Statistics 120
More R

Subsetting

- One of the strong features of R, is the ability to extract data subsets in a flexible way.
- The subsetting in R applies to vectors and also to more general objects.
- The subsetting methods are designed to support statistical data analysis. They are different from the mechanisms found in database systems.

Explicit Subsetting

```
> x = 10 * (1:10)
> x
 [1]  10  20  30  40  50  60  70  80  90 100
```

```
> x[2]
[1] 20
```

```
> x[1:4]
[1] 10 20 30 40
```

```
> x[c(1, 3, 5)]
[1] 10 30 50
```

Exclusion

```
> x[-2]
[1]  10  30  40  50  60  70  80  90 100

> x[-(1:4)]
[1]  50  60  70  80  90 100

> x[c(1, -2)]
Error: only 0's may mix with negative subscripts

> x[0]
numeric(0)
```

Logical Vectors

So far we have only seen vectors containing numbers. It is also possible for vectors to contain logical (true/false) values. These are usually generated by comparisons using the operators.

`<`	less	`>`	greater
`<=`	less or equal	`>=`	greater or equal
`==`	equal	`!=`	not equal
`&`	and	`|`	or

Logical Values

```
> x = 10 * (1:10)
> y = c(rep(1, 5), rep(2, 5))
> x
 [1]  10  20  30  40  50  60  70  80  90 100


> y
 [1] 1 1 1 1 1 2 2 2 2 2


> x[y == 1]
[1] 10 20 30 40 50


> x[y == 2]
[1]  60  70  80  90 100
```

A More Complex Example

Subsetting is often used as follows in data analysis:

```
mean(weight[sex == "male" & age < 30])
```

This can be read naturally as:

> "*Obtain the mean weight for males under 30.*"

Character Strings

- In addition to vectors of numbers and logical values, R also supports vectors of character strings.
- Strings are enclosed between (single or double) quotes.

```
"A string"
'Another string'
```

- Strings are most commonly used as plot labels.

Character Strings

```
> x = "abc"
> y = rep(x, 3)
> y
[1] "abc" "abc" "abc"


> length(y)
[1] 3


> y[2] = "z"
> y
[1] "abc" "z"   "abc"
```

Mixing Data Types

- The elements stored in a vector must all be of the same type (numbers/logicals/strings).
- If items of different types are combined into a vector they are *coerced* to be of the same type.
- The direction of conversion is:

  logical $\rightarrow$ numeric $\rightarrow$ character

Type Coercion Examples

```
> c(TRUE, FALSE, 10, 20)
[1]  1  0 10 20
```

```
> c(TRUE, FALSE, "string")
[1] "TRUE"   "FALSE"  "string"
```

```
> c(TRUE, FALSE, 10, "string")
[1] "TRUE"   "FALSE"  "10"     "string"
```

A Common Idiom

Suppose that the variables `sex` and `age` contain the gender and age of members of a sample of individuals.

The fraction of males under 30 can be computed as follows:

```
pm30 = sum(sex == "male" & age < 30) /
           sum(sex == "male")
```

A Common Idiom

Suppose that the variables `sex` and `age` contain the gender and age of members of a sample of individuals.

The fraction of males under 30 can be computed as follows:

```
pm30 = sum(sex == "male" & age < 30) /
           sum(sex == "male")
```

This fraction can be converted to a percentage as follows:

```
round(100 * pm30, 2)
```

Lists

Vectors are by far the most common objects encountered in R, but sometimes the requirement that all elements must have the same type is too restrictive. There is another structure called a list which can be

```
> L = list(a = 1:3, b = "hello")
> L
$a
[1] 1 2 3

$b
[1] "hello"
```

Manipulating Lists

The only important operation which can be performed on a list is the extraction of a sublist, or element.

```
> L = list(a = 1:3, b = "hello")
> L$a
[1] 1 2 3


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


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

Matrices and Arrays

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.

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

Matrices and Arrays

By default, matrices are created with their values 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
```

Matrix Extents

The number of rows and columns in a matrix can be obtained with the functions `nrow` and `ncol`. Both values can be obtained simultaneously with the function `dim`

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


> ncol(A)
[1] 2


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

Row and Column Labels

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

Control Flow

- Virtually all computer languages support some from of control-flow statements.
- Control statements usually are of two types:
  - **Iteration** – repeatedly carrying out the same task.
  - **Alternation** – carrying out one of two (or more) alternatives.
- R provides control flow statements in the form of "`for`" and "`if`" statements.

For Statements

Here is an example showing how to use a "`for`" statement to add up the elements of a vector.

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

The effect of this is to successively set the value of the variable `i` to each of $1, 2, \ldots,$ `length(x)` and to carry out the computation `total = total + x[i]`.

At the end of the computation the variable `total` will contain the sum of all the elements in `x`

General For Statements

The general form of a `for` statement is:

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

The effect of this is to successively set the value of the variable to each element of the vector and compute the expression.

We could thus compute the sum of elements of a vector as follows:

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

If-Then-Else and If-Then Statements

A simple example of an `if` statement.

```
if (any(x < 0))
    stop("There were negative values in x")
```

A Simple example of an `if-then-else` statement.

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

This could be written more simply as:

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

Functions

One feature which makes R more useful than many statistical programs is that is extensible – it is possible to create new capabilities by defining new functions.

A simple example:

```
> square = function(x) x * x
> square(10)
[1] 100
```

Here we have defined a new function which takes a single argument and returns the product of that value with itself as its value. The function is stored with the name `square`.

More On Functions

The function square works just as well on vectors as it does on scalar values.

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

The function square is just like any other R function. We can define new functions in terms of it.

```
> sumsq = function(x) sum(square(x))
> sumsq(1:10)
[1] 385
```

General Functions

R functions can have multiple arguments. The following function has two arguments.

```
> hypot = function(a, b) sqrt(a^2 + b^2)
> hypot(3, 4)
[1] 5
```

There is no (theoretical) limit to the number of arguments a function can have.

Optional Arguments

It is possible to define *default values* for function arguments. If those arguments are not given values when the functions are called, the default values are used.

```
> sumsq = function(x, a = 0) sum((x - a)^2)
> sumsq(1:10)
[1] 385


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

Argument Naming

Because function arguments are optional it is important to have a way of specifying which argument is which. This can be be done by specifying (partial) names for the arguments.

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

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

Arguments are matched first by name and then by position.

Recursion

Like most modern programming languages, R functions are able to be defined in a recursive fashion. Here is a recursively defined factorial function.

```
> factorial = function(n) if (n <= 1) 1 else n *
      factorial(n - 1)
> factorial(10)
[1] 3628800
```

Beware that there is a (quite low) limit on the depth of recursion which is permitted in R.

Object Orientation

- R has an object system similar to that of Common Lisp or Dylan (and quite different from that of C++ and Java).
- The object system and other system features make it possible to use R for large-scale software projects.
- In this course we will not be making any use of the object system but, if you are interested, you can can find out about it in the system manuals.