2 Simple manipulations; numbers and vectors

2.1 Vectors and assignment

R operates on named *data structures*. The simplest such structure is the numeric *vector*, which is a single entity consisting of an ordered collection of numbers. To set up a vector named `x`, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

This is an *assignment* statement using the *function* `c()` which in this context can take an arbitrary number of vector *arguments* and whose value is a vector got by concatenating its arguments end to end.[1]

A number occurring by itself in an expression is taken as a vector of length one.

Notice that the assignment operator (‘`<-`’), which consists of the two characters ‘`<`’ (“less than”) and ‘`-`’ (“minus”) occurring strictly side-by-side and it ‘points’ to the object receiving the value of the expression. In most contexts the ‘`=`’ operator can be used as an alternative.

Assignment can also be made using the function `assign()`. An equivalent way of making the same assignment as above is with:

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

The usual operator, `<-`, can be thought of as a syntactic short-cut to this.

Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

If an expression is used as a complete command, the value is printed *and lost*[2]. So now if we were to use the command

```
> 1/x
```

the reciprocals of the five values would be printed at the terminal (and the value of `x`, of course, unchanged).

The further assignment

```
> y <- c(x, 0, x)
```

would create a vector `y` with 11 entries consisting of two copies of `x` with a zero in the middle place.

2.2 Vector arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are *recycled* as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
> v <- 2*x + y + 1
```

generates a new vector `v` of length 11 constructed by adding together, element by element, `2*x` repeated 2.2 times, `y` repeated just once, and `1` repeated 11 times.

---

[1] With other than vector types of argument, such as `list` mode arguments, the action of `c()` is rather different. See Section 6.2.1 [Concatenating lists], page 27.

[2] Actually, it is still available as `.Last.value` before any other statements are executed.

The elementary arithmetic operators are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, and so on, all have their usual meaning. `max` and `min` select the largest and smallest elements of a vector respectively. `range` is a function whose value is a vector of length two, namely `c(min(x), max(x))`. `length(x)` is the number of elements in `x`, `sum(x)` gives the total of the elements in `x`, and `prod(x)` their product.

Two statistical functions are `mean(x)` which calculates the sample mean, which is the same as `sum(x)/length(x)`, and `var(x)` which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

or sample variance. If the argument to `var()` is an n-by-p matrix the value is a p-by-p sample covariance matrix got by regarding the rows as independent p-variate sample vectors.

`sort(x)` returns a vector of the same size as `x` with the elements arranged in increasing order; however there are other more flexible sorting facilities available (see `order()` or `sort.list()` which produce a permutation to do the sorting).

Note that `max` and `min` select the largest and smallest values in their arguments, even if they are given several vectors. The *parallel* maximum and minimum functions `pmax` and `pmin` return a vector (of length equal to their longest argument) that contains in each element the largest (smallest) element in that position in any of the input vectors.

For most purposes the user will not be concerned if the "numbers" in a numeric vector are integers, reals or even complex. Internally calculations are done as double precision real numbers, or double precision complex numbers if the input data are complex.

To work with complex numbers, supply an explicit complex part. Thus

```
sqrt(-17)
```

will give `NaN` and a warning, but

```
sqrt(-17+0i)
```

will do the computations as complex numbers.

2.3 Generating regular sequences

R has a number of facilities for generating commonly used sequences of numbers. For example `1:30` is the vector `c(1, 2, ..., 29, 30)`. The colon operator has high priority within an expression, so, for example `2*1:15` is the vector `c(2, 4, ..., 28, 30)`. Put `n <- 10` and compare the sequences `1:n-1` and `1:(n-1)`.

The construction `30:1` may be used to generate a sequence backwards.

The function `seq()` is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is `seq(2,10)` is the same vector as `2:10`.

Arguments to `seq()`, and to many other R functions, can also be given in named form, in which case the order in which they appear is irrelevant. The first two arguments may be named `from=`*value* and `to=`*value*; thus `seq(1,30)`, `seq(from=1, to=30)` and `seq(to=30, from=1)` are all the same as `1:30`. The next two arguments to `seq()` may be named `by=`*value* and `length=`*value*, which specify a step size and a length for the sequence respectively. If neither of these is given, the default `by=1` is assumed.

For example

```
> seq(-5, 5, by=.2) -> s3
```

generates in `s3` the vector `c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)`. Similarly

```
> s4 <- seq(length=51, from=-5, by=.2)
```

generates the same vector in `s4`.

The fifth argument may be named `along=`*`vector`*, which is normally used as the only argument to create the sequence `1, 2, ..., length(`*`vector`*`)`, or the empty sequence if the vector is empty (as it can be).

A related function is `rep()` which can be used for replicating an object in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

which will put five copies of `x` end-to-end in `s5`. Another useful version is

```
> s6 <- rep(x, each=5)
```

which repeats each element of `x` five times before moving on to the next.

2.4 Logical vectors

As well as numerical vectors, R allows manipulation of logical quantities. The elements of a logical vector can have the values `TRUE`, `FALSE`, and `NA` (for "not available", see below). The first two are often abbreviated as `T` and `F`, respectively. Note however that `T` and `F` are just variables which are set to `TRUE` and `FALSE` by default, but are not reserved words and hence can be overwritten by the user. Hence, you should always use `TRUE` and `FALSE`.

Logical vectors are generated by *conditions*. For example

```
> temp <- x > 13
```

sets `temp` as a vector of the same length as `x` with values `FALSE` corresponding to elements of `x` where the condition is *not* met and `TRUE` where it is.

The logical operators are `<`, `<=`, `>`, `>=`, `==` for exact equality and `!=` for inequality. In addition if `c1` and `c2` are logical expressions, then `c1 & c2` is their intersection (*"and"*), `c1 | c2` is their union (*"or"*), and `!c1` is the negation of `c1`.

Logical vectors may be used in ordinary arithmetic, in which case they are *coerced* into numeric vectors, `FALSE` becoming `0` and `TRUE` becoming `1`. However there are situations where logical vectors and their coerced numeric counterparts are not equivalent, for example see the next subsection.

2.5 Missing values

In some cases the components of a vector may not be completely known. When an element or value is "not available" or a "missing value" in the statistical sense, a place within a vector may be reserved for it by assigning it the special value `NA`. In general any operation on an `NA` becomes an `NA`. The motivation for this rule is simply that if the specification of an operation is incomplete, the result cannot be known and hence is not available.

The function `is.na(x)` gives a logical vector of the same size as `x` with value `TRUE` if and only if the corresponding element in `x` is `NA`.

```
> z <- c(1:3,NA);  ind <- is.na(z)
```

Notice that the logical expression `x == NA` is quite different from `is.na(x)` since `NA` is not really a value but a marker for a quantity that is not available. Thus `x == NA` is a vector of the same length as `x` *all* of whose values are `NA` as the logical expression itself is incomplete and hence undecidable.

Note that there is a second kind of "missing" values which are produced by numerical computation, the so-called *Not a Number*, `NaN`, values. Examples are

```
> 0/0
```

or

```
> Inf - Inf
```

which both give NaN since the result cannot be defined sensibly.

In summary, is.na(xx) is TRUE *both* for NA and NaN values. To differentiate these, is.nan(xx) is only TRUE for NaNs.

Missing values are sometimes printed as <NA> when character vectors are printed without quotes.

2.6 Character vectors

Character quantities and character vectors are used frequently in R, for example as plot labels. Where needed they are denoted by a sequence of characters delimited by the double quote character, e.g., "x-values", "New iteration results".

Character strings are entered using either matching double (") or single (') quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using \ as the escape character, so \\ is entered and printed as \\, and inside double quotes " is entered as \". Other useful escape sequences are \n, newline, \t, tab and \b, backspace—see ?Quotes for a full list.

Character vectors may be concatenated into a vector by the c() function; examples of their use will emerge frequently.

The paste() function takes an arbitrary number of arguments and concatenates them one by one into character strings. Any numbers given among the arguments are coerced into character strings in the evident way, that is, in the same way they would be if they were printed. The arguments are by default separated in the result by a single blank character, but this can be changed by the named argument, sep=*string*, which changes it to *string*, possibly empty.

For example

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

makes labs into the character vector

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Note particularly that recycling of short lists takes place here too; thus c("X", "Y") is repeated 5 times to match the sequence 1:10.[3]

2.7 Index vectors; selecting and modifying subsets of a data set

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.

1. **A logical vector**. In this case the index vector is recycled to 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

   ```
   > 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

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

   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.

[3] paste(..., collapse=*ss*) joins the arguments into a single character string putting *ss* in between, e.g., ss <- "|". There are more tools for character manipulation, see the help for sub and substring.

2. **A vector of positive integral quantities**. In this case the values in the index vector must lie in the set {1, 2, . . . , `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]
   ```

   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)]
   ```

   (an admittedly unlikely thing to do) produces a character vector of length 16 consisting of `"x", "y", "y", "x"` repeated four times.
3. **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`.
4. **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.

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)
```

2.8 Other types of objects

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

- *matrices* or more generally *arrays* are multi-dimensional generalizations of vectors. In fact, they *are* vectors that can be indexed by two or more indices and will be printed in special ways. See Chapter 5 [Arrays and matrices], page 18.
- *factors* provide compact ways to handle categorical data. See Chapter 4 [Factors], page 16.
- *lists* are a general form of vector in which the various elements need not be of the same type, and are often themselves vectors or lists. Lists provide a convenient way to return the results of a statistical computation. See Section 6.1 [Lists], page 26.
- *data frames* are matrix-like structures, in which the columns can be of different types. Think of data frames as 'data matrices' with one row per observational unit but with (possibly)

both numerical and categorical variables. Many experiments are best described by data frames: the treatments are categorical but the response is numeric. See Section 6.3 [Data frames], page 27.

- *functions* are themselves objects in R which can be stored in the project's workspace. This provides a simple and convenient way to extend R. See Chapter 10 [Writing your own functions], page 42.