R Language Fundamentals

Data Frames

Steven Buechler

Department of Mathematics
276B Hurley Hall; 1-6233

Fall, 2007

Outline

Clean-up and Filter

Some common tasks in preparing a data frame for analysis are:

- Set appropriate rownames (sample ids) and component names (variables).
- Massage factors by collapsing extraneous levels, renaming levels, labeling missing values.
- Decide on which rows or columns to eliminate based on missing values or the specific analysis. Do we eliminate any record with missing data, or just too much missing data?

Breast Cancer Example

The clinical data from a breast cancer study in Sweden are found in the spreadsheet "Clinical_Upps.csv". This is typical supplementary data found on the Gene Expression Omnibus (GEO) at NCBI, and/or accompanying a paper.

Read in Spreadsheet

and get acquainted

```
> clinUPPS <- read.csv("../Lect2/lect2WorkDir/Clinical_Upps
> dim(clinUPPS)

[1] 249  11

> clinUPPS[1:3, 1:4]

    GSM.ID..A.B.chip.       ID Grd DFS_time
1 GSM110625/GSM110874 X100B08   1    11.83
2 GSM110626/GSM110875 X101B88   3    11.83
3 GSM110627/GSM110876 X102B06   3    11.83
```

Name the Rows

```
> rownames(clinUPPS) <- clinUPPS$ID
```

Inspect Components

```
> names(clinUPPS)

 [1] "GSM.ID..A.B.chip." "ID"
 [3] "Grd"               "DFS_time"
 [5] "DFS_event"         "No_tamox"
 [7] "ER_status"         "Lymph_node"
 [9] "p53_mut_status"    "age_at_diagnosis"
[11] "tumor_size_.mm."
```

Viewing the original spreadsheet is not cheating. It reveals the codes used in recording the data.

Inspect No_tamox

```
> clinUPPS$No_tamox[1:5]
[1]  1  1 NA NA NA
```

Correct No_tamox Coding

The coding of the No_tamox component reads as `1, NA`, when the `NA` really codes "Yes, Tamoxifen". So, recode these `NA`'s as 0.

```
> tam <- clinUPPS$No_tamox
> tam[is.na(clinUPPS$No_tamox)] <- 0
> clinUPPS[, 6] <- tam
> clinUPPS[1:5, 6]
```

```
[1] 1 1 0 0 0
```

Any Other NAs?

```
> sum(is.na(clinUPPS))
[1] 0
```

Nope

Characteristics of ER Status

In this experiment we're comparing estrogen receptor positive (ER+) samples with (ER-) samples. Check that each sample has a determined status.

```
> levels(clinUPPS$ER_status)
```

```
[1] "ER+" "ER-" "ER?"
```

```
> table(clinUPPS$ER_status)
```

```
ER+ ER- ER?
211  34   4
```

We need to remove the ER? samples.

Filter by ER Status

```
> clinUPPS2 <- clinUPPS[clinUPPS$ER_status !=
+     "ER?", ]
> table(clinUPPS2$ER_status)

ER+ ER- ER?
211  34   0
```

Clean-up the ER_status factor to remove the defunct level.

```
> clinUPPS2$ER_status <- factor(as.character(clinUPPS2$ER_s
```

Outline

Grouped Expressions

In writing custom functions and loops it's necessary to group a set of expressions together and generate a single output of this set of expressions. Commands may be grouped inside braces. The output of the grouped expressions is the value of the last expression.

Loops

It may be necessary to loop through an index and perform an operation at each iteration. The most common format uses the `for` construct.

```
for (name in vec) {grouped expression }
```

Most commonly, `vec` is an integer range, like `1:20`, so `name` iterates through those integers, and the grouped expression uses `name` as a variable.

`for` Example

Use a for loop to compute the product of the elements of a given vector.

```
> a <- 1:10
> b <- 1
> for (i in 1:length(a)) {
+     b <- b * a[i]
+ }
> b
```

```
[1] 3628800
```

for Example 2

Given a list such that each component is a character vector, create a list whose components are the first entry of the vector.

```
> LL <- list(one = c("a", "b", "c"), two = c("p",
+     "q", "r"), three = c("x", "y"))
> FF <- vector(mode = "list", length = length(LL))
> for (i in 1:length(LL)) {
+     FF[[i]] <- LL[[i]][1]
+ }
```

for Example 2

```
> FF

[[1]]
[1] "a"

[[2]]
[1] "p"

[[3]]
[1] "x"

> ff <- unlist(FF)
> ff

[1] "a" "p" "x"
```

It's ocassionally useful to collapse a list to a vector when the components will allow it.

Other Control Statements

R has an if-else statement in the format

```
if (logical expr) { group 1 } else { group 2 }
```

Other constructs to look up are `ifelse`, `while`, `repeat` and `break`. You'll need these when writing batch programs.

Function Objects

R allows the user to write functions. These can be applied just like core functions (`sum`, `seq`, etc.). The format for creating the function named `fn` is

```
fn <- function( var1, var2, ... ) { expr group }
```

Here, `var1`, etc., are the objects the function takes as input. The return value of the function is the value of the expression group; i.e., the value of the last command in the group.

First Function

Write a function that computes the product of the entries in a vector.

```
> prod1 <- function(x) {
+     b <- 1
+     for (i in 1:length(x)) {
+         b <- b * x[i]
+     }
+     b
+ }
```

Product Function

Examples:

```
> a1 <- c(2, 4, 7)
> b1 <- prod1(a1)
> b1
```

```
[1] 56
```

Functions as Subroutines

Functions can be used as reusable chuncks of code. Given input they execute a set of commands and return some output.

R commands, including function definitions can be stored in an ordinary text file with a `.R` extension. Simply type the commands as you would at a prompt, except they aren't executed. Then execute in *R*,

```
> source("the file")
```

The commands are executed and any defined functions are ready for use.

Defined Functions in `apply`

More essential is the use of defined functions in `apply` and friends.

Problem Given a matrix A find a vector v whose i^{th} entry is `max(row i) - min(row i)`.

Defined Functions in apply

```
> maxMin <- function(x) {
+     max(x) - min(x)
+ }
> A <- matrix(rnorm(16), nrow = 4, ncol = 4)
> v <- apply(A, 1, maxMin)
> v
[1] 1.3582 2.0209 2.2516 0.8158
```

Defined Functions in apply

More commonly, the function definition is included with the apply.

```
> vv <- apply(A, 1, function(x) {
+     max(x) - min(x)
+ })
> vv
```

```
[1] 1.3582 2.0209 2.2516 0.8158
```

The `lapply` Function

The `lapply` function alows batch operation of a function on the components of a list. This can replace many (slow) `for` loops.

Given a list `LL` with `n` components, suppose we want to apply a function successively to `LL[[i]]`, for each `i`, and to collect all the output. Here, it's assumed that `fn(i)` gives the desired output for the input `LL[[i]]`.

The `lapply` Function

The format:

```
FF <- lapply( 1:n, fn)
```

Then `FF` is a list with `n` components, `FF[[i]]` is the result of applying the function to `LL[[i]]`.

The lapply Function

Example

Consider the for loop example in which all list components are character vectors and we want to extract the first entry from each component. This is done with lapply as follows. Note, LL already exists.

```
> M <- lapply(1:length(LL), function(x) {
+     LL[[x]][1]
+ })
> unlist(M)
```

```
[1] "a" "p" "x"
```