Lecture 2

Data Summarization and Manipulation

Andrew Jaffe
Instructor

Start New Script

Start a new script for lecture 2, and add the appropriate header

Data Output

While its nice to be able to read in a variety of data formats, it's equally important to be able to output data somewhere.

write.table(): prints its required argument x (after converting it to a data frame if it is not one nor a matrix) to a file or connection.

```
write.table(x,file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

Data Output

x: the R data frame or matrix you want to write

file: the file name where you want to R object written. It can be an absolute path, or a filename (which writes the file to your working directory)

sep: what character separates the columns?

- "," = .csv - Note there is also a write.csv() function
- "\t" = tab delimited

row.names: I like setting this to FALSE because I email these to collaborators who open them in Excel

Data Output

For example, from the Homework 1 Dataset:

```
> dat = read.csv("C:/Users/Andrew/Dropbox/WinterRClass/Datasets/OpenBaltimore/Charm_City_Circulator_Ride
+     header = T, as.is = T)
> dat2 = dat[, c("day", "date", "orangeAverage", "purpleAverage", "greenAverage",
+     "bannerAverage", "daily")]
> write.csv(dat2, file = "charmcitycirc_reduced.csv", row.names = FALSE)
```

Note that row.names=TRUE would make the first column contain the row names, here just the numbers 1:nrow(dat2), which is not very useful for excel.

Saving R Data

It's very useful to be able to save collections of R objects for future analyses.

For example, if a task takes several hours(/days) to run, it might be nice to run it once and save the results for downstream analyses.

save(...,file="[name].rda")

where "..." is as many R objects, referenced by unquoted variable names, as you want to save.

For example, from the homework:

```
> save(dat, dat2, file = "charmcirc.rda")
```

Saving R Data

You also probably have noticed the prompt when you close R about saving your workspace. The workspace is the collection of R objects and custom R functions in your current environment. You can check the workspace with ls():

```
> ls()
```

```
[1] "dat"  "dat2" "f"    "x"
```

Saving the workspace will save all of these files in your current working directory as a hidden file called ".Rdata". The function save.image() also saves the entire workspace, but you can give your desired file name as an input (which is nicer because the file is not hidden).

Note that R Studio should be able to open any .rda or .Rdata file. Opening one of these file types from Windows Explorer or OSX's Finder loads all of the objects into your workspace and changes your working directory to wherever the file was located.

Loading R Data

You can easily load any '.rda' or '.Rdata' file with the load() function:

```
> tmp = load("charmcirc.rda")
> tmp
```

```
[1] "dat"  "dat2"
```

```
> ls()
```

```
[1] "dat"  "dat2" "f"    "tmp"  "x"
```

Note that this saves the R object names as character strings in an object called 'tmp', which is nice if you already have a lot of items in your working directory, and/or you don't know exactly which got loaded in

Removing R Data

You can easily remove any R object(s) using the rm() or remove() functions, and they are no longer in your R environment (which you can confirm with running 'ls()')

You can also remove all of the objects you have added to your worksapce with:

```
rm(list = ls())
```

Subsetting Data

Often you only want to look at subsets of a data set at any given time. As a review, elements of an R object are selected using the brackets.

Today we are going to look at more flexible ways of identifying which rows of a dataset to select

Subsetting Data

Note: there is a convenience function for subsetting, called subset(), which takes the R object, the logical statement to identify the index of the rows to take, and then an option to select a subset of the columns:

```
subset(x, subset, select, drop = FALSE, ...)
```

However, the function comes with a warning in the help file:

"Warning: This is a convenience function intended for use interactively. For programming it is better to use the standard subsetting functions like [, and in particular the non-standard evaluation of argument subset can have unanticipated consequences."

Therefore, we are only going to use the brackets for selecting data in this class.

Subsetting Data

You can put a negative integers inside brackets to remove these indices from the data.

```
> x = c(1, 3, 77, 54, 23, 7, 76, 5)
> x[1:3]  # first 3
```

```
[1]  1  3 77
```

```
> x[-2]  # all but the second
```

```
[1]  1 77 54 23  7 76  5
```

Subsetting Data

Note that you have to be careful with this syntax when dropping more than 1 element:

```
> x[-c(1, 2, 3)]  # drop first 3
```

```
[1] 54 23  7 76  5
```

```
> x[-1:3]  # shorthand
```

```
Error: only 0's may be mixed with negative subscripts
```

```
> x[-(1:3)]  # needs parentheses
```

```
[1] 54 23  7 76  5
```

Subsetting Data

Sometimes you want to select a specific sequence of rows from a data frame. Here, the seq() command comes in handy. We already saw one specific application using the colon, but seq() is much more flexible.

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
# Typical usages are:
#seq(from, to)
#seq(from, to, by=)
#seq(from, to, length.out= )
#seq(along.with= )
#seq(from)
#seq(length.out= )
```

where 'from' and 'to' are integers. 'by' can be any numeric value.

seq()

```
> seq(1, 10, by = 2)  # odds
```

```
[1] 1 3 5 7 9
```

```
> seq(2, 10, by = 2)  # evens
```

```
[1]  2  4  6  8 10
```

```
> seq(1, 10, length.out = 3)
```

```
[1]  1.0  5.5 10.0
```

seq()

The 'along.with' argument becomes useful later when we talk about R programming, but here is taste:

```
> x
```

```
[1]  1  3 77 54 23  7 76  5
```

```
> seq(along = x)
```

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

This is essentially a sequence from 1 to length(x)

seq()

'by' can also be negative, but be careful. You can also create sequences from larger numbers to smaller numbers.

```
> seq(1, 10, by = -2)  # odds
```

```
Error: wrong sign in 'by' argument
```

```
> seq(10, 1, by = -2)  # odds
```

```
[1] 10  8  6  4  2
```

```
> seq(10, 1, by = 2)  # evens
```

```
Error: wrong sign in 'by' argument
```

seq()

We can take all of the even rows in a data frame:

```
> head(dat2, 2)  # only the first 2 rows
```

```
      day       date orangeAverage purpleAverage greenAverage
1  Monday 01/11/2010           952            NA           NA
2 Tuesday 01/12/2010           796            NA           NA
  bannerAverage daily
1            NA   952
2            NA   796
```

```
> head(dat2[seq(2, nrow(dat2), by = 2), ], 2)
```

```
      day       date orangeAverage purpleAverage greenAverage
2  Tuesday 01/12/2010           796            NA           NA
4 Thursday 01/14/2010          1214            NA           NA
  bannerAverage daily
2            NA   796
4            NA  1214
```

Selecting on multiple queries

You can select rows where a value is allowed to be several categories. In the homework, we had to subset the Charm City Circulator dataset by each day. How can we select rows that are 1 of 2 days?

The %in% operator proves useful: '%in% is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand.'

```
> (dat$day %in% c("Monday", "Tuesday"))[1:20]  # select entries that are monday or tuesday
```

```
 [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[12] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

```
> which(dat$day %in% c("Monday", "Tuesday"))[1:20]  # which indices are true?
```

```
 [1]  1  2  8  9 15 16 22 23 29 30 36 37 43 44 50 51 57 58 64 65
```

Selecting on multiple queries

What about selecting rows based on the values of two variables? We can 'chain' together logical statements using the following:

- '&' : AND
- '|' : OR

```
> # which Mondays had more than 3000 average riders?
> which(dat$day == "Monday" & dat$daily > 3000)[1:20]
```

```
 [1] 148 155 162 169 176 183 190 197 204 211 218 225 232 239 246 253 260
[18] 267 274 281
```

AND

Which days had more than 10000 riders overall and more than 3000 riders on the purple line?

```
> Index = which(dat$daily > 10000 & dat$purpleAverage > 3000)
> length(Index)  # the number of days
```

```
[1] 280
```

```
> head(dat2[Index, ], 2)  # first 2 rows
```

```
         day       date orangeAverage purpleAverage greenAverage
551   Friday 07/15/2011          4705          6293           NA
552 Saturday 07/16/2011          4624          7622           NA
    bannerAverage daily
551            NA 10998
552            NA 12246
```

OR

Which days had more than 10000 riders overall or more than 3000 riders on the purple line?

```
> Index = which(dat$daily > 10000 | dat$purpleAverage > 3000)
> length(Index)  # the number of days
```

```
[1] 600
```

```
> head(dat2[Index, ], 2)  # first 2 rows
```

```
         day       date orangeAverage purpleAverage greenAverage
180   Friday 07/09/2010          2847          3094           NA
188 Saturday 07/17/2010          1513          3562           NA
    bannerAverage daily
180            NA  5941
188            NA  5076
```

Subsetting with missing data

Note that logical statements cannot evaluate missing values, and therefore returns an NA:

```
> dat$purpleAverage[1:10] > 0
```

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

```
> which(dat$purpleBoardings > 0)[1:10]
```

```
 [1] 148 149 150 151 152 153 154 155 156 157
```

Subsetting with missing data

You can use the complete.cases() function on a data frame, matrix, or vector, which returns a logical vector indicating which cases are complete, i.e., they have no missing values.

Subsetting columns

We touched on this last class. You can select columns using the variable/column names or column index

```
> dat[1:3, c("purpleAverage", "orangeAverage")]
```

```
  purpleAverage orangeAverage
1            NA           952
2            NA           796
3            NA          1212
```

```
> dat[1:3, c(8, 5)]
```

```
  purpleAverage orangeAverage
1            NA           952
2            NA           796
3            NA          1212
```

Subsetting columns

You can also remove a column by setting its value to NULL

```
> tmp = dat2
> tmp$daily = NULL
> tmp[1:3, ]
```

```
        day       date orangeAverage purpleAverage greenAverage
1    Monday 01/11/2010            952            NA           NA
2   Tuesday 01/12/2010            796            NA           NA
3 Wednesday 01/13/2010           1212            NA           NA
  bannerAverage
1            NA
2            NA
3            NA
```

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
```

Sorting and ordering

```
> 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

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"), labels = c("control",
+     "case"))
```

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

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

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

Factors

Factors can be converted to numeric or character very easily

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

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

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

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

Cut

Now that we know 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
```

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            12
```

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$purpleAverage)
> 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
```

Data Summarization

- Basic statistical summarization
  - mean(x): takes the mean of x
  - sd(x): takes the standard deviation of x
  - median(x): takes the median of x
  - quantile(x): displays sample quantities of x. Default is min, IQR, max
  - range(x): displays the range. Same as c(min(x), max(x))
- Basic summarization plots
  - plot(x,y): scatterplot of x and y
  - boxplot(y~x): boxplot of y against levels of x
  - hist(x): histogram of x
  - density(X): kernel density plot of x

Data Summarization on matrices/data frames

- Basic statistical summarization
  - rowMeans(x): takes the means of each row of x
  - colMeans(x): takes the means of each column of x
  - rowSums(x): takes the sum of each row of x
  - colSums(x): takes the sum of each column of x
  - summary(x): for data frames, displays the quantile information
- Basic summarization plots
  - matplot(x,y): scatterplot of two matrices, x and y
  - pairs(x,y): plots pairwise scatter plots of matrices x and y, column by column

colMeans and rowMeans

```
> tmp = dat2[, 3:6]
> colMeans(tmp, na.rm = TRUE)
```

```
orangeAverage purpleAverage  greenAverage bannerAverage
         2994           4013           1951            964
```

```
> head(rowMeans(tmp, na.rm = TRUE))
```

```
[1]  952  796 1212 1214 1644 1490
```

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)

Apply statements

You can apply more general functions to the rows or columns of a matrix or data frame, beyond the mean and sum.

```
apply(X, MARGIN, FUN, ...)
```

X : an array, including a matrix.

MARGIN : a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.

FUN : the function to be applied: see 'Details'. In the case of functions like +, %*%, etc., the function name must be backquoted or quoted.

... : optional arguments to FUN.

Apply statements

```
> tmp = dat2[, 3:6]
> apply(tmp, 2, mean, na.rm = TRUE)  # column means
```

```
orangeAverage purpleAverage  greenAverage bannerAverage
         2994           4013           1951            964
```

```
> apply(tmp, 2, sd, na.rm = TRUE)  # columns sds
```

```
orangeAverage purpleAverage  greenAverage bannerAverage
       1258.7         1442.4          613.7          527.1
```

```
> head(apply(tmp, 2, max, na.rm = TRUE))  # row maxs
```

```
orangeAverage purpleAverage  greenAverage bannerAverage
         6926           8090           5094           4617
```

Other Apply Statements

- tapply(): 'table' apply
- lapply(): 'list' apply [tomorrow]
- sapply(): 'simple' apply [tomorrow]
- Other less used ones...

See more details here: http://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/

tapply()

From the help file: "Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors."

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

Simply put, you can apply functions FUN to X within each categorical level of INDEX. It is very useful for assessing properties of continuous data by levels of categorical data.

tapply()

For example, we can estimate the highest average daily ridership for each day of the week in 1 line in the Circulator dataset.

```
> tapply(dat$daily, dat$day, max, na.rm = TRUE)
```

```
   Friday    Monday  Saturday    Sunday  Thursday   Tuesday Wednesday
    21951     13982     22075     15224     17580     14776     15672
```

Lab A

Bike Lanes Dataset: BikeBaltimore is the Department of Transportation's bike program.

https://data.baltimorecity.gov/Transportation/Bike-Lanes/xzfj-gyms

Download as a CSV (like the Monuments dataset) in your current working directory

Lab A

1. How many bike "lanes" are currently in Baltimore?
2. How many (a) feet and (b) miles of bike "lanes" are currently in Baltimore?
3. How many types of bike lanes are there? Which type has (a) the most number of and (b) longest average bike lane length?
4. How many different projects do the "bike" lanes fall into? Which project category has the longest average bike lane?
5. (a) Numerically and (b) graphically describe the distribution of bike "lane" lengths. Then describe after stratifying by i) type then ii) number of lanes

Lab B

Download the CSV: http://biostat.jhsph.edu/~ajaffe/files/indicatordeadkids35.csv

```
death = read.csv("http://biostat.jhsph.edu/~ajaffe/files/indicatordeadkids35.csv",
                 as.is=T,header=TRUE, row.names=1)
```

Via: http://www.gapminder.org/data/

Definition of indicator: How many children the average couple had that die before the age 35.

Lab B

1. How many countries have data in any year?
2. When did measurements in the US start?
3. How many countries, and which, had data the first year of measuring?
4. Display the average number of children lost per family versus year across all countries.
5. Display the distribution of average country's count across all year.
6. How many entries are less than 1? Which array indices do they correspond to?
7. Bonus: Plot the count for each country across year in a line plot