R literacy for digital soil mapping. Part 2

Soil Security Laboratory

2017

1 Vectors, matrices, and arrays

1.1 Creating and working with vectors

There are several ways to create a vector in R. Where the elements are spaced by exactly 1, just separate the values of the first and last elements with a colon.

```
1:5
```

```
## [1] 1 2 3 4 5
```

The function **seq** (for sequence) is more flexible. Its typical arguments are **from, to**, and **by** (or, in place of **by**, you can specify **length.out**).

```
seq(-10, 10, 2)
```

```
##  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

Note that the **by** argument does not need to be an integer. When all the elements in a vector are identical, use the **rep** function (for repeat).

```
rep(4, 5)
```

```
## [1] 4 4 4 4 4
```

For other cases, use **c** (for concatenate or combine).

```
c(2, 1, 5, 100, 2)
```

```
## [1]   2   1   5 100   2
```

Note that you can name the elements within a vector.

```
c(a = 2, b = 1, c = 5, d = 100, e = 2)
```

```
##   a   b   c   d   e
##   2   1   5 100   2
```

Any of these expressions could be assigned to a symbolic variable, using an assignment operator.

```
v1 <- c(2, 1, 5, 100, 2)
v1
```

```
## [1]   2   1   5 100   2
```

Variable names can be any combination of letters, numbers, and the symbols . and _, but they can not start with a number or with _. Google has a R style guide `https://google.github.io/styleguide/Rguide.xml` which describes good and poor examples of variable name attribution, but generally it is a personal preference on how you name your variables.

```
probably.not_a.good_example.for.a.name.100 <- seq(1, 2, 0.1)
probably.not_a.good_example.for.a.name.100
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

The `c` function is very useful for setting up arguments for other functions, as will be shown later. As with all R functions, both variable names and function names can be substituted into function calls in place of numeric values.

```
x <- rep(1:3)
y <- 4:10
z <- c(x, y)
z
```

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

Although R prints the contents of individual vectors with a horizontal orientation, R does not have "column vectors" and "row vectors", and vectors do not have a fixed orientation. This makes use of vectors in R very flexible.

Vectors do not need to contain numbers, but can contain data with any of the modes mentioned earlier (numeric, logical, character, and complex), as long as all the data in a vector are of the same mode.

Logical vectors are very useful in R for sub-setting data i.e., for isolating some part of an object that meets certain criteria. For relational commands, the shorter vector is repeated as many as necessary to carry out the requested comparison for each element in the longer vector.

```
x <- 1:10
x > 5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Also, note that when logical vectors are used in arithmetic, they are changed (coerced in R terms) into a vector of binary elements: 1 or 0. Continuing with the above example:

```
a <- x > 5
a
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
a * 1.4
```

```
##  [1] 0.0 0.0 0.0 0.0 0.0 1.4 1.4 1.4 1.4 1.4
```

One function that is commonly used on character data is `paste`. It

concatenates character data (and can also work with numerical and logical elements—these become character data).

```
paste("A", "B", "C", "D", TRUE, 42)
```

```
## [1] "A B C D TRUE 42"
```

Note that the `paste` function is very different from `c`. The `paste` function concatenates its arguments into a single character value, while the `c` function combines its arguments into a vector, where each argument becomes a single element. The `paste` function becomes handy when you want to combine the character data that are stored in several symbolic variables.

```
month <- "April"
day <- 29
year <- 1770
paste("Captain Cook, on the ", day, "th day of ", month, ", "
, year, ", sailed into Botany Bay", sep = "")
```

```
## [1] "Captain Cook, on the 29th day of April, 1770, sailed into Botany Bay"
```

This is especially useful with loops, when a variable with a changing value is combined with other data. Loops will be discussed in a later section.

```
group <- 1:10
id <- LETTERS[1:10]
for (i in 1:10) {
    print(paste("group =", group[i], "id =", id[i]))
}
```

```
## [1] "group = 1 id = A"
## [1] "group = 2 id = B"
## [1] "group = 3 id = C"
## [1] "group = 4 id = D"
## [1] "group = 5 id = E"
## [1] "group = 6 id = F"
## [1] "group = 7 id = G"
## [1] "group = 8 id = H"
## [1] "group = 9 id = I"
## [1] "group = 10 id = J"
```

`LETTERS` is a constant that is built into `R`—it is a vector of uppercase letters A through Z (different from `letters`).

1.2 Vector arithmetic, some common functions, and vectorised operations

In `R`, vectors can be used directly in arithmetic operations. Operations are applied on an element-by-element basis. This can be referred to as "vectorised" arithmetic, and along with vectorised functions (described below), it is a quality that makes `R` a very efficient programming language.

```
x <- 6:10
x
```

```
## [1]  6  7  8  9 10
```

```
x + 2
```

```
## [1]  8  9 10 11 12
```

For an operation carried out on two vectors, the mathematical operation is applied on an element-by-element basis.

```
y <- c(4, 3, 7, 1, 1)
y
```

```
## [1] 4 3 7 1 1
```

```
z <- x + y
z
```

```
## [1] 10 10 15 10 11
```

When two vectors having different numbers of elements used in an expression together, R will repeat the smaller vector. For example, with vector of length one, i.e. a single number:

```
x <- 1:10
m <- 0.8
b <- 2
y <- m * x + b
y
```

```
##  [1]  2.8  3.6  4.4  5.2  6.0  6.8  7.6  8.4  9.2 10.0
```

If the number of rows in the smaller vector is not a multiple of the larger vector (often indicative of an error) R will return a warning.

```
x <- 1:10
m <- 0.8
b <- c(2, 1, 1)
y <- m * x + b
```

```
## Warning in m * x + b:  longer object length is not a multiple
of shorter object length
```

```
y
```

```
##  [1]  2.8  2.6  3.4  5.2  5.0  5.8  7.6  7.4  8.2 10.0
```

Some arithmetic operators that are available in R include:

+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
%/%	integer division
%%	modulo (remainder)

`log (a)`	natural log of a
`log10 (a)`	base 10 log of a
`exp (a)`	e^a
`sine (a)`	sine of a
`cos (a)`	cosine of a
`tan (a)`	tangent of a
`sqrt (a)`	square root of a

Some simple functions that are useful for vector math include:

`min`	minimum value of a set of numbers
`max`	maximum of a set of numbers
`pmin`	parallel minima (compares multiple vectors "row-by-row")
`pmax`	parallel maxima
`sum`	sum of all elements
`length`	length of a vector (or number of columns in a data frame)
`nrow`	number of rows in a vector of data frame
`ncol`	number of columns
`mean`	arithmetic mean
`sd`	standard deviation
`rnorm`	generates a vector of normally-distributed random numbers
`signif, ceiling, floor`	rounding

Many, many other functions are available.

`R` also has a few built in constants, including `pi`.

```
pi
```

```
## [1] 3.141593
```

Parentheses can be used to control the order of operations, as in any other programming language.

```
7 - 2 * 4
```

```
## [1] -1
```

is different from:

```
(7 - 2) * 4
```

```
## [1] 20
```

and

```
10^1:5
```

```
## [1] 10  9  8  7  6  5
```

is different from:

```
10^(1:5)
```

```
## [1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

Many functions in R are capable of accepting vectors (or even data frames, arrays, and lists) as input for single arguments, and returning an object with the same structure. These vectorised functions make vector manipulations very efficient. Examples of such functions include `log, sin`, and `sqrt`, For example:

```
x <- 1:10
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

or

```
sqrt(1:10)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

The previous expressions are also equivalent to:

```
sqrt(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

But they are not the same as the following, where all the numbers are interpreted as individual values for multiple arguments.

```
## This will throw an error
sqrt(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

There are also some functions designed for making vectorised operations on lists, matrices, and arrays: these include `apply` and `lapply`.

1.3 Matrices and arrays

Arrays are multi-dimensional collections of elements and matrices are simply two-dimensional arrays. R has several operators and functions for carrying out operations on arrays, and matrices in particular (e.g. matrix multiplication).

To generate a matrix, the `matrix` function can be used. For example:

```
X <- matrix(1:15, nrow = 5, ncol = 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
```

Note that the filling order is by column by default (i.e. each column is filled

before moving onto the next one). The "unpacking" order is the same:

```
as.vector(X)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

If, for any reason, you want to change the filling order, you can use the `by row` argument:

```
X <- matrix(1:15, nrow = 5, ncol = 3, byrow = T)
X
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
## [5,]   13   14   15
```

A similar function is available for higher-order arrays, called `array`. Here is an example with a three-dimensional array:

```
Y <- array(1:30, dim = c(5, 3, 2))
Y
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   16   21   26
## [2,]   17   22   27
## [3,]   18   23   28
## [4,]   19   24   29
## [5,]   20   25   30
```

Arithmetic with matrices and arrays that have the same dimensions is straightforward, and is done on an element-by-element basis. This true for all the arithmetic operators listed in earlier sections.

```
Z <- matrix(1, nrow = 5, ncol = 3)
Z
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

```
## [4,]    1    1    1
## [5,]    1    1    1
```

```
X + Z
```

```
##      [,1] [,2] [,3]
## [1,]    2    3    4
## [2,]    5    6    7
## [3,]    8    9   10
## [4,]   11   12   13
## [5,]   14   15   16
```

This does not work when dimensions do not match:

```
## This will throw an error
Z <- matrix(1, nrow = 3, ncol = 3)
X + Z
```

For mixed vector/array arithmetic, vectors are recycled if needed.

```
Z
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
## [4,]    1    1    1
## [5,]    1    1    1
```

```
x <- 1:9
Z + x
```

```
## Warning in Z + x:  longer object length is not a multiple of
shorter object length
```

```
##      [,1] [,2] [,3]
## [1,]    2    7    3
## [2,]    3    8    4
## [3,]    4    9    5
## [4,]    5   10    6
## [5,]    6    2    7
```

```
y <- 1:3
Z + y
```

```
##      [,1] [,2] [,3]
## [1,]    2    4    3
## [2,]    3    2    4
## [3,]    4    3    2
## [4,]    2    4    3
## [5,]    3    2    4
```

`R` also has operators for matrix algebra. The operator `%*%` carries out matrix multiplication, and the function `solve` can invert matrices.

```
X <- matrix(c(1, 2.5, 6, 7.5, 4.9, 5.6, 9.9, 7.8, 9.3), nrow = 3)
X
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  7.5  9.9
## [2,]  2.5  4.9  7.8
## [3,]  6.0  5.6  9.3
```

```
solve(X)
```

```
##             [,1]       [,2]       [,3]
## [1,]  0.07253886 -0.5492228  0.3834197
## [2,]  0.90385723 -1.9228555  0.6505469
## [3,] -0.59105738  1.5121858 -0.5315678
```

1.4 Exercises

1. Generate a vector of numbers that contains the sequence 1, 2, 3, ...10 (try to use the least amount of code possible to do this). Assign this vector to the variable x, and then carry out the following vector arithmetic.
   (a) $log_{10}x$
   (b) lnx
   (c) $\frac{\sqrt{x}}{2-x}$
2. Use an appropriate function to generate a vector of 100 numbers that go from 0 to 2π, with a constant interval. Assuming this fist vector is called x, create a new vector that contains $sine(2x - 0.5\pi)$. Determine the minimum and maximum of $sine(2x - 0.5\pi)$. Does this match with what you expect?
3. Create 5 vectors, each containing 10 random numbers. Give each vector a different name. Create a new vector where the 1^{st} element contains the sum of the 1^{st} elements in your original 5 vectors, the 2^{nd} element contains the sum of the 2^{nd} elements, etc. Determine the mean of this new vector.
4. Create the following matrix using the least amount of code:

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
```

5. If you are bored, try this. Given the following set of linear equations:

   $27.2x + 32y - 10.8z = 401.2$
   $x - 1.48y = 0$
   $409.1x + 13.5z = 2.83$

Solve for x, y, and z.