

# Overview of R

Kerby Shedden  
October, 2007

# R

- R is a programming language for statistical computing, data analysis, and graphics. It is a re-implementation of the S language, which was developed in the 1980's.
- R is a high level language. The core language has some superficial similarities to C, but many things are handled automatically in R that are not in C.
- R is one of the main tools used for computing in statistical education and research. It is also widely used for data analysis and numerical computing in other fields of scientific research.

## **Using this document**

Rather than simply reading this document, you will learn R much more quickly if you type the code examples (contained in boxes) into R. Even better, experiment with variations of the code examples to see what happens.

## Running R

- **Interactive:** You can use R interactively, by typing short programs directly at the R prompt.
- **Sourcing scripts:** Most of the time you should write your programs using a text editor (like WordPad on a Windows machine). Save your program as a text file with extension “.R,” like “myprog.R,” then run your program by typing

```
source('myprog.R')
```

at the R prompt.

- You will probably need to set the directory path to point to the directory where you saved your script.
- Make sure you save your programs as text files.

## Variables in R

A variable is a symbol, like  $x$ , that holds a value. The value can be any R object. There are many types of objects in R, but we will mainly use numerical objects. In mathematical terms, these can be grouped as scalars, vectors, and matrices:

- **Scalar variable** A scalar is a single number. The following code creates a scalar variable with the numeric value 5:

```
x = 5
```

- **Vector variable** A vector is a sequence of numbers. The following code creates a vector variable with the value [3, 5, 2]:

```
x = c(3, 5, 2)
```

The “c” stands for “concatenate.”

- **Matrix variable** A matrix is a two-way table of numbers. The following code creates a matrix variable with the value

$$\begin{pmatrix} 2 & 5 \\ 3 & 6 \\ 4 & 7 \end{pmatrix}$$

```
x = matrix(c(2, 3, 4, 5, 6, 7), nrow=3, ncol=2)
```

Note that the matrix is filled in by column. If we use

```
x = matrix(c(2, 3, 4, 5, 6, 7), nrow=3, ncol=2, byrow=TRUE)
```

Then we get

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{pmatrix}$$

## Variable names

You can use simple variable names like `x`, `y`, `A`, and `a` (note that `A` and `a` are different names). You can also use longer names like `counter`, `index1`, or `subject_id`.

A variable name may contain digits, but it cannot begin with a digit. It may contain underscores (`_`) but not operators (`*` `-` `+` `<` `>` `=` `&` `|` `%`) punctuation (`(` `)` `{` `}` `,` `.`) or the comment character (`#`).

Be careful about “clobbering” built-in symbols with your own variable names. You could create a variable named `log`, but then you would no longer be able to use the logarithm function.

## Function signatures

A function signature specifies what arguments can be passed into a function. Functions in R have zero or more mandatory arguments, and zero or more keyword arguments.

For example, the `matrix` function we saw earlier has the following signature:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

For the `matrix` function, every argument has a default value (e.g. `nrow` defaults to 1).

## Getting Information from R

- You can get some documentation about almost any R command or function using the `help` command. For example, the following produces some documentation about the `matrix` function.

```
help(matrix)
```

- You can get the current value of a variable by typing its name at the prompt. You can also use the `print` function to display its value. The following displays the value of the variable `x`.

```
print(x)
```

## Comments

A comment is anything you write in your program that is ignored by the computer. Comments help other people understand the programs you write. Anything following a “#” character is a comment.

```
x = c(3, 5, 2) ## These are the doses of the new drug formulation.
```

## Scalar arithmetic

The familiar scalar arithmetic operations are  $+$ ,  $-$ ,  $*$ , and  $/$  (addition, subtraction, multiplication, and division). After the following program is run,  $z$  will have the value 12,  $a$  will have the value 2, and  $w$  will have the value 24.

```
x = 5
y = 7
z = x + y
a = y - x
w = a*z
```

Other arithmetic operations are  $^$  (exponentiation) and  $\% \%$  (remainder).

```
x = 5^2
y = 23 %% 5
```

## Arithmetic expressions

You can evaluate more complicated expressions by following standard mathematical conventions. If in doubt about precedence, use parentheses (but don't over-use them).

$$x = 5$$

$$y = (x+1) / (x-1)^2 + 1/x$$

## Modifying variable values in place

It is possible (and useful) to modify the value of a variable using an expression that involves its current value. After the following program, the value of  $x$  will be 6.

```
x = 5  
x = x + 1
```

## Assignment by value

Variable values are assigned “by value” in R. Therefore after running the following program, the value of  $x$  is 5 and the value of  $y$  is 6.

```
x = 5  
y = x  
y = y+1
```

## Rounding

R provides several rounding functions: `floor` rounds to  $-\infty$ , `ceiling` rounds to  $+\infty$ , `round` rounds to the nearest integer, and `trunc` rounds toward zero.

```
v = ceiling(3.8)
w = floor(3.8)
x = ceiling(-3.8)
y = trunc(-3.8)
z = round(-3.8)
```

## Higher mathematical functions

The square root function is `sqrt`, and fractional powers are also allowed, so `sqrt(x)` is the same as  $x^{0.5}$ . The natural log function is denoted `log`, and the exponential function  $e^x$  is denoted by `exp(x)`. The trigonometric functions are denoted in the usual way. The mathematical constant `pi` is also provided.

```
w = sqrt(2)
x = exp(3)
y = log(x)
z = tan(pi/3)
```

## Overflow and underflow

Numbers can be represented in exponential notation in R, for example, using `1.5e7` for  $1.5 \times 10^7$ .

Representing a real number on a computer requires an approximation. These approximations are poor for very large numbers and very small numbers. For example, the value of `y` after running the following short program will be exactly zero.

```
x = exp(-100)
y = x^10
```

As another example, the value of `x = tan(pi/2)` should be infinity, but you will see that it is actually a very large finite number. Beware.

## Boolean expressions

Boolean expressions evaluate to either TRUE or FALSE. For example,

```
3 + 2 < 5
```

is FALSE,

```
10 - 4 > 5
```

is TRUE, and

```
10 + 4 == 7 + 7
```

is TRUE (note that you must use two equals signs for testing equality to avoid confusion with assignment statements).

## More Boolean expressions

The `&` (and) operator is `TRUE` only if the expressions on both sides of the operator are `TRUE`. For example,

```
(3 < 5) & (2 > 0)
```

is `TRUE`, and

```
(2 < 3) & (5 > 5)
```

is `FALSE`.

The | (or) operator is TRUE if at least one of the expressions surrounding it is TRUE. For example,

```
(3 < 5) | (2 > 3)
```

is TRUE, and

```
(2 < 1) | (5 > 5)
```

is FALSE.

The ! operator (not) evaluates to TRUE for FALSE statements and to FALSE for TRUE statements. For example

```
!(2 < 1)
```

is TRUE and

```
!(3 < 6)
```

is FALSE.

Boolean expressions can be combined, using parentheses to confirm precedence. For example,

```
((5>4) & !(3<2)) | (6>7)
```

is TRUE.

## Generating arithmetic sequences

The `seq` function generates an arithmetic sequence (i.e. a sequence of values with a fixed spacing between consecutive elements). For example

```
z = seq(3, 8)
```

creates a vector variable called `z` that contains the values 3, 4, 5, 6, 7, 8. You can also use

```
z = seq(8, 3)
```

to create the vector of values 8, 7, 6, 5, 4, 3, or

```
z = seq(3, 8, by=2)
```

to create the vector of values 3, 5, 7, where the `by` parameter causes every second value in the range to be returned.

## Generating vectors using the `array` function

Another way to create vectors in R is using the `array` function. One use of the `array` function is to create a vector with the same value repeated a given number of times. For example,

```
z = array(3, 5)
```

constructs a vector of 5 consecutive 3's, `[3,3,3,3,3]`. You can also use the `array` function to concatenate several copies of an array together end-to-end. For example

```
z = array(c(3,5), 10)
```

creates the vector `[3,5,3,5,3,5,3,5,3,5]`. Note that the second parameter (10 in this case) refers to the length of the result, not the number of times that the first parameter is repeated.

## Generating arrays using the `array` function

You can reshape a vector into an array using `array`:

```
V = seq(1, 11, 2)
M = array(V, c(3,2))
```

which yields the array

$$\begin{pmatrix} 1 & 7 \\ 3 & 9 \\ 5 & 11 \end{pmatrix}.$$

Note that the matrix is filled in column-wise, not row-wise.

You can also convert a matrix into a vector using the `array` function.

Finally, you can use the `array` function in place of the `matrix` function. In the following, the values of A and B are equivalent.

```
V = seq(1, 11, 2)
A = matrix(V, nrow=3, ncol=2)
B = array(V, c(3,2))
```

## Vector and matrix arithmetic

Vectors and matrices of the same shape can be operated on arithmetically, with the operations acting element-wise. For example,

```
x = seq(3, 12, 2)
y = seq(10, 6)
z = x + y
```

calculates the vector sum

$$\begin{array}{ccccccc} & x & & + & & y & & = & & z \\ [3, 5, 7, 9, 11] & & & + & & [10, 9, 8, 7, 6] & & = & & [13, 14, 15, 16, 17] \end{array}$$

## Element-wise operations on vectors and arrays

Many of the mathematical functions in R act element-wise on vectors and matrices. For example, in

```
x = c(9, 16, 25, 36)
y = sqrt(x)
```

the value of  $y$  will be  $y = [3, 4, 5, 6]$ . Other functions acting element-wise include `log`, `exp`, and the trigonometric functions.

## Reducing operations on vectors and arrays

The values in a vector can be summed using the `sum` function. For example,

```
A = seq(1, 100, 2)
x = sum(A)
```

calculates the sum of the odd integers between 1 and 100. There is also a product function called `prod`, but it is rarely used.

The `max` and `min` functions calculate the largest and smallest value, respectively, in a vector or matrix.

```
A = array(seq(1, 100, 2), c(25,2))
mx = max(A)
mn = min(A)
```

The functions `mean`, `median`, `sd`, `IQR`, and `var` calculate the corresponding descriptive statistic from the values in a vector or matrix.

## Element-wise Boolean operations

Most Boolean operators act element-wise.

```
V = c(3,2,8,6,5,6,11,0)
I = (V %% 2 == 1)
```

Creates the vector I with value [TRUE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE].

## Counting

Frequently it is useful to count how many elements within a vector satisfy some condition. For example, if we wanted to know how many of the integers between 1 and 100 are divisible by 7, we could use

```
A = seq(100)
B = (A %% 7 == 0)
x = sum(B)
```

Note that when summing the Boolean vector B, true values are counted as 1 and false values are counted as 0.

## Operating on matrices by row or column

The `apply` function applies a function to every row or to every column of a matrix. For example

```
M = array(seq(8), c(4,2))  
RS = apply(M, 1, sum)  
CS = apply(M, 2, sum)
```

takes the matrix  $M$

$$M = \begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{pmatrix}$$

and computes the row sums  $RS = [6, 8, 10, 12]$  or the column sums  $CS = [10, 26]$ .

The first argument to `apply` is a matrix, and the second argument is either 1 (for operating on the rows) or 2 (for operating on the columns).

The third argument to `apply` (`sum` in the example), can be any function that maps vectors to scalars (e.g. `mean`, `max`, `sd`). If you replace `sum` with `max` in the example, you will get  $RS = [5, 6, 7, 8]$  and  $CS = [4, 8]$ .

## Element access and slicing

To access the 5<sup>th</sup> element of the vector  $V$ , use  $V[5]$ . To access the value in row 3, column 2 of a matrix  $M$ , use  $M[3,2]$ .

To access the entire third row of a matrix  $M$  use  $M[3,]$ . To access the entire second column of a matrix  $M$  use  $M[:,2]$ .

To access the  $2 \times 3$  submatrix spanning rows 3 and 4, and columns 5, 6 and 7 of a matrix  $M$ , use  $M[3:4,5:7]$ .

Here are some examples of element access and slicing:

```
M = array(seq(10), c(5,2))  
M[4,] = -1 ## Change every value in the fourth row to -1  
M[1,2] = -9 ## Change the value in the upper-right corner to -9  
M[2:3,] = 0 ## Change every value in the middle 2x2 submatrix to 0
```

## Creating an empty vector

You can create an empty vector by assigning to `NULL`, e.g. `A = NULL`. You can then access elements of `A` by indexing in the usual way, e.g. `A[5]`. The length of `A` will automatically grow to the largest index to which you have assigned, with unassigned positions having the value `NA`.

```
M = NULL
M[3] = 1

## M will now be c(NA, NA, 1)
```

## Learning the shape of a vector or array

The `length` function returns the length of a vector.

The `dim` function tells you how many rows and columns an array has. `dim` returns a vector of two values. The first return value is the number of rows and the second return value is the number of columns.

```
M = seq(10)
M = array(M, c(5,2))

## d will be c(5,2)
d = dim(M)

## nrow will be 5, and ncol will be 2
nrow = dim(M)[1]
ncol = dim(M)[2]
```

## Extending arrays

You can append a row to a matrix with `rbind`, and a column to a matrix with `cbind`.

```
M = array(seq(10), c(5,2))

## Create a new array which is M extended by one row at
## its lower edge.
A = rbind(M, c(37,38))

## Create a new array which is M extended by one column at
## its right edge.
B = cbind(M, c(37,38,39,40,41))
```

Note that a row you are appending to a matrix `M` should have the same number of elements `M` has columns, and a column you are appending to `M` should have the same number of elements as `M` has rows.

## Removing elements and slices from vectors

To remove the value in a specific position from a vector, use a negative index:

```
M = c(3, 1, 2, 6, 5, 8, 7, 9)

## Remove the value in the third position (which is 2)
## Note: does not remove all 3's from the vector.
## The value of A will be c(3,1,6,5,8,7,9)
A = M[-3]

## Remove a slice: the value of B will be c(3, 1, 9)
B = M[-3:-7]

## Remove everything from the third position to the end of
## the array. The value of C will be c(6, 5, 8, 7, 9).
C = M[-3:0]
```

## Loops

Loops are used to carry out a sequence of related operations without having to write the code for each step explicitly.

Suppose we want to sum the integers from 1 to 10. We could use the following.

```
x = 0
for (i in 1:10)
{
    x = x + i
}
```

The `for` statement creates a loop in which the *looping variable* `i` takes on the values 1,2,...,10 in sequence. For each value of the looping variable, the code inside the braces `{}` is executed (this code is called the *body* of the loop). Each execution of the loop body is called an *iteration*.

In the above program, `x` is an *accumulator variable*, meaning that its value is repeatedly updated while the program runs. It's important to remember to initialize accumulator variables (to zero in the example).

To clarify, we can add a `print` statement inside the loop body.

```
x = 0
for (i in 1:10)
{
    x = x + i
    print(c(i,x))
}
```

Run the above code. The output (to the screen) should look like this:

```
1 1
2 3
3 6
4 10
5 15
6 21
7 28
8 36
9 45
10 55
```

## More on loops

Loops can run over any vector of values. For example,

```
x = 1
for (v in c(3, 4, 7, 2))
{
  x = x*v
}
```

calculates the product  $3 \cdot 4 \cdot 7 \cdot 2 = 168$ .

Loops can be nested. For example,

```
x = 0
for (i in seq(4))
{
  for (j in seq(i))
  {
    x = x+i*j
  }
}
```



## while loops

A `while` loop is used when it is not known ahead of time how many loop iterations are needed. For example, suppose we wish to calculate the partial sums of the harmonic series

$$\sum_{j=1}^n 1/j$$

until the partial sum exceeds 5. It is a fact that the harmonic series diverges:

$$\sum_{j=1}^{\infty} 1/j = \infty,$$

so eventually the partial sum must exceed any given constant.

The following program will produce the first value  $n$  such that  $\sum_{j=1}^n 1/j > 5$ .

```
n = 1 ## Don't forget
x = 0 ## To initialize
while (x < 5)
{
  x = x + 1/n ## Note that the order of these two statements
  n = n+1     ## in the block is important.
}
```

## Conditional execution (“if” blocks)

An if block can be used to make on-the-fly decisions about what statements of a program get executed. For example,

```
y = 7
if (y < 10) { x = 2 }
else       { x = 1 }
```

The value of `x` after executing the previous code is 2. This doesn't appear very useful, but if statements can be very useful inside a loop. For example, the following program places the sum of the even integers up to 100 in `A` and the sum of the odd integers up to 100 in `B`.

```
A = 0
B = 0
for (k in (1:100))
{
  if (k %% 2 == 1) { A = A + k }
  else             { B = B + k }
}
```

The following demonstrates an even more complicated if construction.

```
A = 0
B = 0
C = 0
D = 0
for (k in (1:100))
{
  ## An if construct.
  if ((k %% 2 == 1) & (k < 50))      { A = A + k }
  else if ((k %% 2 == 1) & (k >= 50)) { B = B + k }
  else                               { C = C + k }

  ## An independent if construct.
  if (k >= 50) { D = D + k }
}
```

The preceding program places the sum of odd integers between 1 and 49 in A, the sum of odd integers between 50 and 100 in B, the sum of even integers between 1 and 100 in C, and the sum of all integers greater than or equal to 50 in D.

## break and next in loops

A `break` statement is used to exit a loop when a certain condition is met. A `next` statement results in the current iteration being aborted, but the loop continues with the next iteration.

The following program sums the odd integers between 1 and 49.

```
x = 0
for (k in seq(100))
{
  if (k %% 2 == 0) { next } ## Skip even numbers, but keep looping.
  if (x >= 50) { break } ## Quit looping when the sum exceeds 50.
  x = x + k
}
```

## Defining your own functions

You can define your own functions with the `function` construct. The body of the function (the code in the braces) is executed whenever the function is called. The `return` statement produces a value that is returned when the function is called.

```
f = function(x, y) {  
    return(x / (1+y^2))  
}  
  
## a will be 0.2.  
a = f(1, 2)
```

## Hashes

A list is a collection of values that are indexed by arbitrary keys (unlike vectors which are indexed by integer keys). A list in R is equivalent to an “associative array,” “hash table,” or “dictionary” as found in other high-level languages.

```
## Creat a list with two values.  
A = list()  
A$apple = 3  
A$pear = 4  
  
## Add a third value based on the other two values.  
A$fruit = A$apple + A$pear  
  
## An alternate way to use the key.  
A[['fruit']] = A[['apple']] + A[['pear']]
```