2 Variables and data types

In this section, we introduce some of the ways data are organised and handled in R. Some of the concepts here do not only apply to R, but can be applied in some form to other computer languages.

2.1 Variables

Variables are identifiers of items that we have stored in the computer's memory. These items could be a single number, a letter, a word, a series of values or a whole table of value. We can demonstrate this by storing a number (e.g., 2) in a variable called `a`. We do this with the assignment operator is `<-`, such as in

```
> a <- 2
```

We can now ask R to give us the value of the variable `a` by simply typing its name

```
> a
 [1] 2
```

We use variable to store handle information. For example, we can do some simple calculations.

```
> a*a
 [1] 4
> a^3
 [1] 8
> a^0.5
 [1] 1.414214
```

We can define another variable, `b`, to use in the calculations, and yet another, `c`, one to store the result,

```
> b <- 10
> c<-a*b
```

Typing just `c` will show us the result, stored by the variable

```
> c
 [1] 20
```

Exercise 2.1

1. Use R to calculate the value of the Golden Ratio, $(1 + \sqrt{5})/2$. Hint: Use the fact that $\sqrt{5} = 5^{1/2}$.
   Solution:
   We use the following R code

The *Golden Ratio* is a number that has been studied for centuries (and involved in a large amount of folklore). We shall see how it is obtained from a sequence of numbers known as the Fibonacci sequence Wikipedia: Golden Ratio.

```
> (1 + 5^0.5)/2
 [1] 1.618034
```

2. By assigning values to variables `a` and `b` examine the result of the expression `a*b+a`. Which operation, $\times$ or $+$, is performed first?
   Solution:
   The multiply operation is performed first, and then the addition operation (we say multiplication takes precedence over addition). If we wish to perform the addition operation first we must use brackets as follows.

```
> a <- 2
> b <- 3
> a*b+a
 [1] 8
> a*(b+a)
 [1] 10
```

For more information on the available operators and their precedence, see this link.
R manual : Syntax

2.2 Vectors

Vectors are series of items that are stored in a single variable. Each item is an element of the vector and can be accessed by its position in the series. Vectors are found in most programming languages and are also sometimes called arrays.

Single-value variables, such as the ones we had defined above (`a`,`b` and `c`), were in fact vectors of length one. Thus, when printing them out, R put a `[1]` at the start. This stated that the result is a vector with the first element containing the value.

We can create vectors with the command `c`, which stands for 'combine'. The function `c()` takes as arguments in parentheses the elements to be combined into a vector, separated by commas. Using this command, we can create a vector containing a series of numbers,

```
> x <- c(12, 24, 36, 48, 60)
> x
 [1] 12 24 36 48 60
```

We can access a given element of a vector using an **index**. This is an integer which refers to the position of the element in the vector. In R, the first element in the array has an index of 1, so to access the first element of our vector `x`, we use the square brackets `[ ]` operator as follows.

Be warned! In some languages *e.g.* C, C++, FORTRAN, Perl and Python, the first element has an index of 0.

```
> x[1]
 [1] 12
```

Similarly, to access the third element of the vector use an index of 3.

```
> x[3]
 [1] 36
```

Let's try and access an array element that does not yet exist. For example, if we try to access the sixth element of `x` we get the following.

```
> x[6]
 [1] NA
```

R returns the `NA` value which stands for **missing data**. R has lots of mechanisms for handling missing data which can be very useful when performing statistical analyses but we won't go into detail here. We can create the sixth element using a simple assignment.

```
> x[6] <- 72
> x
 [1] 12 24 36 48 60 72
```

R has created the sixth element and assigned it the numerical value of 72. The vector is now 6 elements long.

We can perform all the previous numerical calculations using vector elements. For example the following code adds together the fifth and sixth elements of `x`.

```
> x[5] + x[6]
 [1] 132
```

The combine operator can also join two vectors. For example consider the following code.

```
> a <- c(1,2,3)
> b <- c(4,5,6)
> x <- c(a,b)
> x
 [1] 1 2 3 4 5 6
```

R has shortcuts to generate vectors filled with sequences of numbers which can be very useful. For integer sequences, we can use the **colon operator**.

```
> 1:8
 [1]  1  2  3  4  5  6  7  8
```

Alternatively, we can use the `seq` function and specify the start and end and then increment as follows.

```
> seq(from=1,to=4,by=0.5)
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

To find the size of a vector of any type, use the `length` function.

```
> x <- 1:6
> length(x)
 [1] 6
```

Remember: to access a vector element in R use the square `[ ]` operator.

Note: R uses **square brackets** [ ] to access elements in an array whereas MATLAB uses **round brackets ( )**.

Just as we have done with single-value variables above, we can perform calculations with vectors. For example, we can add a constant value to the vector. As you can see below, this will add the constant to each one of the elements:

```
> x+2
 [1] 3 4 5 6 7 8
```

Similarly, we can multiply each element of the vector by a constant:

```
> x*3
 [1] 3 6 9 12 15 18
```

One extremely useful aspect to R when doing maths with vectors is the concept of **element-wise** operations. When certain operations are given vectors of the same size, R interprets this as a repeated operation on each element in turn. Examples of operations that perform this are addition, subtraction and multiplication. Consider the following code.

```
> x <- c(1,2,3,4)
> y <- c(5,6,7,8)
> x + y
 [1]  6  8 10 12
> x - y
 [1] -4 -4 -4 -4
> x*y
 [1]  5 12 21 32
```

We can see that in all three cases, the operation acts on each element in turn. This is what we mean by element-wise operations.

Exercise 2.2

1. Build a vector containing the first ten integers.
   Solution:
   `> x <- 1:10`

2. Create a vector containing the real numbers between 0 and 1 in steps of 0.1.
   Solution:
   `> seq(from=0,to=1,by=0.1)`

2.3 Matrices

Matrices are conceptually similar to vectors, except that they are structure in two dimensions, with rows and columns. Also, like vectors, they contain values of a same type, for example numbers (for tables that contain different types of data, see the section on data frames below). We won't cover all the details of matrices in R but just mention them in

passing. There are many functions for performing matrix operations and linear algebra.

We can define a matrix as follows.

```
> m <- matrix( nrow=3, ncol=3, data=c(1,2,3,4,5,6,7,8,9) )
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Here we have specified the number of rows (`nrow`), the number of columns (`ncol`) and the entries in the matrix (via the `data` argument). Note that the matrix is filled column-wise by default but we can fill row-wise by adding the `byrow=T` flag.

```
> m <- matrix( nrow=3, ncol=3, data=c(1,2,3,4,5,6,7,8,9), byrow=T )
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

If you look closely at the way R printed the contents of the matrix `m`, you see that it added coordinates in square brackets. You see that the first line is indexed with `[1,]` and the first column with `[,1]`. In general, the elements of a matrix (and of their more sophisticated cousins, data frames, that we'll see later) are accessed by [row number, column number]. So to pick out the entry 6 from the matrix `m` above, we say

```
> m[2,3]
[1] 6
```

(Note that the output is now indexed with `[1]`, the matrix element we have pulled out is a single value, and hence a vector with length one.)

In line with the indexing of the table rows and columns above, we can select an entire row or an entire column. When selecting an entire row we leave the column index blank, such as in `[1,]` (the entire first row). Similarly, when selecting an entire column we leave the row index blank, e.g., `[,2]`.

Exercise 2.3

1. Create a 5×5 matrix containing all the integers from 1 to 25. Fill this column wise and row wise.
   Solution:
   Column-wise:
   `> m <- matrix(nrow=5,ncol=5,data=1:25)`
   Row-wise:

```
> m <- matrix(nrow=5,ncol=5,data=1:25,byrow=T )
```

2.4 Data types

In a program, we often wish to store different types of data. We have already seen how to store and manipulate numerical data but there are other possibilities such as character vectors (also known as strings) or logical information (true and false). R contains these basic data types plus it also implements more sophisticated **containers** for holding data known as **data frames** which we shall look at in Section 4.

In R, there are three basic data types

- numeric: integer or real
- character: strings
- logical: `TRUE` and `FALSE`

Here are examples of the three types. We can use the `class` function to get information on variables.

```
> x <- 5
> y <- "hello world"
> z <- TRUE
> class(x)
 [1] "numeric"
> class(y)
 [1] "character"
> class(z)
 [1] "logical"
```

Strings are suites of characters that are stored and treated as one entity. They are a bit like a word, except that a string could also contain spaces and hence comprise several words (thus, "hello", "world" and "hello world" are each a string). In R, vectors of strings are treated in a similar way to numerical vectors, with each string being one element of the vector. For example, we can use the combine operation.

```
> y1 <- c("hello", "world")
> y1
 [1] "hello" "world"
```

However, note the difference between `y` and `y1` defined immediately above. `y1` has a length of 2 as it contains the string "hello" in the first element and "world" in the second.

Going through the previous pages, you have created a lot of variables. A useful command to display their names and make an inventory of the variables currently held in the workspace is `ls()`.

```
> ls()
 [1] "x" "y" "y1" "z"
```

This can be used to keep track of the variables that we have created so far. A nice feature of the R console is the Workshop Browser which is accessed in the menu by clicking on `Workspace -> Workspace Browser`. This gives the data type and allows browsing of more complex data structure such as data frames which are discussed in Section 4.

It is also possible to save your entire workspace, including all variables you have created. When quitting R, you will be asked whether you would like to save a workspace image. If you agree, this will create an invisible file in your working directory that contains everything you need to continue working where you left off last time. You can also create the same file at any point by typing the command `save.image()` or go to the menu item `Workspace -> Save Default Workspace`. If you'd rather see your file, the easiest is to go to the menu item `Workspace -> Save Workspace File...` and specify a file name.

In order to retrieve the data that you saved, you need to load your saved workspace. If you saved an invisible file, do this by setting the right working directory and then choosing `Workspace -> Load Default Workspace`. Otherwise you cal load a named file with `Workspace -> Load Workspace File...`. Loading the file will bring back any variables you created in the previous session(s), as well as the command history (allowing you to call up past commands using the 'up' arrow).

2.5 Logical data

The logical data type takes the value `TRUE` or `FALSE`. This type is returned from certain comparison operations. For example, "equal" (`==`), "greater than" (`>`) and "less than" (`<`).

This is also known a Boolean type.

```
> a <- 2
> a > 1
 [1] TRUE
> a < 1
 [1] FALSE
```

Here on line 2, we ask the question "is `a` greater than 1" which equates to `TRUE`. On the next line, we ask the question "is `a` less than 1" which equates to `FALSE`. There are both "equals", `==`, and "not-equals", `!=` comparison operators.

```
> a == 2
[1] TRUE
> a != 1
[1] TRUE
```

Here, on line 1, we ask the question "is `a` equal to 2" which equates to `TRUE`. On line 5, we ask the question "is `a` not equal to 1" which equates to `true`.

There is also a "greater than or equal" operator, `>=`, and a "less than or equal" operator,`<=`.

Note that when performing a comparison where we wish to test for equality, we use the double-equals symbol `==` and **not** the single equals `=` since this is an assignment. This is a common cause of bugs in programs! For example look at the following code.

```
> a <- 2
> a == 1
 [1] FALSE
> a
 [1] 2
> a = 1
> a
 [1] 1
```

On line 2 we correctly check to see if the value of `a` is equal to 1. On line 4 we make the mistake of using an assignment operator `=` which simply sets the value of `a` to 1.

Just as we have applied logical operators to single-value variables, we can apply them to vectors or matrices. For example, we could ask which element of a numerical vector is equal to a particular value, as in

```
> a <- c(1:3)
> a == 2
 [1] FALSE TRUE FALSE
```

Just as in mathematical operations with vectors, the logical operator is applied to each element in turn and the out come returned.

Logical operators are useful to select data based on certain criteria, for example by using the function `subset()`. This function takes as arguments a variable from which a subset is to be selected and the condition that selected values need to fulfil. We could for example apply a cut-off to values in a vector, such as in

```
> a <- c(1:6)
> a
[1] 1 2 3 4 5 6
> subset(a,a<4)
[1] 1 2 3
```

This example is slightly contrived, but `subset` can be very useful when applied to entire datasets, in order to analyse particular sets of observations (for example all subjects that are male, all measurements that less than 0 etc.)

Logical operators are also frequently used in programming, for example to perform different tasks depending on the values of input variables. The most common way to do this is in combination with the `if` statement, which is covered in the advanced material in Section 7.3.