Chapter 2

An Introduction to R

2.1 Computing and Graphics

The introduction of cheap, powerful computers has brought about a revolution in the production of graphs. In the past, the production of a quality graph required that someone with special skills spend considerable time drawing it by hand. Now, even novices have access to software tools which can be used to produce high-quality graphs. These tools range from those provided with spreadsheets and general purpose statistical packages, to specialised high-end, presentation graphics and visualisation software.

Unfortunately, although modern software makes it possible to produce good graphs; it doesn't always make it easy. Many programs err by presenting a restricted set of graph types for their users to choose between. This works well if the desired graph fits into one of those categories, but if this is not the case, a user can be forced into creating an inappropriate graph.

In these notes, the software tool we will discuss is a computer language called R, which is a dialect of the S language developed at Bell Laboratories[1] in New Jersey. Not only does R have a large number of pre-built plots, but because it is a complete programming language, it allows users to build completely new types of plots.

2.2 Getting Started With R

Rather than talk abstractly about the language, let's see how it works. The first step in doing this, is to start the R program running. How you do this depends on what type of computer you have. You will find sort summary of how to start R on most common computer systems at the end of these notes.

When R is running it will *prompt* you to type some input. It does this by printing a *greater than* sign.

```
>
```

In the examples that follow, any text which appears on a line which begins with ">" will be what has been typed to R, and anything else will be what the computer types

[1] The research institution who developed the transistor, produced the evidence for the "Big Bang" creation theory, and provided much of the technology found in modern computing systems.

back[2].

A good way to start thinking about R is as an extremely powerful calculator. As the simplest example, let's tell the computer to add 1 and 2.

```
> 1+2
[1] 3
```

The computer responds by typing 3. The preceding bracketed 1 shows you that 3 is the first (and in this case the only) number produced by R.

All the elementary arithmetic operations are available in R. For example, division

```
> 1/2
[1] 0.5
```

and raising to a power.

```
> 17^2
[1] 289
```

(here we have squared seventeen).

The usual rules of arithmetic apply — multiplication and division take place before addition and subtraction and otherwise expressions are evaluated left to right.

```
> 1+2*3
[1] 7
```

The order of evaluation can be altered by using parentheses.

```
> (1+2)*3
[1] 9
```

Most of the elementary mathematical functions are available, for example, square roots and logarithms.

```
> sqrt(2)
[1] 1.414214
> log(10)
[1] 2.302585
```

You can find a summary of some of R's basic arithmetic capabilities, including functions, in table 2.1.

Now that we have a means of performing computations, we need a way to store the results. The values produced by R can be stored by assigning names to them. R uses = to denote assignment. For example,

```
> z = 17
```

stores the value 17 under the name z.

An assignment like the one above creates a *variable* and gives it a value. In the example, the name of the variable is z and its value is 17. If a variable with the given name exists before the assignment, then the effect of the assignment is to change the variable's value.

Variables can be used in expressions in the same way as numbers. For example,

[2]Well, almost everything else. Sometimes R sees that a command is incomplete and it will ask for additional input by typing a plus "+" sign at the start of a line.

Table 2.1: Basic R functions

Assignment	
`y = x`	*assignment (y gets the value of x)*
Arithmetic Operators	
`+`	*addition*
`-`	*subtraction*
`*`	*multiplication*
`/`	*division*
`^`	*exponentiation (raising to a power)*
Elementary Functions	
`log(x)`	*natural logarithms (base e)*
`log10(x)`	*common logarithms (base 10)*
`sqrt(x)`	*square roots*
Trigonometric Functions	
`cos(x)`	*cosine*
`sin(x)`	*sine*
`tan(x)`	*tangent*
`acos(x)`	*arccosine*
`asin(x)`	*arcsine*
`atan(x)`	*arctangent*

```
> z+23
[1] 40
```

In addition to ordinary numbers, R has special codes which indicate infinite and undefined numerical values.

```
> 1/0
[1] Inf

> -1/0
[1] -Inf

> 0/0
[1] NaN
```

In addition to real numbers, R also has complex numbers, logical values and character strings. We won't make use of complex numbers, but logical and character string values will be important.

```
> 10 > 20
[1] F
```

```
> s = "hello"
> s
[1] "hello"
```

2.3 Vectors

2.3.1 Simple Vectors

So far we've used R to work with individual values, but it is also possible to use it manipulate collections of values. One way of creating such a collection is to use the `c` function to combine the values into a *vector*.

```
> x = c(1,2,4,3,1)
> x
[1] 1 2 4 3 1
```

Vectors can be manipulated in the same way as individual values.

```
> 2*x
[1] 2 4 8 6 2
> x/4
[1] 0.25 0.50 1.00 0.75 0.25
```

One function which is useful in connection with vectors is `length`, which returns the number of values contained in a vector.

```
> length(x)
[1] 5
```

Individual elements of vectors can be accessed with a subsetting mechanism. The 5-th element of `x` can be extracted as follows.

```
> x[5]
[1] 1
```

The R subsetting mechanism is actually much more general than this. It is possible to extract several values from a vector by using a vector of subscripts.

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

If the subscripts are all negative, then all elements except those are extracted.

```
> x[-5]
[1] 1 2 4 3
```

Is is also possible to extract values by using logical conditions. For example, the command `x[x>2]` extracts all elements from `x` which are greater than 2.

```
> x[x>2]
[1] 4 3
```

2.3.2 Patterned Sequences

R has a number of ways of generating vectors containing special sequences of values. The one that is most commonly used is the sequence operator "`:`". The expression n_1`:`n_2 returns the sequence of integer values from n_1 to n_2.

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

The vectors created in this way can be quite large, and when printed they may span several lines.

```
> 1:50
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[31] 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
[46] 46 47 48 49 50
```

The value in brackets at the start of each line gives the index of the first value on the line. This can make it easier to locate particular values.

More general sequences can be created by the "`seq`" function. The expression `seq(0,5,by=0.2)` generates the sequence of values from 0 to 5 in steps of 0.2.

```
> seq(0,5,by=0.2)
 [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0
[12] 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0 4.2
[23] 4.4 4.6 4.8 5.0
```

It is also possible to create sequences of a specified length.

```
> seq(0,4,length=9)
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

Another function which is useful for creating patterned sequences is the function "`rep`", which repeats its first argument according to the value of its second. The second argument can either be a single count, giving the number of times to repeat the first,

```
> rep(1:3,5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

or it can be a vector of counts, indicating how many times to repeat each element of the first argument.

```
> rep(1:3,c(3,4,3))
 [1] 1 1 1 2 2 2 2 3 3 3
```

2.3.3 Arithmetic on Vectors*

In section 2.3.1, we saw some very simple examples of arithmetic using vectors (mutiplying a vector by a number). R has very general capabilities for vector arithmetic. Arithmetic operations are carried out on vectors according to an *element recycling rule*.

Under this rule, when vectors of different lengths are combined in an arithmetic operation, the shorter vector is first enlarged to match the length of the longer vector by recycling its elements. Then the vectors are combined element by element.

As an example, consider adding the vectors `1:3` and `1:6`. Since the vectors have different lengths, we must enlarge the shorter one. Three additional values are required, so we recycle the first three elements of the shorter vector. The process takes place as follows:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \overset{\textit{recycle}}{\Longrightarrow} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} \overset{\textit{add}}{\Longrightarrow} \begin{bmatrix} 2 \\ 4 \\ 6 \\ 5 \\ 7 \\ 9 \end{bmatrix}$$

This is how the result appears when computed with R.

```
> 1:3+1:6
[1] 2 4 6 5 7 9
```

While it is possible to use arithmetic operations combine vectors of any lengths, it usually only makes sense when the length of the longer one is a multiple of the length of the shorter one. In fact, R issues a warning message if this is not the case.

2.3.4 Summary Operations on Vectors

Given a set of values stored in a vector, it is quite common to want compute the sum, product, minimum, or maximum of those values. R provides a number of functions which provide this kind of summary. A number of them are given in the table below.

`sum(x)`	the sum of the elements of `x`
`prod(x)`	the product of the elements of `x`
`min(x)`	the minimum of the elements of `x`
`max(x)`	the maximum of the elements of `x`

As a typical example of how these functions might be used, consider computing factorials. A simple way of computing 10! is given by

```
> prod(1:10)
[1] 3628800
```

In fact, the functions above will all operate on an arbitrary number of arguments. This means that we could compute

$$\binom{10}{3} = \frac{10!}{3! \times 7!}$$

as follows

```
> prod(1:10) / prod(1:3, 1:7)
[1] 120
```

In addition to the simple summary functions above, there are also cumulative versions called `cumsum`, `cumprod`, `cummax`, and `cummin`, which produce a vector which consists of the summary computed for; the first element, first two elements, the first three elements and so on.

```
> cumsum(1:10)
 [1]  1  3  6 10 15 21 28 36 45 55

> cumprod(1:10)
 [1]       1       2       6      24     120     720
 [7]    5040   40320  362880 3628800
```

There are a number of summaries which compute quantities of statistical interest. The most important of these are:

`mean(x)`	the mean of the elements of `x`,
`median(x)`	the median of the elements of `x`,
`sd(x)`	the standard deviation of the elements of `x`.

Finally, the `quantile` function computes summaries based on the percentiles of the values in its argument. The simplest use of `quantile` computes the median, upper and lower quartiles, and extremes.

```
> quantile(1:10)
   0%   25%   50%   75%  100%
 1.00  3.25  5.50  7.75 10.00
```

(Note that there are many, slightly different definitions of upper and lower quartile, so these values might not match what you would compute by hand.) An optional second argument to `quantile` can be used to specify a different set of quantiles.

```
> quantile(1:10, 0:10/10)
  0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
 1.0  1.9  2.8  3.7  4.6  5.5  6.4  7.3  8.2  9.1 10.0
```

2.3.5 Character Vectors

All the examples we've presented so far have used numerical values. R has a number of additional data types. The most important of these is the *character* type. Any value in quotes is regarded by R as being a character string.

```
> greeting = "hello, world"
> greeting
[1] "hello, world"
```

This is true, even when the value appears to be numeric.

```
> number = "21"
> number
[1] "21"
```

As with numbers, it is possible to assemble character strings into vectors using the "`c`" function.

```
> c(greeting, number)
[1] "hello, world" "21"
```

It is also possible to combine numbers and character strings into a single vector using "`c`". In this case, the numbers are first converted to strings.

```
> c(greeting,100)
[1] "hello, world" "100"
```

Arithmetic on a mixture of strings and numbers does not work, however.

```
> number+10
Error in number + 10 : non-numeric argument
                                to binary operator
```

Character strings are mainly used for things like labelling graphs. They can also be used to name the individual elements of a vector.

```
> x = 1:5
> x
[1] 1 2 3 4 5
> l = c("a","b","c","d","e")
> l
[1] "a" "b" "c" "d" "e"
> names(x) = l
> x
a b c d e
1 2 3 4 5
```

When a named vector prints, the names of the elements are printed above the elements and the indexing information at the start of a line is dropped.

Names are preserved during subsetting,

```
> x[4:5]
d e
4 5
```

and it is possible to extract subsets by using the names as subscripts.

```
> x[c("a","d")]
a d
1 4
```

2.4 Lists

Vectors are by far the most common objects encountered in R, but sometimes the requirement that all elements have the same type is too restrictive. There is another structure called a list which can be used to group objects of different types into a single object.

Lists are created with the `list` function. The function assembles a list from its arguments. The statment

```
> L = list(1:3, "hello")
```

creates a list containing two elements and assigns it to the variable `L`. This first is a numeric vector with two elements and the second is a character string. The structure of lists is reflected in the way they print — the first element is preceded by `[[1]]`, the second by `[[2]]` etc.

```
> L
[[1]]
[1] 1 2 3

[[2]]
[1] "hello"
```

When a list is created, it is possible to name some or all of its elements. This is done as by naming the arguments in the call to `list`.

```
> L = list(a=1:3, b="hello")
```

The names of a list's elements are used in printing.

```
> L
$a
[1] 1 2 3

$b
[1] "hello"
```

The only important operation which can be performed on a list is the extraction of a sublist, or element. Sublists are extracted with `[ ]` and indiviual elements with `[[ ]]`. It is important to recognise that these return different results.

```
> L[1]
$a
[1] 1 2 3

> L[[1]]
[1] 1 2 3
```

List elements can also be extracted by name. This is actually the most common way of extracting elements.

```
> L$a
[1] 1 2 3
```

The function `names` can be used to obtain the names of the the elements of a list.

```
> names(L)
[1] "a" "b"
```

2.5 Matrices and Arrays

2.5.1 Matrices

In addition to vectors, R has a wide range of data structures. Some of the most commonly used data structures in statistics are matrices. A matrix is a set of values laid out in a regular row×column arrangement. The R function `matrix` takes a vector of values and turns them into a matrix. for example, the expression

```
> A = matrix(1:6, nrow=3, ncol=2)
```

creates a 3×2 matrix. The value can be viewed as follows.

```
> A
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Notice that the values have been placed into the matrix running down successive columns. It is also possible to specify that the matrix be filled by rows.

```
> B = matrix(1:6, nrow=3, ncol=2, byrow=TRUE)
> B
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

The dimensions of a matrix can be obtained in a number of ways.

```
> nrow(A)
[1] 3
> ncol(A)
[1] 2
> dim(A)
[1] 3 2
```

The standard arithmetic operations are all defined for matrices, and take place elementwise.

```
> A+B
     [,1] [,2]
[1,]    2    6
[2,]    5    9
[3,]    8   12
```

In particular, note that `A*B` is the elementwise product of `A` and `B`, not the matrix product.

There are a number of other special matrix operations which are available. The function `t` computes the transpose of its argument.

```
> t(A)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Matrix multiplication can be performed with the `%*%` binary operator.

```
> t(A) %*% B
     [,1] [,2]
[1,]   22   28
[2,]   49   64
```

It it also possible to solve systems of linear equations with the `solve` function. For example the linear system

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

can be solved as follows

```
> A = matrix(c(1, 3, 2, 4), ncol = 2)
> b = c(1, 1)
> solve(A, b)
[1] -1  1
```

If no "right hand side" is given as an argument to `solve` then it is assumed to be an identity matrix of the same size as the "left hand side" coefficient matrix. This results in the computation of the inverse of the coefficient matrix.

```
> solve(A)
     [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

You do, however, need to remember that computers only work to a finite precision and that all computations are subject to roundoff error.

```
> A %*% solve(A)
     [,1]          [,2]
[1,]    1 -4.440892e-16
[2,]    0  1.000000e+00
```

There are many other R functions which support computations on matrices.

`diag`	create a diagonal matrix or extract matrix diagonal
`eigen`	spectral decomposition (eigenvalues/eigenvectors)
`svd`	singular-value decomposition
`qr`	QR decomposition

2.5.2 Subsetting

It is possible to extract submatrices from matrices in a similar way to that for extracting subsets of vectors. The following expression extracts the element from the first row and second column of the matrix `A`.

```
> A[1,2]
```

If vector subscripts are used, then more general submatrices can be extracted. For example, the expression

```
> A[1:2,1:2]
```

extracts the leading 2×2 submatrix of `A`.

If it is desired to extract the full range of subscript values for either rows or columns of a matrix, then that field of a subsetting expression can be left empty. For example:

```
> A[,2:3]
```

extracts the second and third columns of A and

```
A[2:3,]
```

extracts the second and third rows. Subscripting matrices by using logical indices and row or column labels is also possible, but tends to be far less common.

One idiom which is quite common is based on the `row` and `col` functions, which return matrices which have the same shape as their arguments, but are filled with the row or column indices for the matrix.

```
> A = matrix(1, nrow=2, ncol=3)
> A
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
> row(A)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
> col(A)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
```

These can used to access the upper and lower triangles of a matrix.

```
> A[col(A) > row(A)] = 0
> A
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    1    1    0
```

2.5.3 Row and Column Labelling

Matrices can be made rather more useful by using row and column labels. A matrix can be labelled as follows:

```
> A = matrix(1:6, nrow=3)
> dimnames(A) = list(c("sex","drugs","rock&roll"),
+                    c("this", "that"))
```

The primary benefit of labelling can be seen when the matrix is printed.

```
> A
           this that
sex           1    4
drugs         2    5
rock&roll     3    6
```

In R but not S, it is possible to “label the labels”, which can also be useful.

```
> A = matrix(1:6, nrow=3)
> dimnames(A) = list(what=c("sex","drugs","rock&roll"),
```

```
+                     which=c("this", "that"))
> A
           which
what        this that
  sex          1    4
  drugs        2    5
  rock&roll    3    6
```

2.5.4 Arrays

Multiway arrays generalise the notion of matrices. Arrays are created with the `array` function and their subsetting and labelling methods parallel those of matrices. The only major difference is that the notion of transpose must be generalised. The function `aperm` provides such a generalised transpose operation.

2.6 Control Flow

Like most computer languages, R has a number of *control-flow* features designed to make it possible to carry out interesting calculations. These features fall into two general categories — *iteration* and *alternation*. We will look at examples of each of these features.

First however we will look at the notion of *compound expressions*. Compound expressions give a way of treating a sequence of expressions as a single expression. The general form of a compound expression is:

{ *expression*$_1$; ... ; *expression*$_n$ }

A compound expression is evaluated by evaluating each of its component expressions in turn and taking the value of the last one as the value of the compound. (Note that the statements here are separated by semi-colons, but newlines will also serve as separators.)

2.6.1 For-Loop Statements

As part of a computation we often want to repeatedly carry out some computation, perhaps with some slight variation. As an example, suppose we have a vector `x` which contains a set of numerical values, and we want to compute the sum of those values. One way to carry out the calculation is to initialise a variable to zero and to add each element in turn to that variable. The following code shows how we might do this.

```
sum = 0
for(i in 1:length(x))
  sum = sum + x[i]
```

The effect of this calculation is set the variable `i` successively equal to each of the values $1, 2, \ldots,$ `length(x)`, and for each of the successive values to evaluate the expression `sum = sum + x[i]`.

This construction is known as a *for-loop*, and is an example of iteration. The general form of a for-loop is

`for(`*variable* `in` *vector*`)` *expression*

where expression is either a simple or compound expression.

The loop is evaluated by successively setting the value of the *variable* to each element of the *vector* and then evaluating the *expression*. The value of the loop is the value of the last expression computed during evaluation of the loop.

It should be clear that the example above could also be written as

```
sum = 0
for(elt in x)
  sum = sum + elt
```

although this may look slightly odd to anyone used to programming in a traditional programming language.

2.6.2 If-Then-Else and If-Then Statements

In many programs it may be desirable to perform one computation if a particular condition is true, and to otherwise perform some alternative computation. This is made possible through the use of an *if-then-else* statement, which is a particular form of an alternation statement. The general form of the if-then-else statement is:

`if (`*condition*`)` *expression*$_1$ `else` *expression*$_2$

where *condition* is an expression which results in a logical (true/false) value, and *expression*$_1$ and *expression*$_2$ are simple or compound expressions.

An if-then-else statement is evaluated as follows: if *condition* is true, then *expression*$_1$ is evaluated, otherwise *expression*$_2$ is evaluated. The value of an if-then-else statement is the value of whichever of *expression*$_1$ or *expression*$_2$ which was computed.

The expression

```
if (x > 0) y = sqrt(x) else y = -sqrt(-x)
```

provides an example of an if-then-else statement which will look familiar to Pascal, Java, C, or C++ programmers. The statement can however be written more succinctly in R as

```
y = if (x > 0) sqrt(x) else -sqrt(-x)
```

which will look familiar to Lisp or Algol programmers.

There is a simplified form of if-then-else statement which is available when there is no *expression*$_2$ to evaluate. This statement has the general form

`if (`*condition*`)` *expression*

and is completely equivalent to the statement

`if (`*condition*`)` *expression* `else NULL`

2.7 Functions

2.7.1 Simple Functions

R differs from many other statistical software systems because it is designed to be extensible. Users can add new functionality to the system in way which makes it impossible to distinguish that functionality from the capabilities shipped with the system.

New functionality is added to R by defining new *functions*. As a very simple example, let's define a function which squares its argument. We can create this function as follows.

```
> square = function(x) x * x
```

The expression `function(x) x * x` creates a function which is assigned as the value of the variable `square`. The function has a single argument x and the value is multiplied by itself to provide the value of the function. We can use this function in exactly the same way as any other R function.

```
> square(10)
[1] 100
```

Because the operation * acts elementwise on vectors, the new square function will also.

```
> square(1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Using this fact we can write a simple sum-of-squares function.

```
> sumsq = function(x) sum(square(x))
```

2.7.2 General Functions

In general, an R function has the form:

`function` (*arglist*) *body*

where *arglist* is a (comma separated) list of variable names known as the *formal arguments* of the function, and *body* is a simple or compound expression known as the body of the function. The general rule for evaluating a call to a function is to temporarily create a set of variables by associating the arguments passed to the function with the variable names in *arglist*, and then to use these variable definitions to evaluate the function body.

As an example consider the function defined by:

```
> hypot = function(a, b) sqrt(a^2 + b^2)
```

and suppose we make a call to this function by typing:

```
> hypot(3, 4)
```

To evaluate this function call, we first temporarily create variables `a` and `b`, which have the values 3 and 4. We use these variable definitions to evaluate the expression `sqrt(a^2 +b^2)` to obtain the value 5. When the evaluation is complete we remove the temporary definitions of `a` and `b`.

2.7.3 Optional Arguments

In the examples we've presented so far in this section we've assumed that users provide values for all of a function's arguments when they make a call to that function. R has a notion of default argument values which make it possible for the writer of a function to specify reasonable default values for arguments, while still providing the flexibility users the option of overriding these defaults. As an example, consider the following sum-of-squares function.

```
> sumsq = function(x, about=0) sum((x - about)^2)
```

The function definition provides a default definition for the `about` argument. When invoked with just a single argument the function returns the sum of the squared values in that argument.

```
> sumsq(1:10)
[1] 385
```

When provided with a value for the `about` argument, the function computes the sum of squared deviations about that value.

```
> sumsq(1:10, mean(1:10))
[1] 82.5
```

This ability to provide default values for arguments makes it possible to write very complex functions which permit a high degree of customisation, while at the same time making most uses of the functions relatively simple.

2.7.4 Argument Matching

Because it is not necessary to specify all the arguments to R functions, it is important to be clear about which argument corresponds to which formal parameter of the function. The solution is to indicate which formal parameter is associated with an argument by providing a name for the argument. To be specific about which argument is associated with the `about` argument of the `sumsq` function above we could name it as follows.

```
> sumsq(1:10, about=mean(1:10))
[1] 82.5
```

When names are provided for arguments, they are used in preference to position which matching up formal arguments and arguments. For example,

```
> sumsq(about=mean(1:10), 1:10)
```

returns the same answer as the function call above.

The general rule for matching formal and actual arguments is a follows.

1. Use any names provided with the actual arguments to determine the formal arguments associated with the named arguments. Partial matches are acceptable, unless there is an ambiguity.

2. Match the unused actual arguments, in the order given, to any unmatched formal arguments, in the order they appear in the function declaration.

Using these rules, it is easy to see that all the following calls to `sumsq` are equivalent.

```
sumsq(1:10, mean(1:10))
sumsq(1:10, about=mean(1:10))
sumsq(1:10, a=mean(1:10))
sumsq(x=1:10, mean(1:10))
sumsq(mean(1:10), x=1:10)
```