2. Create and Work with Vectors

Vectors

The key feature which makes R very useful for statistics is that it is vectorized. This means that many operations can be performed point-wise on a

vector. The function c() is used to create vectors:

6

> x <- c(1, -1, 3.5, 2)

> x

[1] 1.0 -1.0 3.5 2.0

Then if we want to add 2 to everything in this vector, or to square each

entry:

> x + 2

[1] 3.0 1.0 5.5 4.0

> x^2

[1] 1.00 1.00 12.25 4.00

This is very useful in statistics:

> sum((x - mean(x))^2)

[1] 10.69

```
> x1 <- c(1,3,5,7,9)
> x1
[1] 1 3 5 7 9
>
> gender <- c("male", "female")
> gender
[1] "male"   "female"
>
> 2:7
[1] 2 3 4 5 6 7
>
> seq(from=1, to=7, by=1)
[1] 1 2 3 4 5 6 7
>
> seq(from=1, to=7, by=1/3)
 [1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667 3.000000 3.333333
 [9] 3.666667 4.000000 4.333333 4.666667 5.000000 5.333333 5.666667 6.000000
[17] 6.333333 6.666667 7.000000
>
> seq(from=1, to=7, by=0.25)
 [1] 1.00 1.25 1.50 1.75 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25
[15] 4.50 4.75 5.00 5.25 5.50 5.75 6.00 6.25 6.50 6.75 7.00
>
> rep(1, times=10)
 [1] 1 1 1 1 1 1 1 1 1 1
>
> rep("marin", times=5)
[1] "marin" "marin" "marin" "marin" "marin"
>
```

```
Console ~/
> rep(1:3, times=5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
>
> rep(seq(from=2, to=5, by=0.25), times=5)
 [1] 2.00 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00
[15] 2.25 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00 2.25
[29] 2.50 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00 2.25 2.50
[43] 2.75 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00 2.00 2.25 2.50 2.75
[57] 3.00 3.25 3.50 3.75 4.00 4.25 4.50 4.75 5.00
>
> rep(c("m", "f"), times=5)
 [1] "m" "f" "m" "f" "m" "f" "m" "f" "m" "f"
>
> x <- 1:5
> x
[1] 1 2 3 4 5
>
> y <- c(1,3,5,7,9)
> y
[1] 1 3 5 7 9
>
> x + 10
[1] 11 12 13 14 15
>
> x - 10
[1] -9 -8 -7 -6 -5
>
> x*10
[1] 10 20 30 40 50
>
```

```
Console ~/
> x/2
[1] 0.5 1.0 1.5 2.0 2.5
>
> # if two vectors are of the same length, we may add/subtract/mult/div
> # corresponding elements
>
> x
[1] 1 2 3 4 5
>  y
[1] 1 3 5 7 9
>
> x+y
[1]  2  5  8 11 14
>
> x-y
[1]  0 -1 -2 -3 -4
>
> x*y
[1]  1  6 15 28 45
>
> x/y
[1] 1.0000000 0.6666667 0.6000000 0.5714286 0.5555556
>
> x
[1] 1 2 3 4 5
>
> y
[1] 1 3 5 7 9
>
```

```
Console ~/
> y[3]
[1] 5
>
> y[-3]
[1] 1 3 7 9
>
> y[1:3]
[1] 1 3 5
>
> y[c(1, 5)]
[1] 1 9
>
> y[-c(1, 5)]
[1] 3 5 7
>
> y[y<6]
[1] 1 3 5
>
> matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
>
> matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, byrow=FALSE)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
>
```

```
>
> mat <- matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, byrow=TRUE)
> mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
>
> mat[1, 2]
[1] 2
>
> mat[c(1, 3), 2]
[1] 2 8
>
> mat[2,]
[1] 4 5 6
>
> mat[,1]
[1] 1 4 7
>
> mat*10
     [,1] [,2] [,3]
[1,]   10   20   30
[2,]   40   50   60
[3,]   70   80   90
>
```

Exercise 2.1. The weights of five people before and after a diet are given in the table.

Before 78 72 78 79 105

After 67 65 79 70 93

Read the 'before' and 'after' values into two different vectors called before and after.

Use R to evaluate the amount of weight lost for each participant. What is the average amount of weight lost?

We can also use R's vectorization to create more interesting sequences:

> 2^(0:10)

[1] 1 2 4 8 16 32 64 128 256 512 1024

> 1:3 + rep(seq(from=0,by=10,to=30), each=3)

[1] 1 2 3 11 12 13 21 22 23 31 32 33

The last example demonstrates recycling, which is also an important part of vectorization. If we perform a binary operation (such as +) on two vectors of different lengths, the shorter one is used over and over again until the operation has been applied to every entry in the longer one. If the longer length is not a multiple of the shorter length, a warning is given.

> 1:10 * c(-1,1)

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

> 1:7 * 1:2

Warning: longer object length is not a multiple of shorter object length

[1] 1 4 3 8 5 12 7

Exercise 2.2. Create the following vectors in R using seq() and rep().

(i) 1, 1.5, 2, 2.5, . . . , 12

(ii) 1, 8, 27, 64, . . . , 1000.

(iii) 1, 0, 3, 0, 5, 0, 7, . . . , 0, 49.

Subsetting

It's frequently necessary to extract some of the elements of a larger vector. In R you can use square brackets to select an individual element or group of elements:

> x <- c(5,9,2,14,-4)

> x[3]

[1] 2

> # note indexing starts from 1

> x[c(2,3,5)]

[1] 9 2 -4

> x[1:3]

[1] 5 9 2

> x[3:length(x)]

[1] 2 14 -4

There are two other methods for getting subvectors. The first is using a logical vector (i.e. containing TRUE and FALSE) of the same length:

```
> x > 4

[1] TRUE TRUE FALSE TRUE FALSE

> x[x > 4]

[1] 5 9 14
```

or using negative indices to specify which elements should not be selected:

```
> x[-1]

[1] 9 2 14 -4

> x[-c(1,4)]

[1] 9 2 -4
```

(Note that this is rather different to what other languages such as C or Python would interpret negative indices to mean.)

Exercise 2.3. The built-in vector LETTERS contains the uppercase letters of the alphabet. Produce a vector of (i) the first 12 letters; (ii) the odd 'numbered' letters; (iii) the (English) consonants.

Character Vectors

As you might have noticed in the exercise above, vectors don't have to contain numbers. We can equally create a character vector, in which each entry is a string of text. Strings in R are contained within double quotes ":

```
> x <- c("Hello", "how do you do", "lovely to meet you", 42)

> x

[1] "Hello" "how do you do" "lovely to meet you"

[4] "42"
```

Notice that you cannot mix numbers with strings: if you try to do so the number will be converted into a string. Otherwise character vectors are much like their numerical counterparts.

```
> x[2:3]

[1] "how do you do" "lovely to meet you"

> x[-4]
```

[1] "Hello" "how do you do" "lovely to meet you"

> c(x[1:2], "goodbye")

[1] "Hello" "how do you do" "goodbye"

Matrices

Matrices are much used in statistics, and so play an important role in R. To create a matrix use the function matrix(), specifying elements by column

first:

> matrix(1:12, nrow=3, ncol=4)

[,1] [,2] [,3] [,4]

[1,] 1 4 7 10

[2,] 2 5 8 11

[3,] 3 6 9 12

This is called column-major order. Of course, we need only give one of the dimensions:

> matrix(1:12, nrow=3)

unless we want vector recycling to help us:

> matrix(1:3, nrow=3, ncol=4)

[,1] [,2] [,3] [,4]

[1,] 1 1 1 1

[2,] 2 2 2 2

[3,] 3 3 3 3

Sometimes it’s useful to specify the elements by row first

> matrix(1:12, nrow=3, byrow=TRUE)

There are special functions for constructing certain matrices:

> diag(3)

[,1] [,2] [,3]

[1,] 1 0 0

[2,] 0 1 0

[3,] 0 0 1

```
> diag(1:3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
> 1:5 %o% 1:5
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

The last operator performs an outer product, so it creates a matrix with (i, j)-th entry xiyj . The function outer() generalizes this to any function f on two arguments, to create a matrix with entries f(xi , yj).

```
> outer(1:3, 1:4, "+")
     [,1] [,2] [,3] [,4]
[1,]    2    3    4    5
[2,]    3    4    5    6
[3,]    4    5    6    7
```

Matrix multiplication is performed using the operator %*%, which is quite distinct from scalar multiplication *.

```
> A <- matrix(c(1:8,10), 3, 3)
> x <- c(1,2,3)
> A %*% x # matrix multiplication
     [,1]
[1,]   30
[2,]   36
[3,]   45
```

> A*x # NOT matrix multiplication

[,1] [,2] [,3]

[1,] 1 4 7

[2,] 4 10 16

[3,] 9 18 30

Standard functions exist for common mathematical operations on matrices.

> t(A) # transpose

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

[3,] 7 8 10

> det(A) # determinant

[1] -3

> diag(A) # diagonal

[1] 1 5 10

> solve(A) # inverse

[,1] [,2] [,3]

[1,] -0.6667 -0.6667 1

[2,] -1.3333 3.6667 -2

[3,] 1.0000 -2.0000 1

Exercise 2.4. Construct the matrix

B= [1 2 3

4 2 6

-3 1 -3]

Subsetting of Matrices

Matrices can be subsetted much the same way as vectors, although of course they have two indices. Row number comes first:

> A[2,1]

[1] 2

```
> A[2,2:ncol(A)]
[1] 5 8
> A[,1:2] # blank indices give everything
     [,1] [,2]
[1,] 1    4
[2,] 2    5
[3,] 3   6
> A[c(),1:2] # empty indices give nothing!
[,1] [,2]
```

Notice that, where appropriate, R automatically reduces a matrix to a vector or scalar when you subset it. You can override this using the optional drop argument.

```
> A[2,2:ncol(A),drop=FALSE] # returns a matrix
     [,1] [,2]
[1,]  5     8
```

You can stitch matrices together using the rbind() and cbind() functions. These employ vector recycling:

```
> cbind(A, t(A))
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,] 1 4 7 1 2 3
[2,] 2 5 8 4 5 6
[3,] 3 6 10 7 8 10
> rbind(A, 1, 0)
     [,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 10
[4,] 1 1 1
[5,] 0 0 0
```