CSC 120: Computer Science for the Sciences (R section)

Radford M. Neal, University of Toronto, 2016

http://www.cs.utoronto.ca/∼radford/csc120/

Week 7

Names and NA — Two R Features for Statistics

R is a general purpose programming language. You can write all sorts of programs in R — video games, accounting packages, word processors, programs for navigating rocket ships to Mars, ...

But R is more appropriate for some of these tasks than others. It's probably not the best choice for video game programming — a game may need to respond very quickly to user input, but speed is not R's strong point. On the other hand, some features of R that are not common in other languages are especially useful for statistical applications.

Here are two:

- Names for elements of vectors and lists, and for rows and columns of matrices and data frames.
- Special `NA` values to indicate where data is missing

Names for Elements of Vectors

When doing statistical analysis, referring something by name – for example, “age” — is much more reliable than referring to it by some number.

We’ve already seen that names can be attached to list elements. You can also give names to elements in a vector, either when first creating it, or later. For example:

```
> u <- c (abc=9, def=10, xyz=3)
> u
abc def xyz
  9  10   3
> v <- c(9,10,3)
> names(v) <- c("abc","def","xyz")
> v
abc def xyz
  9  10   3
```

Unfortunately, unlike for lists, you can’t get at elements in vectors by name using `$`. But you can use the name as an index:

```
> v["def"]
def
 10
```

Names for Rows and Columns of Matrices and Data Frames

You can also give names to rows and columns of a matrix:

```
> M <- matrix (1:12, nrow=3, ncol=4)
> rownames(M) <- c("ab","cd","ef")
> colnames(M) <- c("w","x","y","z")
> M
   w x y  z
ab 1 4 7 10
cd 2 5 8 11
ef 3 6 9 12
```

You can then use these names to refer to elements of the matrix:

```
> M["cd","y"]
[1] 8
> M["ab","x"] <- 0
> M["ab",]
 w  x  y  z
 1  0  7 10
```

Adding Attributes to R Objects

An R object can have one or more “attributes”, that record extra information. They are mostly ignored if you don’t look at them, but are there if you look.

An example:

```
> x <- 123                         # Set x to a plain number
> x
[1] 123
> attr(x,"fred") <- "abc"   # Add a "fred" attribute to x
> x
[1] 123
attr(,"fred")
[1] "abc"
> attr(x,"fred")                   # We can get just the attribute if we like
[1] "abc"
> x + 1000                         # The attribute (usually) gets passed on
[1] 1123
attr(,"fred")
[1] "abc"
```

Attributes for Dimensions and Names

You can attach attributes to objects for your own purposes, but R also has some standard uses for attributes.

R uses a `dim` attribute to mark an object as a matrix, and hold how many rows and columns it has. This attribute is not usually shown explicitly, be we can see it if we look using `attr`:

```
  > M <- matrix(0,nrow=3,ncol=5)
  > attr(M,"dim")
[1] 3 5
```

R uses a `names` attribute to hold the names of elements in a list or a vector:

```
  > L <- list (abc=9, def=10, xyz="ha")
  > attr(L,"names")
  [1] "abc" "def" "xyz"
```

Names for rows and columns in a matrix are stored in a `dimnames` attribute.

The Class Attribute

The special `class` attribute tells R that some operations on the object should be done in a special way. We'll cover more about how this works later — and about how it can be used to program in a style known as ‘object-oriented programming”.

For the moment, here's a brief illustration of what can be done:

```
> g <- 123
> attr(g,"class") <- "gobbler"
> print.gobbler <- function (what) {
+     cat ("I'm a gobbler with value", unclass(what), "\n")
+ }
> g
I'm a gobbler with value 123
> g+1000
I'm a gobbler with value 1123
```

We've used the `class` attribute to tell R that objects in our “gobbler” class should be printed in a different way than ordinary numbers. Note that `unclass` gets rid of the class attribute, which lets us handle the number inside a gobbler object in the usual way (though using `unclass` is not strictly necessary here).

Data Frames

One major use of classes is for R's `data.frame` objects, which are the most common way that data is represented in R.

A data frame is sort of like a list and sort of like a matrix. Each "row" of a data frame holds information on some individual, object, case, or whatever. The "columns" of a data frame correspond to variables whose values have been measured for each case. These variables can be numbers, logical (`TRUE`/`FALSE`) values, or character strings (but all values for one variable have the same type).

For example, here's how R prints a small data frame containing the heights and weights of three people:

```
> heights_and_weights
  name height weight
1 Fred     62    144
2 Mary     60    131
3  Joe     71    182
```

A data frame is really a list, with named elements that are the columns of the data frame, but with a `data.frame` class attribute that makes R do things like printing and subscripting differently from an ordinary list.

Getting Data Out of a Data Frame

You can get data from a data frame using subscripting operations similar to those for a matrix (by row and column index), or by operations similar to a list (using names of variables). For example:

```
> heights_and_weights           # The data frame from the last slide
  name height weight
1 Fred     62    144
2 Mary     60    131
3  Joe     71    182
> heights_and_weights$height    # All values of the "height" variable
[1] 62 60 71
> heights_and_weights[2,]       # All values for the 2nd person
  name height weight
2 Mary     60    131
> heights_and_weights[2,3]      # Value of 3rd variable for 2nd person
[1] 131
> heights_and_weights$weight[2] # ... and the same, by variable name
[1] 131
```

Creating a Data Frame

Using `as.data.frame`, you can create a data frame from a list (it just adds the `data.frame` class attribute) or from a matrix (it has to split it up into columns). If you don't provide variable names, R uses `V1`, `V2`, etc.

Examples:

```
> as.data.frame (list (abc=c(1,3,2),
+                      pqr=c(TRUE,FALSE,FALSE),
+                      xyz=c("a","bb","c")))
  abc   pqr xyz
1   1  TRUE   a
2   3 FALSE  bb
3   2 FALSE   c
>
> as.data.frame (matrix (1:12, nrow=3, ncol=4))
  V1 V2 V3 V4
1  1  4  7 10
2  2  5  8 11
3  3  6  9 12
```

If a matrix has row and column names, they become those of the data frame.

Reading Data Into a Data Frame

The `read.table` function creates a data frame using data it reads from a text file.

The file has to contain one line for each row of the data frame, containing a value (eg, a number, `TRUE/FALSE`, a string) for each variable for the case corresponding to that row.

If a `header=TRUE` argument is given to `read.table`, the names of the variables will be taken from the first line of the file.

Here's how we could read the heights and weights data frame from a file on the course web page:

```
heights_and_weights <-
  read.table ("http://www.cs.utoronto.ca/~radford/csc120/data7",
                   header=TRUE)
```

The contents of the file read are as below:

```
name height weight
Fred 62 144
Mary 60 131
Joe 71 182
```

Indicating Missing Values with NA

It is very common for data collected to have some missing values — where the subject declined to answer one of the survey questions, or the interviewer forgot to fill out one page of the form, or where the machine taking the readings was broken that day.

Sometimes these values are indicated by some special number like −999. But this is very unreliable. The person analysing the data may not realize that this is what −999 is supposed to mean, leading to drastically incorrect averages. Or there may be an actual, non-missing, value of −999!

R supports representation of missing data by a special NA value. NA can be the value of an element in a vector, matrix, or data frame. For example:

```
> c(5,1,NA,8,NA)
[1]  5  1 NA  8 NA
```

Arithmetic on NA values

Arithmetic operations where one or both operands are NA produce NA as the result:

```
 > a <- c(5,1,NA,8,NA)
 > a+100
[1] 105 101  NA 108  NA
 > b <- c(10,NA,20,NA,NA)
 > a*b
 [1] 50 NA NA NA NA
```

Comparisons with NA also produce NA, rather than TRUE or FALSE. Trying to use NA as an `if` or `while` condition gives an error:

```
> a == 1
[1] FALSE  TRUE    NA FALSE    NA
> if (a[3]==1) cat("true\n") else cat("false\n")
Error in if (a[3] == 1) cat("true\n") else cat("false\n") :
  missing value where TRUE/FALSE needed
```

Checking For NA

Sometimes you need to check whether a value is NA. But you *can't* do this with something like `if (a == NA) ...` — that will always give an error!

Instead, you can use the `is.na` function. It can be applied to a single value, giving TRUE or FALSE, or a vector of values, giving a logical vector.

For example, R's built-in `airquality` demonstration dataset has some NA values. The following statements create a modified version of the `airquality` data frame in which missing values for solar radiation are replaced by the average of all the non-missing measurements (found with `mean` using the `na.rm` option):

```
ave_solar <- mean (airquality$Solar.R, na.rm=TRUE)
mod_airquality <- airquality
for (i in 1:nrow(mod_airquality))
    if (is.na(mod_airquality$Solar.R[i]))
        mod_airquality$Solar.R[i] <- ave_solar
```

(We'll see later how one can do this more easily using logical indexes.)

NA and NaN

A value will also be “missing” if it is the result of an undefined mathematical operation. R prints such values as NaN, not NA, but `is.na` will be TRUE for them. Operations on NaN produce NaN as a result. Here are some examples:

```
> 0/0
[1] NaN
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> x <- 0/0
> 10*x
[1] NaN
> v <- asin((-2):2)
Warning message:
In asin((-2):2) : NaNs produced
> v
[1]       NaN -1.570796  0.000000  1.570796       NaN
> v / 0
[1]  NaN -Inf  NaN  Inf  NaN
```

Using Numeric Vectors as Subscripts

A subscript used with `[ ]` can be a vector of indexes, rather than just one index, yielding a subset of elements having those indexes, not just one element. Some examples, using some variables defined earlier (note how names are carried on):

```
> v
abc def xyz
  9  10   3
> v[c(1,3)]
abc xyz
  9   3
> M
   w x y  z
ab 1 4 7 10
cd 2 5 8 11
ef 3 6 9 12
> M[c(3,1),c(2,4,4)] # Indexes needn't be in order, can be duplicates
   x  z  z
ef 6 12 12
ab 4 10 10
```

Using Logical Vectors as Subscripts

A subscript can also be a logical vector, which selects elements in positions where this subscript is `TRUE`:

```
> v
abc def xyz
  9  10   3
> v[c(TRUE,FALSE,TRUE)]
abc xyz
  9   3
> v[v>5]
abc def
  9  10
```

R's "and" (`&`) and "or" (`|`) operators can be useful for this:

```
> v[v>5 & v<10]
abc
  9
> v[v>9 | v<7]
def xyz
 10   3
```

Changing Elements with a Logical Vector Subscript

We can also assign to elements of a vector using a logical vector to select elements to change.

Here's how we can use this to make a modified version of the `airquality` data frame with missing values for `Solar.R` filled in:

```
    mod_airquality <- airquality
    mod_airquality$Solar.R [is.na(airquality$Solar.R)] <-
      mean (airquality$Solar.R, na.rm=TRUE)
```