Data Without Borders

Lecture 2

Subsets, indexes, and plots

Today

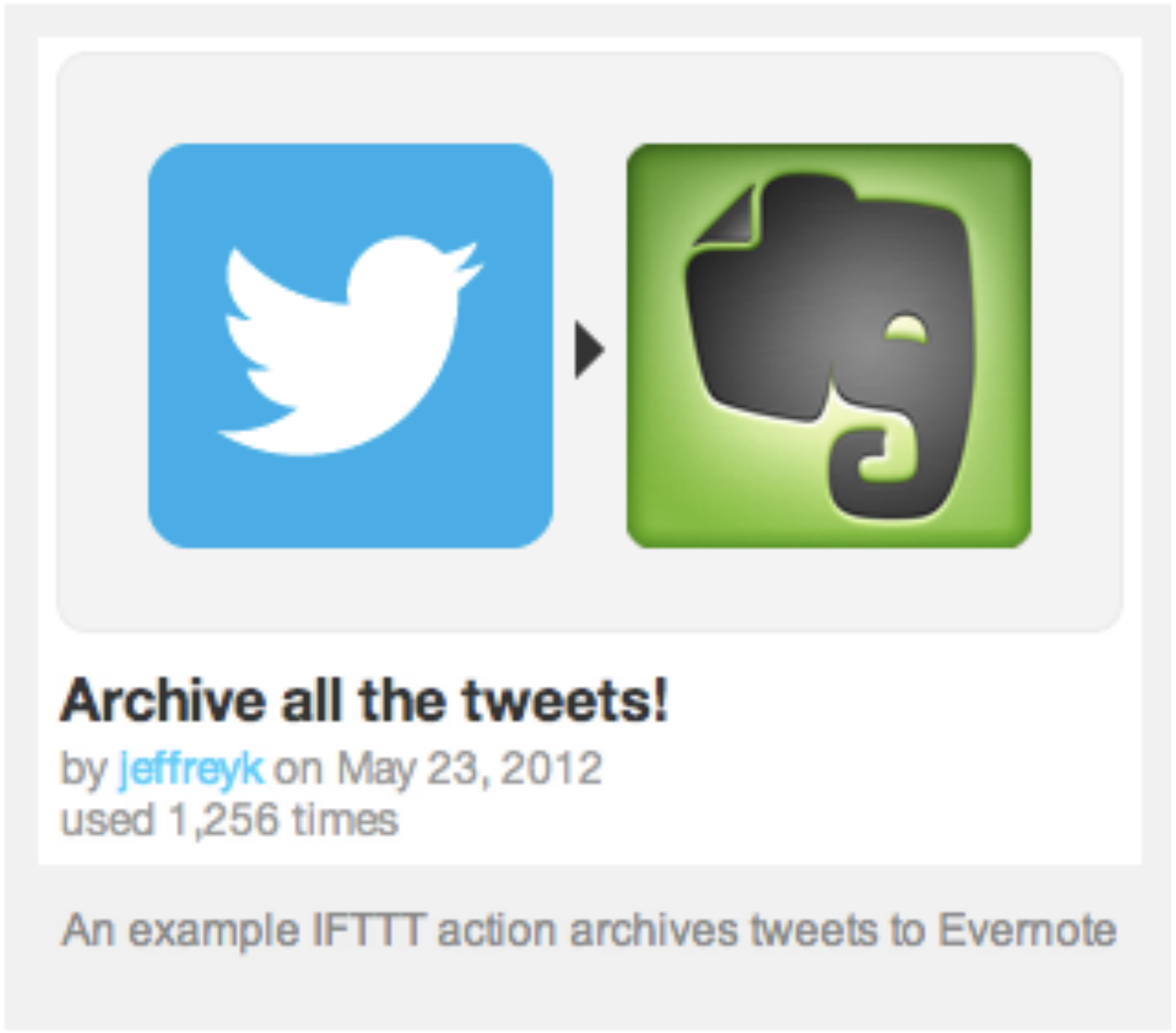
- Review of last week's notes
- Your homework
- More on vectors - subsets and indexes
- Plotting - making plot() prettier and seeing other views of data.
- Fun with strings

Goals

By the end of class today, you should be able to:

- Ask R for any logical subset of rows from your data
- Add and remove columns from data frames
- Know the difference between vectors and data frames.
- Plot various views of your data using plot() as well as other graphing functions

Data-of-the-day



Archive all the tweets!

by jeffreyk on May 23, 2012

used 1,256 times

An example IFTTT action archives tweets to Evernote

http://radar.oreilly.com/2012/09/true-data-liberation.html

Review of Last Week's Notes

- You can load data from the web using `read.csv()`!
- R's basic unit is the function, which can take named arguments to modify their behavior!
- Unlike Processing, in which member functions belong to objects of interest, R applies the same function across many objects - plot(x), plot(table(x)), etc.
- R thinks in terms of data frames and vectors, in addition to scalar data types.
- We can select rows and columns from data by specifying indices or TRUE / FALSE values (more on this today)
- Plotting is super simple!

Questions on the Homework?

NOTE:

- I found a cleaned version of the Stop n Frisk data that we'll be using from now on out. It's *almost* identical to last week's data, except that coded variables (e.g. race) have been replaced with numerical values. You can find a description of all the new variables in the dataset at http://jakeporway.com/teaching/resources/SNF_codes.pdf
- There's also a new dataset with more variables we'll be using today at http://jakeporway.com/teaching/data/snf_2.csv
- Instead of the generic "data" variable we used in Lecture 1 I'll use the variable "snf" for our main data.
- `snf <- read.csv("http://jakeporway.com/teaching/data/snf_2.csv")`

+

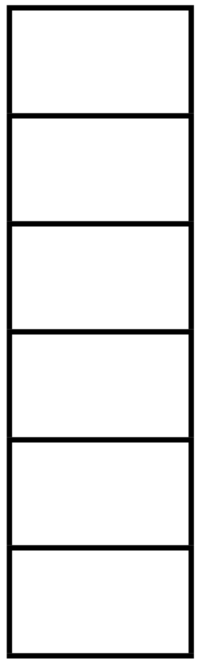
You may have run across this:

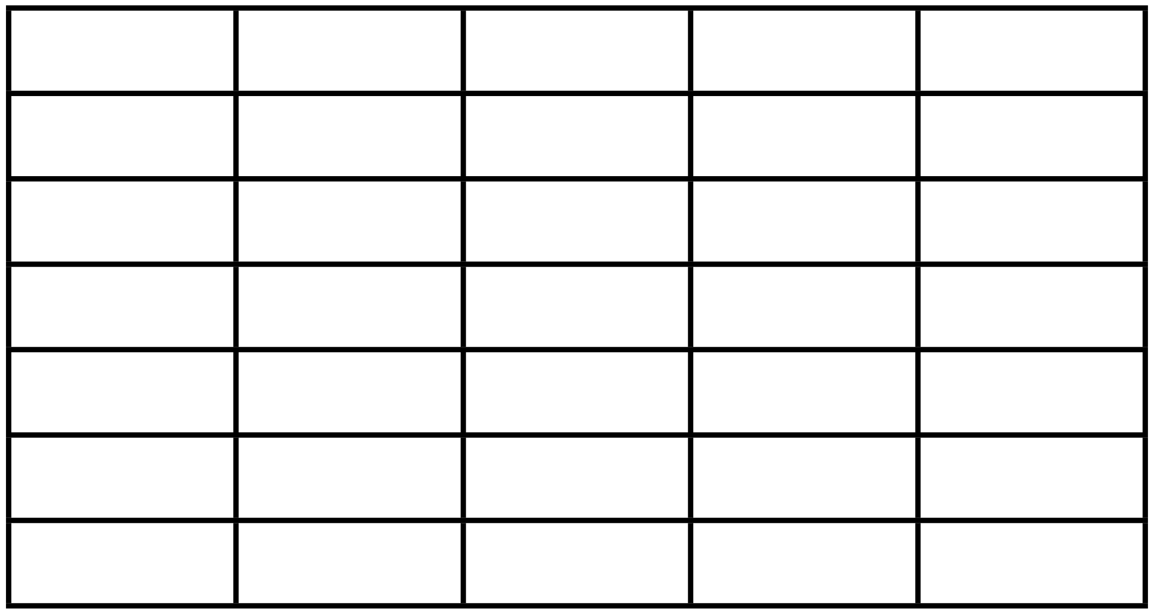
```
> head(snf
+
```

The “+” just means RStudio is waiting for more input.

If you ever want out, just use “Esc”

Vectors vs. Data Frames





Vector:

- 1D
- All same type
- Uses "length()" for size

Data Frame:

- 2D
- A collection of columns
- Columns are of same type
- Uses "dim()" for size

Data Frames

$ is used to index columns of data frames.

Data frames are easy to make:

```
x <- data.frame(1:10, 101:110, letters[1:10])

mini.subset <- data.frame(snf$age, snf$time)
# you would never do it this way, you'd use
# subsetting
```

Vectors are even easier:

```
x <- c(1, 2, 5, 6, 10)
```

table() vs. data frames

table() is a function that counts the number of times each value appears in a vector. The result is a special object called a "table", that is basically a vector with names.

Data frames are 2D representations. We can create them with the data.frame() function.

Indexing

Super important for accessing data in vectors / data frames.

R will return rows if you select them:

	crime.suspected	**frisked**	**searched**	
1	CSCS	0	0	TRUE
2	MISD	1	1	FALSE
3	MISD	1	1	FALSE
4	FEL	1	0	TRUE
5	FEL	1	0	TRUE
6	FEL	1	0	TRUE

Indexing

Super important for accessing data in vectors / data frames.

R will return rows if you select them:

By row number:

c(1, 4, 5, 6)

	crime.suspected	frisked	searched	
1	CSCS	0	0	TRUE
2	MISD	1	1	FALSE
3	MISD	1	1	FALSE
4	FEL	1	0	TRUE
5	FEL	1	0	TRUE
6	FEL	1	0	TRUE

Indexing

Super important for accessing data in vectors / data frames.

R will return rows if you select them:

By row number:

c(1, 4, 5, 6)

	crime.suspected	**frisked**	**searched**
1	CSCS	0	0
2	MISD	1	1
3	MISD	1	1
4	FEL	1	0
5	FEL	1	0
6	FEL	1	0

By TRUE / FALSE for each row:

TRUE
FALSE
FALSE
TRUE
TRUE
TRUE

Indexing

Super important for accessing data in vectors / data frames.

R will return rows if you select them:

By TRUE / FALSE for each row:

By row number:

c(1, 4, 5, 6)

	crime.suspected	frisked	searched	
1	CSCS	0	0	TRUE
2	MISD	1	1	FALSE
3	MISD	1	1	FALSE
4	FEL	1	0	TRUE
5	FEL	1	0	TRUE
6	FEL	1	0	TRUE

Logical expressions (like <, >, ==, !, &, |) create TRUE / FALSE vectors:

Indexing

Super important for accessing data in vectors / data frames.

R will return rows if you select them:

By row number:

c(1, 4, 5, 6)

By TRUE / FALSE for each row:

	crime.suspected	frisked	searched	
1	CSCS	0	0	TRUE
2	MISD	1	1	FALSE
3	MISD	1	1	FALSE
4	FEL	1	0	TRUE
5	FEL	1	0	TRUE
6	FEL	1	0	TRUE

Logical expressions (like <, >, ==, !, &, |) create TRUE / FALSE vectors:

```
searched == 0
```

Indexing

To put a finer point on this:

We can index by TRUE/FALSE vectors

- TRUE/FALSE vectors can be created with logical expressions

We can index by row and column numbers

- Row and column numbers can be created with `which(<logical expression>)`

Subsetting

And by using square brackets, we can subset the data based on *either* logical vectors or numerical vectors:

```
snf[snf$age < 30,] # All the stops of people
                   # under 30

snf[which(snf$age < 30), ] # Totally the same
                           # thing!
```

Logical Vectors vs. Subsets

OK, but here's where things get fun - a lot of times we want to know the number of times something's true (e.g. number of times men are arrested, number of people under 30 who were stopped, etc.) There are two major ways to do this:

```
under.30 <- snf[snf$age < 30,] # Subset of people < 30 stopped
nrow(under.30)  # Because we subset, the number of people < 30 is
                # the number of true rows

# OR

sum(snf$age < 30)  # Whuuuu?
```

Let's Break This Down...

```
sum(snf$age < 30)
```

What is R doing here?

`snf$age < 30` is a logical vector. Cool.

`sum()` adds up all the elements in a vector. OK...

Because you're passing a vector of logicals into a function that adds numbers, R thinks "Hmm, OK, I need to convert this to a number".

It converts every TRUE to 1, every FALSE to 0

Then it adds 'em up.

Let's Break This Down...

```
sum(snf$age < 30)
```

So this raises the question of when to use sum() and when to use length():

length() will give you the length() of a vector, so you want to use it when your vector is a *subset* of the data, e.g. snf $age[snf$age < 30] or which(snf$age < 30)

sum() will actually sum numbers up, but we can abuse it a little to take sums of logical vectors, i.e. all the TRUE values. e.g. sum(snf$age < 30)

Head of the Class

R has basic classes: numerics (including integers and doubles), characters, logicals, and then some fun stuff we'll talk about later. You can check the class of an object with the `class()` function:

```
x <- 3
class(x)
“numeric”

x <- “jake”
class(x)
“character”
```

You can convert between classes with "`as.`", e.g. `as.integer()` or `as.character()`:

```
x <- 3
as.character(x)
“3”

as.integer(snf$age < 30)
```

Logical Expressions for Subsetting

OK, back to subsetting. Just like Processing, R has logical operators (AND, OR, NOT) and comparators (<, >, <=, >=, ==):

```
snf[snf$age < 30 & snf$frisked == 1,]  # Which people
under 30 got frisked?

snf[snf$precinct == 24 | !snf$arrested,] # anyone from
precinct 24 or not arrested
```

R uses a single '&' for AND and a single '|' for OR when combining logical vectors

Vectorization

If you type:

```
sqrt(36)
```

You get the square root of 36. That's cool.

If you type:

```
sqrt(snf$age)
```

You get back a vector of all the square roots of every age. This is kind of neat - give an R function a vector and it'll apply that function to each element of the vector and return the result. This is because R tries to "vectorize" as much as it can, i.e. apply functions over entire vectors (when it makes sense).

Vectorization

Wanna see some other cool vectorizations?

```
x <- runif(10) # runif() gives you as many
               #(r)andom (unif)orm numbers
               # from 0 to 1 as you specify
round(x)       # round each element of x


tolower(snf$crime.suspected)[1:10]
      # note how we can chain subsetting with functions
```

Wanna see another cool side effect of vectorizing?

The Recycling Rule

What does this do?

```
x <- 1:10
x + 3
```

Ho! Vectorizes the addition by 3! OK, hotshot, what about *this*?

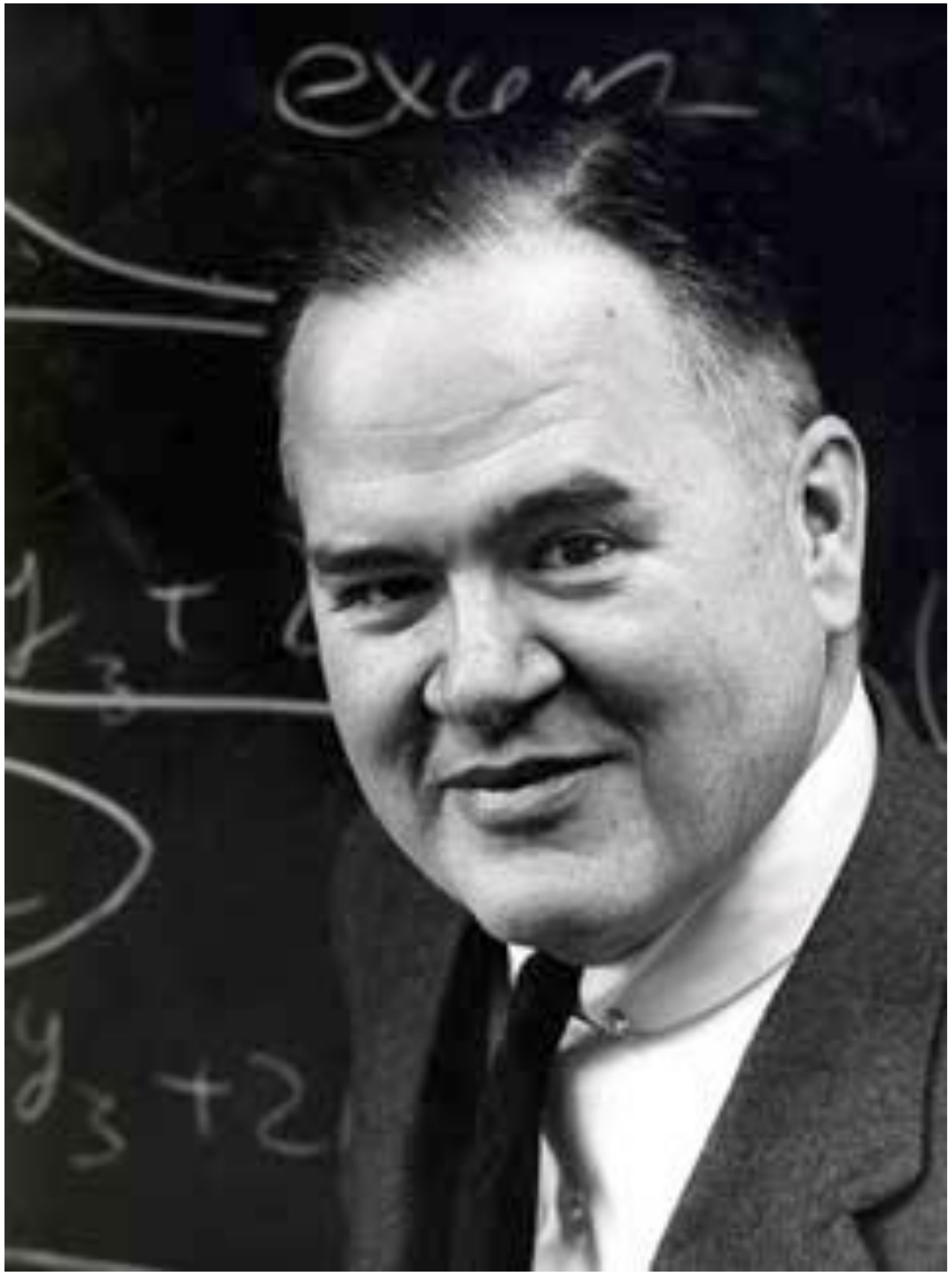
```
y <- 1:3
x + y
```

ZOMG. R *recycled* the smaller of the two vectors to match the length of the longer vector. R needs vectors to be the same length to do most operations on them together, so it will stretch one to match the other. Be aware of this!

Related: Making Sequences

```
one.to.ten <- 1:10
one.to.ten <- seq(1, 10)
who.do.we.appreciate <- seq(2, 8, 2)
we.love.R <- rep("R!", 10)
waltz <- rep(seq(1, 3), 4)

# fancy!  A *lot* going on in here
ranks <- rep(c(2:10, c("J", "Q", "K", "A")), 4)
suits <- rep(c("C","H","S","D"), each=13)
paste(ranks, suits,sep="") # look this function up
```



John Tukey

Plotting

We just saw last class that we could throw thing at **`plot()`** and stuff would appear. Let's get serious about what **`plot()`** can do:

Basics: `plot(x, y)`

e.g. `plot(1:100, runif(100)) # What should this do?`

You can throw other things at plot and it'll try to figure out what you want:

```
plot(table(snf$precinct))
```

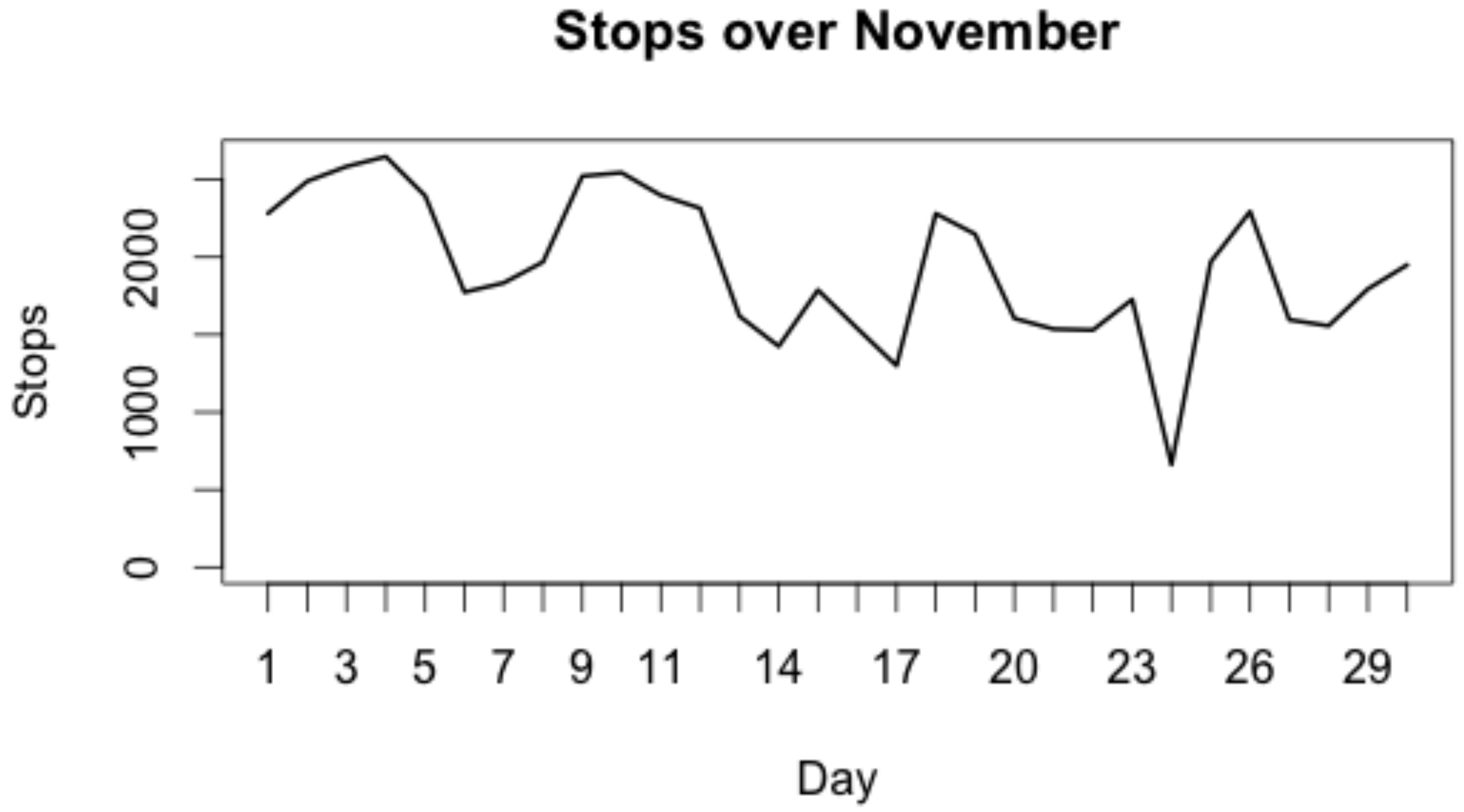
Plotting

There are lots of parameters that **`plot()`** can take to change the appearance of the figure.

Oddly, help for this shows up under **`par()`**, which sets the graphical parameters.

```
plot(table(snf$day), type='l', xlab="Day", ylab="Stops",
main="Stops over November")
```

Plotting Parameters



Plotting Parameters

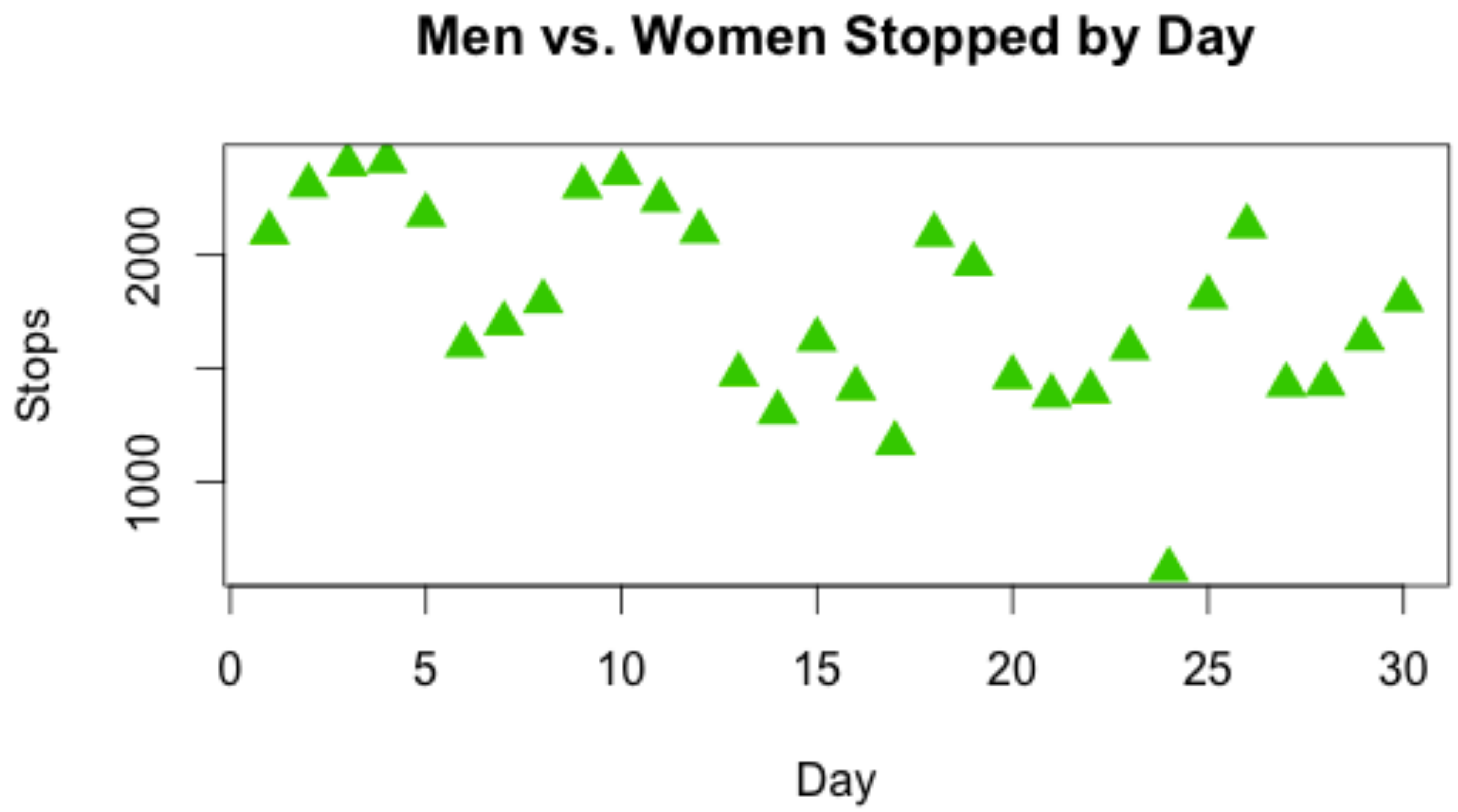
Overlaying plots:

```
men.stops <- as.vector(table(snf$day[snf$sex == "M"]))
women.stops <- as.vector(table(snf$day[snf$sex == "F"]))

plot(men.stops, pch=17, col=3, cex=1.5, xlab="Day",
ylab="Stops", main="Men vs. Women Stopped by Day")

points(women.stops, pch=10, col=6, cex=0.8)
```

Plotting Parameters



Plotting Parameters

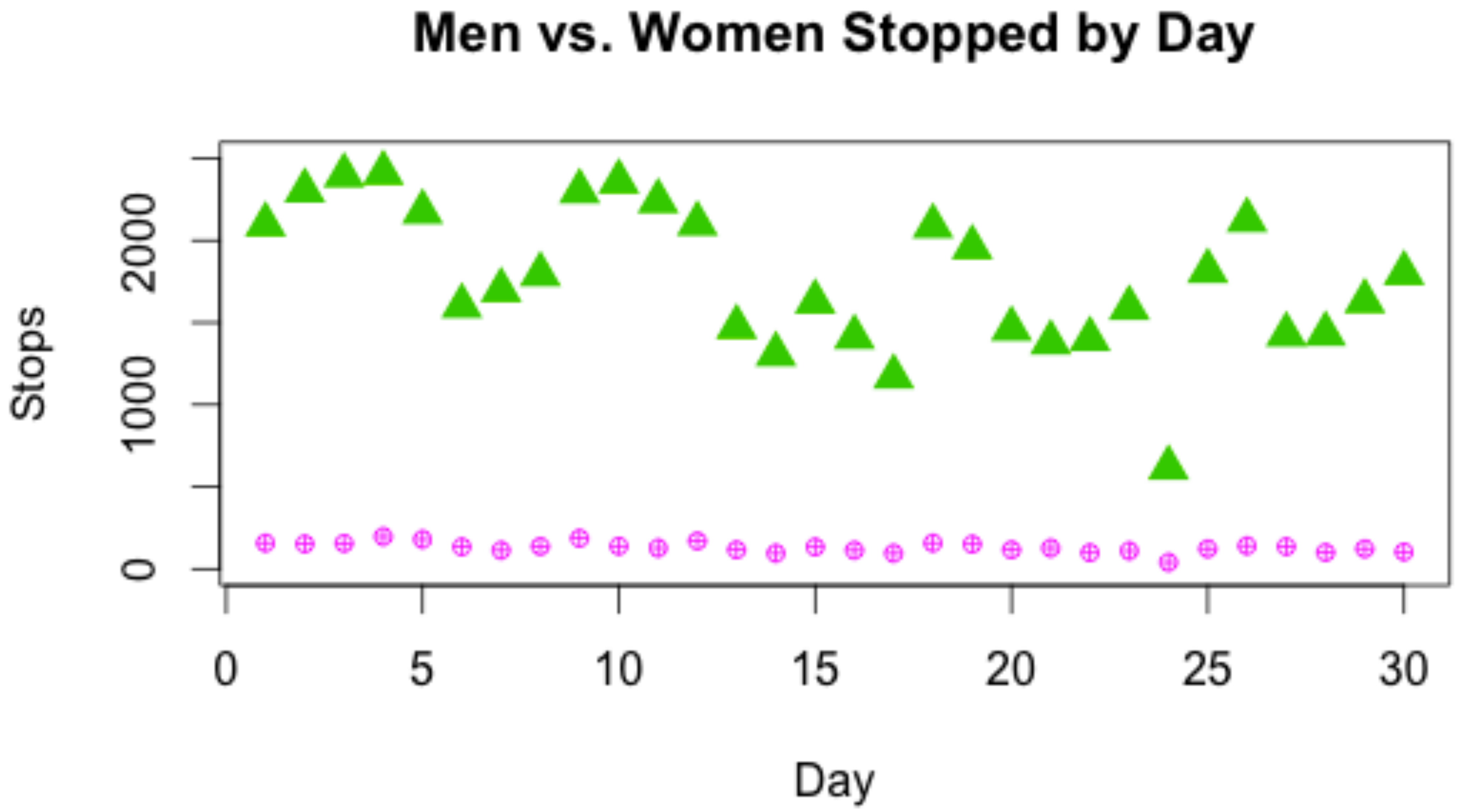
Hmm, where are the ladies? `points()` didn't adjust the x and y...

```
max(men.stops)

plot(men.stops, pch=17, col=3, cex=1.5, xlab="Day",
ylab="Stops", main="Men vs. Women Stopped by Day",
ylim=c(0, 2500))

points(women.stops, pch=10, col=6, cex=0.8)
```

Plotting Parameters



Plotting Parameters

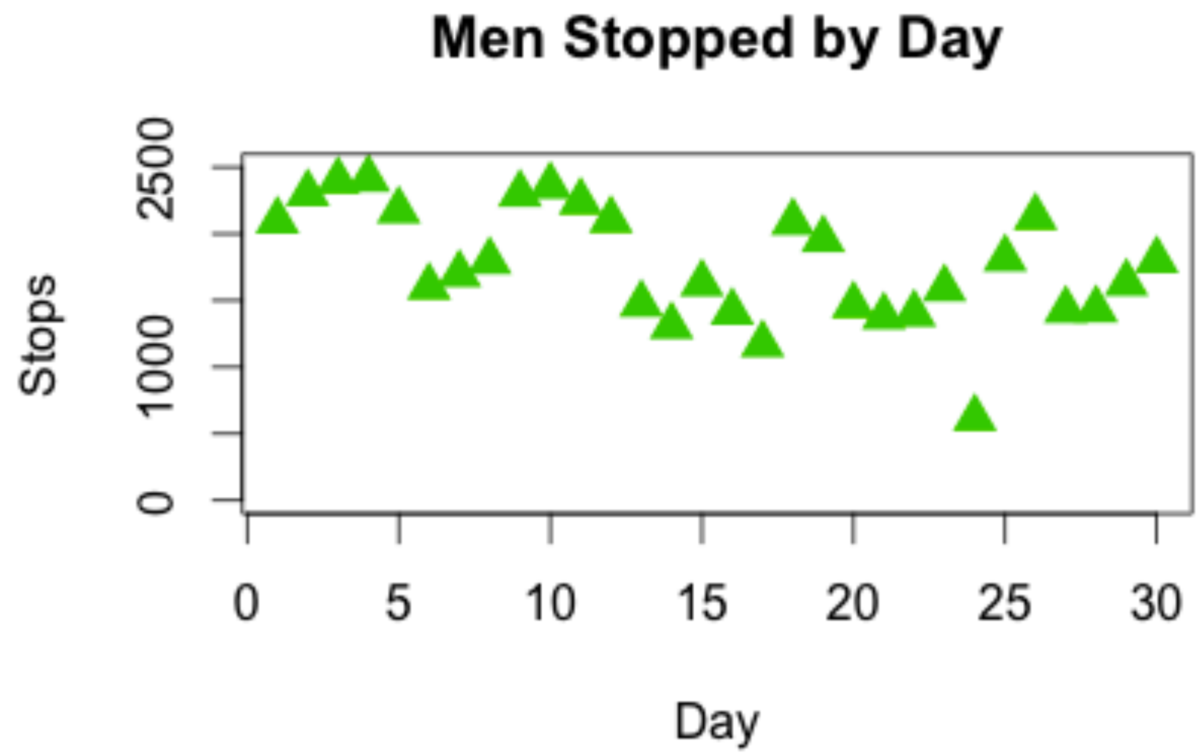
Another way to show similar information

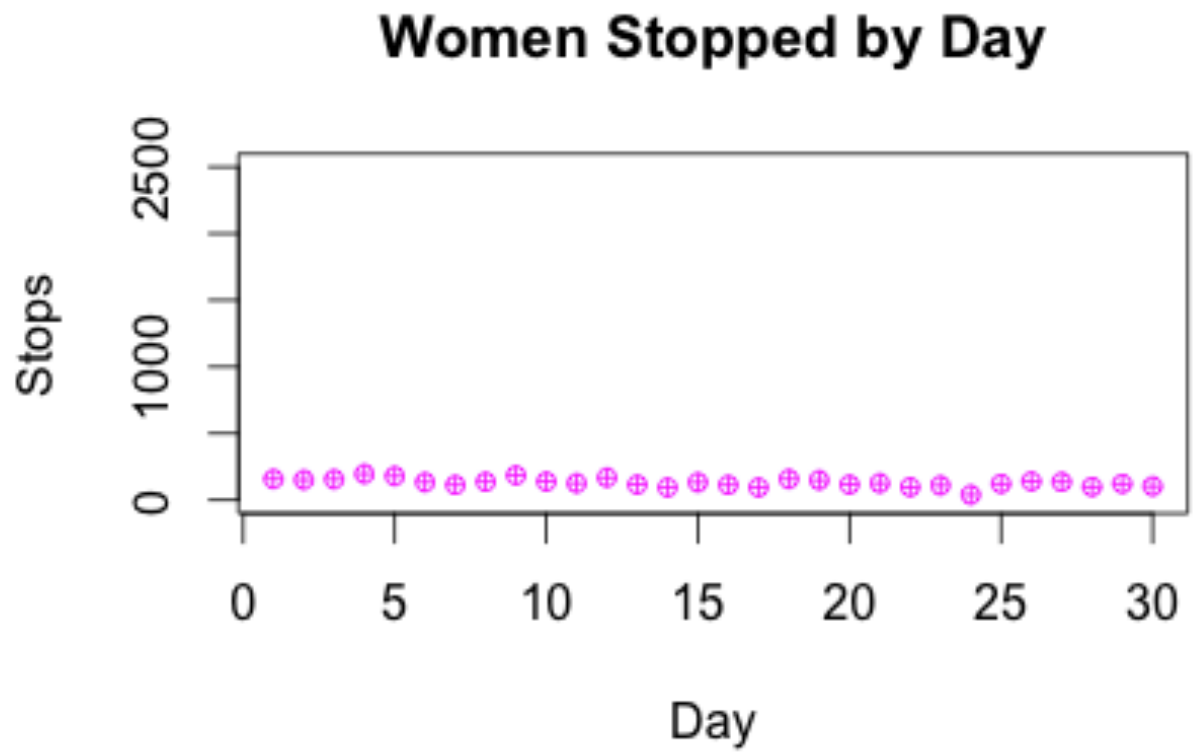
```
par(mfrow=c(1, 2))

plot(men.stops, pch=17, col=3, cex=1.5, xlab="Day",
ylab="Stops", main="Men Stopped by Day", ylim=c(0, 2500))


plot(women.stops, pch=17, col=3, cex=1.5, xlab="Day",
ylab="Stops", main="Women Stopped by Day", ylim=c(0,
2500))
```

Plotting Parameters





Plotting Color

Fun stuff: Just like R will vectorize most things, it'll vectorize arguments to plot(). In particular, the color parameter is the most important.

Let's say we want to look at # of stops on a certain weekdays relative to the rest of the month. First, let's make a data frame of counts by day. This is surprisingly easy to do:

```
stops.by.day <- data.frame(table(snf$day))
```

Plotting Color

Hey, side note, why are the column names in our data frame crazy? By default, `data.frame()` assigns column names based on what it thinks is in there. We can assign the column names like this:

```
names(stops.by.day) <- c(“day”, “stops”)
```

(WHOA. Did you see we just assigned values to the result of a function?? R is awesome!)

Plotting Color

OK, plotting this will give us our familiar graph:

```
plot(stops.by.day$stops)
```

Can we color each day of the week differently?

Well, let's create a vector of colors for each day of the week

```
days <- as.integer(stops.by.day$day) %% 7
   # %% is modulo (seen this before?)
   # I'll explain why we're doing as.integer() here next
   # time (try it without and see what you get)
```

Plotting Color

If we use this vector as the color argument, R will color each point accordingly! Sweet!

```
plot(stops.by.day$stops, col=days)
```

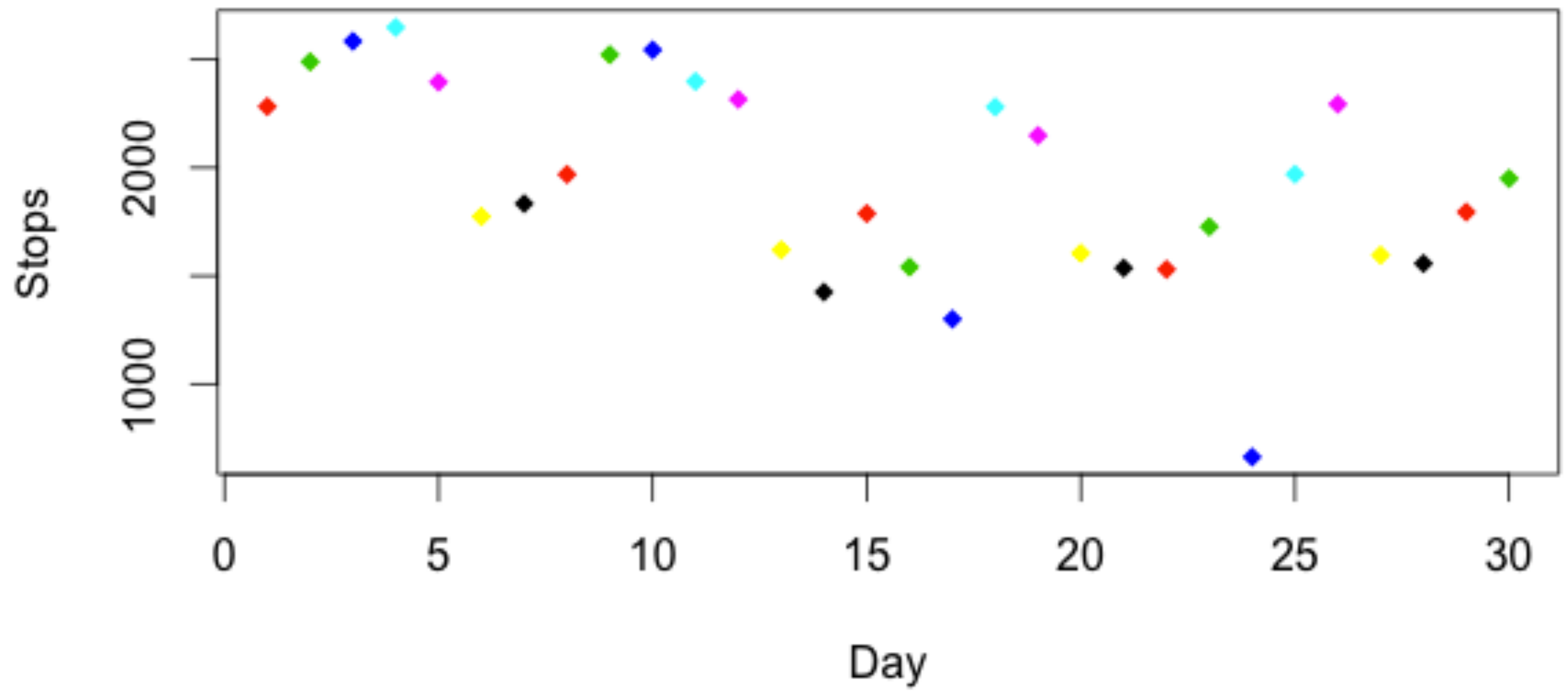
Wait, huh? Some of our points disappeared! Why?

Turns out that `col = 0` prints nothing, so let's just remedy this by bumping all of our colors up by 1:

```
plot(stops.by.day$stops, col=days+1)
```

Ahh, better!

Plotting Color

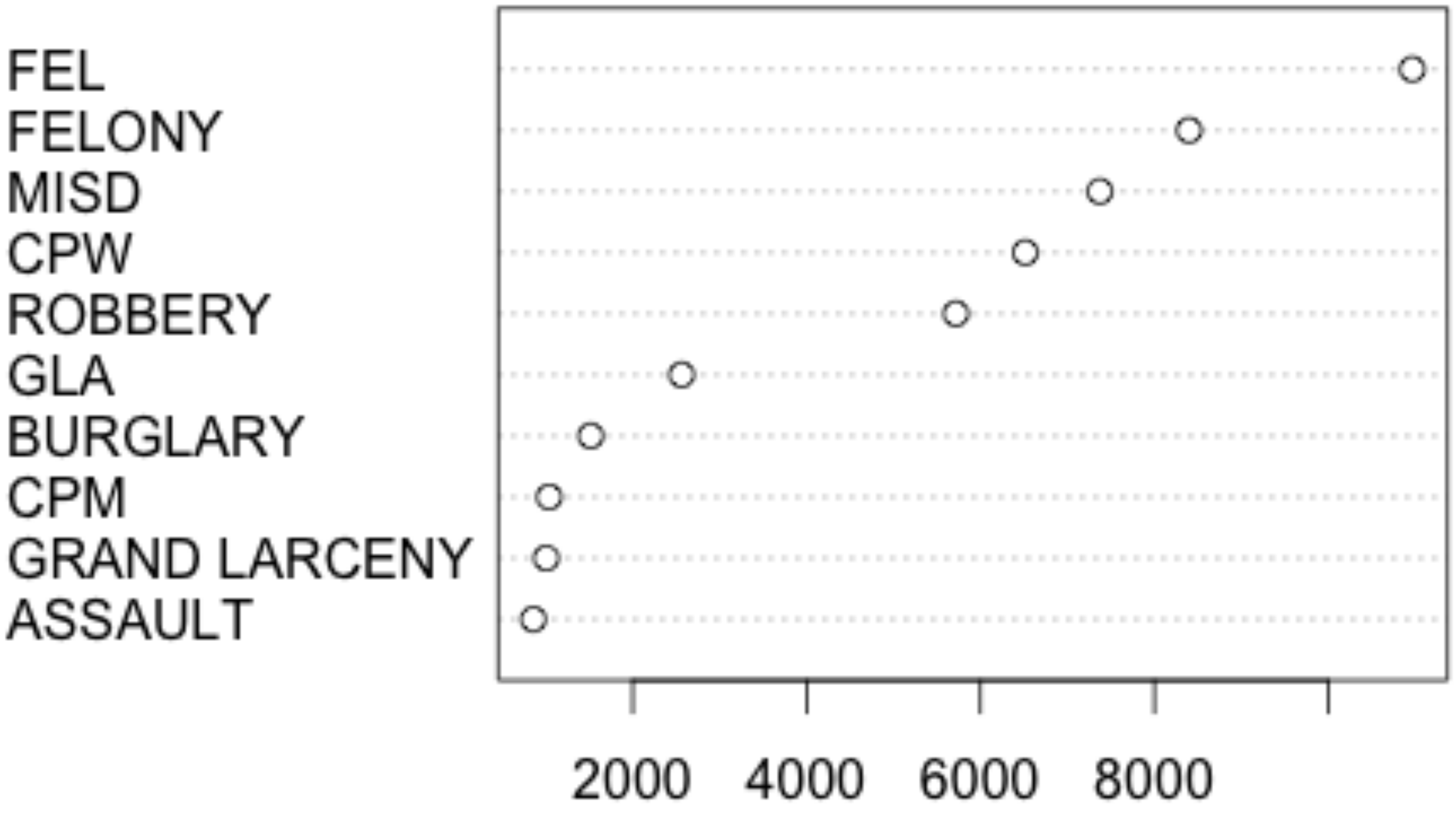


```
plot(stops.by.day$stops, col=days+1, pch=18, xlab="Day",
ylab="Stops")
```

Dotchart

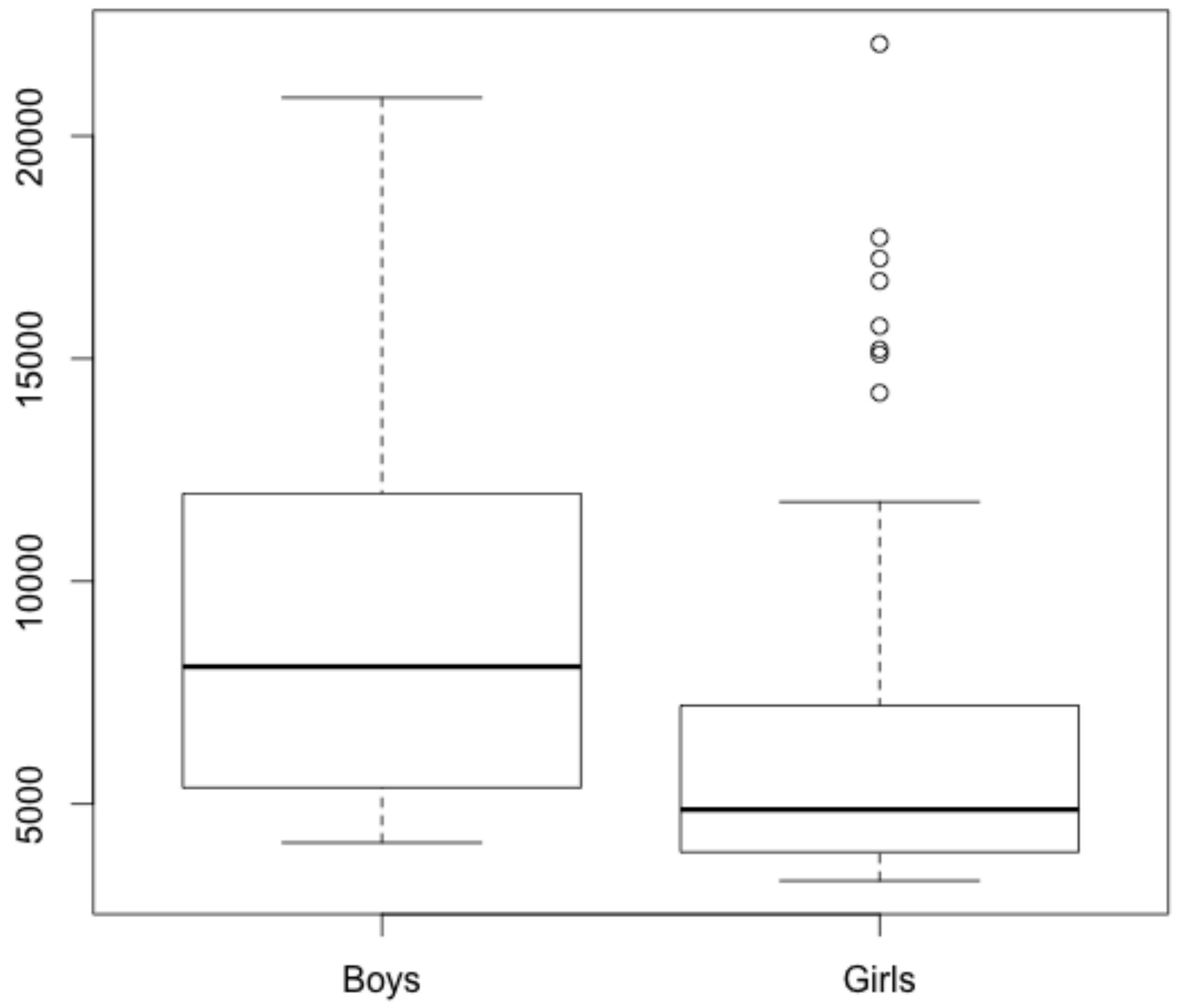
What are the top 10 crimes?

```
crimes <- rev(sort(table(snf$crime.suspected)))
top.crimes <- crimes[1:10]
dotchart(rev(top.crimes))
```



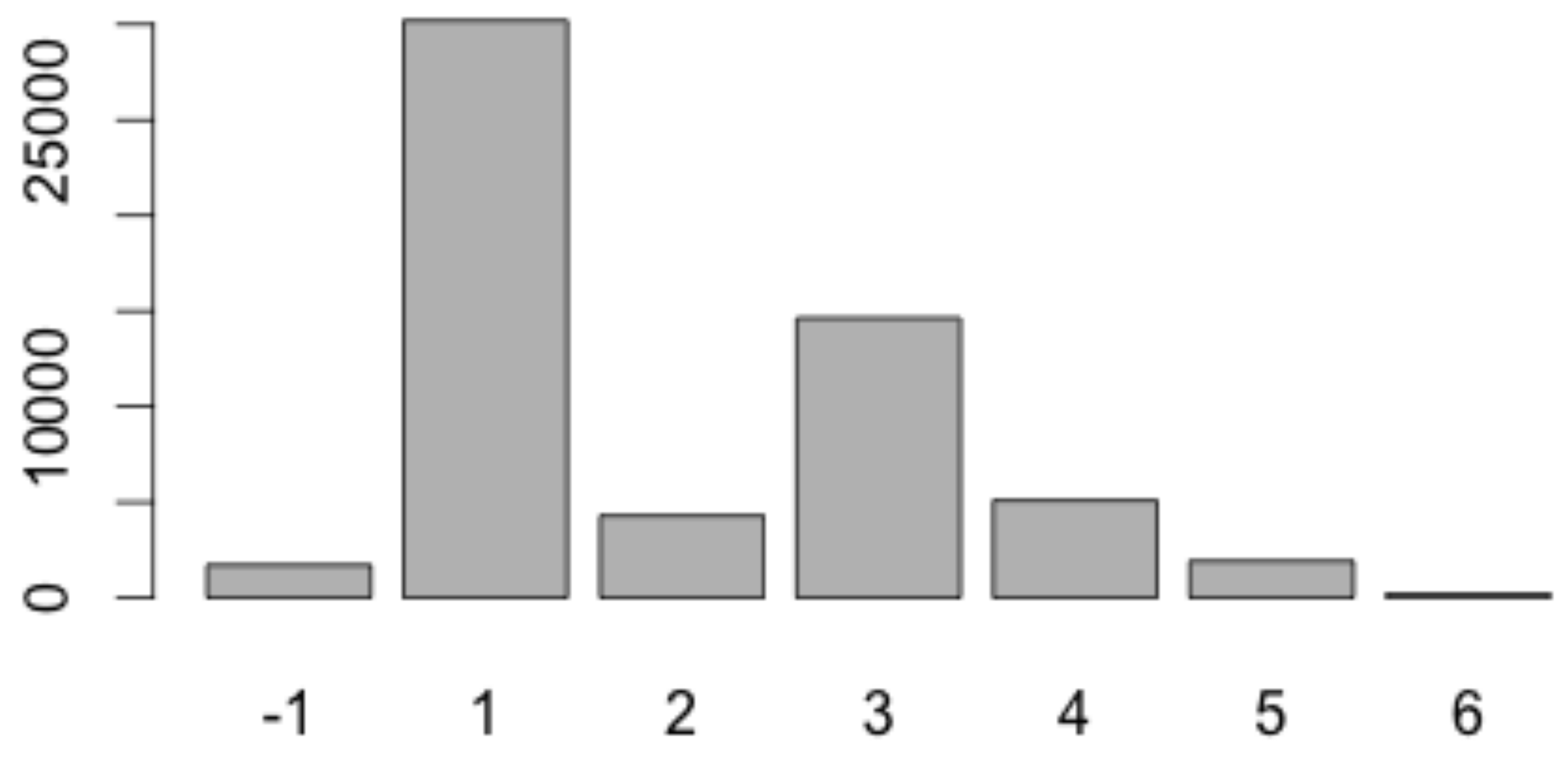
Boxplot

```
men.ages <- snf$age[snf$sex == "M"]
women.ages <- snf$age[snf$sex == "F"]
boxplot(men.ages[men.ages < 80], women.ages[women.ages <
80], names=c("Men", "Women"))
```



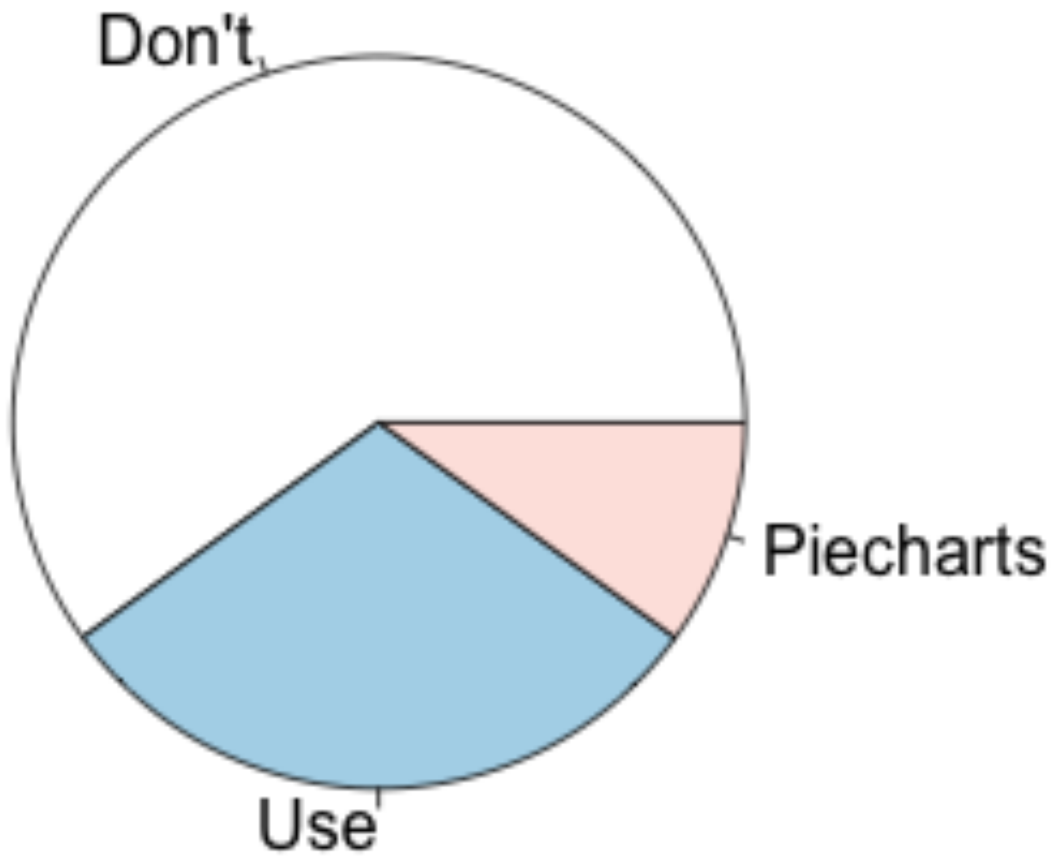
Barchart

```
barplot(table(snf$race))
```



Pie Charts

```
pie(c(0.6, 0.3, 0.1), labels=c("Don't", "Use", "Piecharts"))
```



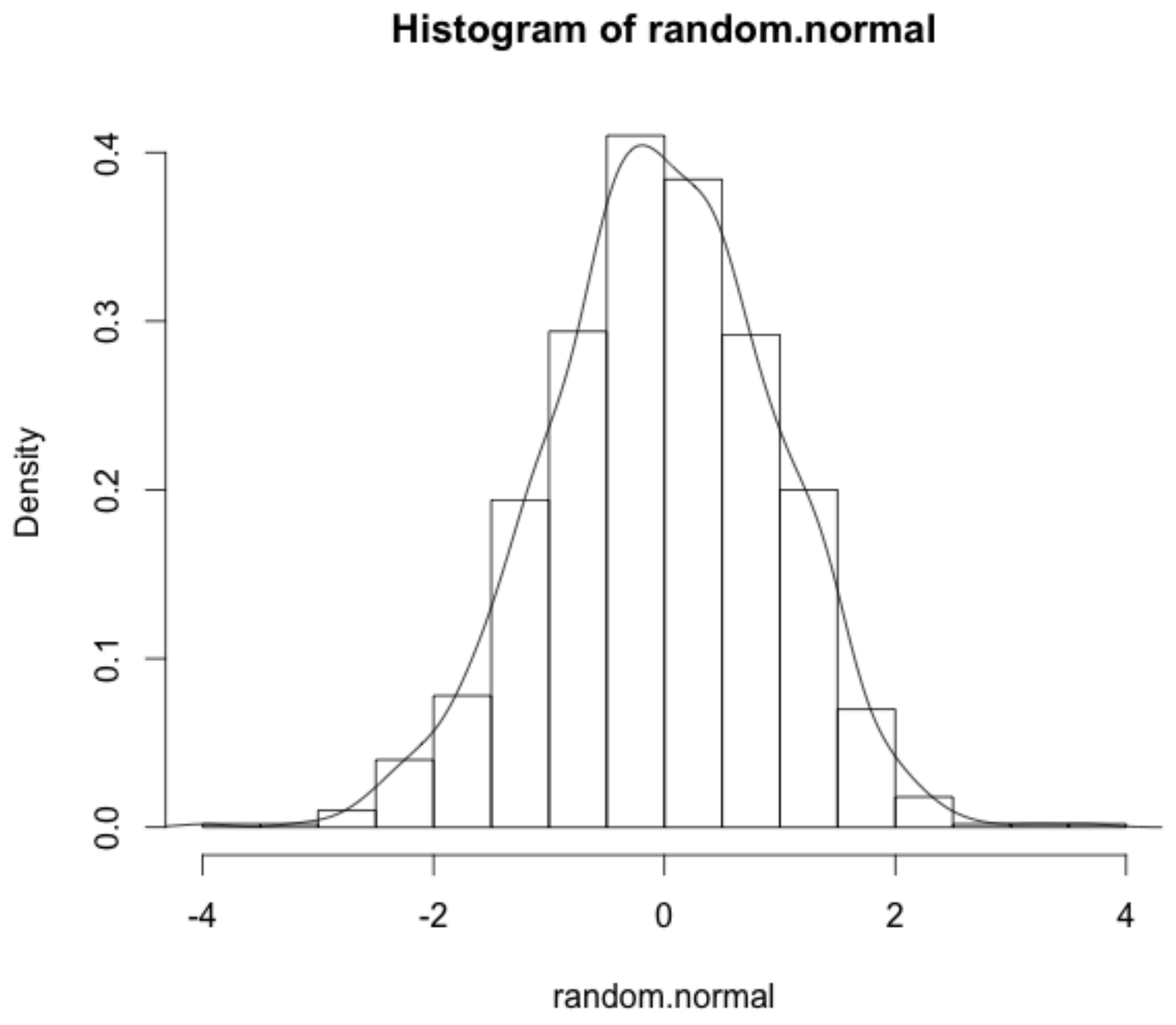
Histograms

One of the most important plots: Histograms (density plots)

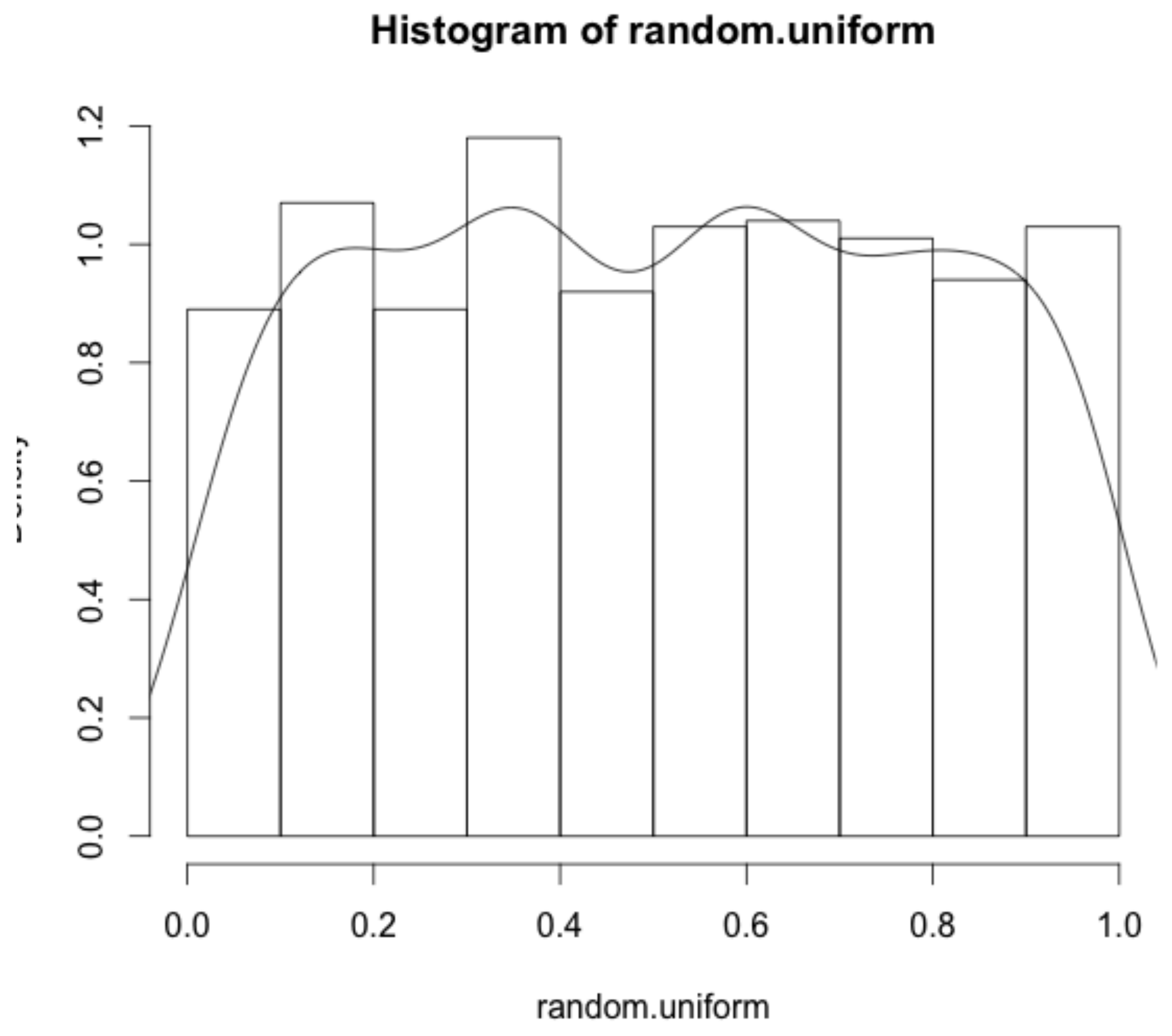
```
random.normal <- rnorm(1000)
head(random.normal)
hist(random.normal, freq=F)    # freq determines whether
                               # we want percentages or
                               # raw counts
lines(density(random.normal))  # we saw points()
                               # overlays points.  lines()
                               # overlays lines

random.uniform <- runif(1000)
head(random.uniform)
hist(random.uniform, freq=F)
lines(density(random.uniform))
```

Histograms



Histogram of random.normal
Density
0.0
0.1
0.2
0.3
0.4
-4
-2
0
2
4
random.normal

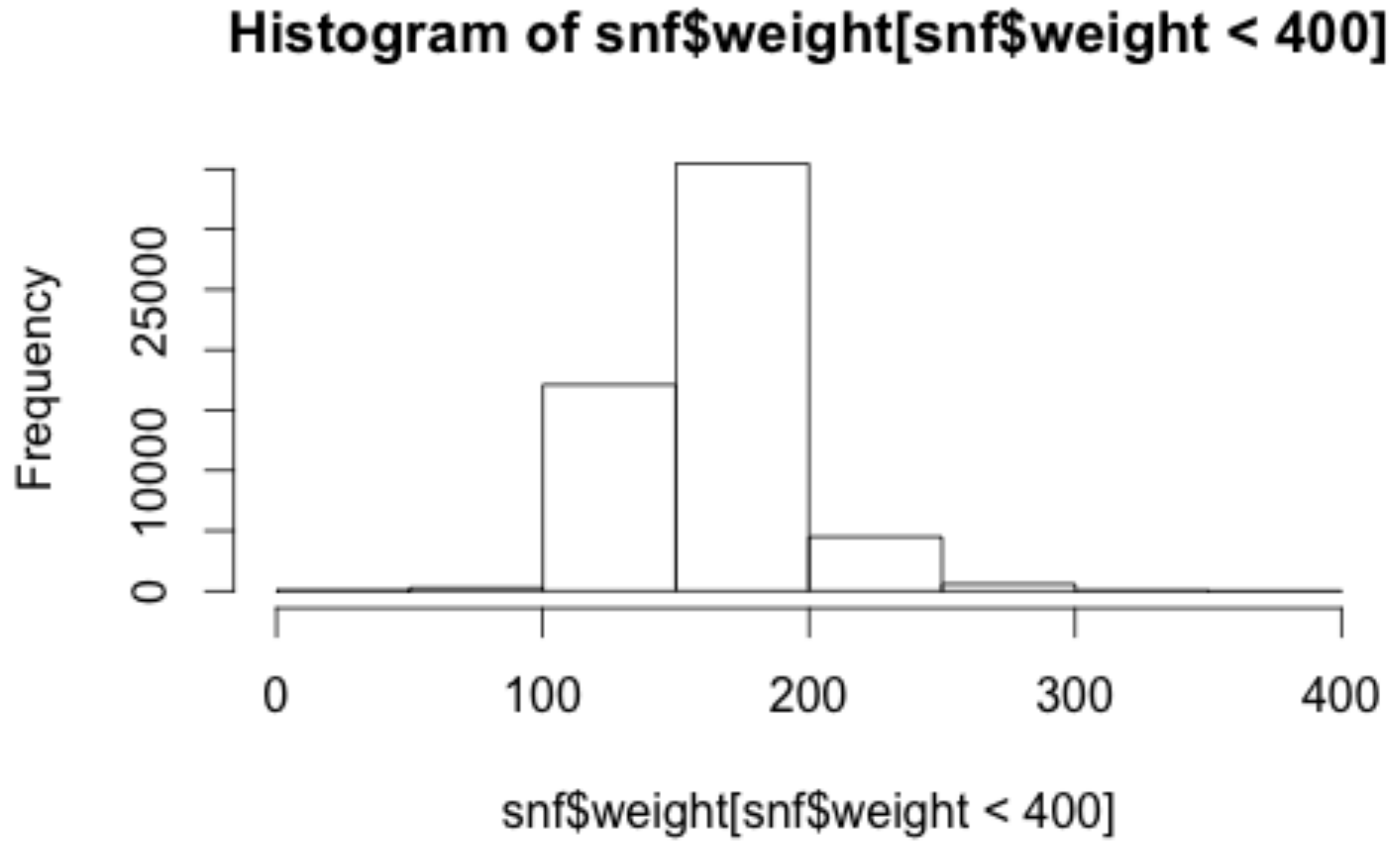


Histogram of random.uniform
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0.0
0.2
0.4
0.6
0.8
1.0
random.uniform

Histograms

BIN SIZE IS SUPER IMPORTANT!

```
weights <- snf$weight[snf$weight < 400]
hist(weights, breaks=10)
```

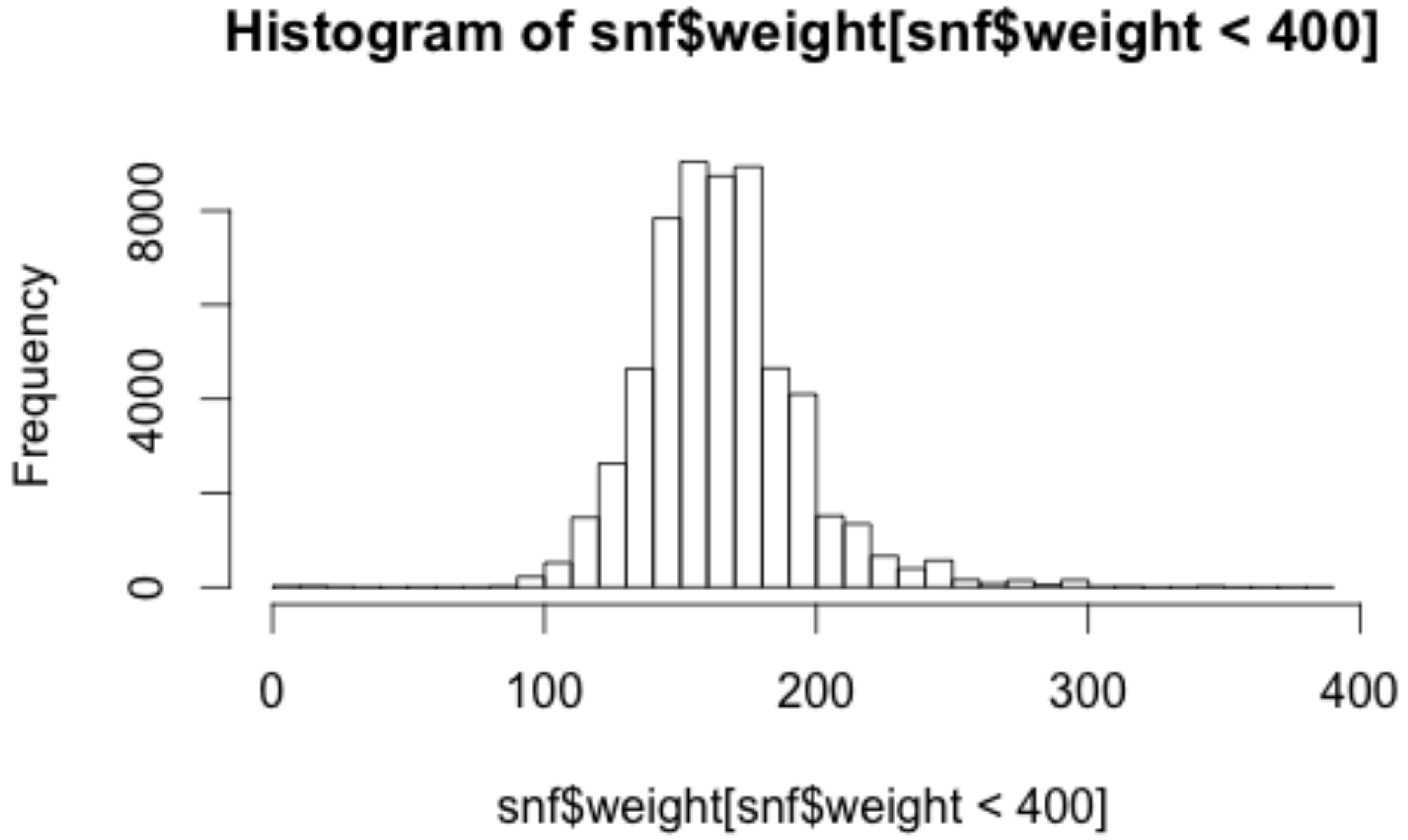


What do you see?

Histograms

BIN SIZE IS SUPER IMPORTANT!

```
hist(weights, breaks=50)
```

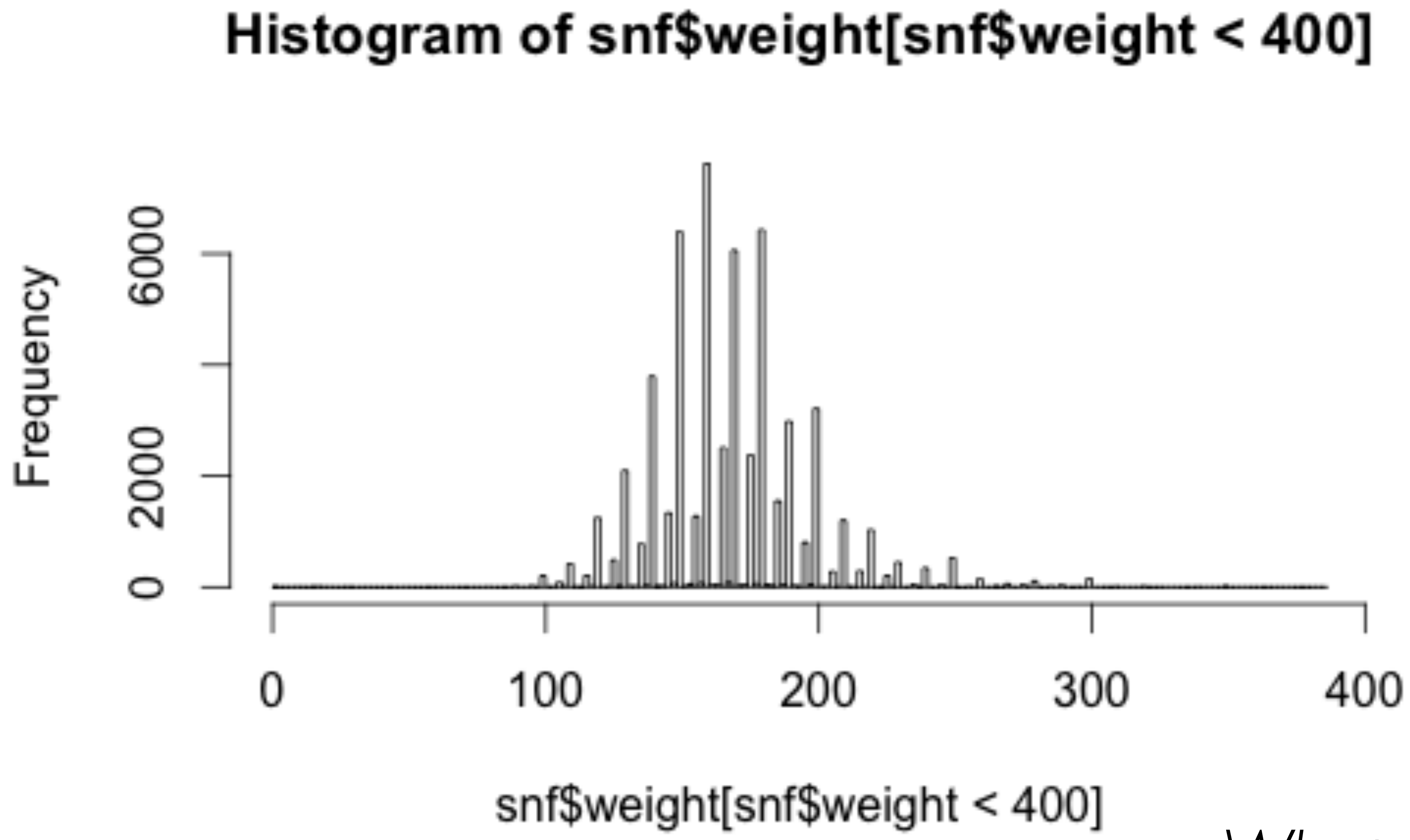


What do you see?

Histograms

BIN SIZE IS SUPER IMPORTANT!

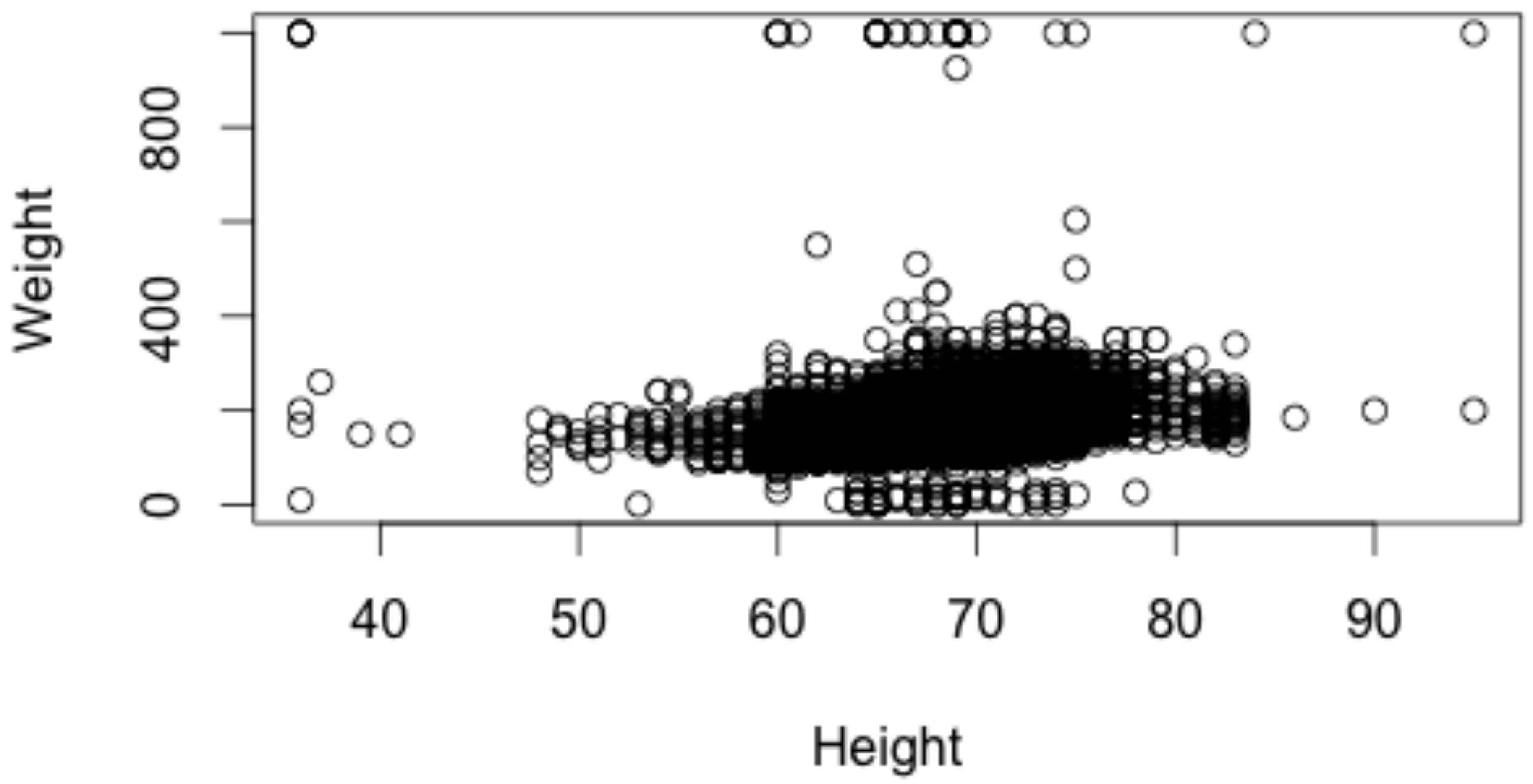
```
hist(weights, breaks=150)
```



What do you see?

2D Plots

```
plot(snf$height, snf$weight, xlab="Height", ylab="Weight")
```



Overview

- Subsetting by logical indices is the way to go (don't forget about which())
- R does vectorizing and recycling, which is super powerful (avoid loops if possible)
- Lots of different ways to fancy up your plots, including different built-in plots.
- Be careful about what your plot forces you to say / ignore about your data!