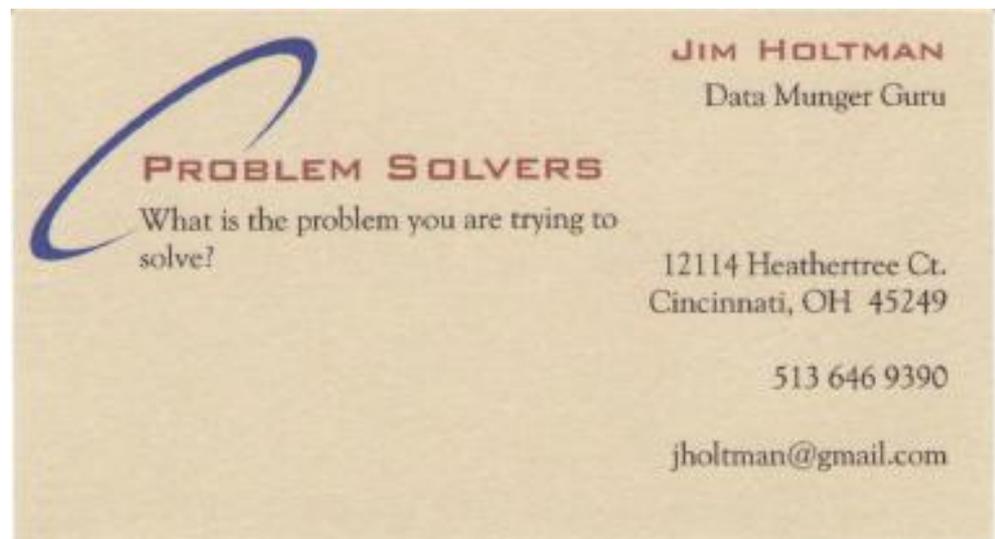


# Data Munging With

**Jim Holtman**  
**Kroger**  
**Data Munger Guru**



# Topics Covered

---

- **What is “data munging”**
- **Summarizing data with various tools**
  - EDA: exploratory data analysis
  - Visualization of the data
- **Measuring performance**
- **Reading in data & Time/Date classes**
- **Debugging**

# Data Munging

**Your desktop dictionary may not include it, but 'munging' is a common term in the programmer's world. Many computing tasks require taking data from one computer system, manipulating it in some way, and passing it to another. Munging can mean manipulating raw data to achieve a final form. It can mean parsing or filtering data, or the many steps required for data recognition.**

**“R” is an open source software package directed at analyzing and visualizing data, but with the power of the language, and available packages, it also provides a powerful means of slicing/dicing the data to get it into a form for analysis.**

# Summarizing Data

- **Various ways of collecting information about relationships of data elements**
- **I am going to use weekly shipments of products to stores**
  - Create the data since I cannot use actual (proprietary) information, but the techniques are the same.
  - 52 weeks of deliveries to 12 stores of 4000 products (~2.5M rows of data)
- **Tools used**
  - ‘tapply’: part of the ‘base’ R
  - ‘data.table’: package that is fast for many of these summarization operations; it has been one that I am using more and more.
  - ‘sqldf’: package that allows SQL access to dataframes; shortens the learning curve on some R activities if you already know SQL.
  - ‘plyr’: package for slicing/dicing that is used by many users.

# ?tapply

tapply {base}

R Documentation

## Apply a Function Over a Ragged Array

### Description

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.

### Usage

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

### Arguments

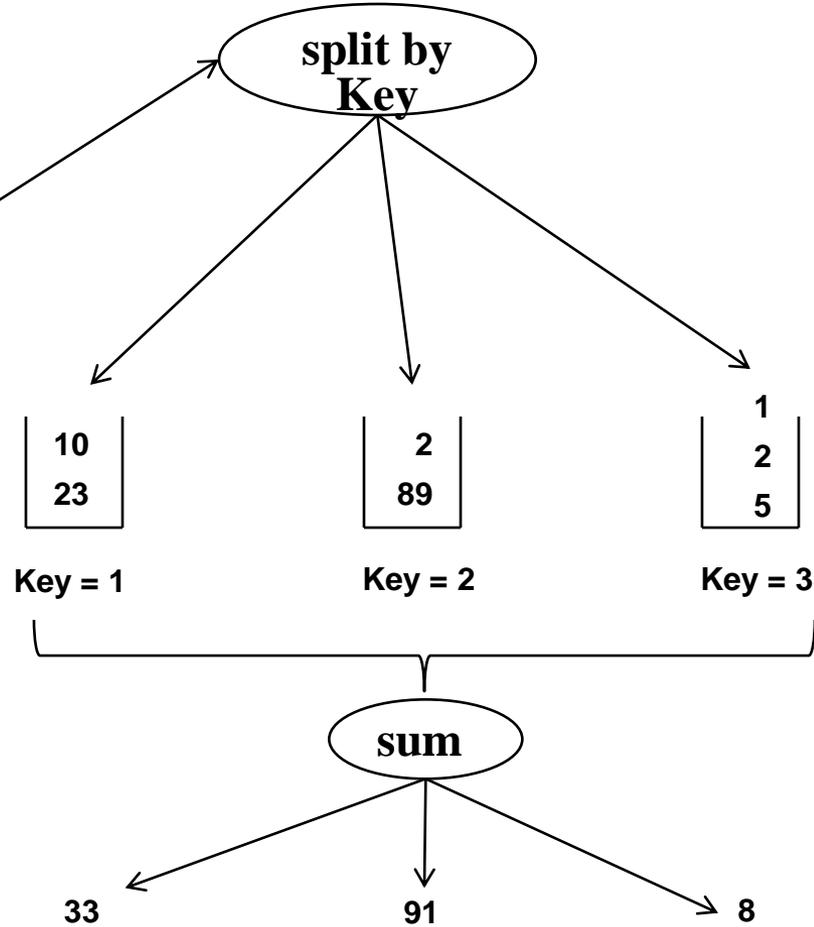
- X** an atomic object, typically a vector.
- INDEX** list of factors, each of same length as X. The elements are coerced to factors by [as.factor](#).
- FUN** the function to be applied, or NULL. In the case of functions like +, %\*%, etc., the function name must be backquoted or quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces.
- ...** optional arguments to FUN: the Note section.
- simplify** If FALSE, tapply always returns an array of mode "list". If TRUE (the default), then if FUN always returns a scalar, tapply returns an array with the mode of the scalar.

# tapply(x\$Count, x\$Key, sum)

```
> x <- data.frame(Key = c(1, 1, 2, 2, 3, 3, 3)  
+                 , Count = c(10, 23, 2, 89, 1, 2, 5)  
+                 )  
> tapply(x$Count, x$Key, sum)  
 1  2  3  
33 91  8
```

**X**

Key	Count
1	10
1	23
2	2
2	89
3	1
3	2
3	5



# ?data.table

R Documentation

## Enhanced data.frame

### Description

`data.table` inherits from `data.frame`. It offers fast subset, fast grouping and fast ordered joins in a short and flexible syntax, for faster development. It was inspired by `A[B]` syntax in R where `A` is a matrix and `B` is a 2-column matrix. Since a `data.table` is a `data.frame` it is compatible with R functions and packages that *only* accept `data.frame`.

The 10 minute quick start guide to `data.table` may be a good place to start; type `vignette("datatable-intro")`.

### Usage

```
data.table(..., keep.rownames=FALSE, check.names=TRUE, key=NULL)
```

```
## S3 method for class 'data.table'  
x[i, j, by=NULL, with=TRUE, nomatch = NA,  
  mult = "all", roll = FALSE, rolltolast = FALSE,  
  which = FALSE, bysameorder = FALSE,  
  verbose=getOption("datatable.verbose", FALSE), drop=NULL]
```

# ?sqldf

sqldf {sqldf}

R Documentation

## SQL select on data frames

### Description

SQL select on data frames

### Usage

```
sqldf(x, stringsAsFactors = TRUE, col.classes = NULL,  
      row.names = FALSE, envir = parent.frame(),  
      method = getOption("sqldf.method"),  
      file.format = list(), dbname, drv = getOption("sqldf.driver"),  
      user, password = "", host = "localhost",  
      dll = getOption("sqldf.dll"), connection = getOption("sqldf.connection"))
```

### Arguments

- x** Character string representing an SQL select statement or character vector whose components each represent a successive SQL statement to be executed. The select statement syntax must conform to the particular database being used. If **x** is missing then it establishes a connection which subsequent **sqldf** statements access. In that case the database is not destroyed until the next **sqldf** statement with no **x**.
- stringsAsFactors** If **TRUE** then output "character" columns are converted to "factor" if the heuristic is unable to determine the class. If **method="raw"** then **stringsAsFactors** is ignored.

# plyr Package

## **plyr: Tools for splitting, applying and combining data**

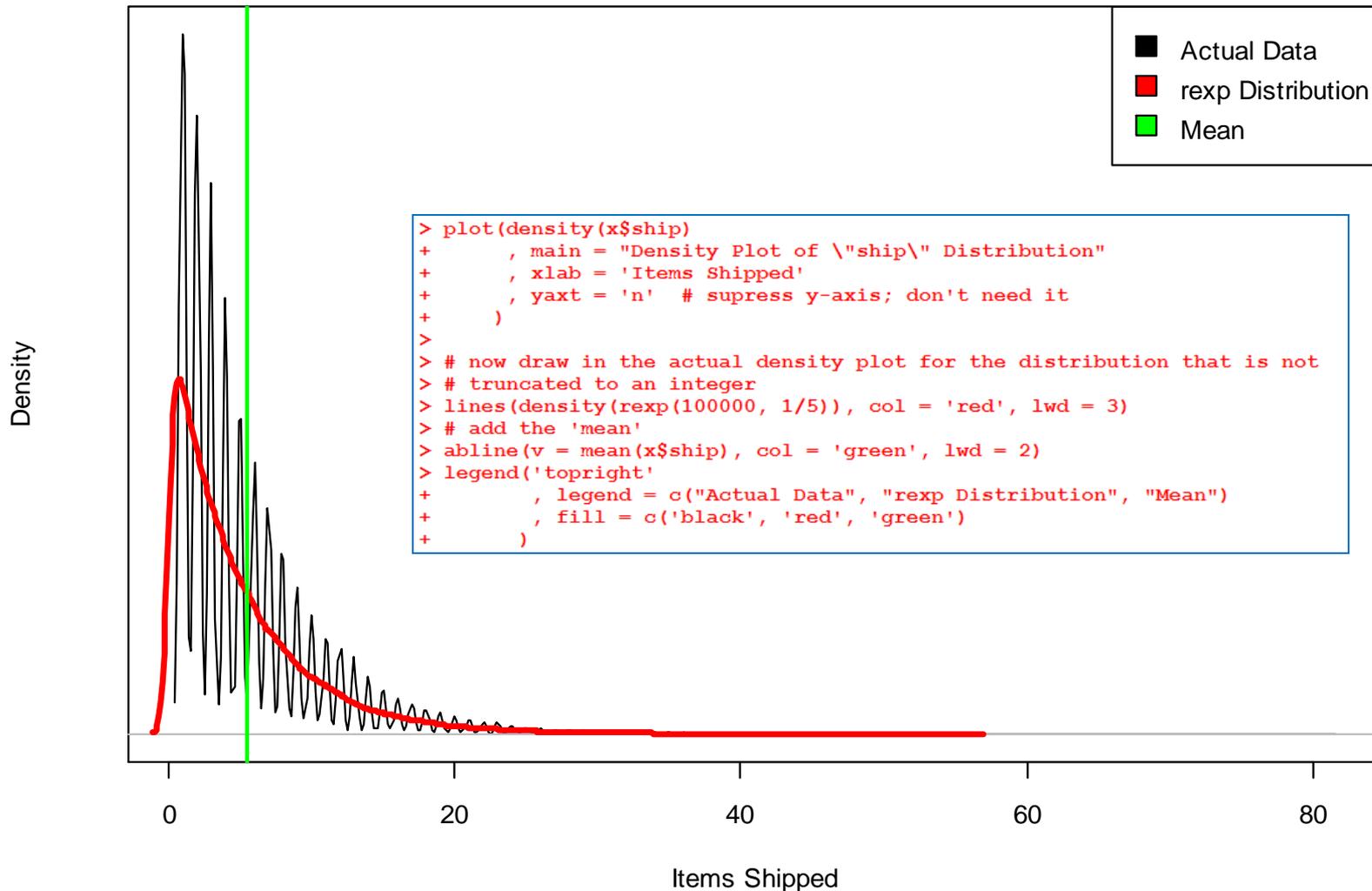
**plyr is a set of tools that solves a common set of problems: you need to break a big problem down into manageable pieces, operate on each pieces and then put all the pieces back together. For example, you might want to fit a model to each spatial location or time point in your study, summarise data by panels or collapse high-dimensional arrays to simpler summary statistics.**

# Setup for Script

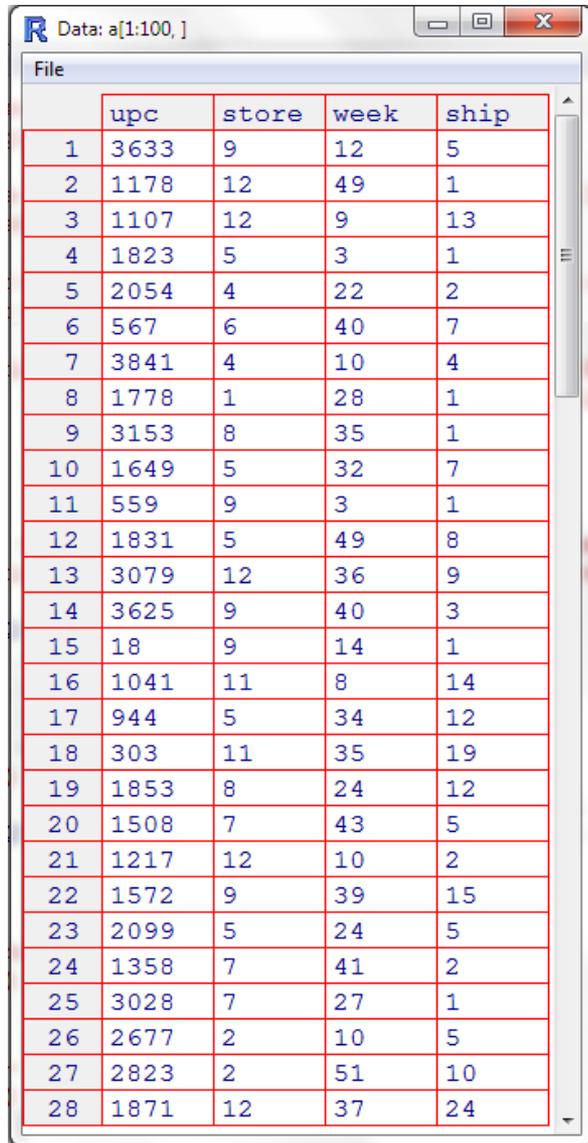
```
> # here is an example of using 4 different ways of aggregating data: tapply,
> # data.table, plyr and sqldf
>
> # create sample data representing real data that I work with in analyzing how
> # physical warehouse should be laid out.
>
> # here I am looking at the weekly shipments from a warehouse which has about 4000
> # unique products going to 12 different store for the period of a year (52 weeks)
>
> x <- expand.grid(upc = 1:4000      # create a dataframe with all combinations
+                , store = 1:12    # of these 3 values
+                , week = 1:52
+                , KEEP.OUT.ATTRS = FALSE
+                )
> # add shipment
> set.seed(1) # generate the same sequence each time
> x$ship <- ceiling(rexp(4000 * 12 * 52, 1/5)) # average of 5 items
> str(x)
'data.frame':   2496000 obs. of  4 variables:
 $ upc   : int  1 2 3 4 5 6 7 8 9 10 ...
 $ store: int  1 1 1 1 1 1 1 1 1 1 ...
 $ week  : int  1 1 1 1 1 1 1 1 1 1 ...
 $ ship  : num  4 6 1 1 3 15 7 3 5 1 ...
> my.func(x$ship)
      Count      Mean      SD      Min      Median      90%      95%      Max $
2.496000e+06 5.520843e+00 4.993964e+00 1.000000e+00 4.000000e+00 1.200000e+01 1.500000e+01 8.100000e+01 $
> require(data.table) # bring in the required packages that we will be using
> require(plyr)
> require(sqldf)
> # create intermediate objects for some of the packages
> x.id <- idata.frame(x) # makes for faster 'plyr' functions
> x.dt <- data.table(x) # compatible with a data.frame
```

# EDA: Distribution of “ship” Data

Density Plot of "ship" Distribution



# How To Determine Shipments Per Week?



The screenshot shows an R Data Viewer window titled "Data: a[1:100,]". The window displays a table with 28 rows and 5 columns. The columns are labeled "upc", "store", "week", and "ship". The first column contains row numbers from 1 to 28. The data is as follows:

	upc	store	week	ship
1	3633	9	12	5
2	1178	12	49	1
3	1107	12	9	13
4	1823	5	3	1
5	2054	4	22	2
6	567	6	40	7
7	3841	4	10	4
8	1778	1	28	1
9	3153	8	35	1
10	1649	5	32	7
11	559	9	3	1
12	1831	5	49	8
13	3079	12	36	9
14	3625	9	40	3
15	18	9	14	1
16	1041	11	8	14
17	944	5	34	12
18	303	11	35	19
19	1853	8	24	12
20	1508	7	43	5
21	1217	12	10	2
22	1572	9	39	15
23	2099	5	24	5
24	1358	7	41	2
25	3028	7	27	1
26	2677	2	10	5
27	2823	2	51	10
28	1871	12	37	24

- **What process would you use to create a summary of shipments per week?**
  - Using C++/Java
  - Using Excel (Pivot Tables?)
  - Using SQL
  - Your “other” favorite language
- **What approach would you use in R?**
  - You want to work on the objects as a whole.
  - Think of how you would split/partition the data and then operate on each group.

# Total Products Ordered Per Week

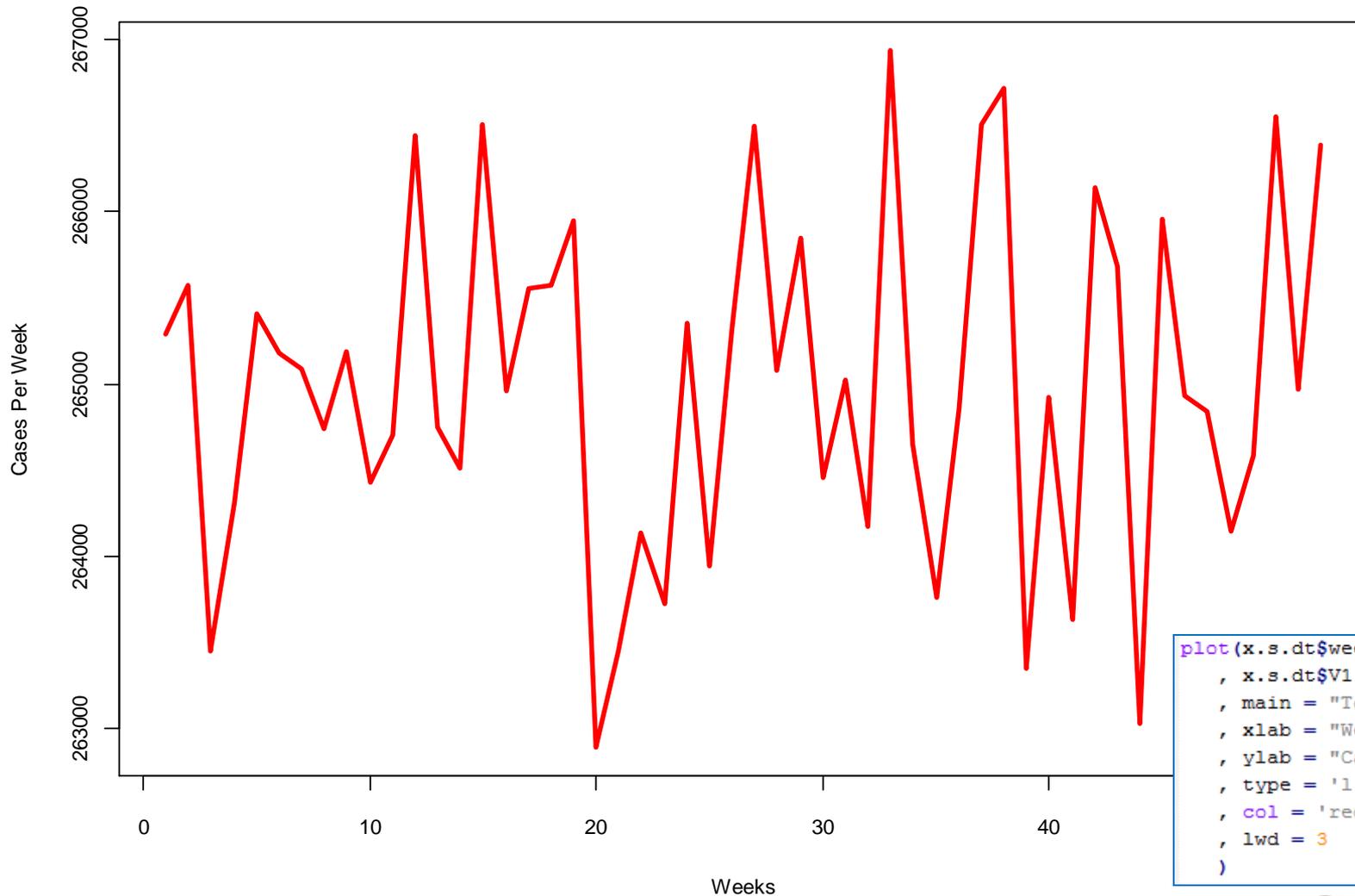
```
> # shipments per week
> system.time(x.s.t <- tapply(x$ship, x$week, sum)) # using tapply
  user  system elapsed
 3.06   0.13   3.23
> system.time(x.s.dt <- x.dt[, sum(ship), by = week]) # using data.table
  user  system elapsed
 0.14   0.02   0.15
> system.time(x.s.pl <- count(x.id, "week", "ship")) # using plyr
  user  system elapsed
 0.75   0.07   0.82
> system.time(x.s.pl.d <- dply(x.id, 'week', function(a) sum(a$ship))) # plyr
  user  system elapsed
 0.56   0.04   0.61
> system.time(x.s.sql <- sqldf('select week, sum(ship) from x group by week'))
  user  system elapsed
17.70   6.61  25.53
> str(x.s.t)
 num [1:52(1d)] 265291 265577 263449 264307 265408 ...
 - attr(*, "dimnames")=List of 1
  ..$ : chr [1:52] "1" "2" "3" "4" ...
> str(x.s.dt)
Classes 'data.table' and 'data.frame': 52 obs. of 2 variables:
 $ week: int  1 2 3 4 5 6 7 8 9 10 ...
 $ V1  : num  265291 265577 263449 264307 265408 ...
> str(x.s.pl)
'data.frame': 52 obs. of 2 variables:
 $ week: int  1 2 3 4 5 6 7 8 9 10 ...
 $ freq: num  265291 265577 263449 264307 265408 ...
> str(x.s.sql)
'data.frame': 52 obs. of 2 variables:
 $ week : int  1 2 3 4 5 6 7 8 9 10 ...
 $ sum(ship): num  265291 265577 263449 264307 265408 ...
```

Anything interesting about the time it took to execute the various commands? Which one would you want to use?

Notice that all the commands above returned the same values.

# Plot of Shipments Per Week

Total Shipments by Week

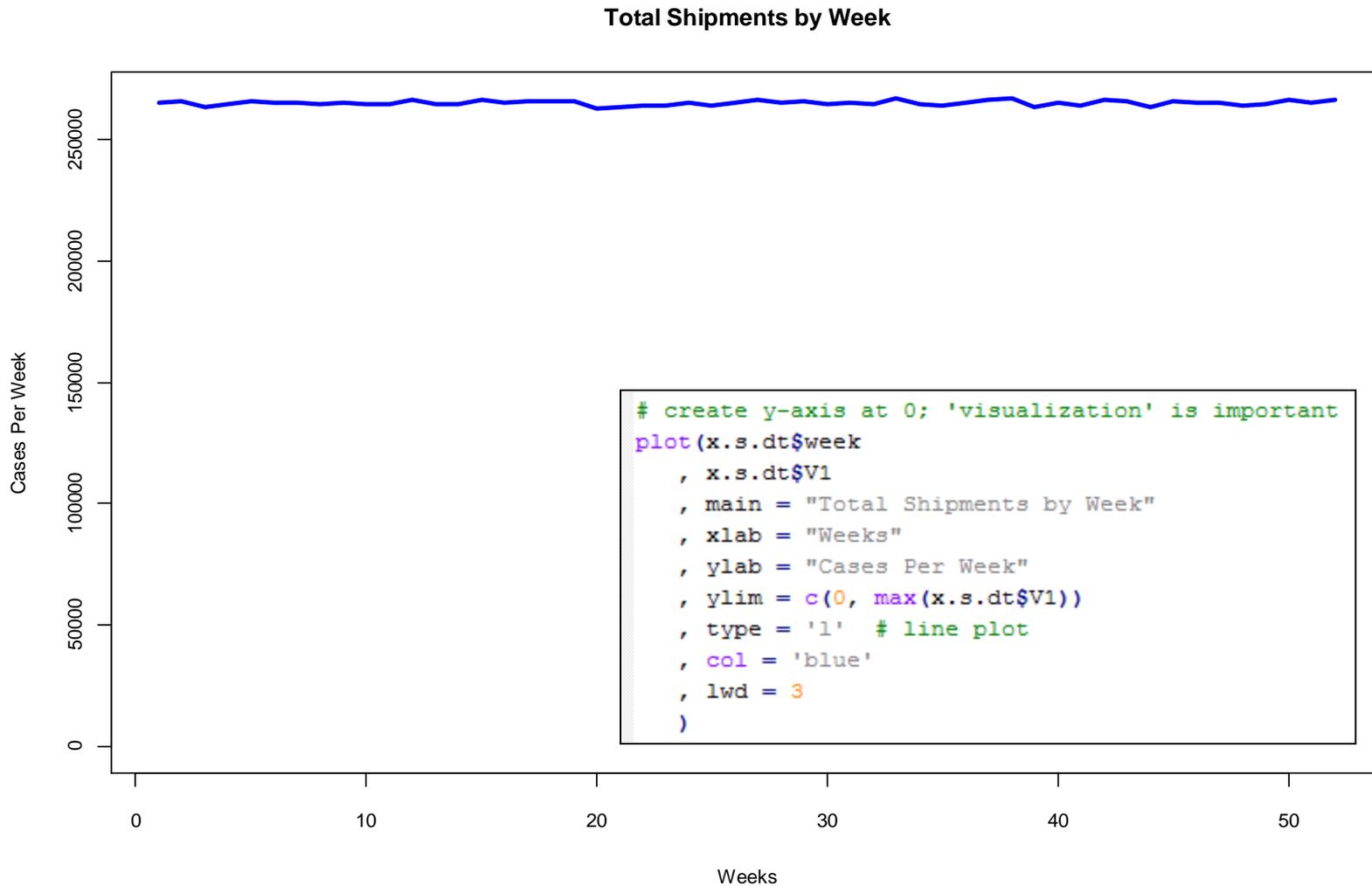


Is there seasonal variation in the data?

Notice the y-axis scaling.

```
plot(x.s.dt$week
, x.s.dt$V1
, main = "Total Shipments by Week"
, xlab = "Weeks"
, ylab = "Cases Per Week"
, type = 'l' # line plot
, col = 'red'
, lwd = 3
)
```

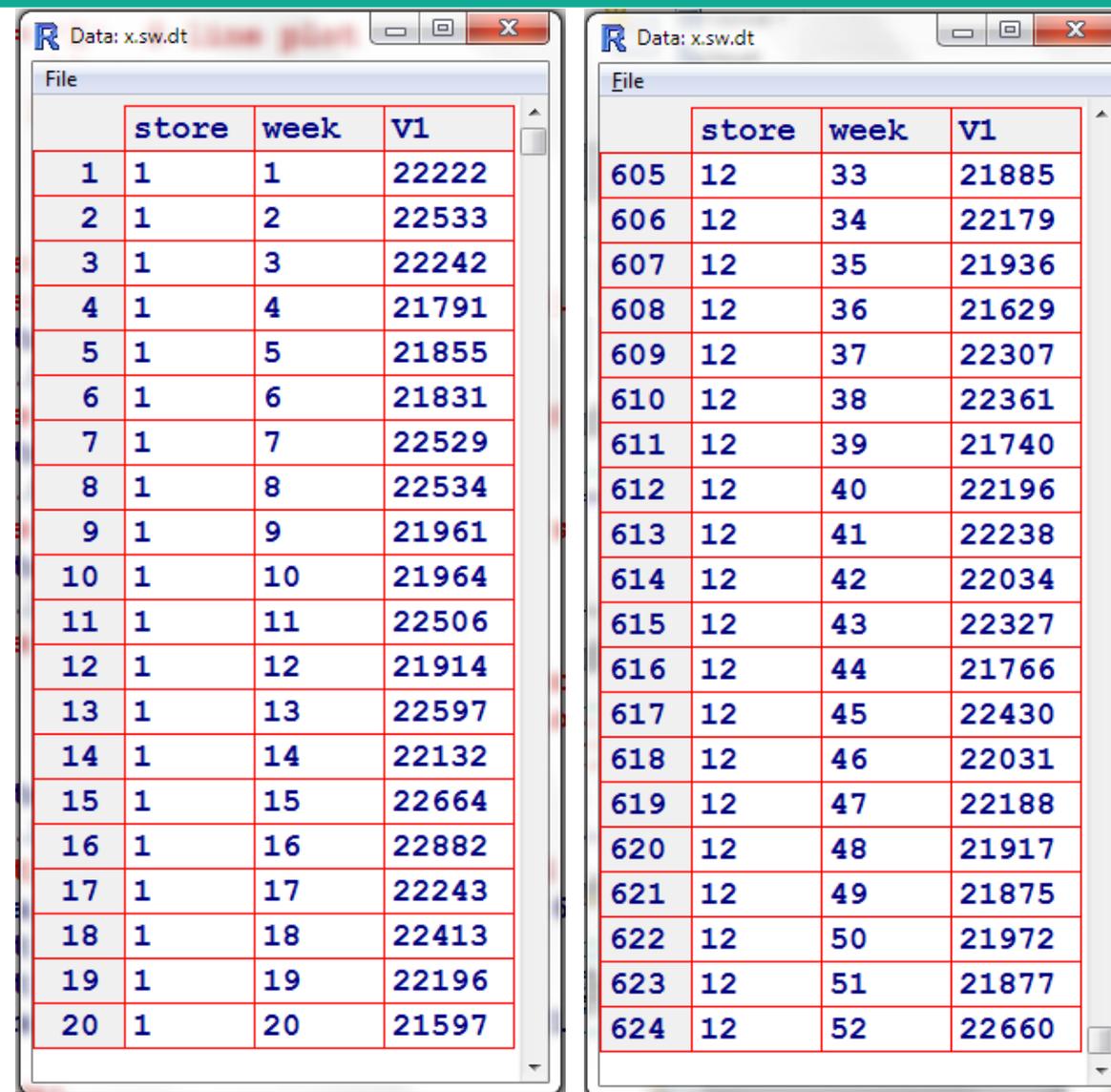
# “Better?” Plot of Shipments



# Products Per Store Per Week

```
> # shipments per store by week
> system.time(x.sw.t <- tapply(x$ship, list(x$week, x$store), sum))
  user  system elapsed
 4.54   0.06   4.88
> system.time(x.sw.dt <- x.dt[, sum(ship), by = list(store, week)])
  user  system elapsed
 0.27   0.02   0.28
> system.time(x.sw.pl <- count(x.id, c('store', 'week'), "ship"))
  user  system elapsed
 1.68   0.14   1.86
> system.time(x.ws.pl.d <- ddply(x.id
+                               , c('store', 'week')
+                               , function(a) sum(a$ship)
+                               ))
  user  system elapsed
 1.57   0.13   1.70
> str(x.sw.dt) # others are similar and the same results
Classes 'data.table' and 'data.frame': 624 obs. of 3 variables:
 $ store: int  1 1 1 1 1 1 1 1 1 1 ...
 $ week : int  1 2 3 4 5 6 7 8 9 10 ...
 $ V1   : num  22222 22533 22242 21791 21855 ...
```

# Use “View” to Look at Your Data



The image shows two R Data Viewer windows side-by-side, both titled "Data: x.sw.dt". Each window displays a data frame with three columns: "store", "week", and "V1". The left window shows rows 1 through 20, and the right window shows rows 605 through 624. The data in both windows appears to be a sequence of values for each column, with "store" being constant (1 or 12) and "week" increasing sequentially.

	store	week	V1
1	1	1	22222
2	1	2	22533
3	1	3	22242
4	1	4	21791
5	1	5	21855
6	1	6	21831
7	1	7	22529
8	1	8	22534
9	1	9	21961
10	1	10	21964
11	1	11	22506
12	1	12	21914
13	1	13	22597
14	1	14	22132
15	1	15	22664
16	1	16	22882
17	1	17	22243
18	1	18	22413
19	1	19	22196
20	1	20	21597

	store	week	V1
605	12	33	21885
606	12	34	22179
607	12	35	21936
608	12	36	21629
609	12	37	22307
610	12	38	22361
611	12	39	21740
612	12	40	22196
613	12	41	22238
614	12	42	22034
615	12	43	22327
616	12	44	21766
617	12	45	22430
618	12	46	22031
619	12	47	22188
620	12	48	21917
621	12	49	21875
622	12	50	21972
623	12	51	21877
624	12	52	22660

**Brings up a separate window that you can scroll through to see all the information in a dataframe.**

**Does this data seem reasonable?**

# Store per Week by UPC (Original Data!)

```
> # shipments per store by week by upc
> system.time(x.swu.t <- tapply(x$ship, list(x$week, x$store, x$upc), sum))
  user  system elapsed
107.05   1.03  109.50
> system.time(x.swu.dt <- x.dt[, sum(ship), by = list(store, week, upc)])
  user  system elapsed
 4.62   0.04   4.77
> system.time(x.swu.pl <- count(x.id, c('store', 'week', 'upc'), 'ship'))
  user  system elapsed
23.80   0.08  24.45
> str(x.swu.dt)
Classes 'data.table' and 'data.frame':  2496000 obs. of  4 variables:
 $ store: int  1 1 1 1 1 1 1 1 1 1 ...
 $ week : int  1 1 1 1 1 1 1 1 1 1 ...
 $ upc  : int  1 2 3 4 5 6 7 8 9 10 ...
 $ V1   : num  4 6 1 1 3 15 7 3 5 1 ...
```

```
> x$ship <- ceiling(rexp(4000 * 12 * 52, 1/5)) # average of 5 items
> str(x)
'data.frame':  2496000 obs. of  4 variables:
 $ upc  : int  1 2 3 4 5 6 7 8 9 10 ...
 $ store: int  1 1 1 1 1 1 1 1 1 1 ...
 $ week : int  1 1 1 1 1 1 1 1 1 1 ...
 $ ship : num  4 6 1 1 3 15 7 3 5 1 ...
```

This is from the original creation of the data and we did get back the same result.

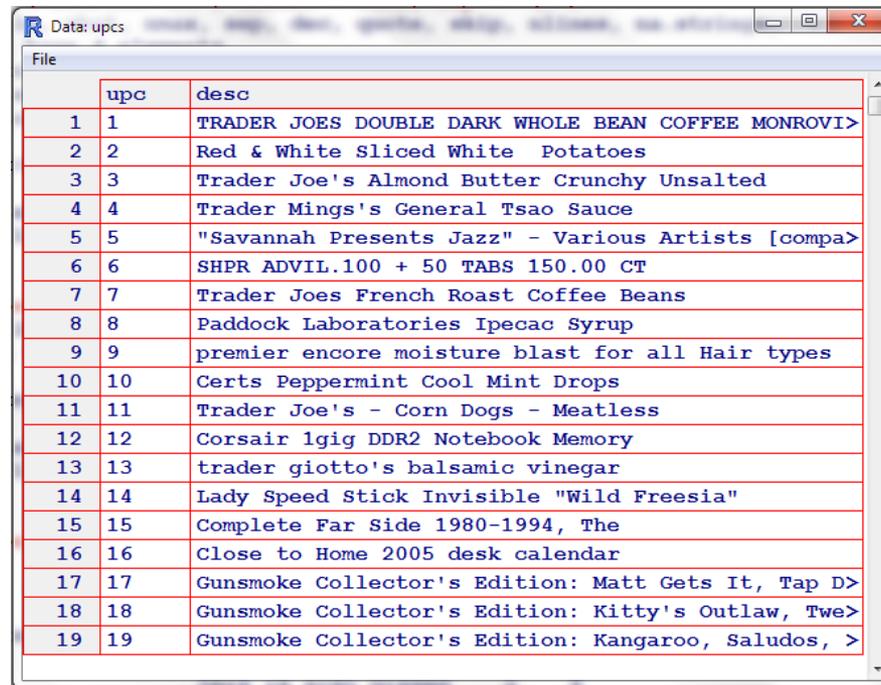
# Let's Add Some Extra Information to the Data

---

- In many cases, you may have data from different tables that you want to 'join' (merge) together based on a common key.
- In this example, I have a file with the names of the 4000 products that I would like to add to the 2.5M row dataframe that I have that defines the shipments.
- In SQL I would do a JOIN; in R I could use the "merge" function, or I could do it with some of the basic functions.
- Functions like "merge" are nice, but "hide" what they are doing. It is good to understand what is happening so if necessary, you can improve the performance of your program.

# Read in the UPC Name File

```
> # read in a database of the UPCs so we can add them to the input data
> upcs <- read.csv("upc.csv", as.is = TRUE)
> str(upcs)
'data.frame':  4000 obs. of  2 variables:
 $ upc : int  1 2 3 4 5 6 7 8 9 10 ...
 $ desc: chr  "TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF" "Red & White Sliced White Pota$
> head(upcs)
  upc          desc
1  1  TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
2  2          Red & White Sliced White Potatoes
3  3  Trader Joe's Almond Butter Crunchy Unsalted
4  4          Trader Mings's General Tsao Sauce
5  5 "Savannah Presents Jazz" - Various Artists [compact disc]
6  6  SHPR ADVIL.100 + 50 TABS 150.00 CT
```



The screenshot shows an R console window with a data frame named 'upcs'. The data frame has two columns: 'upc' and 'desc'. The first six rows of the data frame are displayed in the console output, matching the text above. The screenshot also shows a preview of the data frame in a table format, with the first 19 rows visible.

	upc	desc
1	1	TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVI>
2	2	Red & White Sliced White Potatoes
3	3	Trader Joe's Almond Butter Crunchy Unsalted
4	4	Trader Mings's General Tsao Sauce
5	5	"Savannah Presents Jazz" - Various Artists [compa>
6	6	SHPR ADVIL.100 + 50 TABS 150.00 CT
7	7	Trader Joes French Roast Coffee Beans
8	8	Paddock Laboratories Ipecac Syrup
9	9	premier encore moisture blast for all Hair types
10	10	Certs Peppermint Cool Mint Drops
11	11	Trader Joe's - Corn Dogs - Meatless
12	12	Corsair 1gig DDR2 Notebook Memory
13	13	trader giotto's balsamic vinegar
14	14	Lady Speed Stick Invisible "Wild Freesia"
15	15	Complete Far Side 1980-1994, The
16	16	Close to Home 2005 desk calendar
17	17	Gunsmoke Collector's Edition: Matt Gets It, Tap D>
18	18	Gunsmoke Collector's Edition: Kitty's Outlaw, Twe>
19	19	Gunsmoke Collector's Edition: Kangaroo, Saludos, >

# Using “merge”

- “merge” is general purpose and does a lots of checking/validation that can lead to extended execution times.

```
> # now lets add the description based on the UPC code number
> # one way of doing this is to use the "merge" function which will do a "join"
> # of the data based on a common index between the dataframes
> system.time(newX <- merge(x, upcs, by = "upc"))
  user  system elapsed
21.80   0.41   22.51
> str(newX)
'data.frame':   2496000 obs. of  5 variables:
 $ upc  : int  1 1 1 1 1 1 1 1 1 1 ...
 $ store: int  1 1 1 3 8 3 3 10 2 5 ...
 $ week : int  1 33 17 52 26 20 4 29 3 23 ...
 $ ship : num  4 1 4 3 2 3 3 2 1 3 ...
 $ desc : chr  "TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF" "TRADER JOES DOUBLE DARK WHOLE$
> head(newX)
  upc store week ship
1  1     1     1     4 TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
2  1     1    33     1 TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
3  1     1    17     4 TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
4  1     3    52     3 TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
5  1     8    26     2 TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
6  1     3    20     3 TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF
```

# Using the “base” functions

- Understanding how some of the “base” functions work can lead to improved performance. The technique of creating a set of indices and then using them is powerful and gets to the heart of “R” with “vectorization” of operations. Notice that this is 100X faster than the use of “merge” and gives the same result.

```
> # another way is to determine here the 'upc' in 'x' matches the 'upc' in 'upcs'
> # and then copy over the 'desc' to the result
> newX.match <- x # make a copy so we don't mess it up
> system.time({
+   indx <- match(x$upc, upcs$upc) # determine where they match
+   newX.match$desc <- upcs$desc[indx] # copy over the matching 'desc'
+ })
   user  system elapsed
   0.22   0.00   0.22
> str(newX.match)
'data.frame':   2496000 obs. of  5 variables:
 $ upc  : int  1 2 3 4 5 6 7 8 9 10 ...
 $ store: int  1 1 1 1 1 1 1 1 1 1 ...
 $ week : int  1 1 1 1 1 1 1 1 1 1 ...
 $ ship : num  4 6 1 1 3 15 7 3 5 1 ...
 $ desc : chr  "TRADER JOES DOUBLE DARK WHOLE BEAN COFFEE MONROVIA CALIF" "Red & White Sliced White Pot$
```

# Where Does the Time Go?

## ■ Profiling helps to see what is happening.

```
> Rprof() # turn on profiling
> system.time(newX <- merge(x, upcs, by = "upc"))
  user  system elapsed
31.77   0.50   32.84
> Rprof(NULL) # turn it off
> summaryRprof()
$by.self

```

	self.time	self.pct	total.time	total.pct
nchar	18.62	56.39	18.62	56.39
make.unique	6.22	18.84	7.54	22.83
data.frame	1.70	5.15	21.52	65.17
[.data.frame	1.48	4.48	10.06	30.47
as.character	1.32	4.00	1.32	4.00
anyDuplicated.default	0.86	2.60	0.86	2.60
merge.data.frame	0.72	2.18	32.76	99.21
unlist	0.68	2.06	0.68	2.06
match	0.38	1.15	0.38	1.15
sort.list	0.32	0.97	0.32	0.97
gc	0.26	0.79	0.26	0.79
is.na	0.10	0.30	0.10	0.30
length	0.08	0.24	0.08	0.24
list	0.08	0.24	0.08	0.24
any	0.06	0.18	0.06	0.18
names<-	0.06	0.18	0.06	0.18
attr<-	0.04	0.12	0.04	0.12
c	0.02	0.06	0.02	0.06
row.names<- .data.frame	0.02	0.06	0.02	0.06

Of the 32 secs, 18.6 were consumed by the 'nchar' function which counts the number of characters in a character object. 6.2 secs were in the 'make.unique' which makes character strings unique, which is important when combining dataframes that might have the same names for columns.

As mentioned before, 'merge' is general purpose and does a lot of validation on the data since it is not sure what the caller may be passing in.

# Another Way of Showing the Rprof Data

```
C:\jph\CinDay>perl /perf/bin/readRprof.pl Rprof.out
0 33.0 root
1. 33.0 system.time
2. . 32.8 merge
3. . . 32.8 merge.data.frame
4. . . . 21.5 cbind
5. . . . | 21.5 cbind
6. . . . | . 21.5 data.frame
7. . . . | . . 18.6 nchar
7. . . . | . . . 0.7 unlist
7. . . . | . . . 0.2 data.row.names
8. . . . | . . . . 0.2 anyDuplicated
9. . . . | . . . . . 0.2 anyDuplicated.default
7. . . . | . . . . . 0.2 anyDuplicated
8. . . . | . . . . . 0.2 anyDuplicated.default
7. . . . | . . . . . 0.1 list
7. . . . | . . . . . 0.0 any
7. . . . | . . . . . 0.0 attr<-
7. . . . | . . . . . 0.0 is.na
4. . . . 10.1 [
5. . . . | 10.1 [.data.frame
6. . . . | . 7.5 make.unique
7. . . . | . . 1.3 as.character
6. . . . | . . 0.5 anyDuplicated
7. . . . | . . . 0.5 anyDuplicated.default
6. . . . | . . . 0.3 sort.list
6. . . . | . . . 0.1 is.na
6. . . . | . . . 0.1 vector
7. . . . | . . . . 0.1 length
8. . . . | . . . . . 0.1 length
6. . . . | . . . . . 0.0 any
6. . . . | . . . . . 0.0 c
6. . . . | . . . . . 0.0 attr<-
4. . . . 0.4 match
4. . . . 0.1 names<-
4. . . . 0.0 row.names<-
5. . . . | 0.0 row.names<- .data.frame
2. . 0.3 gc
```

This shows that most of the time (21.5 secs) is spent in 'cbind' putting together the resulting dataframe. It is in there you can see 18.6 secs being used by 'nchar'.

This shows the "calling tree".

The 10.1 secs being used by "[" is the accessing of information in a dataframe. This can be costly if you are doing a lot of it. In many cases, depending on the structure of your data, you are better off (performance wise) is using a 'matrix' instead of a dataframe.

# Hints on Reading in Data

- If you don't need "factors", use "as.is = TRUE" in `read.table` & `read.csv` to read in as "characters".
  - Also goes when creating "data.frames"; use "stringsAsFactors = FALSE"
- If your data has quotes, and is not a 'csv' file, you will probably have to have "quotes = "" as a parameter. If you don't, you will probably see fewer lines read than what you thought you had in your file.
- If your data has "#" as part of data, use "comment.char=""
- If your data lines do not all have the same number of fields, you may have to understand what the 'fill' and 'flush' parameters do.
- 'read.table' tries to determine what type each field is, but it is best to use 'colClasses' to explicitly define the type of each field.

# Sample Performance Data From UNIX

- Blank separated fields from a 'vmstat' command executed every 30 seconds during the day.

```
date time r b w swap free re mf pi po fr de sr intr syscalls cs user sys id
07/27/05 00:13:06 0 0 0 27755440 13051648 20 86 0 0 0 0 0 0 456 2918 1323 0 1 99
07/27/05 00:13:36 0 0 0 27755280 13051480 11 53 0 0 0 0 0 0 399 1722 1411 0 1 99
07/27/05 00:14:06 0 0 0 27753952 13051248 18 88 0 0 0 0 0 0 424 1259 1254 0 1 99
07/27/05 00:14:36 0 0 0 27755304 13051496 17 85 0 0 0 0 0 0 430 1029 1246 0 1 99
07/27/05 00:15:06 0 0 0 27755064 13051232 41 278 0 1 1 0 0 0 452 2047 1386 0 1 99
07/27/05 00:15:36 0 0 0 27753824 13040720 125 1039 0 0 0 0 0 0 664 4097 1901 3 2 95
07/27/05 00:16:06 0 0 0 27754472 13027000 15 91 0 0 0 0 0 0 432 1160 1273 0 1 99
07/27/05 00:16:36 0 0 0 27754568 13027104 17 85 0 0 0 0 0 0 416 1058 1271 0 1 99
07/27/05 00:17:06 0 0 0 27754560 13027096 13 69 0 0 0 0 0 0 425 1198 1268 0 1 99
07/27/05 00:17:36 0 0 0 27754704 13027240 12 51 0 1 1 0 0 0 432 1727 1477 0 1 99
07/27/05 00:18:06 0 0 0 27755096 13027592 27 120 0 0 0 0 0 0 426 1449 1302 0 1 99
07/27/05 00:18:36 0 0 0 27755168 13027664 16 76 0 0 0 0 0 0 420 1002 1278 0 1 99
07/27/05 00:19:06 0 0 0 27755096 13027584 14 86 0 0 0 0 0 0 410 1224 1263 0 1 99
07/27/05 00:19:36 0 0 0 27755344 13027832 7 26 0 0 0 0 0 0 409 1606 1445 0 1 99
07/27/05 00:20:06 0 0 0 27755168 13027624 56 337 0 1 1 0 0 0 438 2112 1406 0 1 98
07/27/05 00:20:36 0 0 0 27755496 13027872 16 77 0 0 0 0 0 0 418 1045 1259 0 1 99
07/27/05 00:21:06 0 0 0 27755648 13028016 14 88 0 0 0 0 0 0 410 1264 1254 0 1 99
07/27/05 00:21:36 0 0 0 27755712 13028088 8 34 0 0 0 0 0 0 418 1666 1427 0 1 99
07/27/05 00:22:06 0 0 0 27755816 13028192 14 76 0 0 0 0 0 0 443 1246 1295 0 1 99
07/27/05 00:22:36 0 0 0 27755816 13028184 19 85 0 1 1 0 0 0 422 1084 1277 0 1 99
```

# Time Classes

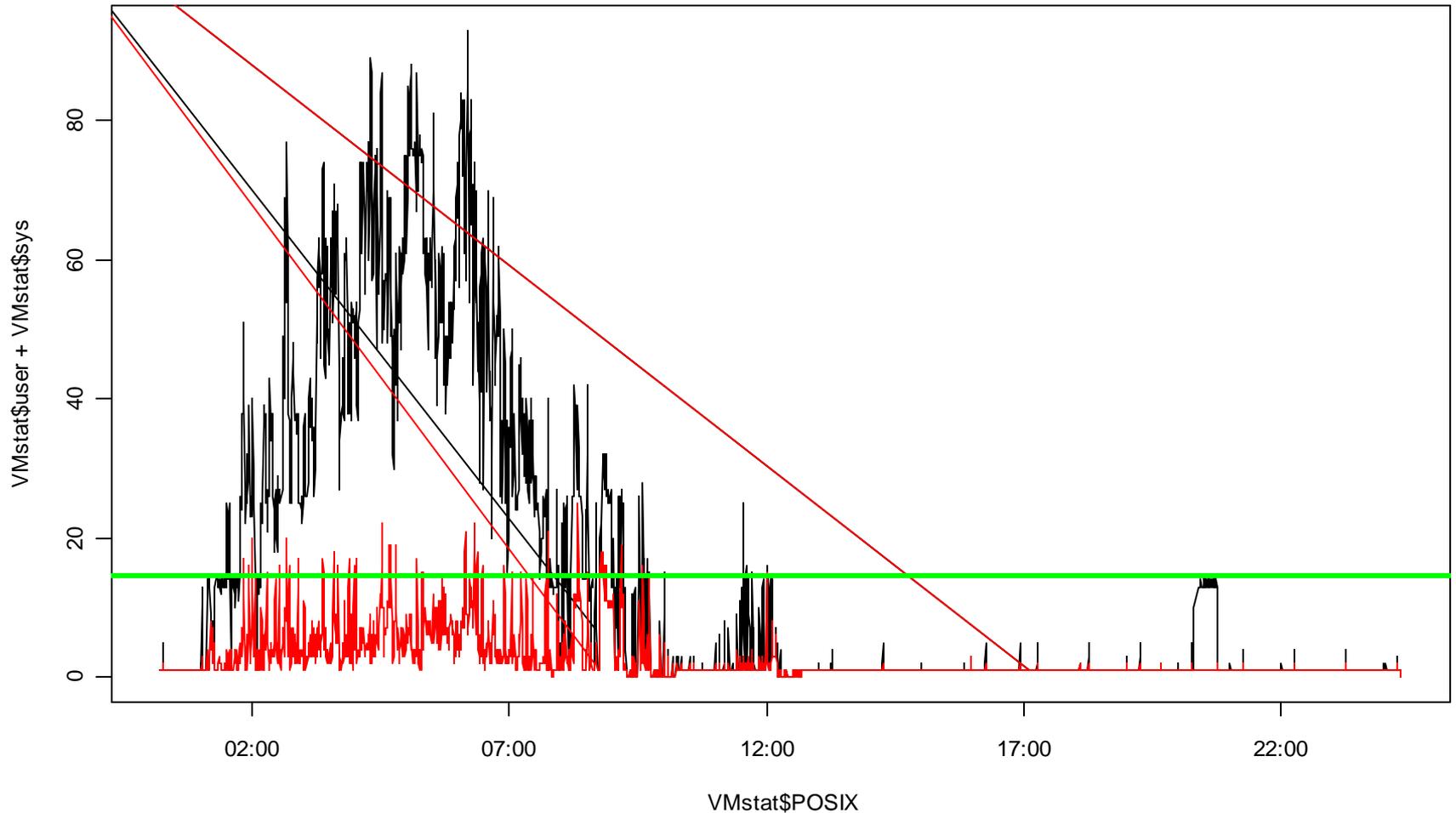
- **Some of your data will probably have some columns with time/date that you will have to handle.**
  - Need to convert from a character string into some time/date “class”
  - There are operations you can perform on dates: differences between them, when is a start of a month/quarter/year, plotting/summarizing by date, etc.
- **There are several different “classes” that can be used, but the two most prevalent one are “POSIX” and “Date”**
  - See the R Journal 4/1 June 2004 for a good discussion on the subject.
  - Using dates has a “learning curve”; the above reference helps.
- **Times and dates are typically read in as character strings and then converted to the appropriate date “class”**
- **I use “POSIXct” for almost all my date related values**
  - This is based on 1/1/1970 as the epoch which is the same as UNIX/LINUX uses and makes the transfer of data between systems easier.

# Read In and Convert the Time

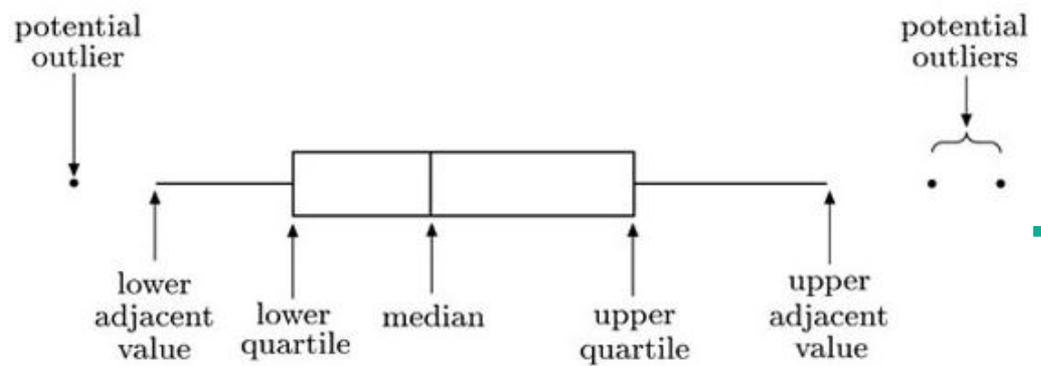
```
> # read in some vmstat data; data has 'header' defining the columns
> my.stats('start', oper = 'push') # function for timing my scripts
start (2) - Rgui : 22:16:26 <0.0 0.0> 18170.4 : 707.6MB
> VMstat <- read.table('vmstat.txt', header=TRUE, as.is=TRUE)
>
> # need to 'paste' together the two fields 'date' and 'time' for conversion
> VMstat$POSIX <- as.POSIXct(paste(VMstat$date, VMstat$time)
+                             , format = "%m/%d/%y %H:%M:%S"
+                             )
> my.stats('done', oper = 'pop')
done (2) - Rgui : 22:16:26 <0.4 0.5> 18170.9 : 711.5MB
> str(VMstat)
'data.frame': 2856 obs. of 21 variables:
 $ date      : chr  "07/27/05" "07/27/05" "07/27/05" "07/27/05" ...
 $ time      : chr  "00:13:06" "00:13:36" "00:14:06" "00:14:36" ...
 $ r         : int  0 0 0 0 0 0 0 0 0 0 ...
 $ b         : int  0 0 0 0 0 0 0 0 0 0 ...
 $ w         : int  0 0 0 0 0 0 0 0 0 0 ...
 $ swap      : int  27755440 27755280 27753952 27755304 27755064 27753824 27754472 27754568 27754560 27754$
 $ free      : int  13051648 13051480 13051248 13051496 13051232 13040720 13027000 13027104 13027096 13027$
 $ re        : int  20 11 18 17 41 125 15 17 13 12 ...
 $ mf        : int  86 53 88 85 278 1039 91 85 69 51 ...
 $ pi        : int  0 0 0 0 0 0 0 0 0 0 ...
 $ po        : int  0 0 0 0 1 0 0 0 0 1 ...
 $ fr        : int  0 0 0 0 1 0 0 0 0 1 ...
 $ de        : int  0 0 0 0 0 0 0 0 0 0 ...
 $ sr        : int  0 0 0 0 0 0 0 0 0 0 ...
 $ intr      : int  456 399 424 430 452 664 432 416 425 432 ...
 $ syscalls  : int  2918 1722 1259 1029 2047 4097 1160 1058 1198 1727 ...
 $ cs        : int  1323 1411 1254 1246 1386 1901 1273 1271 1268 1477 ...
 $ user      : int  0 0 0 0 0 3 0 0 0 0 ...
 $ sys       : int  1 1 1 1 1 2 1 1 1 1 ...
 $ id        : int  99 99 99 99 99 95 99 99 99 99 ...
 $ POSIX     : POSIXct, format: "2005-07-27 00:13:06" "2005-07-27 00:13:36" "2005-07-27 00:14:06" ...
```

# Plot 'user + sys' Over Time

```
plot(VMstat$POSIX, VMstat$user + VMstat$sys, type='l')  
lines(VMstat$POSIX, VMstat$sys, col='red')  
abline(h=mean(VMstat$user + VMstat$sys), col='green', lwd=3)
```



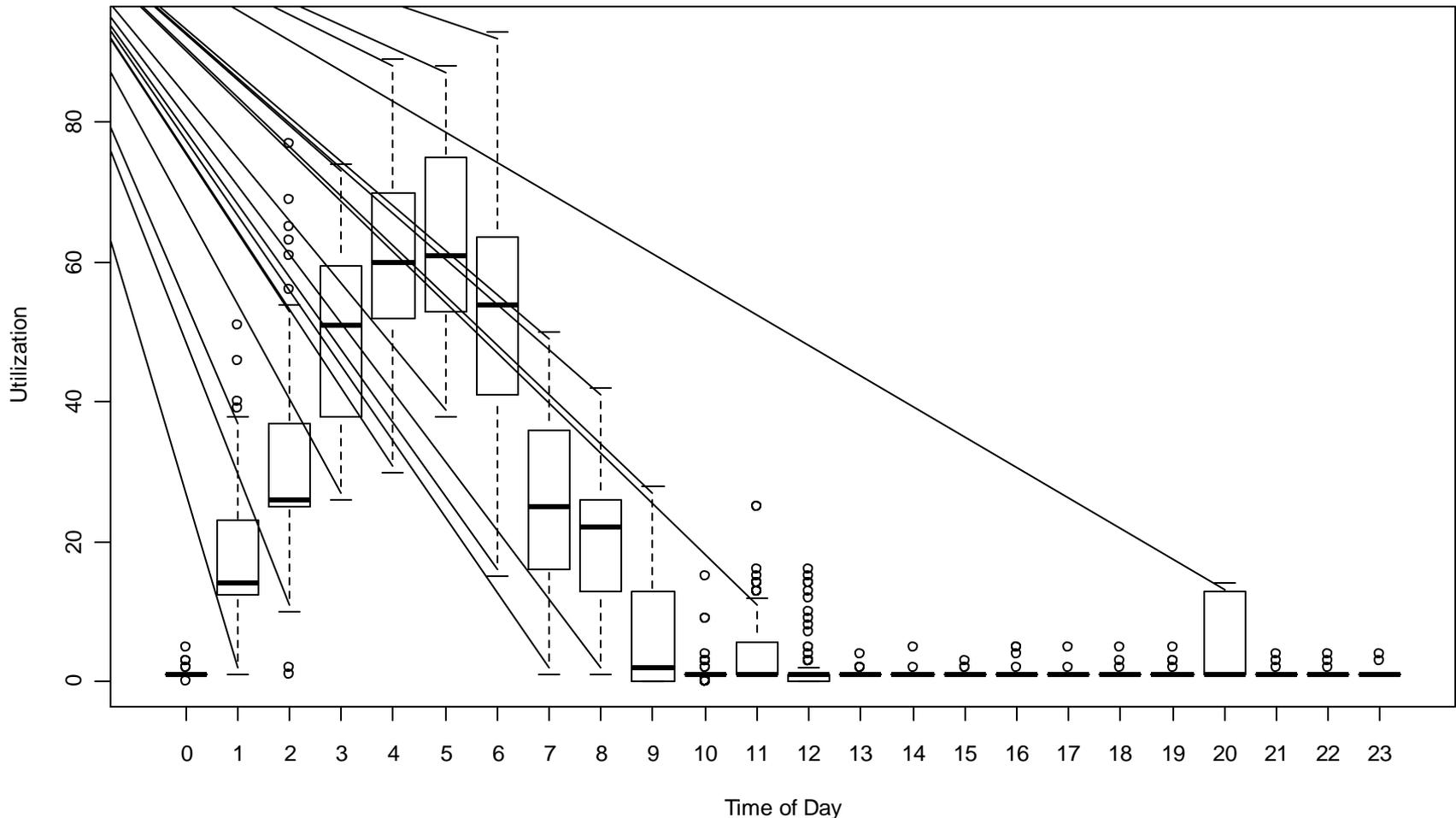
# Boxplots



- Many organizations like to summarize the utilization on some time period. I am going to assume that we would like to see statistics for each one hour period during the day.
- One technique that is used is to create a “box and whiskers” chart of the data. The ‘box’ contains 50% of the data points (between the 25<sup>th</sup> and 75% percentiles). The line in the box is the median value.
- The whiskers extend above/below the box to the last data point or a maximum of 1.5X the size of the box.
- Any data points lying outside the whiskers are plotted as individual points.

# boxplot Showing Utilization in Each Hour

```
VMstat$hour <- as.integer(format(VMstat$POSIX, format = "%H"))  
boxplot(user + sys ~ hour, data=VMstat, ylab="Utilization", xlab="Time of Day")
```



# String Handling/Regular Expressions

- Until recently, the only two languages I needed (out of the over 100 I have written programs in) were R and Perl: Perl to prepare the data for R, and R to analyze the data.
- R currently has most of the regular expression capabilities of Perl, and I have had to revert to Perl less and less since I can do most of my processing in R.
- So with the 4,000 product descriptions that we have, let's count up the number of times each word occurs and prints the 20 most frequently appearing.
- Let's then select one, and list out all that contain that word.

```

> # parse the description, breaking it into words (character strings separated by
> # blanks) and then count how many times each occurs. 'strsplit' will do it.
> words <- strsplit(upcs$desc, ' ') # split on blanks
>
> # this returns a 'list', the first few entries of which are
> str(words[1:8]) # only list the first 8 list entries
List of 8
 $ : chr [1:9] "TRADER" "JOES" "DOUBLE" "DARK" ...
 $ : chr [1:7] "Red" "&" "White" "Sliced" ...
 $ : chr [1:6] "Trader" "Joe's" "Almond" "Butter" ...
 $ : chr [1:5] "Trader" "Mings's" "General" "Tsao" ...
 $ : chr [1:8] "\"Savannah" "Presents" "Jazz\""" "-" ...
 $ : chr [1:7] "SHPR" "ADVIL.100" "+" "50" ...
 $ : chr [1:6] "Trader" "Joes" "French" "Roast" ...
 $ : chr [1:4] "Paddock" "Laboratories" "Ipecac" "Syrup"
>
> # to count them, 'unlist' to create a single character vector and then count
> words <- unlist(words)
> str(words) # now a character vector
chr [1:28343] "TRADER" "JOES" "DOUBLE" "DARK" "WHOLE" "BEAN" "COFFEE" "MONROVIA" ...
>
> # count and print the top 20
> head(sort(table(words), decreasing = TRUE), 20)
words
  -          THE          &          OZ          CT          The          OF
1325         404         342         261         254         221         215         187
  of        BEST Collector's      Epsom        and         1.00         the         for
  157         129         119         117         115         107         100         93
Edition         2         Kroger         Paper
  88          77          73          72

```

```

> # count the number of times a word appear and only list the ones that
> # appear in 32 products (just choose a number)
> word.count <- table(words)
> word.count[word.count == 32]
words
FOR    LEE White
 32    32    32
> # list the rows in 'upcs' that contain "White"
> subset(upcs, grepl("White", desc))
      upc                                     desc
2      2                                     Red & White Sliced White Potatoes
139   139                                   All Whites Egg Whites 16 Oz Carton W/Cap
140   140                                   All Whites Egg Whites 3-4 Oz Cups
143   143                                   All Whites Egg Whites 2-8 Oz Cartons
185   185                                   Bagables Refill White Self-Opening Bags
191   191                                   BIC White Out - Quick Dry Correction Fluid
519   519                                   Starkist Solid White Albacore Tuna in spring water
563   563                                   Charter Club Vail White Comforter
670   670                                   White Diamond Quilted Tree Skirt w/ applique
862   862                                   Princess Beanie Baby 1997 tag, Purple Color, White Rose
885   885                                   Ty Beanie Baby ( Red, White & Blue)
921   921                                   Avia Women's Running Sneakers White/Grey/Light Pink A212WWSQ
929   929                                   Art's Mexican Products - Lightly Salted - White Corn Chips
981   981                                   Trader Joes's Pomegranate White Tea - tea bags
1002  1002                                   AT-101S White Impedance matching volume control
1006  1006                                   Zig-Zag Cigarette Paper, KutCorners Blue, Free Buring, Super-White
1017  1017                                   Zig Zag White with Free White Lights
1050  1050                                   Lindt Lindor Milk Chocolate Truffle With White Center - Holiday Blend
1699  1699                                   Chely Wright - Single White Female CD
1817  1817                                   Woods Household Extension Cord, White
1928  1928                                   Shaun White Snowboarding game - Xbox 360
2082  2082                                   25 White Plastic Pom Poms (4" in diameter)
2367  2367                                   Paper Packs - White Cardstock - 8 1/2" x 11"
2710  2710                                   Polident Mouthwash Whitening For Denture Wearers Bright Mint
2711  2711                                   Polident Mouthwash Whitening For Denture Wearers Peppermint

```

# Debugging

- **All programs have bugs.**
- **When the “error” occurs, you need to “see” the environment in which it happened**
  - May be deep in a series of functions calls
  - Need to go up through each level to see what the parameters were
  - Need to examine the objects in each function environment
- **One way of trapping the error and gaining control is to put the following function call in your script; I have it as part of my Startup so that it is always active:**
  - `options(error = utils::recover)`
  - On a error it will give you the stack trace and let you set the “browser” at the appropriate environment to examine values.
- **Also checkout the ‘debug’ package.**

# Example of Processing Error

```
> # create two functions that call one another so we can create a error and show
> # how to look at values.
> f.1 <- function() f.2(list()) # just call f.2 with NULL list
> f.2 <- function(x) x[[2]] # cause a subscript error
> f.1() # cause error
Error in x[[2]] : subscript out of bounds
```

error message

Enter a frame number, or 0 to exit

```
1: f.1()
2: f.2(list())
```

Calling stacks

```
Selection: 2
Called from: top level
```

```
Browse[1]> ls()
[1] "x"
```

```
Browse[1]> x
list()
```

```
Browse[1]>
```

go to stack frame 2

get list of objects in frame

examine value of "x"

Enter a frame number, or 0 to exit

```
1: f.1()
2: f.2(list())
```

```
Selection: 0
> |
```

# FAQ 7.31

- In the R-Help news group, this is referred to a lot: “[Why doesn't R think these numbers are equal?](#)”

```
> # FAQ 7.31
> myData <- seq(0, 5, 0.1) # increments of 0.1
> myData
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8
[20] 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
[39] 3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0
> myData[49] # 4.8
[1] 4.8
> myData[49] == 4.8 # comes up FALSE!! Why?
[1] FALSE
> myData[49] - 4.8 # see the small rounding error?
[1] 8.881784e-16
> print(myData[49], digits = 20)
[1] 4.80000000000000007105
> print(4.8, digits = 20)
[1] 4.79999999999999998224
> all.equal(myData[49], 4.8) # use when comparing 'numerics'
[1] TRUE
```

“What Every Computer Scientist Should Know About Floating-Point Arithmetic”,  
*ACM Computing Surveys*, 23/1, 5–48, also available via  
<http://www.validlab.com/goldberg/paper.pdf>.

# Subset of R Functions to Start With

abline	cut	levels	quantile	strftime
abs	data.frame	lines	quit	strptime
all	density	list	range	strsplit
all.equal	deparse	lm	rbind	structure
any	dev.off	load	read.csv	substr
apply	diff	ls	read.table	sum
approx	dim	match	regexpr	summary
approxfun	do.call	matplot	rep	supsmu
arrows	duplicated	matrix	return	table
as.integer	eval	max	rle	tapply
as.numeric	exists	mean	rm	terms
as.POSIXct	factor	median	row	text
assign	floor	min	rowMeans	title
attr	flush.console	mtext	rownames	traceback
axis	for	names	rowSums	trunc
barplot	function	nchar	Rprof	trunc.POSIXt
boxplot	gc	ncol	rug	truncate
break	get	next	sample	try
c	grep	nrow	sapply	unclass
cat	help.search	numeric	save	unique
cbind	hist	options	save.image	unlist
ceiling	if	order	scan	which
character	ifelse	pairs	seq	which.max
colMeans	image	palette	set.seed	which.min
colSums	integer	par	setwd	while
count.fields	jitter	parse	sink	window
cummax	lapply	paste	sort	with
cummin	layout	pdf	source	write.csv
cumprod	layout.show	plot	split	
cumsum	length	postscript	sprintf	
curve	level.plot	print	str	