

## INTRODUCTORY R WORKSHEET 2: GETTING STARTED WITH R

Before attempting this worksheet you should have completed *Introductory R Worksheet 1: Getting Started*.

### 1. *Getting into R.*

As described in Worksheet 1, you should log in to the PC/Mac using your University ID and password and begin R using the RStudio program. (If at home, log on as you would usually).

Note that if you get into difficulties with any R command and wish to cancel it after you have pressed the return key, hit the Esc key on the keyboard to interrupt R.

### 2. *Arithmetic Operators in R*

Now that you are in R, the first thing you can do is to use R as a basic calculator. R will perform all the usual arithmetic operations on numbers:

+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation (i.e. raising to a power)

To make sure you know how to apply these operators (and to check that R will give you the expected results), try doing a few simple calculations such as adding, subtracting, multiplying and so on with some chosen pair of numbers. For example try finding 3 cubed (i.e. 3 to the power 3).

R also has the following comparison and logical operators:

==	is equal to
!=	is not equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
&&	and
	or
!	not

These operators are primarily used in identifying subsets of data objects and in writing branching statements (such as *if ... then ...* statements, etc.) which you will need when writing your own functions (which we will discuss in more detail in Worksheet 3). Try typing in the following expressions and interpreting the results:

`3 < 5`

`3 > 5`

`3 > 5 || 3 < 5`

There are also a number of simple *R* univariate functions which will work using “ordinary” numbers as arguments:

<code>abs()</code>	the absolute value of the argument
<code>floor()</code>	the integer part of the argument
<code>sqrt()</code>	the square root of the argument
<code>exp()</code> , <code>log()</code> , <code>log10()</code>	exponential and log functions
<code>sin()</code> , <code>cos()</code> , <code>tan()</code>	trigonometric functions
<code>asin()</code> , <code>acos()</code> , <code>atan()</code>	inverse trigonometric functions
<code>sinh()</code> , <code>cosh()</code> , <code>tanh()</code>	hyperbolic functions
<code>asinh()</code> , <code>acosh()</code> , <code>atanh()</code>	inverse hyperbolic functions

Try applying these functions, e.g. test the square root function by typing:

```
sqrt(4)
```

### 3. *Storing Objects in R*

So far, all we have been doing in *R* is evaluating expressions. We can also create and store the results of these evaluations as permanent objects by using the assignment function, `<-` or `=`. An example of how this function is used is:

```
c <- 5
```

The assignment function tells *R* to first evaluate the right-hand side (which in this case isn’t much of an evaluation) and assign its value to the object named on the left side. Since the assignment arrow is frequently used, it has an alternate one-character form, the equal sign “=”.

Note that *R* doesn’t seem to do anything in response to the above command. What it has done is create a new permanent object called `c` which has the value 5. To see the contents of any object such as `c`, simply type the name of the object.

Arithmetic operations can be performed on this object as we did for numbers. For example, try typing the following commands:

```
c^3
c*3
```

*R* will automatically save any objects you create for use in future sessions. By default it saves them in the `.Data` directory in your account area. You can examine the contents of this directory by using the `objects()` function. You can also look at your objects in the *Workspace Manager*, but here we will focus on the command line interface. Try using this function now by typing:

```
objects()
```

The object called `c` which we created earlier in this session is probably the only object currently in your area. This object will remain in your area until you remove it.

There are a number of other areas which are accessible from your account, such as libraries or collections of *R* commands and the `classdata` library/workspace that is discussed in Worksheet 1. The areas which were set up for you when you started your *R* session can be listed using the command:

```
search()
```

The result of this command is a list of areas, with a position number in front of each area. To list the contents of area 3, use the `objects()` command and specify the number of the area, i.e.:

```
objects(when=3)
```

The default area for the `objects()` command is your `.Data` area which is located at position number 1 in your `search()` list. Use the `objects()` command to explore the other areas and look at the contents of some of the objects in these areas by typing their names.

You will find that some of these objects will be sets of data and others will be functions containing lists of commands. R stores both data and functions as objects. This is why it is essential to type the brackets when you wish to use a function.

To create a new function for finding the square of a number, we would type in the following two lines:

```
square <- function(x) {  
  x^2}
```

There will be more on writing your own R functions in Worksheet 3. Typing `objects()` should now show the two objects you have created this session, i.e. `c` and `square`.

To see what an object contains, just type the name of that object. So, typing `square` without any brackets will show you the contents of this new function, whilst typing `square(c)` will calculate the value of `c` squared. Try typing `square()`, without any number or argument, and see what happens. The warning message occurs because typing a function name followed by parentheses indicates to R that you wish to calculate something using the function (as opposed to seeing its contents), and the `square()` function requires an input value in order to perform its calculations.

New R objects can be called anything you like, which can lead to problems if you use the names of common R functions to store data items. For instance, `c()` and `t()` are the names of two very frequently used R functions and using these names for your own objects can lead to problems as we will see below. Stored R objects can be deleted using the `rm()` function.

#### 4. Types of Data Objects in R.

The simplest type of stored object is a single number, i.e. a scalar. We have already seen an example of a stored scalar; namely, the object we created called `c`. We can change the value of this stored object by simply reassigning it, i.e.:

```
c <- 4
```

The next simplest stored data object is a vector of numbers (a scalar can be regarded as simply a vector with only one component). A number of functions can be used to create special vectors. The functions `seq` and `rep` are useful to create vectors containing sequences of numbers or repeated patterns of numbers. Experiment with the functions `seq()` and `rep()`. Examine the help files on these functions - use `help(seq)` and `help(rep)`.

One of the most versatile methods of creating vectors is to use the `c()` function, which can be used to combine smaller vectors and/or single numbers into larger vectors. The `c()` function also provides a very simple way of entering small amounts of data into R. Use this function to enter the following vectors:

```
x <- c(1, 0, 0)
y <- c(1, 2, 0)
z <- c(1, 2, 5)
```

A vector is a one-dimensional array of numbers. R can also handle two-dimensional arrays (matrices) and even more general arrays of higher dimensions (simply called arrays in R).

Vectors can be combined to form a matrix (and matrices can be combined to form multi-dimensional arrays) in a number of ways. Two of the easiest ways are through the use of the `cbind()` function and the `rbind()` function. Compare the results of the following commands:

```
cbind(x, y, z)
rbind(x, y, z)
```

Store the results of the second command as a new object, as follows:

```
mymat <- rbind(x, y, z)
```

Note that the `cbind()` version can be extracted from this new object by using the `t()` function which creates the transpose of a matrix:

```
t(mymat)
```

Other functions which can be applied to a vector, say `x`, include:

<code>length(x)</code>	number of elements in the argument <code>x</code>
<code>sort(x)</code>	sorts the elements of <code>x</code> into ascending order
<code>order(x)</code>	gives the ranks of the elements of <code>x</code>
<code>max(x), min(x)</code>	maximum or minimum of elements of <code>x</code>
<code>median(x)</code>	median value of the elements of <code>x</code>
<code>sum(x)</code>	sum of elements in <code>x</code>
<code>prod(x)</code>	product of elements in <code>x</code>
<code>mean(x)</code>	mean of all elements in <code>x</code>
<code>mean(x, trim=0.25)</code>	25% trimmed mean of all elements in <code>x</code>
<code>var(x)</code>	sample variance of the elements in <code>x</code>

Functions which can be applied to a matrix, say `amat`, include:

<code>ncol(amat)</code>	number of columns of <code>amat</code>
<code>nrow(amat)</code>	number of rows of <code>amat</code>
<code>t(amat)</code>	transpose of <code>amat</code>
<code>solve(amat)</code>	inverse of <code>amat</code>
<code>cor(amat)</code>	sample correlation matrix of <code>amat</code>
<code>apply(), sweep()</code>	row- or column-wise function application

The `apply()` and `sweep()` functions are particularly useful. The `apply()` function is used to apply the same function to each column or row of a matrix. For example, to calculate the mean of each column of `mymat`, type:

```
apply(mymat, 2, mean)
```

and to calculate the mean of each row of `mymat`, type:

```
apply(mymat, 1, mean)
```

In the above examples, the `mean()` function returns a single value so the result of the `apply()` function is a vector. When the applied function returns a vector, some care is needed. For example, to sort the rows of the matrix `mymat`, use:

```
t(apply(mymat, 1, sort))
```

The transpose is needed here because the first sorted row becomes the first column and so on. However, to sort columns, we can use:

```
apply(mymat, 2, sort)
```

The `sweep()` function subtracts values from each row or column of a matrix. Thus to get a matrix with mean-centred columns, type:

```
sweep(mymat, 2, apply(mymat, 2, mean))
```

Apart from using functions, we can also perform arithmetic with vectors and matrices. The operators `+`, `-` and `*` will work with vectors and matrices in a component-by-component fashion. Try typing the following commands to see what happens:

```
y + z  
y - z  
y * z
```

The matrix multiplication operator is `%*%` and it works in the expected fashion. As an example, type the following command, but try to work out what the result should be, before you press the return button:

```
mymat %*% solve(mymat)
```

Other operations with vectors and matrices are also possible, e.g. the outer product of two vectors. Refer to the help files on the `outer()` function or try the `%o%` operator on the vectors `y` and `z`.

We can also refer to and manipulate parts of data objects separately. For instance, suppose that we had made a mistake in entering the third element of the vector `z`, above, and it should have been a 3, not a 5. We can correct the vector `z` as follows:

```
z[3] <- 3
```

However, we also used the vector `z` when we created the matrix `mymat`, and we can also correct that mistake using:

```
mymat[3, 3] <- 3
```

In this discussion we have only looked at numeric data objects, however *R* can also store data in other modes, e.g. complex numbers, logical values and characters. For more information examine the help files on attributes and mode.

There are two more complicated ways of storing data in *R*: lists, which allow a number of different types of objects to be grouped together; and dataframes which combine some of the features of both matrices and lists.

## 5. *Entering Data into R*

Data is most easily entered into *R* importing the data using the menu items for importing data under the File menu. Data can be imported from files in a variety of formats, including

text files, Excel spreadsheets, and so on. It is also possible to read data in from a file using the command line interface, details of which are given below. For your class, all of the data you will need is already loaded into R at startup if you have followed the directions in Worksheet 1 properly. But it is still useful to be able to import data from a file, so some details are supplied here.

The `scan()` command can be used to read data from a text file (say the file `data.txt`, which doesn't really exist) into a single vector. If the data really belongs in two columns, we could rearrange this vector as a matrix by using the `matrix()` command, which "wraps" a vector into the elements of a matrix, by typing:

```
data.mat <- matrix(scan("data.txt"),ncol=2,byrow=T)
```

Note that the `ncol=2` argument designates the number of columns of the matrix we are creating and the `byrow=T` argument indicates that the elements of the vector are to be "wrapped" into the matrix by filling up the available positions row by row rather than column by column as would have occurred if the `byrow=T` argument had been left out.

Alternatively, we could read the data into a list object with components named "Fred" and "Wilma" using:

```
data.lst <- scan("data.txt",list(Fred=0,Wilma=0))
```

Yet another alternative is to edit the existing data file to include two column headings, *Fred* and *Wilma*. We could then read the file into a dataframe using the command:

```
data.df <- read.table("data.txt",header=T)
```

Suppose we have now successfully read in the appropriate data into an R object `worksheet2.women`. To find out if this object is a matrix, type

```
is.matrix(worksheet2.women)
```

and observe the response. To find out if the object is instead a data frame, type

```
is.data.frame(worksheet2.women)
```

and observe the response. To find out if the object is instead a list, type

```
is.list(worksheet2.women)
```

and observe the response. Whichever of a matrix, list or data frame the object, we can coerce it to be of the type we wish. For example, we can create a matrix from the object as

```
women.mat <- as.matrix(worksheet2.women)
```

and the columns named height and weight using

```
dimnames(women.mat) <- list(NULL,c("height","weight"))
```

or a data frame as

```
women.df <- as.data.frame(worksheet2.women)
names(women.df) <- c("height","weight")
```

or a list as

```
women.lst <- list(height=women.mat[,1], weight=women.mat[,2])
```

We can now extract the heights and weights as separate vectors from the matrix by using appropriate “square bracket” notation:

```
women.mat[,1]
women.mat[,2]
```

Here, the initial blank preceding the comma indicates that the entire column is being referenced rather than any specifically designated row as was the case when we edited the matrix `my.mat` previously. Similarly, we could indicate an entire row of a matrix by using a trailing blank following the comma.

We could extract the same vectors from the list object using either appropriate “double square bracket” notation:

```
women.lst[[1]]
women.lst[[2]]
```

or by using the `$` notation:

```
women.lst$height
women.lst$weight
```

We could also extract the same vectors from the dataframe using either the square bracket notation of a matrix:

```
women.df[,1]
women.df[,2]
```

or the `$` notation of a list:

```
women.df$height
women.df$weight
```

The most flexible of the above alternatives is usually the dataframe, as we can treat a dataframe in a manner similar to an R library by using the `attach()` function:

```
attach(women.df)
```

Having attached the dataframe, we can then refer to the named components as if they were separate R objects, i.e. simply type:

```
height
weight
```

## 6. Graphics in R

The graphics capability of R is one of its powerful features. A full description of the possibilities is way beyond the scope of the present worksheet so we will concentrate on some simple, commonly used graphical procedures. For information on how to set graphical parameters for these procedures, use `help(par)` and also see the help files for each of the graphical functions listed below.

When a graphics command is issued, R automatically opens a graphics window.

One very rudimentary graphic which doesn’t need a graphics device is a stem-and-leaf plot. To see stem-and-leaf plots for the women’s data, type:

```
stem(height)
```

```
stem(weight)
```

To plot histograms of the women's data, type:

```
hist(height)
hist(weight)
```

To obtain side-by-side boxplots the women's heights and weights, type:

```
boxplot(height,weight)
```

To get a scatterplot of `weight` versus `height`, type:

```
plot(height,weight)
```

To join the points on the plot, type:

```
plot(height,weight,type="l")
```

Other more elaborate plots will probably be introduced later in the course. Note that `plot()` is the first example you have encountered of a generic function, one which examines the object it is given to work on and tries to come up with an appropriate treatment. You will probably encounter more of these generic functions (such as the `summary()` function) later in the course.

Contrast what the generic function `plot()` does with the three different forms of the women's data:

```
plot(women.mat)
plot(women.lst)
plot(women.df)
```

Control over the plots is usually achieved by specifying specific graphical parameters as additional arguments to the `plot()` function. The most commonly used of these parameters include the labelling arguments:

```
main="...", sub="...", ylab="...", xlab="..."
```

and the plotting control arguments:

```
type=      "l" to plot lines; "p" for points; "b" for both
pch=       A numeric value specifies a specific plotting symbol:
              0=square, 1=octagon, 2=triangle,3=cross,
              4=X, 5=diamond, 6=inverted triangle, etc.
              A character in double-quotes specifies a user-chosen plotting symbol
              (e.g. "o" will plot a lower-case "o" as the plotting symbol)
lty=       1 for a solid line, 2 or higher for varying degrees of dashed or dotted lines
```

It is also possible to add to a plot once it has been drawn. For example, to add the least-squares regression line to a scatterplot of `weight` versus `height`, type:

```
plot(height,weight)
abline(lsfit(height,weight)$coef)
```

The `abline()` function can be used to include lines of the form  $y = a + bx$  on a plot, including horizontal and vertical lines. The argument to `abline` should be a vector whose first component is  $a$ , the intercept of the line, and whose second component is  $b$ , the slope. The `lsfit()` function requires two main inputs (and also allows various optional input parameters) the first being either



a vector or matrix of predictor variable values and the second being a vector of response values. The `lsfit()` returns a named list containing various relevant information about a least-squares regression of the response variable on the predictor variable(s). As noted above, we can access elements of a list using the `$` operator, and in this instance we have chosen the `coef` element of the list which is a vector containing the intercept and slope of the least-squares regression line. Note that the `lm()` function can also be used to perform least-squares regressions, so we could replace the second line above with:

```
abline(lm(weight ~ height))
```

Also available are a number of presentation plots (`barplot()`, `pie()`), and a large number of other types of plots through an assortment of free, downloadable packages from CRAN.

Before we finish this R session, try plotting some of the datasets stored in the various libraries in your search path. In particular, try:

```
plot(co2)
lines(smooth(co2),lty=2)
```

The `co2` dataset is a time series of carbon dioxide concentrations gathered at Mauna Loa in Hawaii. Try resizing the plot window to see the pattern in this data.

### 7. *Preparing for Worksheet 3*

Worksheet 3 assumes that you have covered the materials in the first two worksheets in detail. Do not attempt Worksheet 3 until you are happy that you understand the first two worksheets.