Module 6

Data Manipulation

Andrew Jaffe
Instructor

Manipulating Data

So far, we've covered how to read in data, and select specific rows and columns. All of these steps help you set up your analysis or data exploration. Now we are going to cover manipulating your data and summarizing it using basic statistics and visualizations.

Sorting and ordering

`sort(x, decreasing=FALSE)`: 'sort (or order) a vector or factor (partially) into ascending or descending order.' Note that this returns an object that has been sorted/ordered

`order(...,decreasing=FALSE)`: 'returns a permutation which rearranges its first argument into ascending or descending order, breaking ties by further arguments.' Note that this returns the indices corresponding to the sorted data.

```
> x = c(1, 4, 7, 6, 4, 12, 9, 3)
> sort(x)
```

```
[1]  1  3  4  4  6  7  9 12
```

```
> order(x)
```

```
[1] 1 8 2 5 4 3 7 6
```

Note you would have to assign the sorted variable to a new variable to retain it

Sorting and ordering

```
> dat = read.csv("data/charmcitycirc_reduced.csv", header = TRUE, as.is = TRUE)
> dat2 = dat[, c("day", "date", "orangeAverage", "purpleAverage", "greenAverage",
+     "bannerAverage", "daily")]
> head(order(dat2$daily, decreasing = TRUE))
```

```
[1] 888 887 886 971 880 866
```

```
> head(sort(dat2$daily, decreasing = TRUE))
```

```
[1] 22075 21951 17580 16714 16366 16150
```

The first indicates the rows of `dat2` ordered by daily average ridership. The second displays the actual sorted values of daily average ridership.

Sorting and ordering

```
> datSorted = dat2[order(dat2$daily, decreasing = TRUE), ]
> datSorted[1:5, ]
```

```
         day       date orangeAverage purpleAverage greenAverage
888 Saturday 06/16/2012          6322          7797         3338
887   Friday 06/15/2012          6926          8090         3485
886 Thursday 06/14/2012          5618          6521         2770
971   Friday 09/07/2012          5718          7007         2688
880   Friday 06/08/2012          5782          6882         2858
    bannerAverage daily
888        4617.0 22075
887        3450.0 21951
886        2672.0 17580
971        1301.0 16714
880         844.5 16366
```

Sorting and ordering

Note that the row names refer to their previous values. You can do something like this to fix:

```
> rownames(datSorted) = NULL
> datSorted[1:5, ]
```

```
       day       date orangeAverage purpleAverage greenAverage
1 Saturday 06/16/2012          6322          7797         3338
2   Friday 06/15/2012          6926          8090         3485
3 Thursday 06/14/2012          5618          6521         2770
4   Friday 09/07/2012          5718          7007         2688
5   Friday 06/08/2012          5782          6882         2858
  bannerAverage daily
1        4617.0 22075
2        3450.0 21951
3        2672.0 17580
4        1301.0 16714
5         844.5 16366
```

Creating categorical variables

the `rep()` ["repeat"] function is useful for creating new variables

```
> bg = rep(c("boy", "girl"), each = 50)
> head(bg)
```

```
[1] "boy" "boy" "boy" "boy" "boy" "boy"
```

```
> bg2 = rep(c("boy", "girl"), times = 50)
> head(bg2)
```

```
[1] "boy"  "girl" "boy"  "girl" "boy"  "girl"
```

```
> length(bg) == length(bg2)
```

```
[1] TRUE
```

Creating categorical variables

One frequently-used tool is creating categorical variables out of continuous variables, like generating quantiles of a specific continuously measured variable.

A general function for creating new variables based on existing variables is the `ifelse()` function, which "returns a value with the same shape as test which is filled with elements selected from either yes or no depending on whether the element of test is `TRUE` or `FALSE`."

```
ifelse(test, yes, no)

# test: an object which can be coerced to logical mode.
# yes: return values for true elements of test.
# no: return values for false elements of test.
```

Creating categorical variables

For example, we can create a new variable that records whether daily ridership on the Circulator was above 10,000.

```
> hi_rider = ifelse(dat$daily > 10000, 1, 0)
> head(hi_rider)
```

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

```
> table(hi_rider)
```

```
hi_rider
  0   1
740 282
```

Creating categorical variables

You can also nest `ifelse()` within itself to create 3 levels of a variable.

```
> riderLevels = ifelse(dat$daily < 10000, "low", ifelse(dat$daily > 20000, "high",
+     "med"))
> head(riderLevels)
```

```
[1] "low" "low" "low" "low" "low" "low"
```

```
> table(riderLevels)
```

```
riderLevels
high  low  med
   2  740  280
```

Creating categorical variables

However, it's much easier to use `cut()` to create categorical variables from continuous variables.

'cut divides the range of x into intervals and codes the values in x according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on.'

```
cut(x, breaks, labels = NULL, include.lowest = FALSE,
right = TRUE, dig.lab = 3,
ordered_result = FALSE, ...)
```

`x`: a numeric vector which is to be converted to a factor by cutting.

`breaks`: either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut.

`labels`: labels for the levels of the resulting category. By default, labels are constructed using "(a,b]" interval notation. If `labels = FALSE`, simple integer codes are returned instead of a factor.

Factors

Factors are used to represent categorical data, and can also be used for ordinal data (ie categories have an intrinsic ordering)

Note that R reads in character strings as factors by default in functions like `read.table()`

'The function factor is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument ordered is TRUE, the factor levels are assumed to be ordered. For compatibility with S there is also a function ordered.'

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x))
```

Factors

Suppose we have a vector of case-control status

```
> cc = factor(c("case", "case", "case", "control", "control", "control"))
> cc
```

```
[1] case    case    case    control control control
Levels: case control
```

```
> levels(cc) = c("control", "case")
> cc
```

```
[1] control control control case    case    case
Levels: control case
```

Factors

Note that the levels are alphabetically ordered by default. We can also specify the levels within the factor call

```
> factor(c("case", "case", "case", "control", "control", "control"), levels = c("control",
+     "case"))
```

```
[1] case    case    case    control control control
Levels: control case
```

```
> factor(c("case", "case", "case", "control", "control", "control"), levels = c("control",
+     "case"), ordered = TRUE)
```

```
[1] case    case    case    control control control
Levels: control < case
```

Factors

Factors can be converted to `numeric` or `character` very easily

```
> x = factor(c("case", "case", "case", "control", "control", "control"), levels = c("control",
+     "case"))
> as.character(x)
```

```
[1] "case"    "case"    "case"    "control" "control" "control"
```

```
> as.numeric(x)
```

```
[1] 2 2 2 1 1 1
```

Cut

Now that we know more about factors, `cut()` will make more sense:

```
> x = 1:100
> cx = cut(x, breaks = c(0, 10, 25, 50, 100))
> head(cx)
```

```
[1] (0,10] (0,10] (0,10] (0,10] (0,10] (0,10]
Levels: (0,10] (10,25] (25,50] (50,100]
```

```
> table(cx)
```

```
cx
  (0,10]  (10,25]  (25,50] (50,100]
      10       15       25       50
```

Cut

We can also leave off the labels

```
> cx = cut(x, breaks = c(0, 10, 25, 50, 100), labels = FALSE)
> head(cx)
```

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

```
> table(cx)
```

```
cx
 1  2  3  4
10 15 25 50
```

Cut

Note that you have to specify the endpoints of the data, otherwise some of the categories will not be created

```
> cx = cut(x, breaks = c(10, 25, 50), labels = FALSE)
> head(cx)
```

```
[1] NA NA NA NA NA NA
```

```
> table(cx)
```

```
cx
 1  2
15 25
```

```
> table(cx, useNA = "ifany")
```

```
cx
   1    2 <NA>
  15   25   60
```

Adding to data frames

```
> dat2$riderLevels = cut(dat2$daily, breaks = c(0, 10000, 20000, 1e+05))
> dat2[1:2, ]
```

```
      day       date orangeAverage purpleAverage greenAverage
1  Monday 01/11/2010           952            NA           NA
2 Tuesday 01/12/2010           796            NA           NA
  bannerAverage daily riderLevels
1            NA   952   (0,1e+04]
2            NA   796   (0,1e+04]
```

```
> table(dat2$riderLevels, useNA = "always")
```

```
   (0,1e+04] (1e+04,2e+04] (2e+04,1e+05]          <NA>
         731           280             2           133
```

Making 2D objects

We can make matrices from "scratch" using the `matrix()` function.

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
  dimnames = NULL)
```

`data`: a data vector.

`nrow`: the number of rows

`ncol`: the number of columns

`byrow`: does the data fill in the matrix across the rows or down the columns?

Matrices

```
> m1 = matrix(1:9, nrow = 3, ncol = 3, byrow = FALSE)
> m1
```

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

```
> m2 = matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
> m2
```

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

Adding rows and columns

More generally, you can add columns (or another matrix/data frame) to a data frame or matrix using `cbind()` ('column bind'). You can also add rows (or another matrix/data frame) using `rbind()` ('row bind').

Note that the vector you are adding has to have the same length as the number of rows (for `cbind()`) or the number of columns (`rbind()`)

When binding two matrices, they must have either the same number of rows or columns

```
> cbind(m1, m2)
```

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

Adding rows and columns

```
> rbind(m1, m2)
```

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

Adding columns manually

```
> dat2$riderLevels = NULL
> rider = cut(dat2$daily, breaks = c(0, 10000, 20000, 1e+05))
> dat2 = cbind(dat2, rider)
> dat2[1:2, ]
```

```
      day       date orangeAverage purpleAverage greenAverage
1  Monday 01/11/2010           952            NA           NA
2 Tuesday 01/12/2010           796            NA           NA
  bannerAverage daily     rider
1            NA   952 (0,1e+04]
2            NA   796 (0,1e+04]
```

Making a data frame

```
data.frame(col1 = [vector], col2 = [vector], ..., stringsAsFactors=FALSE)
```

```
> df = data.frame(Date = dat$day, orangeLine = dat$orangeAverage, purpleLine = dat$purpleAvera
> df[1:5, ]
```

```
       Date orangeLine purpleLine
1    Monday        952         NA
2   Tuesday        796         NA
3 Wednesday       1212         NA
4  Thursday       1214         NA
5    Friday       1644         NA
```

Other manipulations

- `abs(x)`: absolute value
- `sqrt(x)`: square root
- `ceiling(x)`: ceiling(3.475) is 4
- `floor(x)`: floor(3.475) is 3
- `trunc(x)`: trunc(5.99) is 5
- `round(x, digits=n)`: round(3.475, digits=2) is 3.48
- `signif(x, digits=n)`: signif(3.475, digits=2) is 3.5
- `cos(x)`, `sin(x)`, `tan(x)` also `acos(x)`, `cosh(x)`, `acosh(x)`, etc.
- `log(x)`: natural logarithm
- `log10(x)`: common logarithm
- `exp(x)`: ex

(via: http://statmethods.net/management/functions.html)