes) or complex. Matrices and arrays are similar to vectors but with 2 or more dimensions. There are also attributes for the dimensions (dim) and, optionally the dimension names (dimnames); a matrix has 2 dimensions (i.e. row and column) and arrays 2 or more dimensions.

Methods for finding out what class a particular object belongs to, what type of data it contains and what storage mode is used to store the information are important. As an example we generate some random data and use it to create an array. You will see that the storage mode of the objects contents is 'numeric' whilst the specific data type is 'double'. The object oriented approach allows simple classes to be extended to form more complex classes, e.g. vectors can be coerced into a data.frame The 'is' method can be used to view the full class inheritance tree of the object.

```
> x = array(rnorm(9),dim=c(3,3),
+                    dimnames=list(row=c("x","y","z"),col=1:3))
> x

   col
row          1         2          3
  x -1.6716369 1.2882357 -0.4899754
  y -0.9667871 0.3914129  0.6129542
  z -0.6851321 0.8843904  0.1489437

> # what is it?
> class(x)

[1] "matrix"

> typeof(x)

[1] "double"

> mode(x)

[1] "numeric"

> is(x)

[1] "matrix"    "array"     "structure" "vector"
```

Data frames are similar to data sets as used by statistical software such as SAS and contain variables (columns) by observations (row). The columns may contain vectors of different types, e.g. numeric, logical, character or factors. They must all have the length. In addition, a data frame generally has a names attribute labelling the variables and a row.names attribute for labelling the observations. A data frame can also contain a list of the same length as the other components.A list is a collection of objects that may be of any type and length.

It is worth mentioning in passing the model.frame method (or function) which in certain packges returns a data.frame with the variable needed to use the formula in a model, such as a linear regression, stored as columns.

3.1. Vectors

R is a vector orientated language and the assignment symbol ”=” or ”<-” creates a new object. Typing only the name of the object outputs its values. Calculations are performed on

all elements of a vector thus the square of all elements in vector b can be computed simply by passing the vector b to the appropriate method as shown below.

```
> b = 1:5
> length(b)

[1] 5

> b

[1] 1 2 3 4 5

> b^2

[1]  1  4  9 16 25
```

R is case sensitive and the correct case must be used when calling functions and methods and referencing variables. Variables names can contain letters, digits, ànd _, however the variable name must start with a letter.

A vector must contain elements of the same type, e.g. a vector of doubles as above. The vector oriented approach of the the R language means that functions such as log will be applied to all elements in the vector passed as an argument e.g.

```
> log(b)

[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

4. Methods

R is an object oriented language and an important concept is that everything in R is an object and that every object belongs to a class. Without getting into a detailed explanation of Object Oriented Programming (OOP) many packages are based on the S4 classes using OOP . An S4 class contains structures composed of certain types of data and methods or functions that manipulate the classes. What class an object belongs to determines what operations can be performed on it and what a generic method like plot() or or summary() produces.

4.1. Plotting

Since R is an object orientated language it knows how to make appropraite plots for any object, since it uses methods, i.e. generic functions that behave appropriately depending on

the object passed as an argument. In this case a trivial exercise but with more complicated objects this is a useful feature.

```
> plot(b)
```

5. Examples

Recycling is an important feature of vector arithmetic. When two vectors of unequal length are combined, the elements of the shorter vector are reused. In the example below, after the third element in a, the first element in b is reused. This process is known as automatic recursion and can be a very powerful tool in R. But beware of it! R will use automatic recursion and will assume that the user is aware of it. If the length of the smaller object is a multiple of the larger object, R will use automatic recursion. If the length of the smaller object is not a multiple of the larger object, R will still perform the action but will give a warning that the sizes of the two objects are not compatible.

```
> # recycling
> a = 1:6
> b = 1:3
> a + b

[1] 2 4 6 5 7 9

> c = 1:4
> a + c

[1] 2 4 6 8 6 8
```

A variety of missing values can be specified in R. Values can take Inf or -Inf for infinity, NA for a missing value and NaN for not a number. Dividing a positive number by 0 will produce Inf value and dividing a negative number by 0 will produce a -Inf value. NaN values can be created by trying to calculate the log of a negative number, for example.

```
> a = c(3, 4, NA, Inf)
> a

[1]   3   4  NA Inf

> tt = c(1, NA, 3, 4, NA, 6, NaN)
> is.na(tt)

[1] FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE

> is.nan(tt)

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

The sum of vector a will in the example above be defined as NA, since NA is propagated. The na.rm argument will remove the NAs and then sum the valid values in the vector.

```
> a=1
> sum(a)

[1] 1

> sum(a, na.rm = T)

[1] 1
```

```
> a=1
> sum(a)

[1] 1

> sum(a, na.rm = T)

[1] 1
```

Similar rules apply for string manipulation, for which additional methods such as paste are available

```
> b = c("yes", "no", "maybe")
> paste("is it", b)

[1] "is it yes"   "is it no"    "is it maybe"

> paste("is it", b, sep = ",")

[1] "is it,yes"   "is it,no"    "is it,maybe"

> paste("is it", b, sep = " ", collapse = ", ")

[1] "is it yes, is it no, is it maybe"
```

A variety of methods exist for creating objects with different characteristics for example rnorm creates random numbers

```
> a = rnorm(100)
```

A vector can be summaried by calculating it mean and standard deviation and plotted as a histogram. Many basic functions exist in R to create a variety of summary statistics. In addition a number of basic plotting functions are also available for quick and easy data visualisation e.g.

```
```

max() and min() return the maximum and minimum values in a vector, and pmax and pmin perform a pairwise comparison, i.e. comparing the vector with 0.0.

```
[1] 2.323448

  [1] 0.843975077 0.000000000 0.000000000 0.759440249 0.000000000 0.000000000
  [7] 0.655559150 0.000000000 0.904518208 0.274524576 0.000000000 0.722728758
 [13] 0.000000000 0.000000000 2.323447546 0.144805359 0.000000000 0.000000000
 [19] 0.400645130 0.174289045 0.244995167 1.303992901 0.000000000 0.000000000
 [25] 0.000000000 0.240908525 0.000000000 0.000000000 0.000000000 0.097104036
 [31] 0.000000000 0.000000000 0.244775841 0.099222651 0.051801262 0.067313287
 [37] 0.421458008 0.505639647 0.368358780 1.039287635 1.428110936 0.282612636
 [43] 0.755235077 1.448248034 0.463330361 0.000000000 1.381180585 0.000000000
 [49] 0.000000000 0.000000000 0.000000000 0.548693523 0.835459354 0.000000000
 [55] 0.000000000 0.000000000 1.628117285 0.381719741 0.000000000 0.000000000
 [61] 0.405100547 0.000000000 0.000000000 0.616288867 0.174181917 0.000000000
 [67] 0.000000000 0.000000000 0.000000000 0.000000000 1.338148939 0.548750484
 [73] 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000 0.654645567
 [79] 0.000000000 1.079811249 0.000000000 0.000000000 0.000000000 0.341869018
 [85] 1.389407289 0.000000000 0.000000000 0.000000000 0.709884383 0.000000000
 [91] 0.000000000 0.000000000 0.000000000 0.004408523 0.547156202 0.000000000
 [97] 0.000000000 1.125207139 0.000000000 0.000000000
```

Comparing elements of vectors can be done with ==, and ! is the logical inverse

```
> c(T, F, T, T, F, F) == TRUE

[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE

> !c(T, F, T, T, F, F)

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

A vector can be subset using logical values or using a comparison operator, in this instance to get all values greater than two or by sub-setting the vector by its index, i.e. to get the first three elements

```
> a = c(T, T, F, F, T)
> b = 1:5
> b[a]

[1] 1 2 5

> b[b > 2]

[1] 3 4 5

> b[1:3]

[1] 1 2 3
```

Alternatively by giving a list of numbers, or by excluding an element

```
> b[c(1, 2, 1, 3, 4, 1, 1, 5)]

[1] 1 2 1 3 4 1 1 5

> b[-3]

[1] 1 2 4 5
```

Vectors of one type can be converted to vectors of other types through coercion, i.e. from integer to logical

```
> b = c(0, b)
> lb = as.logical(b)
> lb

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

5.1. Matrices and Arrays

Matrices and arrays are vectors with attributes that define the dimensions (dims) and optionally the dimension names (dimnames), recalling two-dimensional array (i.e. matrix) x that we created before use the following

```
> # create an array
> x = array(rnorm(9),dim=c(3,3),
+                     dimnames=list(row=c("x","y","z"),col=1:3))
```

The matrix function takes an argument that determines how the matrix is filled, i.e. by row or by column. Note that by default a matrix is filled by column, you can change the way that the matrix is filled by setting the argument 'byrow' to TRUE. The kind of data structure (i.e. number of rows and columns) is determined by the attributes of the object.

```
> # dimensions
> dim(x)

[1] 3 3

> dimnames(x)

$row
[1] "x" "y" "z"

$col
[1] "1" "2" "3"

> attributes(x)

$dim
[1] 3 3

$dimnames
$dimnames$row
[1] "x" "y" "z"

$dimnames$col
[1] "1" "2" "3"

> # alternatives for creating an array
> a = matrix(1:16, nrow = 4)
> aa= matrix(1:16, nrow = 4, byrow = TRUE)
> a

     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

> aa

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

It is possible to remove dimensions, for example to get a vector rather than a matrix, by changing the attributes of the object. A matrix can be subset to obtain specific values, i.e. a vector defined by row or column A vector can also be reshaped to get a higher dimensional array, note that you cannot go out of a vectors bounds, since this is protected

```
> # changing attributes
> a=1
> dim(a) = NULL
> attributes(a)

NULL

> a

[1] 1

> attr(aa, "dim") = c(2, 4, 2)
> attributes(aa)

$dim
[1] 2 4 2

> aa

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    9    2   10
[2,]    5   13    6   14

, , 2

     [,1] [,2] [,3] [,4]
[1,]    3   11    4   12
[2,]    7   15    8   16

> aa[2, 3,1]

[1] 6

> aa[2,,]

     [,1] [,2]
[1,]    5    7
[2,]   13   15
[3,]    6    8
[4,]   14   16

> aa[1:2,, ]

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    9    2   10
[2,]    5   13    6   14

, , 2

     [,1] [,2] [,3] [,4]
[1,]    3   11    4   12
[2,]    7   15    8   16
```

5.2. Lists

Lists are simply collections of objects. The objects contained in any given list can be of different types. In this example we create a list of length 3 containing string, integer and logical types.

```
> fish = list(name = "cod", age = 3, male = FALSE)
> fish

$name
[1] "cod"

$age
[1] 3

$male
[1] FALSE
```

You can retrieve an element of a list in different ways, either by name or by specifying its index using square brackets.

```
> fish$name

[1] "cod"

> fish[["name"]]

[1] "cod"

> fish[[1]]

[1] "cod"
```

Note how we use double square brackets to index the list. Indexing a list with single square brackets will return an object of class list that has length 1 and contains the element of the list that we have requested. When using double square brackets we are returned an object in the class of the contents of the element of the list that we are accessing.

```
> class(fish$name)

[1] "character"

> class(fish[["name"]])

[1] "character"

> class(fish[[1]])

[1] "character"

> class(fish[1])

[1] "list"
```

Using double square brackets returns the original type, whilst single square brackets returns a list. You can also create arrays of type list e.g.

```
> array(list(), c(2, 3))

     [,1] [,2] [,3]
[1,] NULL NULL NULL
[2,] NULL NULL NULL
```

5.3. Data frames

In a data.frame all vectors need to be of the same length, this is equivalent to a SAS data set or simple Excel sheet. First we create a list with equal vector length, then make the data.frame.

```
> a = list(age = 1:10, weight = c(0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.6, 0.7, 0.9))
> a

$age
 [1]  1  2  3  4  5  6  7  8  9 10

$weight
 [1] 0.05 0.10 0.20 0.30 0.40 0.50 0.60 0.60 0.70 0.90

> b = as.data.frame(a)
> b

   age weight
1    1   0.05
2    2   0.10
3    3   0.20
4    4   0.30
5    5   0.40
6    6   0.50
7    7   0.60
8    8   0.60
9    9   0.70
10  10   0.90
```

It is also possible to make data.frame in one go and because of recycling you dont need to replicate all elements.

```
> fish = list(name = "cod", age = 3, male = FALSE)
> fish

$name
[1] "cod"

$age
[1] 3

$male
[1] FALSE
```

You can subset a data.frame like you would a list i.e. to get a vector

```
> fish$name

[1] "cod"

> fish[["name"]]

[1] "cod"

> fish[[1]]

[1] "cod"
```

Sometimes it will be useful to determine what class an object belongs to and to add comments.

```
> class(fish$name)

[1] "character"

> class(fish[["name"]])

[1] "character"

> class(fish[[1]])

[1] "character"

> class(fish[1])

[1] "list"

> attr(fish, "rem") = "This is my dataframe"
```

You can edit a data.frame as you would spreadsheet using fix and import and export dataframes, by default if no path, then file is put in setwd() dir

```
> args(fix)

function (x, ...)
NULL

> write.table(b, )

"age" "weight"
"1" 1 0.05
"2" 2 0.1
"3" 3 0.2
"4" 4 0.3
"5" 5 0.4
"6" 6 0.5
"7" 7 0.6
"8" 8 0.6
"9" 9 0.7
"10" 10 0.9

> args(write.table)

function (x, file = "", append = FALSE, quote = TRUE, sep = " ",
    eol = "\n", na = "NA", dec = ".", row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"), fileEncoding = "")
NULL

> args(read.table)

function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
    encoding = "unknown", text)
NULL
```

5.4. Coercion

Coercion is the act of creating one type of R object from another, this can be changing the basic type, i.e. a string into a number, or a matrix into a data.frame.

Coercion just requires sticking an ???as.??? in front of the type you want to create e.g. for both the atomic types (i.e. vectors of type integer, double,..)

```
> as.double("2")

[1] 2

> as.character(2)

[1] "2"

> df = data.frame(string = c("eeny", "meeny", "miny", "mo"), integer = 1:4)
> as.matrix(df)

     string  integer
[1,] "eeny"  "1"
[2,] "meeny" "2"
[3,] "miny"  "3"
[4,] "mo"    "4"
```

6. Importing Data

Very often the data that you want to analyse will be available in a file format that is supported by another piece of software such as a spreadsheet, a database or a statistical package. There are a variety of methods for importing data of many different formats into your R session. See the guide http://cran.r-project.org/doc/manuals/R-data.pdf for a full review.

A number of packages have been developed to enable import of data from other formats. For example the 'foreign' package allows a number of alterrnative data formats to be read into R including SAS formatted data and SAS transfer files, and the package 'RODBC' provides database functionality.

6.1. Text

Here we consider the most common data format i.e. comma or space separated data saved as a flat text file. The required methods are read.table and read.csv, although your data do not necessarily have to be comma separated. Spaces, tabs and any other form of separator may be used, but they must be used consistently. Files containing a combination of different separators will be very difficult to read using the methods below. There are a number of functions provided in the base distribution of R for importing data into your work session. These include scan, readLines and read.table. Perhaps the simplest method is to use scan which simply reads all of the data contained in the file into a single vector. The data can then be coerced into the required type as shown. The readLines method will do something very similar but will read in only the specified number of lines. The read.table method is particularly useful. It returns a data.frame object containing the data. If the header is specified as TRUE then the first line of data to be read in will be used for the column names in

the returned data.frame object. read.csv() is similar to read.table and is basically a wrapper that defines the seperators ad ???,??? and sets header=TRUE by default e.g. to read in the catch distribution data. You will need to change the file path given in the example below to the location of the file on your machine.

```
> dirData=paste(system.file(package="saCourse", mustWork=TRUE),"examples",sep="/")
> ## read.csv
> fileCdis =paste(dirData,"cdis.csv",sep="")
> cdis=read.csv(fileCdis)
> head(cdis)
```

read.table() can also read from http and fttp, e.g. the NAO data on the NOAA website as it also accepts web addresses e.g.

```
> nao =read.table("http://www.cdc.noaa.gov/data/correlation/nao.data", skip=1,
+ nrow=62,
+ na.strings="-99.90")
```

scan() is more flexible but requires a bit more effort, here we read in data from a file that holds the catch-at-age data for mediterannean swordfish and create an array with appropriate dims and dimnames

```
> fileSwo="swoCN.dat"
> yrs =scan(fileSwo,skip=2,nlines=1)
> ages=scan(fileSwo,skip=3,nlines=1)
> dat =scan(fileSwo,skip=5)
> caa =array(dat,dim     =c(diff(ages)+1, diff(yrs)+1),
+                dimnames=list(age =ages[1]:ages[2],year=yrs[1]:yrs[2]))
```

6.2. Spreadsheets

As is almost always the case with R, there are several different ways to perform the same task. Data can be read in from an Excel spreadsheet using one of a number of methods. The packages RODBC, xlsReadWrite, xlsx and gdata all contain routines for accessing data in spreadsheet form. In the example below we use the read.xls method available in the gdata package to read data from the excel file swordfish.xls. This is because it uses the same defaults as for read.csv() making it easier to use. You will need to change the file path given in the example below to the location of the file on your machine. If you haven't already installed the gdata package you will need to do this first, and also install perl

```
> install.packages('gdata')
> library(gdata)
> yrs =read.xls("swordfish.xls",sheet="CatchN",skip=1,nrow=1)[1:2]
> ages =read.xls("swordfish.xls",sheet="CatchN",skip=2,nrow=1)[1:2]
> dat =read.xls("swordfish.xls",sheet="CatchN",skip=5,header=F)
> caaXl=array(dat,dim
+ =c(diff(ages)+1, diff(yrs)+1),
+ dimnames=list(age =ages[1]:ages[2],year=yrs[1]:yrs[2]))
> is(ages)
> is(yrs)
> is(dat)
> ages=unlist(ages)
> yrs =unlist(yrs)
> dat =t(as.matrix(dat))
> caaXl=array(dat,dim
+ =c(diff(ages)+1, diff(yrs)+1),
+ dimnames=list(age =ages[1]:ages[2],year=yrs[1]:yrs[2]))
> caaXl
```

As you can see, the example given above counts the number of sheets in the excel file, obtains the names of each of those sheets and finally accesses the data in the second sheet, ommitting the first four rows of data and extracting the following 5 rows of data. The results are returned as a data.frame to the object we have called xlsdata.

6.3. Databases

Databases prove more flexibility, since you can access different tables and views. Where a view is a virtual table, i.e. it is constructed from other tables in the database.

```
> chlT3sz   =odbcConnectAccess(fileT3sz)
> sqlTables(chlT3sz)
> t3sz     = sqlQuery(chlT3sz, "select * from [t2szFreqs]")
> head( t3sz)
> names(t3sz)
> head(sqlQuery(chlT3sz, "select * from [Species]"))
> head(sqlQuery(chlT3sz, "select * from [t2szFreqs]"))
> head(sqlQuery(chlT3sz, "select * from [t2szProcs]"))
> head(sqlQuery(chlT3sz, "select * from [t2szStrata]"))
> head(sqlQuery(chlT3sz, "select * from [cat_szDetail]"))
> head(sqlQuery(chlT3sz, "select * from [cat_szSummary]"))
> head(sqlQuery(chlT3sz, "select * from [subSets]"))
> head(sqlQuery(chlT3sz, "select * from [t2szProcs Query]"))
```

An example of creating a database and making a query

```
> library(RSQLite)
> ## Initialise the SQLite engine
> SQLite()
> ## connect DB
> dbMC ="c:/temp/t.dbf"
> conMC =dbConnect(dbDriver("SQLite"), dbname=dbMC)
> df=data.frame(n=rep(1:4,25),a=rep(c("a","b","c","d"),each=25))
> dbWriteTable(conMC,"EastMC",df,append=TRUE)
> dbListTables(conMC)
> file.info(dbMC)
> query="SELECT * FROM 'EastMC' WHERE n IN (2) AND a IN ('a','b') LIMIT 10"
> x= dbGetQuery(conMC, query)
```

A utility function for database queries

```
> setGeneric("sqlVar", function(object, ...)
+ standardGeneric("sqlVar"))
> setMethod("sqlVar", signature("character"),
+ function(object, ...) paste("('",paste(object,collapse="','"),"')",sep=""))
> setMethod("sqlVar", signature("numeric"),
+ function(object, ...) paste("(",paste(object,collapse=","),")",sep=""))
> sqlVar(c("b"))
> sqlVar(2)
> query= paste("SELECT * FROM 'EastMC' WHERE n IN", sqlVar(2), "AND a IN", sqlVar(c("a","b")), "LIMIT
+ 10")
> x= dbGetQuery(conMC, query)
> dbDisconnect(conMC)
```

7. Manipulating Objects

R has many functions and methods for manipulating data objects such as arrays, data.frames and lists. Although FLR is based on object oriented (OO) programming where data (i.e. slots) and actions (i.e. methods) are grouped together in S4 classes, these methods can still be used with FLR. For example an FLQuant is derived from an array and so can be manipulated using functions written for arrays, while an FLR class has similarities to a list (in that it can contain a variety of data types) and functions that work for lists have been overloaded so that they work for FLR classes. In R it is recommended that rather than using ???for loops??? that you use apply and sweep. To the new user this can appear to be confusing but using them speeds up code and helps conceptually by moving towards a "whole object" view. The apply family of functions allow functions to be applied on subsets of different types of R classes (i.e. lapply, tapply, sapply, rapply mapply etc). Sweep is useful when trying to perform an operation on two arrays that might have different dimensions.

We show how these and related functions can be used within FLR. Firstly showing how R functions used for arrays can be used with the FLQuant class, we then describe additional features and functions that have been added to FLR. Following this we describe functions that have been added for the FLR classes themselves.Let's start by loading FLCore and some data sets distributed with it.

7.1. apply

The apply function in base R applies a function to the margins of an array and returns a vector or array or list of the values obtained.

```
> ?apply
> example(apply)
```

7.2. sweep

sweep return an array derived from an input array by sweeping out a summary statistic The function is useful when performing an operation on two arrays which dont have the same dimensions, for example when calculating selection pattern by first scaling fishing mortality-at-age by the average value or looking at sex ratios where numbers by sex are divided by total numbers

```
> ?sweep
> example(sweep)
```

7.3. Re-shaping

see http://plyr.had.co.nz/ for a set of tools for data manipulation. For example to split up a big data structure into homogeneous pieces, apply a function to each piece and then combine all the results back together.

8. Functions

Very often you will want to write a function in R to perform a given task or routine that may be required several times. If you have previously written code for other languages you may be familiar with the conventional approach of writing 'for loops' whereby the code loops successively through each element of the object and performs the same action on each. The example below is a trivial case, simply adding 2 to a numeric vector using the conventional programming approach. In R, this is generally the slow and inefficient method.

```
> for (i in 1:length(x))
+ x[i] = x[i] + 2
```

A far more effective approach is to use the vectorised arithmetic of R. But as mentioned above, beware of automatic recursion.

```
> x = c(2, 3, 4, 5)
> x + 2
```

In the example below we create a very simple function that calculates the cube of any value, or vector of values passed to it. A function comprises 2 main components: the function arguments and a function body. The function body contains error trapping, the main processes of the function and the return statement.

```
> cube = function(x, na.rm = TRUE) {
+ if (na.rm == TRUE)
+ x = x[!is.na(x)]
+ return(x^3)
+ }
> a = 1:5
> cube(a)
```

Our function takes two arguments, x and na.rm and that by default na.rm is set to TRUE. The function performs some simple error trapping conditional on the value of na.rm and returns an object of the same class as x, cubed. If we want to see the formal arguments to any function we can use the args() method, as below.

```
> args(cube)
```

8.1. Debugging

As soon as you write a function you will find that it doesnt work as expected due to a bug.

```
> err = function(a){
+     loga = log(a[,1])
+
+     res  = mean(log(a))
+
+     return(res)}
> err(rnorm(5,0,1))
```

You can debug functions written in R using the environment browser. This will interrupt the execution of an expression and allow the inspection of the environment where ???browser??? was called from. A call to ???browser??? can be included in the body of a function. When reached, this causes a pause in the execution of the current expression and allows access to the R interpreter, enabling you to interrogate the objects that have been declared locally within the function. For example if we have the following function which takes a numeric vector and performs a number of simple tasks eventually returning the object res ... You will see that the function returns an error: Error in a[, 1] : incorrect number of dimensions To debug the function we simply insert a call to browser() within the function as below. This will halt execution of the function at the point of the call to browser(). Entering 'n' at the command prompt will execute the remainder of the function line by line. Entering a 'c' will execute the remaining code and exit the function.

```
> err2 = function(a,b){
+ browser()
+ v1 = mean(a)
+ v2 = median(a)
+ loga = log(a[,1])
+ res= mean(log(a))
+ return(res)}
> err2(rnorm(5,0,1))
```

By stepping through the code and interrogating the objects that are created you should be able to detect the line at which the function fails. One disadvantage of browser() is that on encountering the point of failure the environment immediately returns to the top level. A more sophisticated debugger is available in the 'debug' package

```
> install.packages('debug')
> library(debug)
> mtrace(err)
> err(rnorm(5,0,1))
> mtrace.off(err)
```

With debug there is no need to insert break points or commands within the function. Instead you declare the function for debugging using 'mtrace' as shown below

Affiliation:

Laurence Kell
ICCAT Secretariat
C/Corazón de María, 8.
28002 Madrid
Spain
E-mail: Laurie.Kell@iccat.int