Overview and History of R

A dialog of S

What is S?

- S is a language that was developed by John Chambers and others at Bell Labs.
- S was initiated in 1976 as an internal statistical analysis environment—originally implemented as Fortran libraries.
- Early versions of the language did not contain functions for statistical modeling.
- In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language). The book *Statistical Models in S* by Chambers and Hastie (the white book) documents the statistical analysis functionality.
- Version 4 of the S language was released in 1998 and is the version we use today. The book *Programming with Data* by John Chambers (the green book) documents this version of the language.

What is R?

- 1991: Created in New Zealand by Ross Ihaka and Robert Gentleman. Their experience developing R is documented in a 1996 *JCGS* paper.
- 1993: First announcement of R to the public.
- 1995: Martin Mächler convinces Ross and Robert to use the GNU General Public License to make R free software.
- 1996: A public mailing list is created (R-help and R-devel)
- 1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
- 2000: R version 1.0.0 is released.
- 2013: R version 3.0.2 is released on December 2013.

Free Software

With *free software*, you are granted

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

http://www.fsf.org

Drawbacks of R

- Essentially based on 40 year old technology.
- Little built in support for dynamic or 3-D graphics (but things have improved greatly since the "old days").
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's *your* job!
    - (Or you need to pay someone to do it)
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too
- Not ideal for all possible situations (but this is a drawback of all software packages).

Design of the R System

The R system is divided into 2 conceptual parts:

1. The "base" R system that you download from CRAN
2. Everything else.

R functionality is divided into a number of *packages*.

- The "base" R system contains, among other things, the **base** package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include **utils**, **stats**, **datasets**, **graphics**, **grDevices**, **grid**, **methods**, **tools**, **parallel**, **compiler**, **splines**, **tcltk**, **stats4**.
- There are also "Recommend" packages: **boot**, **class**, **cluster**, **codetools**, **foreign**, **KernSmooth**, **lattice**, **mgcv**, **nlme**, **rpart**, **survival**, **MASS**, **spatial**, **nnet**, **Matrix**.
- There are about 4000 packages on CRAN that have been developed by users and programmers around the world.
- There are also many packages associated with the Bioconductor project (http://bioconductor.org).
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Resources for R

Available from CRAN (http://cran.r-project.org)

- An Introduction to R
- Writing R Extensions
- R Data Import/Export
- R Installation and Administration (mostly for building R from sources)
- R Internals (not for the faint of heart)

Useful books for R

Standard texts

- Chambers (2008). *Software for Data Analysis*, Springer. (your textbook)
- Chambers (1998). *Programming with Data*, Springer.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
- Venables & Ripley (2000). *S Programming*, Springer.
- Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-PLUS*, Springer.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.

Other resources

- Springer has a series of books called *Use R!*.
- A longer list of books is at http://www.r-project.org/doc/bib/R-books.html

R Console Input and Evaluation

At the R prompt we type expressions. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
> x <-  ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored.

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
> x <- 5  ## nothing printed
> x       ## auto-printing occurs
[1] 5
> print(x)  ## explicit printing
[1] 5
```

The `[1]` indicates that `x` is a vector and `5` is the first element.

Printing

```
> x <- 1:20
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20
```

The `:` operator is used to create integer sequences.

Data Types
R Objects and Attributes

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic object is a vector

- A vector can only contain objects of the same class
- BUT: The one exception is a *list*, which is represented as a vector but can contain objects of different classes (indeed, that's usually why we use them)

Empty vectors can be created with the `vector()` function.

Numbers

- Numbers in R a generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the `L` suffix
- Ex: Entering `1` gives you a numeric object; entering `1L` explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g. `1 / 0`; `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is 0
- The value `NaN` represents an undefined value ("not a number"); e.g. 0 / 0; `NaN` can also be thought of as a missing value (more on that later)

Attributes

R objects can have attributes

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/metadata

Attributes of an object can be accessed using the `attributes()` function.

Vectors and Lists

The `c()` function can be used to create vectors of objects.

```
> x <- c(0.5, 0.6)       ## numeric
> x <- c(TRUE, FALSE)    ## logical
> x <- c(T, F)           ## logical
> x <- c("a", "b", "c")  ## character
> x <- 9:29              ## integer
> x <- c(1+0i, 2+4i)     ## complex
```

Using the `vector()` function

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

What about the following?

```
> y <- c(1.7, "a")   ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE)  ## character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Nonsensical coercion results in `NA`s.

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Cbind-ing and rbind-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*.

- Factors are treated specially by modelling functions like `lm()` and `glm()`
- Using factors with labels is *better* than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no yes no
Levels: no yes
> table(x)
x
no yes
 2   3
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
              levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

Missing Values

Missing values are denoted by **NA** or **NaN** for undefined mathematical operations.

- **is.na()** is used to test objects if they are **NA**
- **is.nan()** is used to test for **NaN**
- **NA** values have a class also, so there are integer **NA**, character **NA**, etc.
- A **NaN** value is also **NA** but the converse is not true

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Data frames are used to store tabular data

- They are represented as a special type of list where every element of the list has to have the same length
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
- Data frames also have a special attribute called `row.names`
- Data frames are usually created by calling `read.table()` or `read.csv()`
- Can be converted to a matrix by calling `data.matrix()`

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Names Attribute

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("foo", "bar", "norf")
> x
foo bar norf
  1   2    3
> names(x)
[1] "foo"  "bar"  "norf"
```

List Names

Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3
```

Matrix Names

And matrices.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Summary of Data Types

- atomic classes: numeric, logical, character, integer, complex \
- vectors, lists
- factors
- missing values
- data frames
- names

Reading Tabular Data

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (`inverse` of `dump`)
- `dget`, for reading in R code files (`inverse` of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

Writing Data

There are analogous functions for writing data to files

- write.table
- writeLines
- dump
- dput
- save
- serialize

Read.table

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `file`, the name of a file, or a connection
- `header`, logical indicating if the file has a header line
- `sep`, a string indicating how the columns are separated
- `colClasses`, a character vector indicating the class of each column in the dataset
- `nrows`, the number of rows in the dataset
- `comment.char`, a character string indicating the comment character
- `skip`, the number of lines to skip from the beginning
- `stringsAsFactors`, should character variables be coded as factors?

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
data <- read.table("foo.txt")
```

R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table Telling R all these things directly makes R run faster and more efficiently.
- `read.csv` is identical to read.table except that the default separator is a comma.

Reading Larger Datasets with read.table

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.
- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick an dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

Know Your Computer System

In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

Calculating Memory Requirements

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

1,500,000 × 120 × 8 bytes/numeric

= 1440000000 bytes

= 1440000000 / 2^{20} bytes/MB

= 1,373.29 MB

= 1.34 GB

And then double it (memory required to read in the dataset)

=2.68 GB total

Textual Formats

- `dumping` and dputing are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- `Unlike` writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- `Textual` formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient

dput-ting R Objects

Another way to pass data around is by deparsing the R object with dput and reading it back in using `dget`.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)
structure(list(a = 1,
               b = structure(1L, .Label = "a",
                             class = "factor")),
          .Names = c("a", "b"), row.names = c(NA, -1L),
          class = "data.frame")
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
   a  b
1  1  a
```

Dumping R Objects

Multiple objects can be deparsed using the dump function and read back in using `source`.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a  b
1 1  a
> x
[1] "foo"
```

Connections – Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

File Connections

```
> str(file)
function (description = "", open = "", blocking = TRUE,
          encoding = getOption("encoding"))
```

- `description` is the name of the file
- `open` is a code indicating
  - "r" read only
  - "w" writing (and initializing a new file)
  - "a" appending
  - "rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

Connections

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
con <- file("foo.txt", "r")
data <- read.csv(con)
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

Reading Lines of a Text File

```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"      "10-point" "10th"      "11-point"
 [5] "12-point" "16-point" "18-point" "1st"
 [9] "2"         "20-point"
```

`writeLines` takes a character vector and writes each element one line at a time to a text file.

`readLines` can be useful for reading in lines of webpages

```
## This might take time
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
> head(x)
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"
[2] ""
[3] "<html>"
[4] "<head>"
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html;charset=utf-8
```

Subsetting – Basic

There are a number of operators that can be used to extract subsets of R objects.

- `[` always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
- `$` is used to extract elements of a list or data frame by name; semantics are similar to hat of `[[`.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
>
```

Lists

```
6/14
Subsetting Lists
> x <- list(foo = 1:4, bar = 0.6)
> x[1]
$foo
[1] 1 2 3 4

> x[[1]]
[1] 1 2 3 4

> x$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
> x["bar"]
$bar
[1] 0.6
```

Multiple Elements of a List

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$foo
[1] 1 2 3 4

$baz
[1] "hello"
```

Double Bracket Operator

The `[[` operator can be used with *computed* indices; `$` can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]]  ## computed index for 'foo'
[1] 1 2 3 4
> x$name     ## element 'name' doesn't exist!
NULL
> x$foo
[1] 1 2 3 4  ## element 'foo' does exist
```

Subsetting Nested Elements of a List

The `[[` can take an integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> x[[c(1, 3)]]
[1] 14
> x[[1]][[3]]
[1] 14

> x[[c(2, 1)]]
[1] 3.14
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (*i*,*j*) type indices.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1 × 1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,] 3
```

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

Drop = FALSE (returns subset as matrix, rather than vector)

Partial Matching

Partial matching of names is allowed with `[[` and `$`.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Removing Missing Values (NA)

A common task is to remove missing values (`NA`s).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

Subset of All Objects With No NA

What if there are multiple things and you want to take the subset with no missing values?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

Removing Rows with NA Values

```
> airquality[1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> good <- complete.cases(airquality)
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
```

Vectorized Operations

Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1] 7 9 11 13
> x > 2
[1] FALSE FALSE  TRUE  TRUE
> x >= 2
[1] FALSE  TRUE  TRUE  TRUE
> y == 8
[1] FALSE FALSE TRUE FALSE
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Matrix Operations

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y       ## element-wise multiplication
     [,1] [,2]
[1,]   10   30
[2,]   20   40
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y     ## true matrix multiplication
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

Control Structures in R

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an interation of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

If

```
if(<condition>) {
        ## do something
} else {
        ## do something else
}
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

This is a valid if/else structure.

```
if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

So is this one.

```
y <- if(x > 3) {
        10
} else {
        0
}
```

Of course, the else clause is not necessary.

```
if(<condition1>) {

}

if(<condition2>) {

}
```

For Loops

`for` loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
for(i in 1:10) {
        print(i)
}
```

This loop takes the `i` variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

These three loops have the same behavior.

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {
        print(x[i])
}

for(i in seq_along(x)) {
        print(x[i])
}

for(letter in x) {
        print(letter)
}

for(i in 1:4) print(x[i])
```

Nested for Loops

`for` loops can be nested.

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.

While Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0
while(count < 10) {
        print(count)
        count <- count + 1
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!

Sometimes there will be more than one condition in the test.

```
z <- 5

while(z >= 3 && z <= 10) {
        print(z)
        coin <- rbinom(1, 1, 0.5)

        if(coin == 1) {  ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}
```

Conditions are always evaluated from left to right.

Repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a `repeat` loop is to call `break`.

```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- computeEstimate()

        if(abs(x1 - x0) < tol) {
                break
        } else {
                x0 <- x1
        }
}
```

The loop in the previous slide is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

Next, Return

`next` is used to skip an iteration of a loop

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

`return` signals that a function should exit and return a given value

Control Structures Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if they are theoretically correct.
- Control structures mentiond here are primarily useful for writing programs; for command-line interactive work, the *apply functions are more useful.

Functions

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

```
f <- function(<arguments>) {
        ## Do something interesting
}
```

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have *named arguments* which potentially have *default values*.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

You can mix positional matching with matching by name. When an argument is matched by name, it is "taken out" of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

First looks for exact name match, then partial match, then positional match

Defining a Function

```
f <- function(a, b = 1, c = 2, d = NULL) {

}
```

In addition to not specifying a default value, you can also set an argument value to `NULL`.

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

```
f <- function(a, b) {
    a^2
}
f(2)
```

```
## [1] 4
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the 2 gets positionally matched to `a`.

```
f <- function(a, b) {
    print(a)
    print(b)
}
f(45)
```

```
## [1] 45
```

```
## Error: argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

The "..." Argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions.

- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)
}
```

- Generic functions use ... so that extra arguments can be passed to methods (more on this later).

```
> mean
function (x, ...)
UseMethod("mean")
```

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> args(cat)
function (..., file = "", sep = " ", fill = FALSE,
    labels = NULL, append = FALSE)
```

Arguments After the "..." Argument

One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (..., sep = " ", collapse = NULL)

> paste("a", "b", sep = ":")
[1] "a:b"

> paste("a", "b", se = ":")
[1] "a b :"
```

R Intro / Basics

A Diversion on Binding Values to Symbol

How does R know which value to assign to which symbol? When I type

```
> lm <- function(x) { x * x }
> lm
function(x) { x * x }
```

how does R know what value to assign to the symbol `lm`? Why doesn't it give it the value of `lm` that is in the *stats* package?

When R tries to bind a value to a symbol, it searches through a series of `environments` to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

1. Search the global environment for a symbol name matching the one requested.
2. Search the namespaces of each of the packages on the search list

The search list can be found by using the `search` function.

```
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Bringing Values to Symbol

- The *global environment* or the user's workspace is always the first element of the search list and the *base* package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named c and a function named c.

Scoping Rules

The scoping rules for R are the main feature that make it different from the original S language.

- The scoping rules determine how a value is associated with a free variable in a function
- R uses *lexical scoping* or *static scoping*. A common alternative is *dynamic scoping*.
- Related to the scoping rules is how R uses the search *list* to bind a value to a symbol
- Lexical scoping turns out to be particularly useful for simplifying statistical computations

Lexical Scoping

Consider the following function.

```
f <- function(x, y) {
        x^2 + y / z
}
```

This function has 2 formal arguments `x` and `y`. In the body of the function there is another symbol `z`. In this case `z` is called a *free* variable. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned insided the function body).

Lexical scoping in R means that

the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

- An *environment* is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple "children"
- the only environment without a parent is the empty environment
- A function + an environment = a *closure* or *function closure*.

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the *parent environment*.
- The search continues down the sequence of parent environments until we hit the *top-level environment*; this usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the *empty environment*. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Why does all this matter?

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace
- This behavior is logical for most people and is usually the "right thing" to do
- However, in R you can have functions defined *inside other functions*
  - Languages like C don't let you do this
- Now things get interesting — In this case the environment in which a function is defined is the body of another function!

```
make.power <- function(n) {
        pow <- function(x) {
                x^n
        }
        pow
}
```

This function returns another function as its value.

```
> cube <- make.power(3)
> square <- make.power(2)
> cube(3)
[1] 27
> square(3)
[1] 9
```

Exploring a Function Closure

What's in a function's environment?

```
> ls(environment(cube))
[1] "n"   "pow"
> get("n", environment(cube))
[1] 3

> ls(environment(square))
[1] "n"   "pow"
> get("n", environment(square))
[1] 2
```

Lexical vs. Dynamic Scoping

```
y <- 10

f <- function(x) {
        y <- 2
        y^2 + g(x)
}

g <- function(x) {
        x*y
}
```

What is the value of

```
f(3)
```

- With lexical scoping the value of `y` in the function `g` is looked up in the environment in which the function was defined, in this case the global environment, so the value of `y` is 10.
- With dynamic scoping, the value of `y` is looked up in the environment from which the function was *called* (sometimes referred to as the *calling environment*).
  - In R the calling environment is known as the *parent frame*
- So the value of `y` would be 2.

When a function is *defined* in the global environment and is subsequently *called* from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
> g <- function(x) {
+ a <- 3
+ x+a+y
+ }
> g(2)
Error in g(2) : object "y" not found
> y <- 3
> g(2)
[1] 8
```

Other Lexical Scoping Languages

Other languages that support lexical scoping

- Scheme
- Perl
- Python
- Common Lisp (all languages converge to Lisp)

Consequences of Lexical Scoping

- In R, all objects must be stored in memory
- All functions must carry a pointer to their respective defining environments, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.

Application: Optimization

Why is any of this information useful?

- Optimization routines in R like `optim`, `nlm`, and `optimize` require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)
- However, an object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

Maximizing a Normal Likelihood

Write a "constructor" function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
        params <- fixed
        function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
}
```

Note: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood.

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> nLL <- make.NegLogLik(normals)
> nLL
function(p) {
                params[!fixed] <- p
                mu <- params[1]
                sigma <- params[2]
                a <- -0.5*length(data)*log(2*pi*sigma^2)
                b <- -0.5*sum((data-mu)^2) / (sigma^2)
                -(a + b)
        }
<environment: 0x165bla4>
> ls(environment(nLL))
[1] "data"   "fixed"  "params"
```

Estimating Parameters

```
> optim(c(mu = 0, sigma = 1), nLL)$par
      mu    sigma
1.218239 1.787343
```

Fixing $\sigma = 2$

```
> nLL <- make.NegLogLik(normals, c(FALSE, 2))
> optimize(nLL, c(-1, 3))$minimum
[1] 1.217775
```

Fixing $\mu = 1$

```
> nLL <- make.NegLogLik(normals, c(1, FALSE))
> optimize(nLL, c(1e-6, 10))$minimum
[1] 1.800596
```

Plotting the Likelihood

```
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")

nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l")
```

Lexical Scoping Summary

- Objective functions can be "built" which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists — useful for interactive and exploratory work.
- Code can be simplified and cleand up
- Reference: Robert Gentleman and Ross Ihaka (2000). "Lexical Scope and Statistical Computing," *JCGS*, 9, 491–508.

Coding Standards in R

1. Always use text files / text editor
2. Indent your code
3. Limit the width of your code (80 columns?)
4. Limit the length of individual functions

- Indenting improves readability
- Fixing line length (80 columns) prevents lots of nesting and very long functions
- Suggested: Indents of 4 spaces at minimum; 8 spaces ideal

Dates and Times in R

R has developed a special representation of dates and times

- Dates are represented by the `Date` class
- Times are represented by the `POSIXct` or the `POSIXlt` class
- Dates are stored internally as the number of days since 1970-01-01
- Times are stored internally as the number of seconds since 1970-01-01

Dates

Dates are represented by the Date class and can be coerced from a character string using the `as.Date()` function.

```
x <- as.Date("1970-01-01")
x
## [1] "1970-01-01"
unclass(x)
## [1] 0
unclass(as.Date("1970-01-02"))
## [1] 1
```

Times

Times are represented using the `POSIXct` or the `POSIXlt` class

- `POSIXct` is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame
- `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

There are a number of generic functions that work on dates and times

- `weekdays`: give the day of the week
- `months`: give the month name
- `quarters`: give the quarter number ("Q1", "Q2", "Q3", or "Q4")

Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.

```
x <- Sys.time()
x
## [1] "2013-01-24 22:04:14 EST"
p <- as.POSIXlt(x)
names(unclass(p))
## [1] "sec"   "min"   "hour"  "mday"  "mon"
## [6] "year"  "wday"  "yday"  "isdst"
p$sec
## [1] 14.34
```

You can also use the `POSIXct` format.

```
x <- Sys.time()
x  ## Already in 'POSIXct' format
## [1] "2013-01-24 22:04:14 EST"
unclass(x)
## [1] 1359083054
x$sec
## Error: $ operator is invalid for atomic vectors
p <- as.POSIXlt(x)
p$sec
## [1] 14.37
```

Finally, there is the `strptime` function in case your dates are written in a different format

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-10 10:40:00 EST" "2011-12-09 09:10:00 EST"
```

```
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

I can *never* remember the formatting strings. Check `?strptime` for details.

Summary

- Dates and times have special classes in R that allow for numerical and statistical calculations
- Dates use the `Date` class
- Times use the `POSIXct` and `POSIXlt` class
- Character strings can be coerced to Date/Time classes using the `strptime` function or the `as.Date`, `as.POSIXlt`, or `as.POSIXct`

Loop Functions

Writing for, while loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

- `lapply`: Loop over a list and evaluate a function on each element
- `sapply`: Same as `lapply` but try to simplify the result
- `apply`: Apply a function over the margins of an array
- `tapply`: Apply a function over subsets of a vector
- `mapply`: Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

Lapply

`lapply` takes three arguments: (1) a list `x`; (2) a function (or the name of a function) `FUN`; (3) other arguments via its ... argument. If `x` is not a list, it will be coerced to a list using `as.list`.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x7ff7a195lc00>
## <environment: namespace:base>
```

The actual looping is done internally in C code.

`lapply` always returns a list, regardless of the class of the input.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
[1] 3

$b
[1] 0.0296824
```

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

```
> x <- 1:4
> lapply(x, runif)
[[1]]
[1] 0.2675082

[[2]]
[1] 0.2186453 0.5167968

[[3]]
[1] 0.2689506 0.1811683 0.5185761

[[4]]
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
[[1]]
[1] 3.302142

[[2]]
[1] 6.848960 7.195282

[[3]]
[1] 3.5031416 0.8465707 9.7421014

[[4]]
[1] 1.195114 3.594027 2.930794 2.766946
```

`lapply` and friends make heavy use of *anonymous* functions.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
> x
$a
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$b
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

An anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) elt[,1])
$a
[1] 1 2

$b
[1] 1 2 3
```

Sapply

`sapply` will try to simplify the result of `lapply` if possible.

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5

$b
[1] 0.06082667

$c
[1] 1.467083

$d
[1] 5.074749
```

```
> sapply(x, mean)
         a          b          c          d
2.50000000 0.06082667 1.46708277 5.07474950

> mean(x)
[1] NA
Warning message:
In mean.default(x) : argument is not numeric or logical: returning NA
```

Apply

`apply` is used to a evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!

```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- `X` is an array
- `MARGIN` is an integer vector indicating which margins should be "retained".
- `FUN` is a function to be applied
- ... is for other arguments to be passed to `FUN`

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
 [1]  0.04868268  0.35743615 -0.09104379
 [4] -0.05381370 -0.16552070 -0.18192493
 [7]  0.10285727  0.36519270  0.14898850
[10]  0.26767260

> apply(x, 1, sum)
 [1] -1.94843314  2.60601195  1.51772391
 [4] -2.80386816  3.73728682 -1.69371360
 [7]  0.02359932  3.91874808 -2.39902859
[10]  0.48685925 -1.77576824 -3.34016277
[13]  4.04101009  0.46515429  1.83687755
[16]  4.36744690  2.21993789  2.60983764
[19] -1.48607630  3.58709251
```

Col/row sums and means

For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

The shortcut functions are *much* faster, but you won't notice unless you're using a large matrix.

Other Ways to Apply

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
          [,1]       [,2]       [,3]        [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686
75%  0.9258157  0.07065724  0.3050407 -0.06585436
          [,5]       [,6]      [,7]       [,8]
25% -0.05999553 -0.6588380 -0.653250 0.01749997
75%  0.52928743  0.3727449  1.255089 0.72318419
          [,9]      [,10]      [,11]      [,12]
25% -1.2467955 -0.8378429 -1.0488430 -0.7054902
75%  0.3352377  0.7297176  0.3113434  0.4581150
         [,13]      [,14]      [,15]      [,16]
25% -0.1895108 -0.5729407 -0.5968578 -0.9517069
75%  0.5326299  0.5064267  0.4933852  0.8868922
         [,17]      [,18]      [,19]     [,20]
```

Apply average of matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
           [,1]       [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908

> rowMeans(a, dims = 2)
           [,1]       [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748  0.04364908
```

Mapply

mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
          USE.NAMES = TRUE)
```

- **FUN** is a function to apply
- ... contains arguments to apply over
- **MoreArgs** is a list of other arguments to **FUN**.
- **SIMPLIFY** indicates whether the result should be simplified

The following is tedious to type

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

Vectoring a Function

```
> noise <- function(n, mean, sd) {
+ rnorm(n, mean, sd)
+ }
> noise(5, 1, 2)
[1]  2.4831198  2.4790100  0.6855190 -1.2117759
[5] -0.2743532

> noise(1:5, 1:5, 2)
[1] -4.2128648 -0.3989266  4.2507057  1.1572738
[5]  3.7413584
```

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658

[[2]]
[1] 0.7113482 2.7555797

[[3]]
[1] 2.769527 1.643568 4.597882

[[4]]
[1] 4.476741 5.650653 3.962813 1.204284

[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

Which is the same as

```
list(noise(1, 1, 2), noise(2, 2, 2),
     noise(3, 3, 2), noise(4, 4, 2),
     noise(5, 5, 2))
```

Tapply

`tapply` is used to apply a function over subsets of a vector. I don't know why it's called `tapply`.

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

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they are coerced to factors)
- `FUN` is a function to be applied
- ... contains other arguments to be passed `FUN`
- `simplify`, should we simplify the result?

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
        1         2         3
0.1144464 0.5163468 1.2463678
```

Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
$`1`
[1] 0.1144464

$`2`
[1] 0.5163468

$`3`
[1] 1.246368
```

Find group ranges.

```
> tapply(x, f, range)
$`1`
[1] -1.097309  2.694970

$`2`
[1] 0.09479023 0.79107293

$`3`
[1] 0.4717443 2.5887025
```

Split

`split` takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
> str(split)
function (x, f, drop = FALSE, ...)
```

- `x` is a vector (or list) or data frame
- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$`1`
 [1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
 [5]  0.2849881  0.9383361 -1.0973089  2.6949703
 [9]  1.5976789 -0.1321970

$`2`
 [1] 0.09479023 0.79107293 0.45857419 0.74849293
 [5] 0.34936491 0.35842084 0.78541705 0.57732081
 [9] 0.46817559 0.53183823

$`3`
 [1] 0.6795651 0.9293171 1.0318103 0.4717443
 [5] 2.5887025 1.5975774 1.3246333 1.4372701
```

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
$`1`
[1] 0.1144464

$`2`
[1] 0.5163468

$`3`
[1] 1.246368
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

```
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$`5`
   Ozone  Solar.R     Wind
      NA       NA 11.62258
$`6`
    Ozone   Solar.R      Wind
       NA 190.16667  10.26667
$`7`
     Ozone    Solar.R       Wind
        NA 216.483871   8.941935
```

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
               5         6          7        8        9
Ozone         NA        NA         NA       NA       NA
Solar.R       NA 190.16667 216.483871       NA 167.4333
Wind    11.62258  10.26667   8.941935 8.793548  10.1800

> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],
                                 na.rm = TRUE))
                5           6           7           8          9
Ozone    23.61538    29.44444    59.115385   59.961538   31.44828
Solar.R 181.29630   190.16667   216.483871  171.857143  167.43333
Wind     11.62258    10.26667     8.941935    8.793548   10.18000
```

Splitting on More than One Level

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
 [1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1, f2)
 [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5
```

Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))
List of 10
 $ 1.1: num [1:2] -0.378  0.445
 $ 2.1: num(0)
 $ 1.2: num [1:2] 1.4066 0.0166
 $ 2.2: num(0)
 $ 1.3: num -0.355
 $ 2.3: num 0.315
 $ 1.4: num(0)
 $ 2.4: num [1:2] -0.907  0.723
 $ 1.5: num(0)
 $ 2.5: num [1:2] 0.732 0.360
```

Split Empty Levels can be Dropped

Empty levels can be dropped.

```
> str(split(x, list(f1, f2), drop = TRUE))
List of 6
 $ 1.1: num [1:2] -0.378  0.445
 $ 1.2: num [1:2] 1.4066 0.0166
 $ 1.3: num -0.355
 $ 2.3: num 0.315
 $ 2.4: num [1:2] -0.907  0.723
 $ 2.5: num [1:2] 0.732 0.360
```

Debugging

Indications that something's not right

- `message`: A generic notification/diagnostic message produced by the `message` function; execution of the function continues
- `warning`: An indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the `warning` function
- `error`: An indication that a fatal problem has occurred; execution stops; produced by the `stop` function
- `condition`: A generic concept for indicating that something unexpected can occur; programmers can create their own conditions

Warning

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Something's Wrong

```
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

```
printmessage <- function(x) {
          if(x > 0)
                print("x is greater than zero")
          else
                print("x is less than or equal to zero")
          invisible(x)
}
> printmessage(1)
[1] "x is greater than zero"
> printmessage(NA)
Error in if (x > 0) { : missing value where TRUE/FALSE needed
```

```
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than or equal to zero")
        invisible(x)
}
```

```
printmessage2 <- function(x) {
       if(is.na(x))
               print("x is a missing value!")
       else if(x > 0)
               print("x is greater than zero")
       else
               print("x is less than or equal to zero")
       invisible(x)
}
> x <- log(-1)
Warning message:
In log(-1) : NaNs produced
> printmessage2(x)
[1] "x is a missing value!"
```

How do you know that something is wrong with your function?

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Debugging Tools in R

The primary tools for debugging functions in R are

- `traceback`: prints out the function call stack after an error occurs; does nothing if there's no error
- `debug`: flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- `browser`: suspends the execution of a function wherever it is called and puts the function in debug mode
- `trace`: allows you to insert debugging code into a function a specific places
- `recover`: allows you to modify the error behavior so that you can browse the function call stack

These are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting print/cat statements in the function.

Traceback

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
>
```

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

Debug

```
> debug(lm)
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    ...
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]>
```

```
Browse[2]> n
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
```

Recover

```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec =
3: file(file, "rt")

Selection:
```

Summary

Summary

- There are three main indications of a problem/condition: `message`, `warning`, `error`
  - only an `error` is fatal
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation
- Interactive debugging tools `traceback`, `debug`, `browser`, `trace`, and `recover` can be used to find problematic code in functions
- Debugging tools are not a substitute for thinking!

Str Function

`str`: Compactly display the internal structure of an R object

- A diagnostic function and an alternative to 'summary'
- It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists.
- Roughly one line per basic object

What's in this object?

```
> str(lm)
function (formula, data, subset, weights, na.action, method = "qr", model = TRUE
,
    x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL,
    offset, ...)
> str(ls)
function (name, pos = -1, envir = as.environment(pos), all.names = FALSE,
    pattern)
> x <- rnorm(100, 2, 4)
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-7.6690 -0.8765  1.7600  1.8780  4.7290 11.0200
> str(x)
 num [1:100] 7.299 -4.668 -5.345 -0.584 5.177 ...
> f <- gl(40, 10)
> str(f)
 Factor w/ 40 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
> summary(f)
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
27 28 29 30 31 32 33 34 35 36 37 38 39 40
10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> str(airquality)
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
> m <- matrix(rnorm(100), 10, 10)
> str(m)
 num [1:10, 1:10] 0.6197 1.0626 0.0667 -0.7731 0.1197 ...
> m[, 1]
 [1]  0.61974138  1.06261394  0.06672329 -0.77314190  0.11971410  1.55841883
 [7] -1.06157291  0.81664997 -0.95719759  0.29296738
```

```
> s <- split(airquality, airquality$Month)
> str(s)
List of 5
 $ 5:'data.frame':      31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
  ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
  ..$ Wind   : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
  ..$ Temp   : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
  ..$ Month  : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 6:'data.frame':      30 obs. of  6 variables:
  ..$ Ozone  : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
  ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
  ..$ Wind   : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
  ..$ Temp   : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
  ..$ Month  : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
  ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
 $ 7:'data.frame':      31 obs. of  6 variables:
```

Simulation

Generating Random Numbers

Functions for probability distributions in R

- `rnorm`: generate random Normal variates with a given mean and standard deviation
- `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- `pnorm`: evaluate the cumulative distribution function for a Normal distribution
- `rpois`: generate random Poisson variates with a given rate

```
> x <- rnorm(10)
> x
 [1] 1.38380206 0.48772671 0.53403109 0.66721944
 [5] 0.01585029 0.37945986 1.31096736 0.55330472
 [9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
 [1] 23.38812 20.16846 21.87999 20.73813 19.59020
 [6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  18.32   19.73   20.55   20.67   21.67   23.39
```

Probability Distribution Functions

Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

- `d` for density
- `r` for random number generation
- `p` for cumulative distribution
- `q` for quantile function

Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

If Φ is the cumulative distribution function for a standard Normal distribution, then `pnorm(q)` = $\Phi(q)$ and `qnorm(p)` = $\Phi^{-1}(p)$.

Random Numbers using Set.seed

Setting the random number seed with `set.seed` ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

Always set the random number seed when conducting a simulation!

Poisson Distributions

Generating Poisson data

```
> rpois(10, 1)
 [1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
 [1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
 [1] 20 11 21 20 20 21 17 15 24 20

> ppois(2, 2)  ## Cumulative distribution
[1] 0.6766764  ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347   ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662  ## Pr(x <= 6)
```

Simulating a Linear Model

Suppose we want to simulate from the following linear model

$$y = \beta_0 + \beta_1 x + \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
    Min.  1st Qu.   Median
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```

If X is Binary

What if **x** is binary?

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
    Min.  1st Qu.   Median
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```

Simulating a Poisson Model

Y ~ Poisson(μ)

$\log \mu = \beta_0 + \beta_1 x$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$. We need to use the `rpois` function for this

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.00    1.00    1.00    1.55    2.00    6.00
> plot(x, y)
```

Simulating a Random Sampling

The `sample` function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
> sample(1:10)  ## permutation
 [1] 4 710 6 9 2 8 3 1 5
> sample(1:10)
 [1]  2  3  4  1  9  5 10  8  6  7
> sample(1:10, replace = TRUE)  ## Sample w/replacement
 [1] 2 9 7 8 2 8 5 9 7 8
```

Summary

- Drawing samples from specific probability distributions can be done with `r*` functions
- Standard distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc.
- The `sample` function can be used to draw random samples from arbitrary vectors
- Setting the random number generator seed via set.seed is critical for reproducibility

R Profiler – Optimizing Software
Why is My Code Sop Slow?

- Profiling is a systematic way to examine how much time is spend in different parts of a program
- Useful when trying to optimize your code
- Often code runs fine once, but what if you have to put it in a loop for 1,000 iterations? Is it still fast enough?
- Profiling is better than guessing

On Optimizing Your Code

- Getting biggest impact on speeding up code depends on knowing where the code spends most of its time
- This cannot be done without performance analysis or profiling

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil
--Donald Knuth

- Design first, then optimize
- Remember: Premature optimization is the root of all evil
- Measure (collect data), don't guess.
- If you're going to be scientist, you need to apply the same principles here!

Using system.time()

- Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression
- Computes the time (in seconds) needed to execute an expression
  - If there's an error, gives time until the error occurred
- Returns an object of class `proc_time`
  - **user time**: time charged to the CPU(s) for this expression
  - **elapsed time**: "wall clock" time
- Usually, the user time and elapsed time are relatively close, for straight computing tasks
- Elapsed time may be *greater than* user time if the CPU spends a lot of time waiting around
- Elapsted time may be *smaller than* the user time if your machine has multiple cores/processors (and is capable of using them)
  - Multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL)
  - Parallel processing via the **parallel** package

```
## Elapsed time > user time
system.time(readLines("http://www.jhsph.edu"))
   user  system elapsed
  0.004   0.002   0.431

## Elapsed time < user time
hilbert <- function(n) {
        i <- 1:n
        1 / outer(i - 1, i, "+")
}
x <- hilbert(1000)
system.time(svd(x))
   user  system elapsed
  1.605   0.094   0.742
```

Timing Longer Expressions

```
system.time({
    n <- 1000
    r <- numeric(n)
    for (i in 1:n) {
        x <- rnorm(n)
        r[i] <- mean(x)
    }
})
```

```
##    user  system elapsed
##   0.097   0.002   0.099
```

- Using `system.time()` allows you to test certain functions or code blocks to see if they are taking excessive amounts of time
- Assumes you already know where the problem is and can call `system.time()` on it
- What if you don't know where to start?

Using Rprof()

- The `Rprof()` function starts the profiler in R
  - R must be compiled with profiler support (but this is usually the case)
- The `summaryRprof()` function summarizes the output from `Rprof()` (otherwise it's not readable)
- DO NOT use `system.time()` and `Rprof()` together or you will be sad
- Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spend in each function
- Default sampling interval is 0.02 seconds
- NOTE: If your code runs very quickly, the profiler is not useful, but then you probably don't need it in that case

R Profiler Raw Output

```
## lm(y ~ x)

sample.interval=10000
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"list" "eval" "eval" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"na.omit" "model.frame.default" "model.frame" "eval" "eval" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
"lm.fit" "lm"
```

Using SummaryRprof()

- The `summaryRprof()` function tabulates the R profiler output and calculates how much time is spend in which function
- There are two methods for normalizing the data
- "by.total" divides the time spend in each function by the total run time
- "by.self" does the same but first subtracts out time spent in functions above in the call stack

By.total

How many times function spends in top level functions

```
$by.total
                      total.time total.pct self.time self.pct
"lm"                        7.41    100.00      0.30     4.05
"lm.fit"                    3.50     47.23      2.99    40.35
"model.frame.default"       2.24     30.23      0.12     1.62
"eval"                      2.24     30.23      0.00     0.00
"model.frame"               2.24     30.23      0.00     0.00
"na.omit"                   1.54     20.78      0.24     3.24
"na.omit.data.frame"        1.30     17.54      0.49     6.61
"lapply"                    1.04     14.04      0.00     0.00
"[.data.frame"              1.03     13.90      0.79    10.66
"["                         1.03     13.90      0.00     0.00
"as.list.data.frame"        0.82     11.07      0.82    11.07
"as.list"                   0.82     11.07      0.00     0.00
```

By.self

How many times function spends in top level functions, after subtracting out all of the lower level functions it calls

```
$by.self
                      self.time self.pct total.time total.pct
"lm.fit"                    2.99    40.35       3.50     47.23
"as.list.data.frame"        0.82    11.07       0.82     11.07
"[.data.frame"              0.79    10.66       1.03     13.90
"structure"                 0.73     9.85       0.73      9.85
"na.omit.data.frame"        0.49     6.61       1.30     17.54
"list"                      0.46     6.21       0.46      6.21
"lm"                        0.30     4.05       7.41    100.00
"model.matrix.default"      0.27     3.64       0.79     10.66
"na.omit"                   0.24     3.24       1.54     20.78
"as.character"              0.18     2.43       0.18      2.43
"model.frame.default"       0.12     1.62       2.24     30.23
"anyDuplicated.default"     0.02     0.27       0.02      0.27
```

summaryRprof() Output

```
$sample.interval
[1] 0.02

$sampling.time
[1] 7.41
```

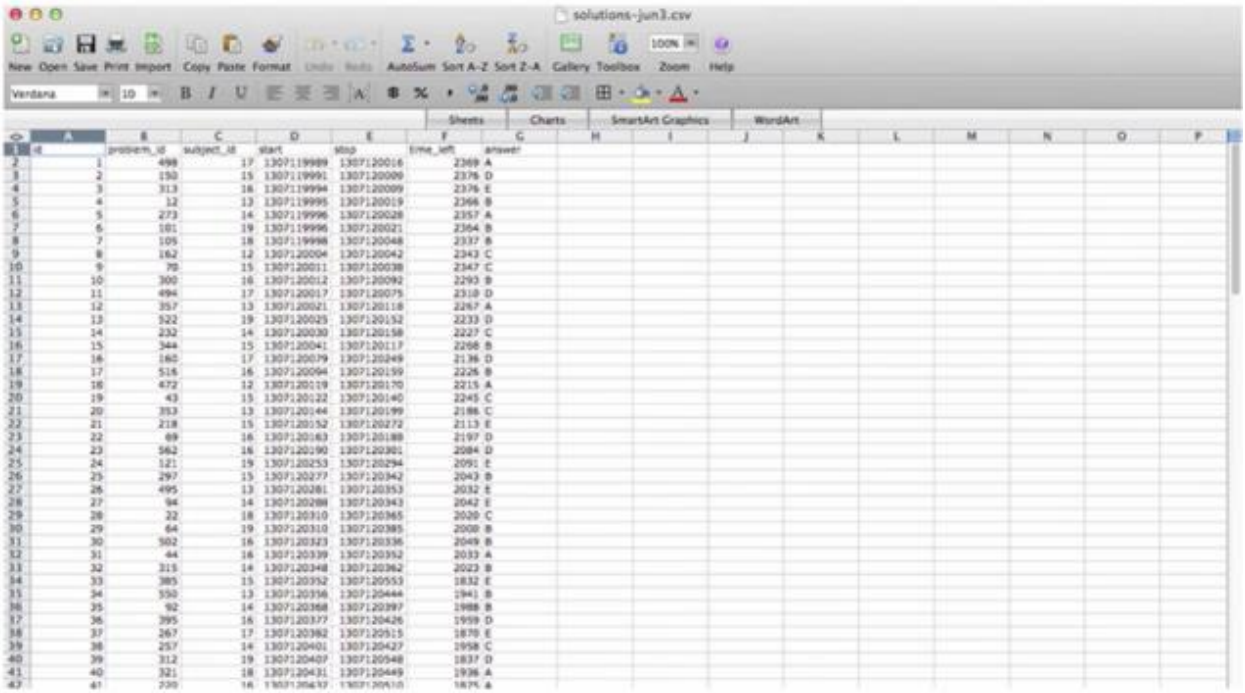
Summary

- `Rprof()` runs the profiler for performance of analysis of R code
- `summaryRprof()` summarizes the output of `Rprof()` and gives percent of time spent in each function (with two types of normalization)
- Good to break your code into functions so that the profiler can give useful information about where time is being spent
- C or Fortran code is not profiled

Motivation and Pre-Requisites

- This course covers the basic ideas behind getting data ready for analysis
  - Finding and extracting raw data
  - Tidy data principles and how to make data tiny
  - Practical implementation through a range of R packages
- What this course depends on
  - The Data Scientist's Toolbox
  - R Programming
- What would be useful
  - Exploratory analysis
  - Reporting Data and Reproducible Research

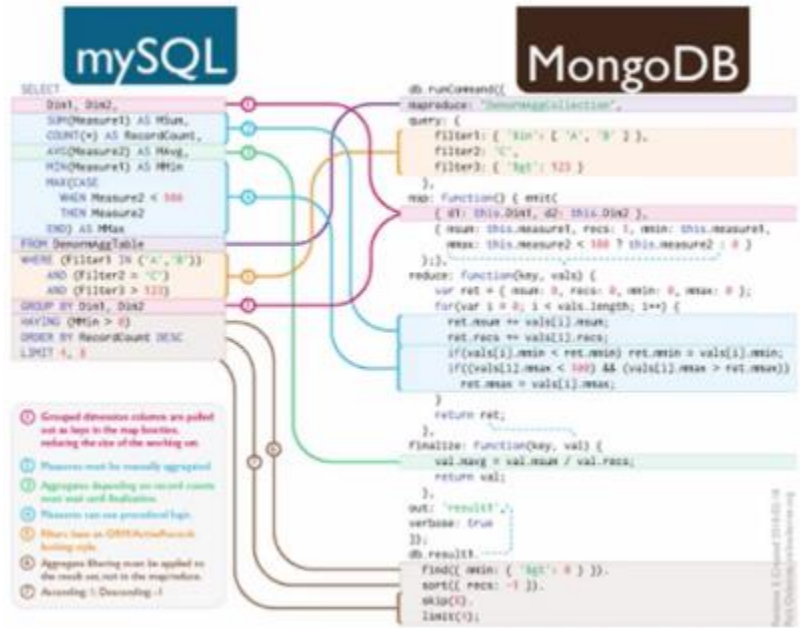
What You Wish Data Looked Like



What Data Actually Looks Like

```
@HWI-EAS121:4:100:1783:550#0/1
CGTTACGAGATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACGGATCTCGTATGCGGTCTGCTGCGTGACAAGACAGGGG
+HWI-EAS121:4:100:1783:550#0/1
aaaaa`b_aa`aa`YaX]aZ`aZM^Z]YRa]YSG[[ZREQLHESDHNDDHNMEEDDMPENITKFLFEEDDDHEJQMEDDD
@HWI-EAS121:4:100:1783:1611#0/1
GGGTGGGCATTTCCACTCGCAGTATGGGTTGCCGCACGACAGGCAGCGGTCAGCCTGCGCTTTGGCCTGGCCTTCGGAAA
+HWI-EAS121:4:100:1783:1611#0/1
a```^\__`_`___^a``a`^a_^__]a_]\]`a_______`_^^`]X]_]XTV_\]]NX_XVX]]_TTTTG[VTHPN]VFDZ
@HWI-EAS121:4:100:1783:322#0/1
CGTTTATGTTTTTGAATATGTCTTATCTTAACGGTTATATTTTAGATGTTGGTCTTATTCTAACGGTCATATATTTTCTA
+HWI-EAS121:4:100:1783:322#0/1
abaa`^aaaaabbbaabababbbbbb`bbbb_bbbbbbbb`bbbaV^_a``a``]``aT]a__V\]]_]^a`]a_abbaV__
@HWI-EAS121:4:100:1783:1394#0/1
GGGTCTTTATTGGTCTGGTGATCCCCCATATTCTCCGGTTGTGTGGTTTAACCGATCATCGCGCATTACTTCCCGGCTGC
+HWI-EAS121:4:100:1783:1394#0/1
```[aa\b^^[]aabbb][`a_abbb`a``bbbbbabaabaaaab_VZa_^___bab_X`[a\HV_[_]_[^_X\T_VQQ
@HWI-EAS121:4:100:1783:207#0/1
CCCTGGGAGATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACCGATCTCGTATGCCGTCTTCTGCTTGAAAAAAAAAACA
+HWI-EAS121:4:100:1783:207#0/1
abba`Xa\^\\`aa]ba__bba[a_O_a`aa`aa`a]^V]X_a^YS\R_\H_[]\ZTDUZZUSOPX]]POP\GS\WSHHD
@HWI-EAS121:4:100:1783:455#0/1
GGGTAATTCAGGGACAATGTAATGGCTGCACAAAAAAATACATCTTTCATGTTCCATTGCACCATTGACAAATACATATT
+HWI-EAS121:4:100:1783:455#0/1
abb_babbabaabbbbbbbbbbbbbbba\`b`\abbbabbbbabbbbbbaabbbbb`bb`ab_O_bab_Q_bbabaa_a
```
```

Where is Data



https://data.baltimorecity.gov/

Goal of the Course

Raw data -> Processing script -> tidy data -> data analysis -> data communication

Raw and Processed Data

Raw data

- The original source of the data
- Often hard to use for data analyses
- Data analysis *includes* processing
- Raw data may only need to be processed once

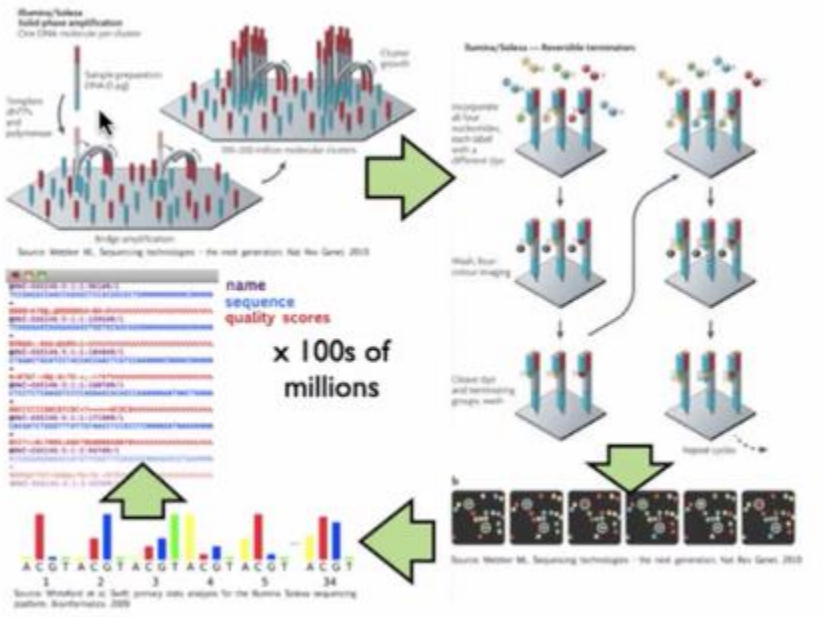
http://en.wikipedia.org/wiki/Raw_data

Processed data

- Data that is ready for analysis
- Processing can include merging, subsetting, transforming, etc.
- There may be standards for processing
- All steps should be recorded

Example of a processing pipeline

http://www.illumina.com.cn/support/sequencing/sequencing_instruments/hiseq_1000.asp



Components of Tidy Data

1. The raw data.
2. A tidy data set
3. A code book describing each variable and its values in the tidy data set.
4. An explicit and exact recipe you used to go from 1 -> 2,3.

Raw Data

- The strange binary file your measurement machine spits out
- The unformatted Excel file with 10 worksheets the company you contracted with sent you
- The complicated JSON data you got from scraping the Twitter API
- The hand-entered numbers you collected looking through a microscope

You know the raw data is in the right format if you

1. Ran no software on the data
2. Did not manipulate any of the numbers in the data
3. You did not remove any data from the data set
4. You did not summarize the data in any way

https://github.com/jtleek/datasharing

Tidy Data

1. Each variable you measure should be in one column
2. Each different observation of that variable should be in a different row
3. There should be one table for each "kind" of variable
4. If you have multiple tables, they should include a column in the table that allows them to be linked

Some other important tips

- Include a row at the top of each file with variable names.
- Make variable names human readable AgeAtDiagnosis instead of AgeDx
- In general data should be saved in one file per table.

Code Book

1. Information about the variables (including units!) in the data set not contained in the tidy data
2. Information about the summary choices you made
3. Information about the experimental study design you used

Some other important tips

- A common format for this document is a Word/text file.
- There should be a section called "Study design" that has a thorough description of how you collected the data.
- There must be a section called "Code book" that describes each variable and its units.

Instruction List

- Ideally a computer script (in R :-), but I suppose Python is ok too...)
- The input for the script is the raw data
- The output is the processed, tidy data
- There are no parameters to the script

In some cases it will not be possible to script every step. In that case you should provide instructions like:

1. Step 1 - take the raw file, run version 3.1.2 of summarize software with parameters a=1, b=2, c=3
2. Step 2 - run the software separately for each sample
3. Step 3 - take column three of outputfile.out for each sample and that is the corresponding row in the output data set

Downloading Files

Get/set Your Working Directory

- A basic component of working with data is knowing your working directory
- The two main commands are `getwd()` and `setwd()`.
- Be aware of relative versus absolute paths
  - **Relative** - `setwd("./data")`, `setwd("../")`
  - **Absolute** - `setwd("/Users/jtleek/data/")`
- Important difference in Windows `setwd("C:\\Users\\Andrew\\Downloads")`

Checking for and Creating Directories

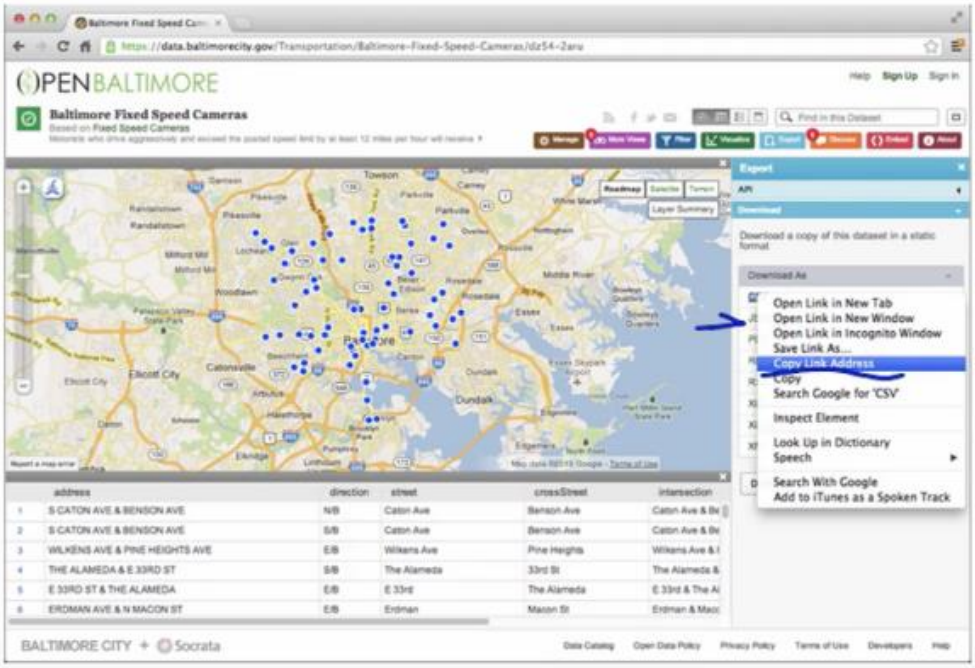
- `file.exists("directoryName")` will check to see if the directory exists
- `dir.create("directoryName")` will create a directory if it doesn't exist
- Here is an example checking for a "data" directory and creating it if it doesn't exist

```
if (!file.exists("data")) {
    dir.create("data")
}
```

Getting Data from the Internet – download.file()

- Downloads a file from the internet
- Even if you could do this by hand, helps with reproducibility
- Important parameters are *url*, *destfile*, *method*
- Useful for downloading tab-delimited, csv, and other files

Example – Baltimore Camera Data




```
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?accessType=DOWNLOAD"
download.file(fileUrl, destfile = "./data/cameras.csv", method = "curl")
list.files("./data")
```

```
## [1] "cameras.csv"
```

```
dateDownloaded <- date()
dateDownloaded
```

```
## [1] "Sun Jan 12 21:37:44 2014"
```

- If the url starts with *http* you can use download.file()
- If the url starts with *https* on Windows you may be ok
- If the url starts with *https* on Mac you may need to set *method="curl"*
- If the file is big, this might take a while
- Be sure to record when you downloaded.

Reading
Local Files – 1 Download Files

```
if (!file.exists("data")) {
    dir.create("data")
}
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?accessType=DOWNLOAD"
download.file(fileUrl, destfile = "cameras.csv", method = "curl")
dateDownloaded <- date()
```

Loading Flat Files – read.table()

- This is the main function for reading data into R
- Flexible and robust but requires more parameters
- Reads the data into RAM - big data can cause problems
- Important parameters *file*, *header*, *sep*, *row.names*, *nrows*
- Related: *read.csv()*, *read.csv2()*

Baltimore Example

```
cameraData <- read.table("./data/cameras.csv")
```

```
## Error: line 1 did not have 13 elements
```

```
head(cameraData)
```

```
## Error: object 'cameraData' not found
```

```
cameraData <- read.table("./data/cameras.csv", sep = ",", header = TRUE)
head(cameraData)
```

```
##                         address direction      street  crossStreet
## 1       S CATON AVE & BENSON AVE       N/B   Caton Ave   Benson Ave
## 2       S CATON AVE & BENSON AVE       S/B   Caton Ave   Benson Ave
## 3 WILKENS AVE & PINE HEIGHTS AVE       E/B Wilkens Ave Pine Heights
## 4        THE ALAMEDA & E 33RD ST       S/B The Alameda      33rd St
## 5        E 33RD ST & THE ALAMEDA       E/B      E 33rd  The Alameda
## 6        ERDMAN AVE & N MACON ST       E/B      Erdman     Macon St
##                intersection                       Location.1
## 1     Caton Ave & Benson Ave (39.2693779962, -76.6688185297)
## 2     Caton Ave & Benson Ave (39.2693157898, -76.6689698176)
## 3 Wilkens Ave & Pine Heights  (39.2720252302, -76.676960806)
## 4     The Alameda  & 33rd St (39.3285013141, -76.5953545714)
## 5      E 33rd  & The Alameda (39.3283410623, -76.5953594625)
## 6         Erdman  & Macon St (39.3068045671, -76.5593167803)
```

read.csv sets *sep=","* and *header=TRUE*

```
cameraData <- read.csv("./data/cameras.csv")
head(cameraData)
```

```
##                         address direction      street  crossStreet
## 1       S CATON AVE & BENSON AVE       N/B   Caton Ave   Benson Ave
## 2       S CATON AVE & BENSON AVE       S/B   Caton Ave   Benson Ave
## 3 WILKENS AVE & PINE HEIGHTS AVE       E/B Wilkens Ave Pine Heights
## 4        THE ALAMEDA & E 33RD ST       S/B The Alameda      33rd St
## 5        E 33RD ST & THE ALAMEDA       E/B      E 33rd  The Alameda
## 6        ERDMAN AVE & N MACON ST       E/B      Erdman     Macon St
##                intersection                       Location.1
## 1     Caton Ave & Benson Ave (39.2693779962, -76.6688185297)
## 2     Caton Ave & Benson Ave (39.2693157898, -76.6689698176)
## 3 Wilkens Ave & Pine Heights  (39.2720252302, -76.676960806)
## 4     The Alameda  & 33rd St (39.3285013141, -76.5953545714)
## 5      E 33rd  & The Alameda (39.3283410623, -76.5953594625)
## 6         Erdman  & Macon St (39.3068045671, -76.5593167803)
```

Important Parameters

- *quote* - you can tell R whether there are any quoted values quote="" means no quotes.
- *na.strings* - set the character that represents a missing value.
- *nrows* - how many rows to read of the file (e.g. nrows=10 reads 10 lines).
- *skip* - number of lines to skip before starting to read

In my experience, the biggest trouble with reading flat files are quotation marks ` or " placed in data values, setting quote="" often resolves these.

Excel Files

```
if(!file.exists("data")){dir.create("data")}
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-2aru/rows.xlsx?accessType=DOWNLOAD"
download.file(fileUrl,destfile="./data/cameras.xlsx",method="curl")
dateDownloaded <- date()
```

Read.xlsx(), read.xlsx2(), {xlsx package}

```
library(xlsx)
cameraData <- read.xlsx("./data/cameras.xlsx",sheetIndex=1,header=TRUE)
head(cameraData)
```

```
                       address direction      street  crossStreet                intersection
1       S CATON AVE & BENSON AVE       N/B   Caton Ave   Benson Ave     Caton Ave & Benson Ave
2       S CATON AVE & BENSON AVE       S/B   Caton Ave   Benson Ave     Caton Ave & Benson Ave
3 WILKENS AVE & PINE HEIGHTS AVE       E/B Wilkens Ave Pine Heights Wilkens Ave & Pine Heights
4        THE ALAMEDA & E 33RD ST       S/B The Alameda      33rd St     The Alameda  & 33rd St
5        E 33RD ST & THE ALAMEDA       E/B      E 33rd  The Alameda      E 33rd  & The Alameda
6
1 (39.2693779962, -76.6688185297)
2 (39.2693157898, -76.6689698176)
3  (39.2720252302, -76.676960806)
4 (39.3285013141, -76.5953545714)
5 (39.3283410623, -76.5953594625)
6 (39.3068045671, -76.5593167803)
```

Reading Specific Rows and Columns

```
colIndex <- 2:3
rowIndex <- 1:4
cameraDataSubset <- read.xlsx("./data/cameras.xlsx",sheetIndex=1,
                              colIndex=colIndex,rowIndex=rowIndex)
cameraDataSubset
```

```
  direction      street
1       N/B   Caton Ave
2       S/B   Caton Ave
3       E/B Wilkens Ave
```

Further Notes

- The *write.xlsx* function will write out an Excel file with similar arguments.
- *read.xlsx2* is much faster than *read.xlsx* but for reading subsets of rows may be slightly unstable.
- The XLConnect package has more options for writing and manipulating Excel files
- The XLConnect vignette is a good place to start for that package
- In general it is advised to store your data in either a database or in comma separated files (.csv) or tab separated files (.tab/.txt) as they are easier to distribute.

XML

- Extensible markup language
- Frequently used to store structured data
- Particularly widely used in internet applications
- Extracting XML is the basis for most web scraping
- Components
  - Markup - labels that give the text structure
  - Content - the actual text of the document

Tags, Elements, and Attributes

- Tags correspond to general labels
  - Start tags `<section>`
  - End tags `</section>`
  - Empty tags `<line-break />`
- Elements are specific examples of tags
  - `<Greeting> Hello, world </Greeting>`
- Attributes are components of the label
  - `<img src="jeff.jpg" alt="instructor"/>`
  - `<step number="3"> Connect A to B. </step>`

Read The File Into R

```
library(XML)
fileUrl <- "http://www.w3schools.com/xml/simple.xml"
doc <- xmlTreeParse(fileUrl,useInternal=TRUE)
rootNode <- xmlRoot(doc)
xmlName(rootNode)
```

```
[1] "breakfast_menu"
```

```
names(rootNode)
```

```
  food   food   food   food   food
"food" "food" "food" "food" "food"
```

Directly Access Parts of XML Document

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

```
<food>
  <name>Belgian Waffles</name>
  <price>$5.95</price>
  <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
  <calories>650</calories>
</food>
```

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

```
<name>Belgian Waffles</name>
```

Programmatically Extract Parts of the File

```
xmlSApply(rootNode,xmlValue)
```

```
                              "Belgian Waffles$5.95Two of our famous Belgian Waffles with plenty of r

                    "Strawberry Belgian Waffles$7.95Light Belgian waffles covered with strawberries an

"Berry-Berry Belgian Waffles$8.95Light Belgian waffles covered with an assortment of fresh berries an

                                              "French Toast$4.50Thick slices made from our homemade

                         "Homestyle Breakfast$6.95Two eggs, bacon or sausage, toast, and our ever-popu
```

XPath

- */node* Top level node
- *//node* Node at any level
- *node[@attr-name]* Node with an attribute name
- *node[@attr-name='bob']* Node with attribute name attr-name='bob'

Information from: http://www.stat.berkeley.edu/~statcur/Workshop2/Presentations/XML.pdf

Get Items On Menu and Prices

```
xpathSApply(rootNode,"//name",xmlValue)
```

```
[1] "Belgian Waffles"            "Strawberry Belgian Waffles"  "Berry-Berry Belgian Waffles"
[4] "French Toast"               "Homestyle Breakfast"
```

```
xpathSApply(rootNode,"//price",xmlValue)
```

```
[1] "$5.95" "$7.95" "$8.95" "$4.50" "$6.95"
```

Extract Content By Attributes

```
fileUrl <- "http://espn.go.com/nfl/team/_/name/bal/baltimore-ravens"
doc <- htmlTreeParse(fileUrl,useInternal=TRUE)
scores <- xpathSApply(doc,"//li[@class='score']",xmlValue)
teams <- xpathSApply(doc,"//li[@class='team-name']",xmlValue)
scores
```

```
 [1] "49-27"    "14-6"     "30-9"     "23-20"    "26-23"    "19-17"    "19-16"    "24-18"
 [9] "20-17 OT" "23-20 OT" "19-3"     "22-20"    "29-26"    "18-16"    "41-7"     "34-17"
```

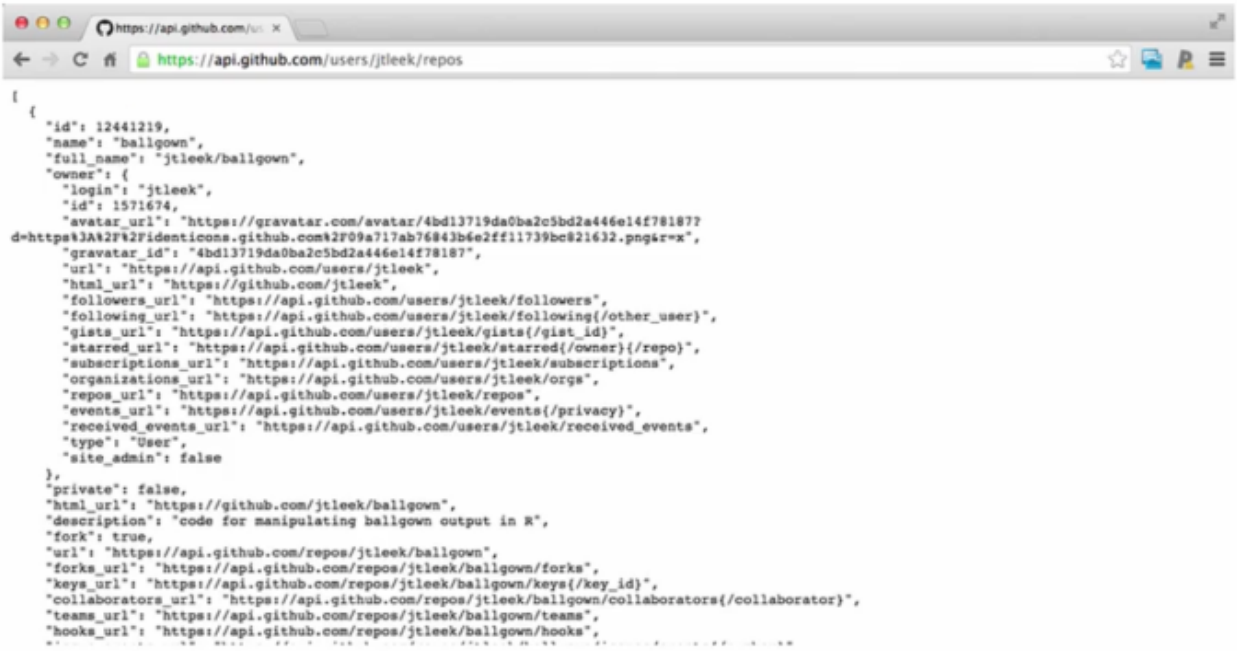
```
teams
```

```
 [1] "Denver"     "Cleveland"  "Houston"    "Buffalo"    "Miami"      "Green Bay"
 [7] "Pittsburgh" "Cleveland"  "Cincinnati" "Chicago"    "New York"   "Pittsburgh"
[13] "Minnesota"  "Detroit"    "New England" "Cincinnati"
```

- Official XML tutorials short, long
- An outstanding guide to the XML package

Reading JSON

- Javascript Object Notation
- Lightweight data storage
- Common format for data from application programming interfaces (APIs)
- Similar structure to XML but different syntax/format
- Data stored as
  - Numbers (double)
  - Strings (double quoted)
  - Boolean (*true* or *false*)
  - Array (ordered, comma separated enclosed in square brackets *[]*)
  - Object (unorderd, comma separated collection of key:value pairs in curley brackets *{}*)



Reading Data from JSON {jsonlite package}

```
library(jsonlite)
jsonData <- fromJSON("https://api.github.com/users/jtleek/repos")
names(jsonData)
```

```
 [1] "id"               "name"             "full_name"        "owner"
 [5] "private"          "html_url"         "description"      "fork"
 [9] "url"              "forks_url"        "keys_url"         "collaborators_url"
[13] "teams_url"        "hooks_url"        "issue_events_url" "events_url"
[17] "assignees_url"    "branches_url"     "tags_url"         "blobs_url"
[21] "git_tags_url"     "git_refs_url"     "trees_url"        "statuses_url"
[25] "languages_url"    "stargazers_url"   "contributors_url" "subscribers_url"
[29] "subscription_url" "commits_url"      "git_commits_url"  "comments_url"
[33] "issue_comment_url" "contents_url"    "compare_url"      "merges_url"
[37] "archive_url"      "downloads_url"    "issues_url"       "pulls_url"
[41] "milestones_url"   "notifications_url" "labels_url"     "releases_url"
[45] "created_at"       "updated_at"       "pushed_at"        "git_url"
[49] "ssh_url"          "clone_url"        "svn_url"          "homepage"
[53] "size"             "stargazers_count" "watchers_count"   "language"
[57] "has_issues"       "has_downloads"    "has_wiki"         "forks_count"
```

Nested Objects in JSON

```
names(jsonData$owner)
```

```
 [1] "login"      "id"         "avatar_url"         "gravatar_id"
 [5] "url"        "html_url"   "followers_url"      "following_url"
 [9] "gists_url"  "starred_url" "subscriptions_url"  "organizations_url"
[13] "repos_url"  "events_url" "received_events_url" "type"
[17] "site_admin"
```

```
jsonData$owner$login
```

```
 [1] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
[11] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
[21] "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek" "jtleek"
```

Writing Data Frames to JSON

```
myjson <- toJSON(iris, pretty=TRUE)
cat(myjson)
```


```
[
    {
        "Sepal.Length" : 5.1,
        "Sepal.Width" : 3.5,
        "Petal.Length" : 1.4,
        "Petal.Width" : 0.2,
        "Species" : "setosa"
    },
    {
        "Sepal.Length" : 4.9,
        "Sepal.Width" : 3,
        "Petal.Length" : 1.4,
        "Petal.Width" : 0.2,
        "Species" : "setosa"
```


Convert Back to JSON

```
iris2 <- fromJSON(myjson)
head(iris2)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

- http://www.json.org/
- A good tutorial on jsonlite - http://www.r-bloggers.com/new-package-jsonlite-a-smarter-json-encoderdecoder/
- jsonlite vignette

Using Data.table Package

- Inherets from data.frame
  - All functions that accept data.frame work on data.table
- Written in C so it is much faster
- Much, much faster at subsetting, group, and updating

Creating Data Tables Just Like Data Frames

```
library(data.table)
DF = data.frame(x=rnorm(9),y=rep(c("a","b","c"),each=3),z=rnorm(9))
head(DF,3)
```

```
       x y       z
1 0.4159 a -0.05855
2 0.8433 a  0.13732
3 1.0585 a  2.16448
```

```
DT = data.table(x=rnorm(9),y=rep(c("a","b","c"),each=3),z=rnorm(9))
head(DT,3)
```

```
          x y      z
1: -0.27721 a 0.2530
2:  1.00158 a 1.5093
3: -0.03382 a 0.4844
```

See All The Data Tables in Memory

```
tables()
```

```
    NAME NROW MB COLS  KEY
[1,] DT     9 1  x,y,z
Total: 1MB
```

Subsetting Rows

```
DT[2,]
```

```
      x y     z
1: 1.002 a 1.509
```

```
DT[DT$y=="a",]
```

```
          x y      z
1: -0.27721 a 0.2530
2:  1.00158 a 1.5093
3: -0.03382 a 0.4844
```

```
DT[c(2,3)]
```

```
          x y      z
1:  1.00158 a 1.5093
2: -0.03382 a 0.4844
```

Subsetting Columns in data.table

```
DT[,c(2,3)]
```

```
[1] 2 3
```

- The subsetting function is modified for data.table
- The argument you pass after the comma is called an "expression"
- In R an expression is a collection of statements enclosed in curley brackets

```
{
  x = 1
  y = 2
}
k = {print(10); 5}
```

```
[1] 10
```

```
print(k)
```

```
[1] 5
```

Calculating Values for Variables with Expressions

```
DT[,list(mean(x),sum(z))]
```

```
       V1     V2
1: 0.05637 0.5815
```

```
DT[,table(y)]
```

```
y
a b c
3 3 3
```

Adding New Columns

```
DT[,w:=z^2]
```

```
         x y       z        w
1: -0.27721 a  0.25300 0.064009
2:  1.00158 a  1.50933 2.278091
3: -0.03382 a  0.48437 0.234619
4: -0.70493 b -1.22755 1.506885
5: -1.36402 b -0.64624 0.417631
6: -0.26224 b -0.51427 0.264475
7: -0.10929 c  1.21445 1.474901
8:  1.40234 c  0.07493 0.005614
9:  0.85494 c -0.56652 0.320948
```

```
DT2 <- DT
DT[, y:= 2]
```

```
         x y       z        w
1: -0.27721 2  0.25300 0.064009
2:  1.00158 2  1.50933 2.278091
3: -0.03382 2  0.48437 0.234619
4: -0.70493 2 -1.22755 1.506885
5: -1.36402 2 -0.64624 0.417631
6: -0.26224 2 -0.51427 0.264475
7: -0.10929 2  1.21445 1.474901
8:  1.40234 2  0.07493 0.005614
9:  0.85494 2 -0.56652 0.320948
```

Careful

```
head(DT,n=3)
```

```
          x y      z       w
1: -0.27721 2 0.2530 0.06401
2:  1.00158 2 1.5093 2.27809
3: -0.03382 2 0.4844 0.23462
```

```
head(DT2,n=3)
```

```
          x y      z       w
1: -0.27721 2 0.2530 0.06401
2:  1.00158 2 1.5093 2.27809
3: -0.03382 2 0.4844 0.23462
```

Multiple Operations

```
DT[,m:= {tmp <- (x+z); log2(tmp+5)}]
```

```
          x y        z        w     m
1: -0.27721 2  0.25300 0.064009 2.315
2:  1.00158 2  1.50933 2.278091 2.909
3: -0.03382 2  0.48437 0.234619 2.446
4: -0.70493 2 -1.22755 1.506885 1.617
5: -1.36402 2 -0.64624 0.417631 1.580
6: -0.26224 2 -0.51427 0.264475 2.078
7: -0.10929 2  1.21445 1.474901 2.610
8:  1.40234 2  0.07493 0.005614 2.695
9:  0.85494 2 -0.56652 0.320948 2.403
```

Plyr Like Operations

```
DT[,b:= mean(x+w),by=a]
```

```
          x y       z        w     m     a      b
1: -0.27721 2  0.25300 0.064009 2.315 FALSE 0.2018
2:  1.00158 2  1.50933 2.278091 2.909  TRUE 1.9545
3: -0.03382 2  0.48437 0.234619 2.446 FALSE 0.2018
4: -0.70493 2 -1.22755 1.506885 1.617 FALSE 0.2018
5: -1.36402 2 -0.64624 0.417631 1.580 FALSE 0.2018
6: -0.26224 2 -0.51427 0.264475 2.078 FALSE 0.2018
7: -0.10929 2  1.21445 1.474901 2.610 FALSE 0.2018
8:  1.40234 2  0.07493 0.005614 2.695  TRUE 1.9545
9:  0.85494 2 -0.56652 0.320948 2.403  TRUE 1.9545
```

Special Variables

.N An integer, length 1, containing the numbe r

```
set.seed(123);
DT <- data.table(x=sample(letters[1:3], 1E5, TRUE))
DT[, .N, by=x]
```

```
   x     N
1: a 33387
2: c 33201
3: b 33412
```

Keys

```
DT <- data.table(x=rep(c("a","b","c"),each=100), y=rnorm(300))
setkey(DT, x)
DT['a']
```

```
    x       y
  1: a  0.25959
  2: a  0.91751
  3: a -0.72232
  4: a -0.80828
  5: a -0.14135
  6: a  2.25701
  7: a -2.37955
  8: a -0.45425
  9: a -0.06007
 10: a  0.86090
 11: a -1.78466
 12: a -0.13074
```

Joins

```
DT1 <- data.table(x=c('a', 'a', 'b', 'dt1'), y=1:4)
DT2 <- data.table(x=c('a', 'b', 'dt2'), z=5:7)
setkey(DT1, x); setkey(DT2, x)
merge(DT1, DT2)
```

```
   x y z
1: a 1 5
2: a 2 5
3: b 3 6
```

Fast Reading

```
big_df <- data.frame(x=rnorm(1E6), y=rnorm(1E6))
file <- tempfile()
write.table(big_df, file=file, row.names=FALSE, col.names=TRUE, sep="\t", quote=FALSE)
system.time(fread(file))
```

```
   user  system elapsed
  0.312   0.015   0.326
```

```
system.time(read.table(file, header=TRUE, sep="\t"))
```

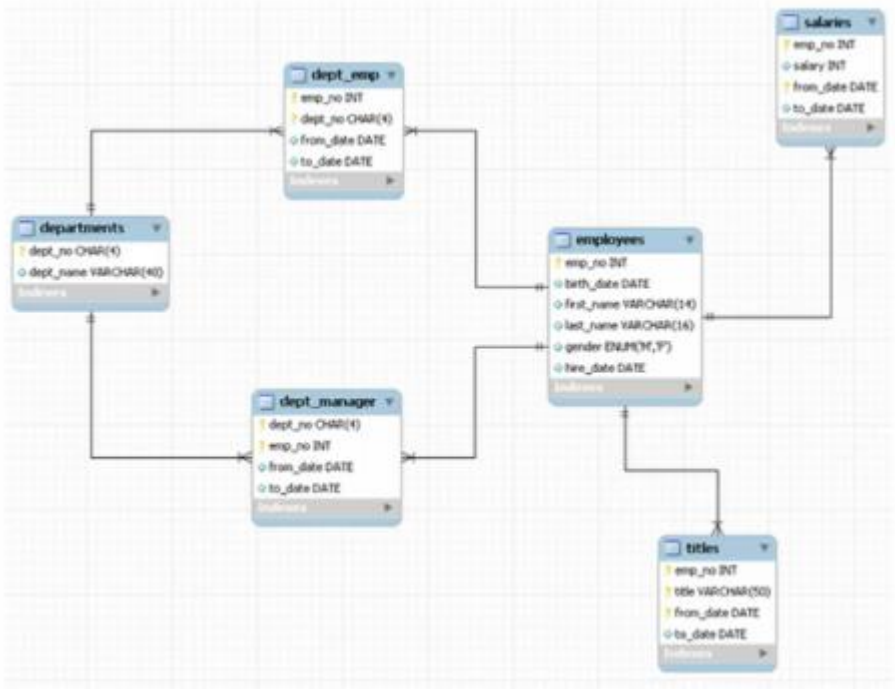
```
   user  system elapsed
  5.702   0.048   5.755
```

Summary and Further Reading

- The latest development version contains new functions like `melt` and `dcast` for data.tables
  - https://r-forge.r-project.org/scm/viewvc.php/pkg/NEWS?view=markup&root=datatable
- Here is a list of differences between data.table and data.frame
  - http://stackoverflow.com/questions/13618488/what-you-can-do-with-data-frame-that-you-cant-in-data-table
- Notes based on Raphael Gottardo's notes https://github.com/raphg/Biostat-578/blob/master/Advanced_data_manipulation.Rpres, who got them from Kevin Ushey.

Reading
MySQL

- Free and widely used open source database software
- Widely used in internet based applications
- Data are structured in
  - Databases
  - Tables within databases
  - Fields within tables
- Each row is called a record



Connecting and Listing Databases

```
ucscDb <- dbConnect(MySQL(),user="genome",
                    host="genome-mysql.cse.ucsc.edu")
result <- dbGetQuery(ucscDb,"show databases;"); dbDisconnect(ucscDb);
```

```
[1] TRUE
```

```
result
```

```
             Database
1   information_schema
2              ailMel1
3              allMis1
4              anoCar1
5              anoCar2
6              anoGam1
```

Connecting to hg19 and listing tables

```
hg19 <- dbConnect(MySQL(),user="genome", db="hg19",
                    host="genome-mysql.cse.ucsc.edu")
allTables <- dbListTables(hg19)
length(allTables)
```

```
[1] 10949
```

```
allTables[1:5]
```

```
[1] "HInv"         "HInvGeneMrna" "acembly"      "acemblyClass" "acemblyPep"
```

Get Dimensions of a Specific Table

```
dbListFields(hg19,"affyU133Plus2")
```

```
 [1] "bin"         "matches"     "misMatches"  "repMatches"  "nCount"      "qNumInsert"
 [7] "qBaseInsert" "tNumInsert"  "tBaseInsert" "strand"      "qName"       "qSize"
[13] "qStart"      "qEnd"        "tName"       "tSize"       "tStart"      "tEnd"
[19] "blockCount"  "blockSizes"  "qStarts"     "tStarts"
```

```
dbGetQuery(hg19, "select count(*) from affyU133Plus2")
```

```
  count(*)
1    58463
```

Read from the Table

```
affyData <- dbReadTable(hg19, "affyU133Plus2")
head(affyData)
```

```
  bin matches misMatches repMatches nCount qNumInsert qBaseInsert tNumInsert tBaseInsert strand
1 585     530          4          0     23          3          41          3         898      -
2 585    3355         17          0    109          9          67          9       11621      -
3 585    4156         14          0     83         16          18          2          93      -
4 585    4667          9          0     68         21          42          3        5743      -
5 585    5180         14          0    167         10          38          1          29      -
6 585     468          5          0     14          0           0          0           0      -
         qName qSize qStart qEnd tName     tSize tStart  tEnd blockCount
1  225995_x_at   637      5  603  chr1 249250621  14361 15816          5
2  225035_x_at  3635      0 3548  chr1 249250621  14381 29483         17
3  226340_x_at  4318      3 4274  chr1 249250621  14399 18745         18
4 1557034_s_at  4834     48 4834  chr1 249250621  14406 24893         23
5    231811_at  5399      0 5399  chr1 249250621  19688 25078         11
6    236841_at   487      0  487  chr1 249250621  27542 28029          1
```

Select a Specific Subset

```
query <- dbSendQuery(hg19, "select * from affyU133Plus2 where misMatches between 1 and 3")
affyMis <- fetch(query); quantile(affyMis$misMatches)
```

```
  0%  25%  50%  75% 100%
   1    1    2    2    3
```

```
affyMisSmall <- fetch(query,n=10); dbClearResult(query);
```

```
[1] TRUE
```

```
dim(affyMisSmall)
```

```
[1] 10 22
```


Don't Forget to Close Connection

```
dbDisconnect(hg19)
```

```
[1] TRUE
```

Further Resources

- RMySQL vignette http://cran.r-project.org/web/packages/RMySQL/RMySQL.pdf
- List of commands http://www.pantz.org/software/mysql/mysqlcommands.html
  - **Do not, do not, delete, add or join things from ensembl. Only select.**
  - In general be careful with mysql commands
- A nice blog post summarizing some other commands http://www.r-bloggers.com/mysql-and-r/

Reading from HDF5

- Used for storing large data sets
- Supports storing a range of data types
- Heirarchical data format
- *groups* containing zero or more data sets and metadata
  - Have a *group header* with group name and list of attributes
  - Have a *group symbol table* with a list of objects in group
- *datasets* multidmensional array of data elements with metadata
  - Have a *header* with name, datatype, dataspace, and storage layout
  - Have a *data array* with the data

```
source("http://bioconductor.org/biocLite.R")
biocLite("rhdf5")
```

```
library(rhdf5)
created = h5createFile("example.h5")
created
```

```
[1] TRUE
```

- This will install packages from Bioconductor http://bioconductor.org/, primarily used for genomics but also has good "big data" packages
- Can be used to interface with hdf5 data sets.
- This lecture is modeled very closely on the rhdf5 tutorial that can be found here http://www.bioconductor.org/packages/release/bioc/vignettes/rhdf5/inst/doc/rhdf5.pdf

Create Groups

```
created = h5createGroup("example.h5","foo")
created = h5createGroup("example.h5","baa")
created = h5createGroup("example.h5","foo/foobaa")
h5ls("example.h5")
```

```
  group   name     otype dclass dim
0     /    baa H5I_GROUP
1     /    foo H5I_GROUP
2  /foo foobaa H5I_GROUP
```

Write to Groups

```
A = matrix(1:10,nr=5,nc=2)
h5write(A, "example.h5","foo/A")
B = array(seq(0.1,2.0,by=0.1),dim=c(5,2,2))
attr(B, "scale") <- "liter"
h5write(B, "example.h5","foo/foobaa/B")
h5ls("example.h5")
```

```
        group   name       otype  dclass       dim
0           /    baa   H5I_GROUP
1           /    foo   H5I_GROUP
2        /foo      A H5I_DATASET INTEGER     5 x 2
3        /foo foobaa   H5I_GROUP
4 /foo/foobaa      B H5I_DATASET   FLOAT 5 x 2 x 2
```

Write a Data Set

```
df = data.frame(1L:5L,seq(0,1,length.out=5),
  c("ab","cde","fghi","a","s"), stringsAsFactors=FALSE)
h5write(df, "example.h5","df")
h5ls("example.h5")
```

```
       group   name       otype   dclass       dim
0          /    baa   H5I_GROUP
1          /     df H5I_DATASET COMPOUND         5
2          /    foo   H5I_GROUP
3       /foo      A H5I_DATASET  INTEGER     5 x 2
4       /foo foobaa   H5I_GROUP
5 /foo/foobaa      B H5I_DATASET    FLOAT 5 x 2 x 2
```

Reading Data

```
readA = h5read("example.h5","foo/A")
readB = h5read("example.h5","foo/foobaa/B")
readdf= h5read("example.h5","df")
readA
```

```
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Writing and Reading Chunks

```
h5write(c(12,13,14),"example.h5","foo/A",index=list(1:3,1))
h5read("example.h5","foo/A")
```

```
     [,1] [,2]
[1,]   12    6
[2,]   13    7
[3,]   14    8
[4,]    4    9
[5,]    5   10
```

- hdf5 can be used to optimize reading/writing from disc in R
- The rhdf5 tutorial:
  - http://www.bioconductor.org/packages/release/bioc/vignettes/rhdf5/inst/doc/rhdf5.pdf
- The HDF group has informaton on HDF5 in general http://www.hdfgroup.org/HDF5/

Reading Data from the Web

Web scraping

Webscraping: Programatically extracting data from the HTML code of websites.

- It can be a great way to get data How Netflix reverse engineered Hollywood
- Many websites have information you may want to programaticaly read
- In some cases this is against the terms of service for the website
- Attempting to read too many pages too quickly can get your IP address blocked

readLines()

```
con = url("http://scholar.google.com/citations?user=HI-I6C0AAAAJ&hl=en")
htmlCode = readLines(con)
close(con)
htmlCode
```

```
[1] "<!DOCTYPE html><html><head><title>Jeff Leek - Google Scholar Citations</title><meta name=\"r
```

Parsing with XML

```
library(XML)
url <- "http://scholar.google.com/citations?user=HI-I6C0AAAAJ&hl=en"
html <- htmlTreeParse(url, useInternalNodes=T)

xpathSApply(html, "//title", xmlValue)
```

```
[1] "Jeff Leek - Google Scholar Citations"
```

```
xpathSApply(html, "//td[@id='col-citedby']", xmlValue)
```

```
 [1] "Cited by" "397"      "259"      "237"      "172"      "138"      "125"      "122"
 [9] "109"      "101"      "34"       "26"       "26"       "24"       "19"       "13"
[17] "12"       "10"       "10"       "7"        "6"
```

GET from the httr package

```
library(httr); html2 = GET(url)
content2 = content(html2,as="text")
parsedHtml = htmlParse(content2,asText=TRUE)
xpathSApply(parsedHtml, "//title", xmlValue)
```

```
[1] "Jeff Leek - Google Scholar Citations"
```

Accessing Websites with Passwords

```
pg2 = GET("http://httpbin.org/basic-auth/user/passwd",
    authenticate("user","passwd"))
pg2
```

```
Response [http://httpbin.org/basic-auth/user/passwd]
  Status: 200
  Content-type: application/json
{
  "authenticated": true,
  "user": "user"
}
```

```
names(pg2)
```

```
[1] "url"         "handle"      "status_code" "headers"     "cookies"     "content"
```

Using handles

```
google = handle("http://google.com")
pg1 = GET(handle=google,path="/")
pg2 = GET(handle=google,path="search")
```

http://cran.r-project.org/web/packages/httr/httr.pdf

- R Bloggers has a number of examples of web scraping http://www.r-bloggers.com/?s=Web+Scraping
- The httr help file has useful examples http://cran.r-project.org/web/packages/httr/httr.pdf

Reading from APIs

Accessing Twitter from R

```
myapp = oauth_app("twitter",
                  key="yourConsumerKeyHere",secret="yourConsumerSecretHere")
sig = sign_oauth1.0(myapp,
                    token = "yourTokenHere",
                     token_secret = "yourTokenSecretHere")
homeTL = GET("https://api.twitter.com/1.1/statuses/home_timeline.json", sig)
```

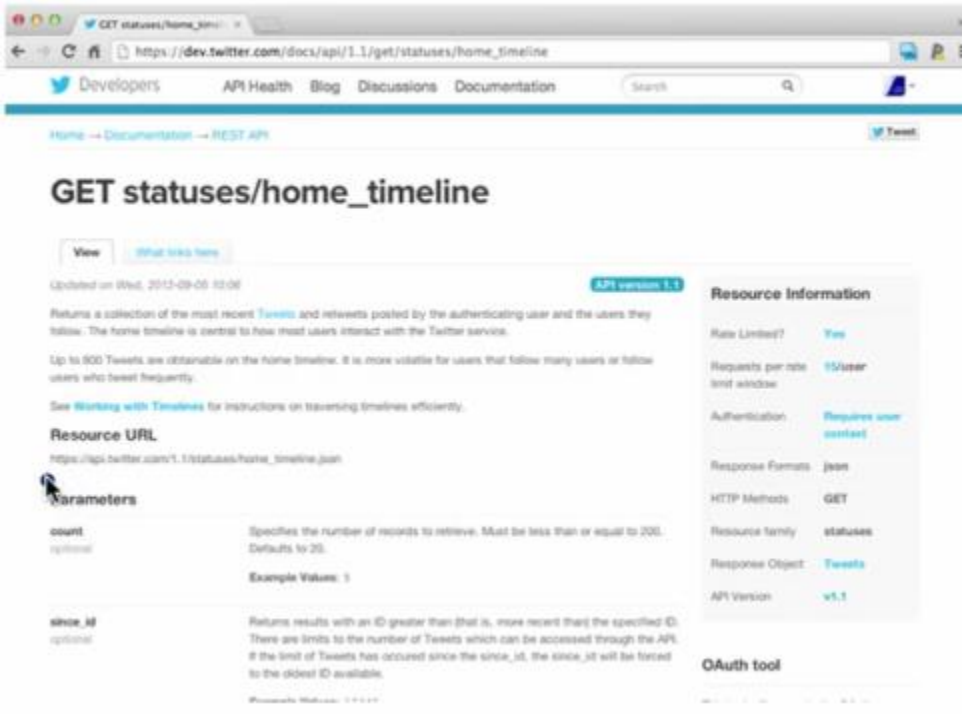
Converting the JSON Object

```
json1 = content(homeTL)
json2 = jsonlite::fromJSON(toJSON(json1))
json2[1,1:4]
```

```
                     created_at           id             id_str
1 Mon Jan 13 05:18:04 +0000 2014 4.225984e+17 422598398940684288

1 Now that P. Norvig's regex golf IPython notebook hit Slashdot, let's see if our traffic spike
```

Identify Which URL to Use



In General, Look at the Documentation

- httr allows `GET`, `POST`, `PUT`, `DELETE` requests if you are authorized
- You can authenticate with a user name or a password
- Most modern APIs use something like oauth
- httr works well with Facebook, Google, Twitter, Githb, etc.

Reading from Other Sources

- Roger has a nice video on how there are R packages for most things that you will want to access.
- Here I'm going to briefly review a few useful packages
- In general the best way to find out if the R package exists is to Google "data storage mechanism R package"
  - For example: "MySQL R package"

Interacting More Directly with Files

- file - open a connection to a text file
- url - open a connection to a url
- gzfile - open a connection to a .gz file
- bzfile - open a connection to a .bz2 file
- *?connections* for more information
- Remember to close connections

Foreign Package

- Loads data from Minitab, S, SAS, SPSS, Stata,Systat
- Basic functions *read.foo*
  - read.arff (Weka)
  - read.dta (Stata)
  - read.mtp (Minitab)
  - read.octave (Octave)
  - read.spss (SPSS)
  - read.xport (SAS)
- See the help page for more details http://cran.r-project.org/web/packages/foreign/foreign.pdf

Other Database Packages

- RPostresSQL provides a DBI-compliant database connection from R. Tutorial- https://code.google.com/p/rpostgresql/, help file-http://cran.r-project.org/web/packages/RPostgreSQL/RPostgreSQL.pdf
- RODBC provides interfaces to multiple databases including PostgreQL, MySQL, Microsoft Access and SQLite. Tutorial - http://cran.r-project.org/web/packages/RODBC/vignettes/RODBC.pdf, help file - http://cran.r-project.org/web/packages/RODBC/RODBC.pdf
- RMongo http://cran.r-project.org/web/packages/RMongo/RMongo.pdf (example of Rmongo http://www.r-bloggers.com/r-and-mongodb/) and rmongodb provide interfaces to MongoDb.

Reading Images

- jpeg - http://cran.r-project.org/web/packages/jpeg/index.html
- readbitmap - http://cran.r-project.org/web/packages/readbitmap/index.html
- png - http://cran.r-project.org/web/packages/png/index.html
- EBImage (Bioconductor) - http://www.bioconductor.org/packages/2.13/bioc/html/EBImage.html

Reading GIS Data

- rdgal - http://cran.r-project.org/web/packages/rgdal/index.html
- rgeos - http://cran.r-project.org/web/packages/rgeos/index.html
- raster - http://cran.r-project.org/web/packages/raster/index.html

Reading Music Data

- tuneR - http://cran.r-project.org/web/packages/tuneR/
- seewave - http://rug.mnhn.fr/seewave/

Subsetting and Sorting

```
set.seed(13435)
X <- data.frame("var1"=sample(1:5),"var2"=sample(6:10),"var3"=sample(11:15))
X <- X[sample(1:5),]; X$var2[c(1,3)] = NA
X
```

```
  var1 var2 var3
1    2   NA   15
4    1   10   11
2    3   NA   12
3    5    6   14
5    4    9   13
```

```
X[,1]
```

```
[1] 2 1 3 5 4
```

```
X[,"var1"]
```

```
[1] 2 1 3 5 4
```

```
X[1:2,"var2"]
```

```
[1] NA 10
```

Logicals ands and ors

```
X[(X$var1 <= 3 & X$var3 > 11),]
```

```
  var1 var2 var3
1    2   NA   15
2    3   NA   12
```

```
X[(X$var1 <= 3 | X$var3 > 15),]
```

```
  var1 var2 var3
1    2   NA   15
4    1   10   11
2    3   NA   12
```

Dealing with Missing Values

```
X[which(X$var2 > 8),]
```

```
  var1 var2 var3
4    1   10   11
5    4    9   13
```

Sorting

```
sort(X$var1)
```

```
[1] 1 2 3 4 5
```

```
sort(X$var1,decreasing=TRUE)
```

```
[1] 5 4 3 2 1
```

```
sort(X$var2,na.last=TRUE)
```

```
[1]  6  9 10 NA NA
```

Ordering

```
X[order(X$var1),]
```

```
  var1 var2 var3
4    1   10   11
1    2   NA   15
2    3   NA   12
5    4    9   13
3    5    6   14
```

Ordering with plyr

```
library(plyr)
arrange(X,var1)
```

```
  var1 var2 var3
1    1   10   11
2    2   NA   15
3    3   NA   12
4    4    9   13
5    5    6   14
```

```
arrange(X,desc(var1))
```

```
  var1 var2 var3
1    5    6   14
2    4    9   13
```

Adding rows and columns

```
X$var4 <- rnorm(5)
X
```

```
  var1 var2 var3     var4
1    2   NA   15  0.18760
4    1   10   11  1.78698
2    3   NA   12  0.49669
3    5    6   14  0.06318
5    4    9   13 -0.53613
```

```
Y <- cbind(X,rnorm(5))
Y
```

```
  var1 var2 var3     var4 rnorm(5)
1    2   NA   15  0.18760  0.62578
4    1   10   11  1.78698 -2.45084
2    3   NA   12  0.49669  0.08909
3    5    6   14  0.06318  0.47839
5    4    9   13 -0.53613  1.00053
```

```
Y <- cbind(X,rnorm(5))
Y
```

```
  var1 var2 var3     var4 rnorm(5)
1    2   NA   15  0.18760  0.62578
4    1   10   11  1.78698 -2.45084
2    3   NA   12  0.49669  0.08909
3    5    6   14  0.06318  0.47839
5    4    9   13 -0.53613  1.00053
```

- R programming in the Data Science Track
- Andrew Jaffe's lecture notes http://www.biostat.jhsph.edu/~ajaffe/lec_winterR/Lecture%202.pdf

Summarizing Data

Get from Web

```
if(!file.exists("./data")){dir.create("./data")}
fileUrl <- "https://data.baltimorecity.gov/api/views/k5ry-ef3g/rows.csv?accessType=DOWNLOAD"
download.file(fileUrl,destfile="./data/restaurants.csv",method="curl")
restData <- read.csv("./data/restaurants.csv")
```

Look at a Bit of the Data

```
head(restData,n=3)
```

```
   name zipCode neighborhood councilDistrict policeDistrict                      Location.1
1   410   21206    Frankford               2   NORTHEASTERN 4509 BELAIR ROAD\nBaltimore, MD\n
2  1919   21231  Fells Point               1   SOUTHEASTERN    1919 FLEET ST\nBaltimore, MD\n
3 SAUTE   21224       Canton               1   SOUTHEASTERN   2844 HUDSON ST\nBaltimore, MD\n
```

```
tail(restData,n=3)
```

```
                name zipCode  neighborhood councilDistrict policeDistrict
1325 ZINK'S CAF\u0090   21213 Belair-Edison              13   NORTHEASTERN
1326     ZISSIMOS BAR   21211       Hampden               7       NORTHERN
1327           ZORBAS   21224     Greektown               2   SOUTHEASTERN
                           Location.1
1325 3300 LAWNVIEW AVE\nBaltimore, MD\n
```

Make Summary

```
summary(restData)
```

```
                        name         zipCode            neighborhood councilDistrict
MCDONALD'S                :   8   Min.   :-21226   Downtown    :128   Min.   : 1.00
POPEYES FAMOUS FRIED CHICKEN:   7   1st Qu.: 21202   Fells Point : 91   1st Qu.: 2.00
SUBWAY                    :   6   Median : 21218   Inner Harbor: 89   Median : 9.00
KENTUCKY FRIED CHICKEN    :   5   Mean   : 21185   Canton      : 81   Mean   : 7.19
BURGER KING               :   4   3rd Qu.: 21226   Federal Hill: 42   3rd Qu.:11.00
DUNKIN DONUTS             :   4   Max.   : 21287   Mount Vernon: 33   Max.   :14.00
(Other)                   :1293                    (Other)     :863
     policeDistrict                      Location.1
SOUTHEASTERN:385    1101 RUSSELL ST\nBaltimore, MD\n:   9
CENTRAL     :288    201 PRATT ST\nBaltimore, MD\n   :   8
SOUTHERN    :213    2400 BOSTON ST\nBaltimore, MD\n :   8
NORTHERN    :157    300 LIGHT ST\nBaltimore, MD\n   :   5
NORTHEASTERN: 72    300 CHARLES ST\nBaltimore, MD\n :   4
EASTERN     : 67    301 LIGHT ST\nBaltimore, MD\n   :   4
```

Mpre in Depth Information

```
str(restData)
```

```
'data.frame':   1327 obs. of  6 variables:
 $ name           : Factor w/ 1277 levels "#1 CHINESE KITCHEN",..: 9 3 992 1 2 4 5 6 7 8 ...
 $ zipCode        : int  21206 21231 21224 21211 21223 21218 21205 21211 21205 21231 ...
 $ neighborhood   : Factor w/ 173 levels "Abell","Arlington",..: 53 52 18 66 104 33 98 133 98 157
 $ councilDistrict: int  2 1 1 14 9 14 13 7 13 1 ...
 $ policeDistrict : Factor w/ 9 levels "CENTRAL","EASTERN",..: 3 6 6 4 8 3 6 4 6 6 ...
 $ Location.1     : Factor w/ 1210 levels "1 BIDDLE ST\nBaltimore, MD\n",..: 835 334 554 755 492
```

Quantiles of quantitative variables

```
quantile(restData$councilDistrict,na.rm=TRUE)
```

```
  0%  25%  50%  75% 100%
   1    2    9   11   14
```

```
quantile(restData$councilDistrict,probs=c(0.5,0.75,0.9))
```

```
50% 75% 90%
  9  11  12
```

Make Table

```
table(restData$zipCode,useNA="ifany")
```

```
-21226  21201  21202  21205  21206  21207  21208  21209  21210  21211  21212  21213  21214  21215
     1    136    201     27     30      4      1      8     23     41     28     31     17     54
 21216  21217  21218  21220  21222  21223  21224  21225  21226  21227  21229  21230  21231  21234
    10     32     69      1      7     56    199     19     18      4     13    156    127      7
 21237  21239  21251  21287
     1      3      2      1
```

```
table(restData$councilDistrict,restData$zipCode)
```

Check for Missing Values

```
sum(is.na(restData$councilDistrict))
```

```
[1] 0
```

```
any(is.na(restData$councilDistrict))
```

```
[1] FALSE
```

```
all(restData$zipCode > 0)
```

```
[1] FALSE
```

Row and Column Sums

```
colSums(is.na(restData))
```

```
          name        zipCode   neighborhood councilDistrict  policeDistrict     Location.1
             0              0              0              0              0              0
```

```
all(colSums(is.na(restData))==0)
```

```
[1] TRUE
```

Values with Specific Characteristics

```
table(restData$zipCode %in% c("21212"))
```

```
FALSE  TRUE
 1299    28
```

```
table(restData$zipCode %in% c("21212","21213"))
```

```
FALSE  TRUE
 1268    59
```

```
restData[restData$zipCode %in% c("21212","21213"),]
```

	name	zipCode	neighborhood	councilDistrict
29	BAY ATLANTIC CLUB	21212	Downtown	11
39	BERMUDA BAR	21213	Broadway East	12
92	ATWATER'S	21212	Chinquapin Park-Belvedere	4
111	BALTIMORE ESTONIAN SOCIETY	21213	South Clifton Park	12
187	CAFE ZEN	21212	Rosebank	4
220	CERIELLO FINE FOODS	21212	Chinquapin Park-Belvedere	4
266	CLIFTON PARK GOLF COURSE SNACK BAR	21213	Darley Park	14
276	CLUB HOUSE BAR & GRILL	21213	Orangeville Industrial Area	13
289	CLUBHOUSE BAR & GRILL	21213	Orangeville Industrial Area	13
291	COCKY LOU'S	21213	Broadway East	12
362	DREAM TAVERN, CARRIBEAN U.S.A.	21213	Broadway East	13
373	DUNKIN DONUTS	21212	Homeland	4
383	EASTSIDE SPORTS SOCIAL CLUB	21213	Broadway East	13
417	FIELDS OLD TRAIL	21212	Mid-Govans	4

Cross Tabs

```
data(UCBAdmissions)
DF = as.data.frame(UCBAdmissions)
summary(DF)
```

```
     Admit       Gender   Dept       Freq
 Admitted:12   Male  :12   A:4   Min.   :  8
 Rejected:12   Female:12   B:4   1st Qu.: 80
                           C:4   Median :170
                           D:4   Mean   :189
                           E:4   3rd Qu.:302
                           F:4   Max.   :512
```

```
xt <- xtabs(Freq ~ Gender + Admit,data=DF)
xt
```

```
        Admit
Gender   Admitted Rejected
  Male       1198     1493
  Female      557     1278
```

Flat Tables

```
warpbreaks$replicate <- rep(1:9, len = 54)
xt = xtabs(breaks ~.,data=warpbreaks)
xt
```

```
, , replicate = 1

    tension
wool  L  M  H
   A 26 18 36
   B 27 42 20

, , replicate = 2

    tension
wool  L  M  H
   A 30 21 21
   B 14 26 21
```

```
ftable(xt)
```

```
            replicate  1  2  3  4  5  6  7  8  9
wool tension
A    L                26 30 54 25 70 52 51 26 67
     M                18 21 29 17 12 18 35 30 36
     H                36 21 24 18 10 43 28 15 26
B    L                27 14 29 19 29 31 41 20 44
     M                42 26 19 16 39 28 21 39 29
     H                20 21 24 17 13 15 15 16 28
```

Size of a Data Set

```
fakeData = rnorm(1e5)
object.size(fakeData)
```

```
800040 bytes
```

```
print(object.size(fakeData),units="Mb")
```

```
0.8 Mb
```

Creating New Variables

- Often the raw data won't have a value you are looking for
- You will need to transform the data to get the values you would like
- Usually you will add those values to the data frames you are working with
- Common variables to create
  - Missingness indicators
  - "Cutting up" quantitative variables
  - Applying transforms

Creating Sequences

Sometimes you need an index for your data set

```
s1 <- seq(1,10,by=2) ; s1
```

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

```
s2 <- seq(1,10,length=3); s2
```

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

```
x <- c(1,3,8,25,100); seq(along = x)
```

```
[1] 1 2 3 4 5
```

Subsetting Variables

```
restData$nearMe = restData$neighborhood %in% c("Roland Park", "Homeland")
table(restData$nearMe)
```

```
FALSE  TRUE
 1314    13
```

Creating Binary Variables

```
restData$zipWrong = ifelse(restData$zipCode < 0, TRUE, FALSE)
table(restData$zipWrong,restData$zipCode < 0)
```

```
       FALSE TRUE
  FALSE  1326    0
  TRUE      0    1
```

Creating Categorical Variables

```
restData$zipGroups = cut(restData$zipCode,breaks=quantile(restData$zipCode))
table(restData$zipGroups)
```

```
(-2.123e+04,2.12e+04]  (2.12e+04,2.122e+04] (2.122e+04,2.123e+04] (2.123e+04,2.129e+04]
                  337                   375                   282                   332
```

```
table(restData$zipGroups,restData$zipCode)
```

```
                      -21226 21201 21202 21205 21206 21207 21208 21209 21210 21211 21212 21213
  (-2.123e+04,2.12e+04]      0   136   201     0     0     0     0     0     0     0     0     0
  (2.12e+04,2.122e+04]       0     0     0    27    30     4     1     8    23    41    28    31
  (2.122e+04,2.123e+04]      0     0     0     0     0     0     0     0     0     0     0     0
```

Easier Cutting

```
library(Hmisc)
restData$zipGroups = cut2(restData$zipCode,g=4)
table(restData$zipGroups)
```

```
[-21226,21205) [ 21205,21220) [ 21220,21227) [ 21227,21287]
           338            375            300            314
```

Creating Factor Variables

```
restData$zcf <- factor(restData$zipCode)
restData$zcf[1:10]
```

```
 [1] 21206 21231 21224 21211 21223 21218 21205 21211 21205 21231
32 Levels: -21226 21201 21202 21205 21206 21207 21208 21209 21210 21211 21212 21213 21214 ... 212
```

```
class(restData$zcf)
```

```
[1] "factor"
```

Levels of Factor Variables

```
yesno <- sample(c("yes","no"),size=10,replace=TRUE)
yesnofac = factor(yesno,levels=c("yes","no"))
relevel(yesnofac,ref="yes")
```

```
 [1] yes yes yes yes no  yes yes yes no  no
Levels: yes no
```

```
as.numeric(yesnofac)
```

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

Cutting Produces Factor Variables

```
library(Hmisc)
restData$zipGroups = cut2(restData$zipCode,g=4)
table(restData$zipGroups)
```

```
[-21226,21205) [ 21205,21220) [ 21220,21227) [ 21227,21287]
           338            375            300            314
```

Using the Mutate Function

```
library(Hmisc); library(plyr)
restData2 = mutate(restData,zipGroups=cut2(zipCode,g=4))
table(restData2$zipGroups)
```

```
[-21226,21205) [ 21205,21220) [ 21220,21227) [ 21227,21287]
           338            375            300            314
```

Common Transforms

- `abs(x)` absolute value
- `sqrt(x)` square root
- `ceiling(x)` ceiling(3.475) is 4
- `floor(x)` floor(3.475) is 3
- `round(x,digits=n)` roun(3.475,digits=2) is 3.48
- `signif(x,digits=n)` signif(3.475,digits=2) is 3.5
- `cos(x), sin(x)` etc.
- `log(x)` natural logarithm
- `log2(x)`, `log10(x)` other common logs
- `exp(x)` exponentiating x

http://www.biostat.jhsph.edu/~ajaffe/lec_winterR/Lecture%202.pdf
http://statmethods.net/management/functions.html

- A tutorial from the developer of plyr - http://plyr.had.co.nz/09-user/
- Andrew Jaffe's R notes http://www.biostat.jhsph.edu/~ajaffe/lec_winterR/Lecture%202.pdf

Reshaping Data
Goal is to Tidy Data

1. Each variable forms a column
2. Each observation forms a row
3. Each table/file stores data about one kind of observation (e.g. people/hospitals).

http://vita.had.co.nz/papers/tidy-data.pdf

Start with Reshaping

```
library(reshape2)
head(mtcars)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Melting Data Frames

```
mtcars$carname <- rownames(mtcars)
carMelt <- melt(mtcars,id=c("carname","gear","cyl"),measure.vars=c("mpg","hp"))
head(carMelt,n=3)
```

	carname	gear	cyl	variable	value
1	Mazda RX4	4	6	mpg	21.0
2	Mazda RX4 Wag	4	6	mpg	21.0
3	Datsun 710	4	4	mpg	22.8

```
tail(carMelt,n=3)
```

	carname	gear	cyl	variable	value
62	Ferrari Dino	5	6	hp	175
63	Maserati Bora	5	8	hp	335
64	Volvo 142E	4	4	hp	109

Casting Data Frames

```
cylData <- dcast(carMelt, cyl ~ variable)
cylData
```

```
  cyl mpg hp
1   4  11 11
2   6   7  7
3   8  14 14
```

```
cylData <- dcast(carMelt, cyl ~ variable,mean)
cylData
```

```
  cyl   mpg     hp
1   4 26.66  82.64
2   6 19.74 122.29
3   8 15.10 209.21
```

Averaging Values

```
head(InsectSprays)
```

```
  count spray
1    10     A
2     7     A
3    20     A
4    14     A
5    14     A
6    12     A
```

```
tapply(InsectSprays$count,InsectSprays$spray,sum)
```

```
  A   B   C   D   E   F
174 184  25  59  42 200
```

Another Way – Split

```
spIns =  split(InsectSprays$count,InsectSprays$spray)
spIns
```

```
$A
 [1] 10  7 20 14 14 12 10 23 17 20 14 13

$B
 [1] 11 17 21 11 16 14 17 17 19 21  7 13

$C
 [1] 0 1 7 2 3 1 2 1 3 0 1 4

$D
 [1]  3  5 12  6  4  3  5  5  5  5  2  4

$E
 [1] 3 5 3 5 3 6 1 1 3 2 6 4
```

Another Way – Combine

```
unlist(sprCount)
```

```
  A   B   C   D   E   F
174 184  25  59  42 200
```

```
sapply(spIns,sum)
```

```
  A   B   C   D   E   F
174 184  25  59  42 200
```

Another Way – Plyr Package

```
ddply(InsectSprays,.(spray),summarize,sum=sum(count))
```

```
  spray sum
1     A 174
2     B 184
3     C  25
4     D  59
5     E  42
6     F 200
```

Creating a New Variable

```
spraySums <- ddply(InsectSprays,.(spray),summarize,sum=ave(count,FUN=sum))
dim(spraySums)
```

```
[1] 72  2
```

```
head(spraySums)
```

```
  spray sum
1     A 174
2     A 174
3     A 174
4     A 174
5     A 174
6     A 174
```

- A tutorial from the developer of plyr - http://plyr.had.co.nz/09-user/
- A nice reshape tutorial http://www.slideshare.net/jeffreybreen/reshaping-data-in-r
- A good plyr primer - http://www.r-bloggers.com/a-quick-primer-on-split-apply-combine-problems/
- See also the functions
  - acast - for casting as multi-dimensional arrays
  - arrange - for faster reordering without using order() commands
  - mutate - adding new variables

Managing Data Frames with dplyr

The data frame is a key data structure in statistics and in R.

- There is one observation per row
- Each column represents a variable or measure or characteristic
- Primary implementation that you will use is the default R implementation
- Other implementations, particularly relational databases systems

- Developed by Hadley Wickham of RStudio
- An optimized and distilled version of `plyr` package (also by Hadley)
- Does not provide any "new" functionality per se, but **greatly** simplifies existing functionality in R
- Provides a "grammar" (in particular, verbs) for data manipulation
- Is **very** fast, as many key operations are coded in C++

Dplyr Verbs

- `select`: return a subset of the columns of a data frame
- `filter`: extract a subset of rows from a data frame based on logical conditions
- `arrange`: reorder rows of a data frame
- `rename`: rename variables in a data frame
- `mutate`: add new variables/columns or transform existing variables
- `summarise` / `summarize`: generate summary statistics of different variables in the data frame, possibly within strata

There is also a handy `print` method that prevents you from printing a lot of data to the console.

Dplyr Properties

- The first argument is a data frame.
- The subsequent arguments describe what to do with it, and you can refer to columns in the data frame directly without using the $ operator (just use the names).
- The result is a new data frame
- Data frames must be properly formatted and annotated for this to all be useful

Select Function

```
> head(select(chicago, city:dptp))
  city tmpd   dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
4 chic 29.0 28.625
5 chic 32.0 28.875
6 chic 40.0 35.125
> head(select(chicago, -(city:dptp)))
        date pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
1 1987-01-01         NA   34.00000 4.250000  19.98810
2 1987-01-02         NA         NA 3.304348  23.19099
3 1987-01-03         NA   34.16667 3.333333  23.81548
4 1987-01-04         NA   47.00000 4.375000  30.43452
5 1987-01-05         NA         NA 4.750000  30.33333
6 1987-01-06         NA   48.00000 5.833333  25.77233
```

Filter Function

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> head(chic.f, 10)
   city tmpd dptp       date pm25tmean2 pm10tmean2  o3tmean2 no2tmean2
1  chic   23 21.9 1998-01-17      38.10   32.46154  3.180556  25.30000
2  chic   28 25.8 1998-01-23      33.95   38.69231  1.750000  29.37630
3  chic   55 51.3 1998-04-30      39.40   34.00000 10.786232  25.31310
4  chic   59 53.7 1998-05-01      35.40   28.50000 14.295125  31.42905
5  chic   57 52.0 1998-05-02      33.30   35.00000 20.662879  26.79861
6  chic   57 56.0 1998-05-07      32.10   34.50000 24.270422  33.99167
7  chic   75 65.8 1998-05-15      56.50   91.00000 38.573007  29.03261
8  chic   61 59.0 1998-06-09      33.80   26.00000 17.890810  25.49668
9  chic   73 60.3 1998-07-13      30.30   64.50000 37.018865  37.93056
10 chic   78 67.1 1998-07-14      41.40   75.00000 40.080902  32.59054
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
> head(chic.f)
  city tmpd dptp       date pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
1 chic   81 71.2 1998-08-23    39.6000       59.0 45.86364  14.32639
2 chic   81 70.4 1998-09-06    31.5000       50.5 50.66250  20.31250
3 chic   82 72.2 2001-07-20    32.3000       58.5 33.00380  33.67500
4 chic   84 72.9 2001-08-01    43.7000       81.5 45.17736  27.44239
5 chic   85 72.6 2001-08-08    38.8375       70.0 37.98047  27.62743
6 chic   84 72.6 2001-08-09    38.2000       66.0 36.73245  26.46742
```

Arrange Function

```
> chicago <- arrange(chicago, date)
> head(chicago)
  city tmpd   dptp       date pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
1 chic 31.5 31.500 1987-01-01         NA   34.00000 4.250000  19.98810
2 chic 33.0 29.875 1987-01-02         NA         NA 3.304348  23.19099
3 chic 33.0 27.375 1987-01-03         NA   34.16667 3.333333  23.81548
4 chic 29.0 28.625 1987-01-04         NA   47.00000 4.375000  30.43452
5 chic 32.0 28.875 1987-01-05         NA         NA 4.750000  30.33333
6 chic 40.0 35.125 1987-01-06         NA   48.00000 5.833333  25.77233
> tail(chicago)
     city tmpd dptp       date pm25tmean2 pm10tmean2  o3tmean2 no2tmean2
6935 chic   35 29.6 2005-12-26    8.40000        8.5 14.041667  16.81944
6936 chic   40 33.6 2005-12-27   23.56000       27.0  4.468750  23.50000
6937 chic   37 34.5 2005-12-28   17.75000       27.5  3.260417  19.28563
6938 chic   35 29.4 2005-12-29    7.45000       23.5  6.794837  19.97222
6939 chic   36 31.0 2005-12-30   15.05714       19.2  3.034420  22.80556
6940 chic   35 30.1 2005-12-31   15.00000       23.5  2.531250  13.25000
```

```
> chicago <- arrange(chicago, desc(date))
> head(chicago)
  city tmpd dptp       date pm25tmean2 pm10tmean2  o3tmean2 no2tmean2
1 chic   35 30.1 2005-12-31   15.00000       23.5  2.531250  13.25000
2 chic   36 31.0 2005-12-30   15.05714       19.2  3.034420  22.80556
3 chic   35 29.4 2005-12-29    7.45000       23.5  6.794837  19.97222
4 chic   37 34.5 2005-12-28   17.75000       27.5  3.260417  19.28563
5 chic   40 33.6 2005-12-27   23.56000       27.0  4.468750  23.50000
6 chic   35 29.6 2005-12-26    8.40000        8.5 14.041667  16.81944
> tail(chicago)
     city tmpd   dptp       date pm25tmean2 pm10tmean2 o3tmean2 no2tmean2
6935 chic 40.0 35.125 1987-01-06         NA   48.00000 5.833333  25.77233
6936 chic 32.0 28.875 1987-01-05         NA         NA 4.750000  30.33333
6937 chic 29.0 28.625 1987-01-04         NA   47.00000 4.375000  30.43452
6938 chic 33.0 27.375 1987-01-03         NA   34.16667 3.333333  23.81548
6939 chic 33.0 29.875 1987-01-02         NA         NA 3.304348  23.19099
6940 chic 31.5 31.500 1987-01-01         NA   34.00000 4.250000  19.98810
```

Rename Function

```
> chicago <- rename(chicago, pm25 = pm25tmean2, dewpoint = dptp)
```

Mutate Function

```
> chicago <- mutate(chicago, pm25detrend = pm25-mean(pm25, na.rm = TRUE))
> head(select(chicago, pm25, pm25detrend))
      pm25 pm25detrend
1 15.00000   -1.230958
2 15.05714   -1.173815
3  7.45000   -8.780958
4 17.75000    1.519042
5 23.56000    7.329042
6  8.40000   -7.830958
```

Group_by Function

```
> chicago <- mutate(chicago, tempcat = factor(1 * (tmpd > 80), labels = c("cold", "hot"))
+ )
> hotcold <- group_by(chicago, tempcat)
> hotcold
Source: local data frame [6,940 x 10]
Groups: tempcat

   city tmpd dewpoint       date     pm25 pm10tmean2  o3tmean2 no2tmean2 pm25detrend tempcat
1  chic   35     30.1 2005-12-31 15.00000       23.5  2.531250  13.25000   -1.230958    cold
2  chic   36     31.0 2005-12-30 15.05714       19.2  3.034420  22.80556   -1.173815    cold
3  chic   35     29.4 2005-12-29  7.45000       23.5  6.794837  19.97222   -8.780958    cold
4  chic   37     34.5 2005-12-28 17.75000       27.5  3.260417  19.28563    1.519042    cold
5  chic   40     33.6 2005-12-27 23.56000       27.0  4.468750  23.50000    7.329042    cold
6  chic   35     29.6 2005-12-26  8.40000        8.5 14.041667  16.81944   -7.830958    cold
7  chic   35     32.1 2005-12-25  6.70000        8.0 14.354167  13.79167   -9.530958    cold
8  chic   37     35.2 2005-12-24 30.77143       25.2  1.770833  31.98611   14.540471    cold
9  chic   41     32.6 2005-12-23 32.90000       34.5  6.906250  29.08333   16.669042    cold
10 chic   22     23.3 2005-12-22 36.65000       42.5  5.385417  33.73026   20.419042    cold
```

Summarize Function

```
> summarize(hotcold, pm25 = mean(pm25), o3 = max(o3tmean2), no2 = median(no2tmean2))
Source: local data frame [3 x 4]

  tempcat    pm25       o3      no2
1    cold      NA 66.587500 24.54924
2     hot      NA 62.969656 24.93870
3      NA 47.7375  9.416667 37.44444
> summarize(hotcold, pm25 = mean(pm25, na.rm = TRUE), o3 = max(o3tmean2), no2 = median(no2tmean2))
Source: local data frame [3 x 4]

  tempcat     pm25       o3      no2
1    cold 15.97807 66.587500 24.54924
2     hot 26.48118 62.969656 24.93870
3      NA 47.73750  9.416667 37.44444
```

Summary

```
> chicago %>% mutate(month = as.POSIXlt(date)$mon + 1) %>% group_by(month) %>% summarize(pm25 = mean(pm25
, na.rm = TRUE), o3 = max(o3tmean2), no2 = median(no2tmean2))
Source: local data frame [12 x 4]

   month     pm25       o3      no2
1      1 17.76996 28.22222 25.35417
2      2 20.37513 37.37500 26.78034
3      3 17.40818 39.05000 26.76984
4      4 13.85879 47.94907 25.03125
5      5 14.07420 52.75000 24.22222
6      6 15.86461 66.58750 25.01140
7      7 16.57087 59.54167 22.38442
8      8 16.93380 53.96701 22.98333
9      9 15.91279 57.48864 24.47917
10    10 14.23557 47.09275 24.15217
11    11 15.15794 29.45833 23.56537
12    12 17.52221 27.70833 24.45773
```

Benefits of dpryr

Once you learn the dplyr "grammar" there are a few additional benefits

- dplyr can work with other data frame "backends"
- `data.table` for large fast tables
- SQL interface for relational databases via the DBI package

Merging Data

```
if(!file.exists("./data")){dir.create("./data")}
fileUrl1 = "https://dl.dropboxusercontent.com/u/7710864/data/reviews-apr29.csv"
fileUrl2 = "https://dl.dropboxusercontent.com/u/7710864/data/solutions-apr29.csv"
download.file(fileUrl1,destfile="./data/reviews.csv",method="curl")
download.file(fileUrl2,destfile="./data/solutions.csv",method="curl")
reviews = read.csv("./data/reviews.csv"); solutions <- read.csv("./data/solutions.csv")
head(reviews,2)
```

```
  id solution_id reviewer_id      start       stop time_left accept
1  1           3          27 1304095698 1304095758      1754      1
2  2           4          22 1304095188 1304095206      2306      1
```

```
head(solutions,2)
```

```
  id problem_id subject_id      start       stop time_left answer
1  1        156         29 1304095119 1304095169      2343      B
```


Merge()

- Merges data frames
- Important parameters: *x,y,by,by.x,by.y,all*

```
names(reviews)
```

```
[1] "id"          "solution_id" "reviewer_id" "start"       "stop"        "time_left"
[7] "accept"
```

```
names(solutions)
```

```
[1] "id"         "problem_id" "subject_id" "start"      "stop"       "time_left"  "answer"
```

```
mergedData = merge(reviews,solutions,by.x="solution_id",by.y="id",all=TRUE)
head(mergedData)
```

```
  solution_id id reviewer_id    start.x     stop.x time_left.x accept problem_id subject_id
1           1  4          26 1304095267 1304095423        2089      1        156         29
2           2  6          29 1304095471 1304095513        1999      1        269         25
3           3  1          27 1304095698 1304095758        1754      1         34         22
4           4  2          22 1304095188 1304095206        2306      1         19         23
5           5  3          28 1304095276 1304095320        2192      1        605         26
6           6 16          22 1304095303 1304095471        2041      1        384         27
     start.y     stop.y time_left.y answer
1 1304095119 1304095169        2343      B
2 1304095119 1304095183        2329      C
3 1304095127 1304095146        2366      C
4 1304095127 1304095150        2362      D
5 1304095127 1304095167        2345      A
6 1304095131 1304095270        2242      C
```

Default – merge all common column names

```
intersect(names(solutions),names(reviews))
```

```
[1] "id"        "start"     "stop"      "time_left"
```

```
mergedData2 = merge(reviews,solutions,all=TRUE)
head(mergedData2)
```

```
  id      start       stop time_left solution_id reviewer_id accept problem_id subject_id answer
1  1 1304095119 1304095169      2343          NA          NA     NA        156         29      B
2  1 1304095698 1304095758      1754           3          27      1         NA         NA   <NA>
3  2 1304095119 1304095183      2329          NA          NA     NA        269         25      C
4  2 1304095188 1304095206      2306           4          22      1         NA         NA   <NA>
5  3 1304095127 1304095146      2366          NA          NA     NA         34         22      C
6  3 1304095276 1304095320      2192           5          28      1         NA         NA   <NA>
```

Using Join in the plyr Package

Faster, but less full featured - defaults to left join, see help file for more

```
df1 = data.frame(id=sample(1:10),x=rnorm(10))
df2 = data.frame(id=sample(1:10),y=rnorm(10))
arrange(join(df1,df2),id)
```

	id	x	y
1	1	0.2514	0.2286
2	2	0.1048	0.8395
3	3	-0.1230	-1.1165
4	4	1.5057	-0.1121
5	5	-0.2505	1.2124
6	6	0.4699	-1.6038
7	7	0.4627	-0.8060
8	8	-1.2629	-1.2848
9	9	-0.9258	-0.8276
10	10	2.8065	0.5794

If You Have Multiple Data Frames

```
df1 = data.frame(id=sample(1:10),x=rnorm(10))
df2 = data.frame(id=sample(1:10),y=rnorm(10))
df3 = data.frame(id=sample(1:10),z=rnorm(10))
dfList = list(df1,df2,df3)
join_all(dfList)
```

	id	x	y	z
1	6	0.39093	-0.16670	0.56523
2	1	-1.90467	0.43811	-0.37449
3	7	-1.48798	-0.85497	-0.69209
4	10	-2.59440	0.39591	-0.36134
5	3	-0.08539	0.08053	1.01247
6	4	-1.63165	-0.13158	0.21927
7	5	-0.50594	0.24256	-0.44003
8	9	-0.85062	-2.08066	-0.96950
9	2	-0.63767	-0.10069	0.09002
10	8	1.20439	1.29138	-0.88586

- The quick R data merging page - http://www.statmethods.net/management/merging.html
- plyr information - http://plyr.had.co.nz/
- Types of joins - http://en.wikipedia.org/wiki/Join_(SQL))

Editing Text Variables

Fixing Character Vectors – tolower(), toupper()

```
if(!file.exists("./data")){dir.create("./data")}
fileUrl <- "https://data.baltimorecity.gov/api/views/dz54-2aru/rows.csv?accessType=DOWNLOAD"
download.file(fileUrl,destfile="./data/cameras.csv",method="curl")
cameraData <- read.csv("./data/cameras.csv")
names(cameraData)
```

```
[1] "address"      "direction"    "street"       "crossStreet"  "intersection" "Location.1"
```

```
tolower(names(cameraData))
```

```
[1] "address"      "direction"    "street"       "crossstreet"  "intersection" "location.1"
```

Fixing Character Vectors – strsplit()

- Good for automatically splitting variable names
- Important parameters: *x*, *split*

```
splitNames = strsplit(names(cameraData),"\\.")
splitNames[[5]]
```

```
[1] "intersection"
```

```
splitNames[[6]]
```

```
[1] "Location" "1"
```

Quick aside – Lists

```
mylist <- list(letters = c("A", "b", "c"), numbers = 1:3, matrix(1:25, ncol = 5))
head(mylist)
```

```
$letters
[1] "A" "b" "c"

$numbers
[1] 1 2 3

[[3]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Fixing character vectors – sapply()

- Applies a function to each element in a vector or list
- Important parameters: *X,FUN*

```
splitNames[[6]][1]
```

```
[1] "Location"
```

```
firstElement <- function(x){x[1]}
sapply(splitNames,firstElement)
```

```
[1] "address"      "direction"    "street"       "crossStreet"  "intersection" "Location"
```

Peer Review Data

```
fileUrl1 <- "https://dl.dropboxusercontent.com/u/7710864/data/reviews-apr29.csv"
fileUrl2 <- "https://dl.dropboxusercontent.com/u/7710864/data/solutions-apr29.csv"
download.file(fileUrl1,destfile="./data/reviews.csv",method="curl")
download.file(fileUrl2,destfile="./data/solutions.csv",method="curl")
reviews <- read.csv("./data/reviews.csv"); solutions <- read.csv("./data/solutions.csv")
head(reviews,2)
```

```
  id solution_id reviewer_id      start       stop time_left accept
1  1           3          27 1304095698 1304095758      1754      1
2  2           4          22 1304095188 1304095206      2306      1
```

```
head(solutions,2)
```

```
  id problem_id subject_id      start       stop time_left answer
1  1        156         29 1304095119 1304095169      2343      B
```

Fixing character vectors – sub()

- Important parameters: *pattern*, *replacement*, *x*

```
names(reviews)
```

```
[1] "id"          "solution_id" "reviewer_id" "start"       "stop"        "time_left"
[7] "accept"
```

```
sub("_","",names(reviews),)
```

```
[1] "id"         "solutionid" "reviewerid" "start"      "stop"       "timeleft"   "accept"
```

Only replaces first one

Fixing character vectors – gsub()

```
testName <- "this_is_a_test"
sub("_","",testName)
```

```
[1] "thisis_a_test"
```

```
gsub("_","",testName)
```

```
[1] "thisisatest"
```

Replaces all

Finding values – grep(), grepl()

```
grep("Alameda",cameraData$intersection)
```

```
[1]  4  5 36
```

```
table(grepl("Alameda",cameraData$intersection))
```

```
FALSE  TRUE
   77     3
```

```
cameraData2 <- cameraData[!grepl("Alameda",cameraData$intersection),]
```

More on grep()

```
grep("Alameda",cameraData$intersection,value=TRUE)
```

```
[1] "The Alameda  & 33rd St"   "E 33rd  & The Alameda"    "Harford \n & The Alameda"
```

```
grep("JeffStreet",cameraData$intersection)
```

```
integer(0)
```

```
length(grep("JeffStreet",cameraData$intersection))
```

```
[1] 0
```

http://www.biostat.jhsph.edu/~ajaffe/lec_winterR/Lecture%203.pdf

More useful string functions

```
library(stringr)
nchar("Jeffrey Leek")
```

```
[1] 12
```

```
substr("Jeffrey Leek",1,7)
```

```
[1] "Jeffrey"
```

```
paste("Jeffrey","Leek")
```

```
[1] "Jeffrey Leek"
```

```
paste0("Jeffrey","Leek")
```

```
[1] "JeffreyLeek"
```

```
str_trim("Jeff      ")
```

```
[1] "Jeff"
```

Important points about text in data sets

- Names of variables should be
  - All lower case when possible
  - Descriptive (Diagnosis versus Dx)
  - Not duplicated
  - Not have underscores or dots or white spaces
- Variables with character values
  - Should usually be made into factor variables (depends on application)
  - Should be descriptive (use TRUE/FALSE instead of 0/1 and Male/Female versus 0/1 or M/F)

Regular Expressions

- Regular expressions can be thought of as a combination of literals and *metacharacters*
- To draw an analogy with natural language, think of literal text forming the words of this language, and the metacharacters defining its grammar
- Regular expressions have a rich set of metacharacters

Literals

Simplest pattern consists only of literals. The literal “nuclear” would match to the following lines:

```
Ooh. I just learned that to keep myself alive after a
nuclear blast! All I have to do is milk some rats
then drink the milk. Aweosme. :}

Laozi says nuclear weapons are mas macho

Chaos in a country that has nuclear weapons — not good.

my nephew is trying to teach me nuclear physics, or
possibly just trying to show me how smart he is
so I'll be proud of him [which I am].

lol if you ever say "nuclear" people immediately think
DEATH by radiation LOL
```

Regular Expressions

- Simplest pattern consists only of literals; a match occurs if the sequence of literals occurs anywhere in the text being tested
- What if we only want the word “Obama”? or sentences that end in the word “Clinton”, or “clinton” or “clinto”?

We need a way to express

- whitespace word boundaries
- sets of literals
- the beginning and end of a line
- alternatives (“war” or “peace”) Metacharacters to the rescue!

Metacharacters

Some metacharacters represent the start of a line

```
^i think
```

will match the lines

```
i think we all rule for participating
i think i have been outed
i think this will be quite fun actually
i think i need to go to work
i think i first saw zombo in 1999.
```

$ represents the end of a line

```
morning$
```

will match the lines

```
well they had something this morning
then had to catch a tram home in the morning
dog obedience school in the morning
and yes happy birthday i forgot to say it earlier this morning
I walked in the rain this morning
good morning
```

Character Classes with []

We can list a set of characters we will accept at a given point in the match

```
[Bb][Uu][Ss][Hh]
```

will match the lines

```
The democrats are playing, "Name the worst thing about Bush!"
I smelled the desert creosote bush, brownies, BBQ chicken
BBQ and bushwalking at Molonglo Gorge
Bush TOLD you that North Korea is part of the Axis of Evil
I'm listening to Bush - Hurricane (Album Version)
```

```
^[Ii] am
```

will match

```
i am so angry at my boyfriend i can't even bear to
look at him

i am boycotting the apple store

I am twittering from iPhone

I am a very vengeful person when you ruin my sweetheart.

I am so over this. I need food. Mmmm bacon...
```

Similarly, you can specify a range of letters [a-z] or [a-zA-Z]; notice that the order doesn't matter

```
^[0-9][a-zA-Z]
```

will match the lines

```
7th inning stretch
2nd half soon to begin. OSU did just win something
3am - cant sleep - too hot still.. :(
5ft 7 sent from heaven
1st sign of starvagtion
```

When used at the beginning of a character class, the “” is also a metacharacter and indicates matching characters NOT in the indicated class

```
[^?.]$
```

will match the lines

```
i like basketballs
6 and 9
dont worry... we all die anyway!
Not in Baghdad
helicopter under water? hmmm
```

“.” is used to refer to any character. So

```
9.11
```

will match the lines

```
its stupid the post 9-11 rules
if any 1 of us did 9/11 we would have been caught in days.
NetBios: scanning ip 203.169.114.66
Front Door 9:11:46 AM
Sings: 0118999881999119725...3 !
```

This does not mean “pipe” in the context of regular expressions; instead it translates to “or”; we can use it to combine two expressions, the subexpressions being called alternatives

```
flood|fire
```

will match the lines

```
is firewire like usb on none macs?
the global flood makes sense within the context of the bible
yeah ive had the fire on tonight
... and the floods, hurricanes, killer heatwaves, rednecks, gun nuts, etc.
```

We can include any number of alternatives...

```
flood|earthquake|hurricane|coldfire
```

will match the lines

```
Not a whole lot of hurricanes in the Arctic.
We do have earthquakes nearly every day somewhere in our State
hurricanes swirl in the other direction
coldfire is STRAIGHT!
'cause we keep getting earthquakes
```

The alternatives can be real expressions and not just literals

```
^[Gg]ood|[Bb]ad
```

will match the lines

```
good to hear some good knews from someone here
Good afternoon fellow american infidels!
good on you-what do you drive?
Katie... guess they had bad experiences...
my middle name is trouble, Miss Bad News
```

Subexpressions are often contained in parentheses to constrain the alternatives

```
^([Gg]ood|[Bb]ad)
```

will match the lines

```
bad habbit
bad coordination today
good, becuase there is nothing worse than a man in kinky underwear
Badcop, its because people want to use drugs
Good Monday Holiday
Good riddance to Limey
```

The question mark indicates that the indicated expression is optional

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

will match the lines

```
i bet i can spell better than you and george bush combined
BBC reported that President George W. Bush claimed God told him to invade I
a bird in the hand is worth two george bushes
```

In the following

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

we wanted to match a "." as a literal period; to do that, we had to "escape" the metacharacter, preceding it with a backslash In general, we have to do this for any metacharacter we want to include in our match

The * and + signs are metacharacters used to indicate repetition; * means "any number, including none, of the item" and + means "at least one of the item"

```
(.*)
```

will match the lines

```
anyone wanna chat? (24, m, germany)
hello, 20.m here... ( east area + drives + webcam )
(he means older men)
()
```

The * and + signs are metacharacters used to indicate repetition; * means "any number, including none, of the item" and + means "at least one of the item"

```
[0-9]+ (.*)[0-9]+
```

will match the lines

```
working as MP here 720 MP battallion, 42nd birgade
so say 2 or 3 years at colleage and 4 at uni makes us 23 when and if we fin
it went down on several occasions for like, 3 or 4 *days*
Mmmm its time 4 me 2 go 2 bed
```

{ and } are referred to as interval quantifiers; the let us specify the minimum and maximum number of matches of an expression

```
[Bb]ush( +[^ ]+ +){1,5} debate
```

will match the lines

```
Bush has historically won all major debates he's done.
in my view, Bush doesn't need these debates..
bush doesn't need the debates? maybe you are right
That's what Bush supporters are doing about the debate.
Felix, I don't disagree that Bush was poorly prepared for the debate.
indeed, but still, Bush should have taken the debate more seriously.
Keep repeating that Bush smirked and scowled during the debate
```

- m,n means at least m but not more than n matches
- m means exactly m matches
- m, means at least m matches

- In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a "|", but also can be used to "remember" text matched by the subexpression enclosed
- We refer to the matched text with \1, \2, etc.

So the expression

```
+([a-zA-Z]+) +\1 +
```

will match the lines

```
time for bed, night night twitter!
blah blah blah blah
my tattoo is so so itchy today
i was standing all all alone against the world outside...
hi anybody anybody at home
estudiando css css css css.... que desastritooooo
```

The * is “greedy” so it always matches the *longest* possible string that satisfies the regular expression. So

```
^s(.*)s
```

matches

```
sitting at starbucks
setting up mysql and rails
studying stuff for the exams
spaghetti with marshmallows
stop fighting with crackers
sore shoulders, stupid ergonomics
```

The greediness of * can be turned off with the ?, as in

```
^s(.*?)s$
```

Summary

- Regular expressions are used in many different languages; not unique to R.
- Regular expressions are composed of literals and metacharacters that represent sets or classes of characters/words
- Text processing via regular expressions is a very powerful way to extract data from “unfriendly” sources (not all data comes as a CSV file)
- Used with the functions `grep`,`grepl`,`sub`,`gsub` and others that involve searching for text strings (Thanks to Mark Hansen for some material in this lecture.)

Working with Dates

```
d1 = date()
d1
```

```
[1] "Sun Jan 12 17:48:33 2014"
```

```
class(d1)
```

```
[1] "character"
```

Date class

```
d2 = Sys.Date()
d2
```

```
[1] "2014-01-12"
```

```
class(d2)
```

```
[1] "Date"
```

Formatting Dates

`%d` = day as number (0-31), `%a` = abbreviated weekday,`%A` = unabbreviated weekday, `%m` = month (00-12), `%b` = abbreviated month, `%B` = unabbrevidated month, `%y` = 2 digit year, `%Y` = four digit year

```
format(d2,"%a %b %d")
```

```
[1] "Sun Jan 12"
```

Creating dates

```
x = c("1jan1960", "2jan1960", "31mar1960", "30jul1960"); z = as.Date(x, "%d%b%Y")
z
```

```
[1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"
```

```
z[1] - z[2]
```

```
Time difference of -1 days
```

```
as.numeric(z[1]-z[2])
```

```
[1] -1
```

Converting to Julian

```
weekdays(d2)
```

```
[1] "Sunday"
```

```
months(d2)
```

```
[1] "January"
```

```
julian(d2)
```

```
[1] 16082
attr(,"origin")
[1] "1970-01-01"
```

Lubridate

```
library(lubridate); ymd("20140108")
```

```
[1] "2014-01-08 UTC"
```

```
mdy("08/04/2013")
```

```
[1] "2013-08-04 UTC"
```

```
dmy("03-04-2013")
```

```
[1] "2013-04-03 UTC"
```

Dealing with times

```
ymd_hms("2011-08-03 10:15:03")
```

```
[1] "2011-08-03 10:15:03 UTC"
```

```
ymd_hms("2011-08-03 10:15:03",tz="Pacific/Auckland")
```

```
[1] "2011-08-03 10:15:03 NZST"
```

```
?Sys.timezone
```

http://www.r-statistics.com/2012/03/do-more-with-dates-and-times-in-r-with-lubridate-1-1-0/

Some functions have slightly different syntax

```
x = dmy(c("1jan2013", "2jan2013", "31mar2013", "30jul2013"))
wday(x[1])
```

```
[1] 3
```

```
wday(x[1],label=TRUE)
```

```
[1] Tues
Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat
```

Notes / Further Resources

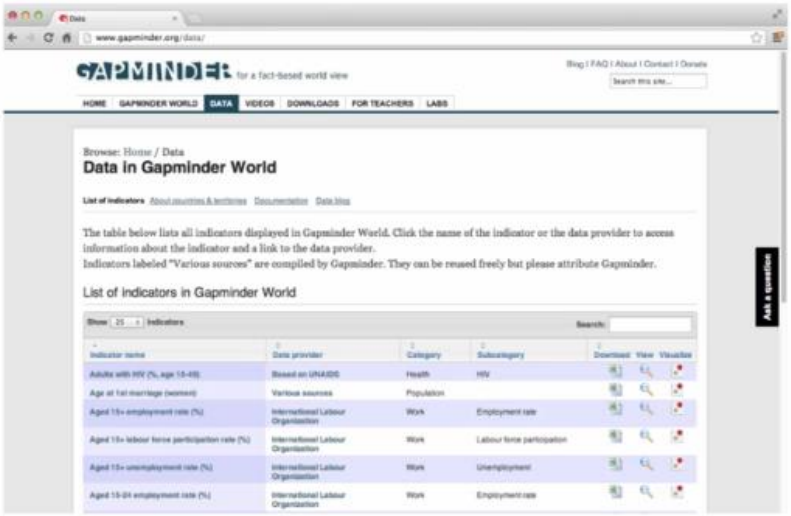
- More information in this nice lubridate tutorial http://www.r-statistics.com/2012/03/do-more-with-dates-and-times-in-r-with-lubridate-1-1-0/
- The lubridate vignette is the same content http://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html
- Ultimately you want your dates and times as class "Date" or the classes "POSIXct", "POSIXlt". For more information type `?POSIXlt`

Data Resources

Government Sites

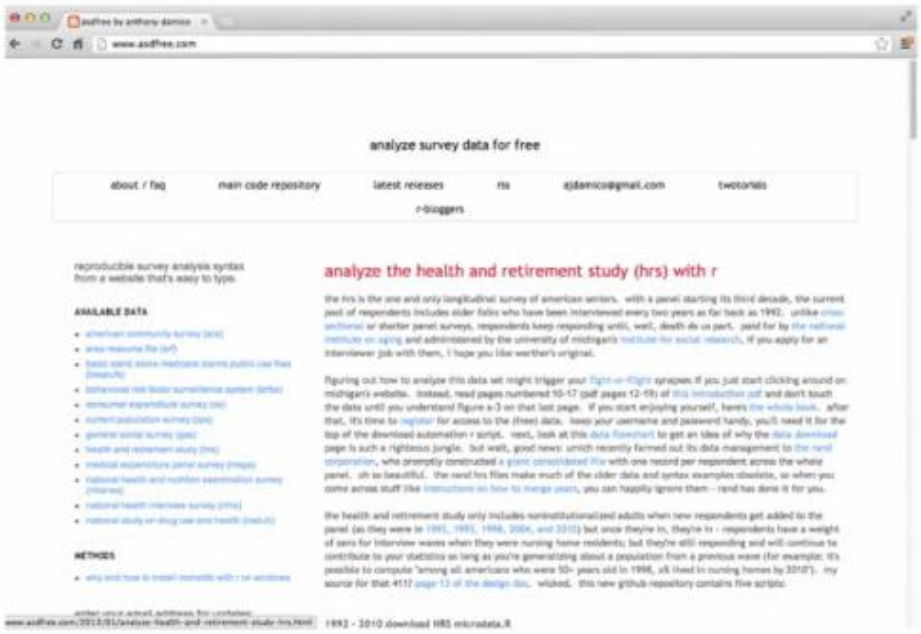
- United Nations http://data.un.org/
- U.S. http://www.data.gov/
  - List of cities/states with open data
- United Kingdom http://data.gov.uk/
- France http://www.data.gouv.fr/
- Ghana http://data.gov.gh/
- Australia http://data.gov.au/
- Germany https://www.govdata.de/
- Hong Kong http://www.gov.hk/en/theme/psi/datasets/
- Japan http://www.data.go.jp/
- Many more http://www.data.gov/opendatasites

Gapminder



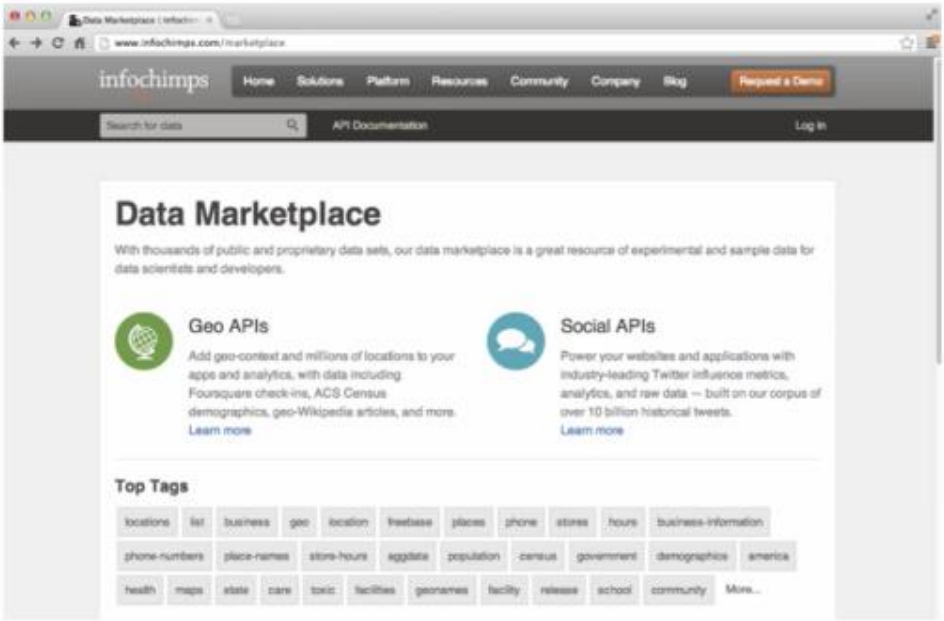
http://www.gapminder.org/

Survey Data from USA



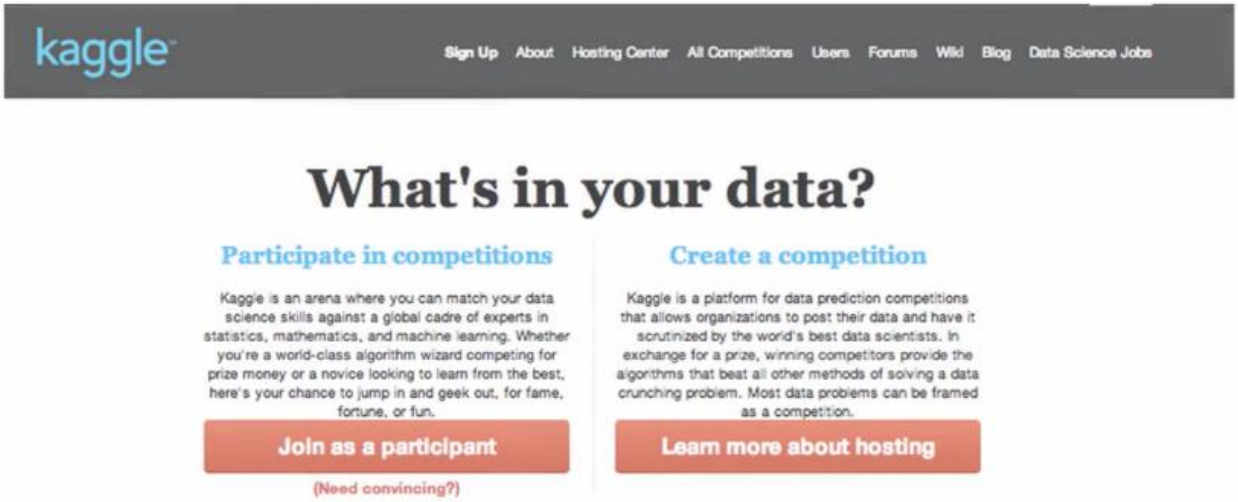
http://www.asdfree.com/

Infochimps



http://www.infochimps.com/marketplace

Kaggle



http://www.kaggle.com/

Collections by data scientists

- Hilary Mason http://bitly.com/bundles/hmason/1
- Peter Skomoroch https://delicious.com/pskomoroch/dataset
- Jeff Hammerbacher http://www.quora.com/Jeff-Hammerbacher/Introduction-to-Data-Science-Data-Sets
- Gregory Piatetsky-Shapiro http://www.kdnuggets.com/gps.html
- http://blog.mortardata.com/post/67652898761/6-dataset-lists-curated-by-data-scientists

More Specialized Collections

- Stanford Large Newtork Data
- UCI Machine Learning
- KDD Nugets Datasets
- CMU Statlib
- Gene expression omnibus
- ArXiv Data
- Public Data Sets on Amazon Web Services

Some API's with R interfaces

- twitter and twitteR package
- figshare and rfigshare
- PLoS and rplos
- rOpenSci
- Facebook and RFacebook
- Google maps and RGoogleMaps

Changing Working Directory

File → Change Dir

```
> getwd
function ()
.Internal(getwd())
<bytecode: 0x00000000261af828>
<environment: namespace:base>
> dir()
[1] "~$Week01.docx"        "artofdatascience.pdf" "exdata.pdf"
[4] "practice01.r"         "Week01.docx"
> ls()
character(0)
> source("practice01.R")
> ls()
[1] "myfunction" "second"
> secpmd(4)
Error: could not find function "secpmd"
> second(4)
[1] 3.790873
> second(4)
[1] 1.674745
> second(4)
[1] 4.649308
> myfunction()
[1] -0.126019
>
```

Principles of Analytic Graphics

- Principle 1: Show comparisons
  - Evidence for a hypothesis is always *relative* to another competing hypothesis.
  - Always ask "Compared to What?"
- Principle 2: Show causality, mechanism, explanation, systematic structure
  - What is your causal framework for thinking about a question?
- Principle 3: Show multivariate data
  - Multivariate = more than 2 variables
  - The real world is multivariate
  - Need to "escape flatland"
- Principle 4: Integration of evidence
  - Completely integrate words, numbers, images, diagrams
  - Data graphics should make use of many modes of data presentation
  - Don't let the tool drive the analysis
- Principle 5: Describe and document the evidence with appropriate labels, scales, sources, etc.
  - A data graphic should tell a complete story that is credible
- Principle 6: Content is king
  - Analytical presentations ultimately stand or fall depending on the quality, relevance, and integrity of their content

Exploratory Graphs

- To understand data properties
- To find patterns in data
- To suggest modeling strategies
- To "debug" analyses
- To communicate results

- They are made quickly
- A large number are made
- The goal is for personal understanding
- Axes/legends are generally cleaned up (later)
- Color/size are primarily used for information

Data

Annual average PM2.5 averaged over the period 2008 through 2010

```
pollution <- read.csv("data/avgpm25.csv", colClasses = c("numeric", "character",
    "factor", "numeric", "numeric"))
head(pollution)
```

```
##     pm25  fips region longitude latitude
## 1  9.771 01003   east    -87.75    30.59
## 2  9.994 01027   east    -85.84    33.27
## 3 10.689 01033   east    -87.73    34.73
## 4 11.337 01049   east    -85.80    34.46
## 5 12.120 01055   east    -86.03    34.02
## 6 10.828 01069   east    -85.35    31.19
```

Do any counties exceed the standard of $12\ \mu g/m^3$?

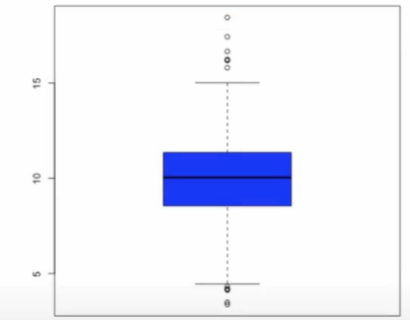
Five Number Summary

```
summary(pollution$pm25)
```

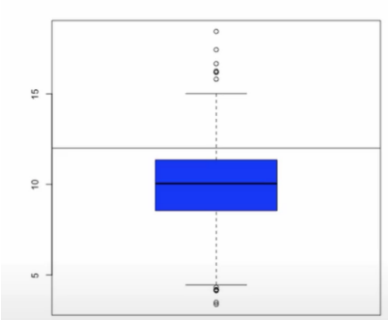
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    3.38    8.55   10.00    9.84   11.40   18.40
```

Box/Histogram/Bar Plots

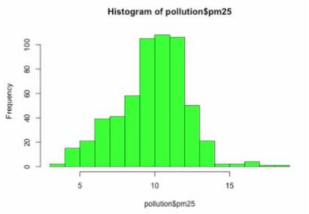
```
boxplot(pollution$pm25, col = "blue")
```



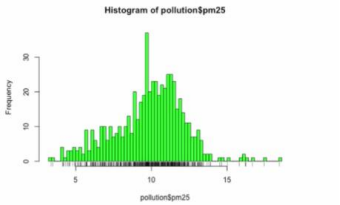
```
boxplot(pollution$pm25, col = "blue")
abline(h = 12)
```



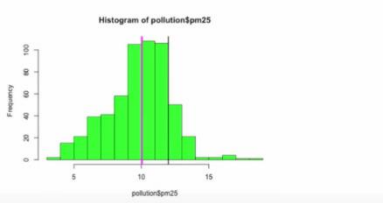
```
hist(pollution$pm25, col = "green")
```



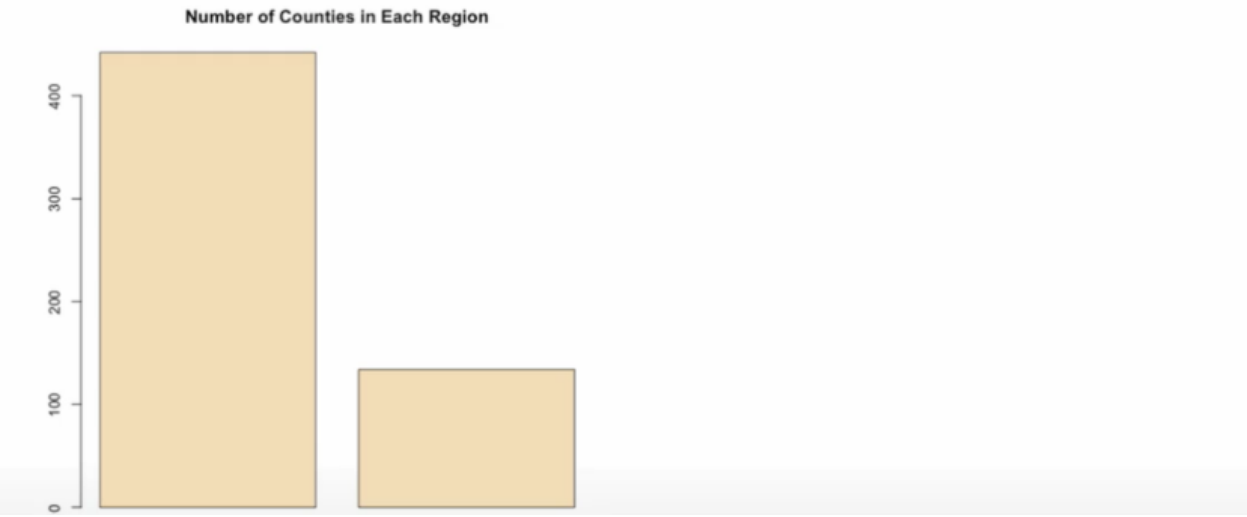

```
hist(pollution$pm25, col = "green", breaks = 100)
rug(pollution$pm25)
```




```
hist(pollution$pm25, col = "green")
abline(v = 12, lwd = 2)
abline(v = median(pollution$pm25), col = "magenta", lwd = 4)
```




```
barplot(table(pollution$region), col = "wheat", main = "Number of Counties in Each Region")
```



Simple Summaries of Data

Two dimensions

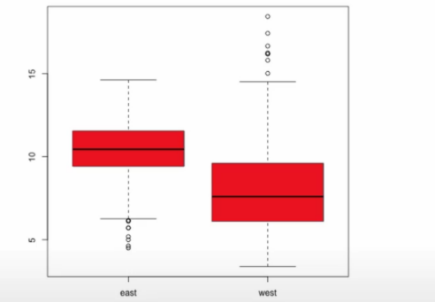
- Multiple/overlayed 1-D plots (Lattice/ggplot2)
- Scatterplots
- Smooth scatterplots

> 2 dimensions

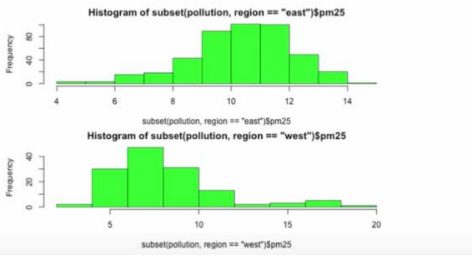
- Overlayed/multiple 2-D plots; coplots
- Use color, size, shape to add dimensions
- Spinning plots
- Actual 3-D plots (not that useful)

Multiple Plots

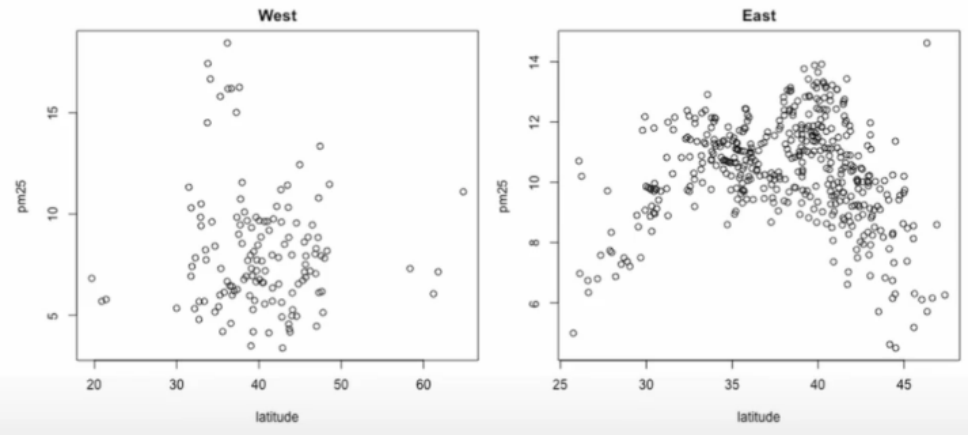
```
boxplot(pm25 ~ region, data = pollution, col = "red")
```




```
par(mfrow = c(2, 1), mar = c(4, 4, 2, 1))
hist(subset(pollution, region == "east")$pm25, col = "green")
hist(subset(pollution, region == "west")$pm25, col = "green")
```

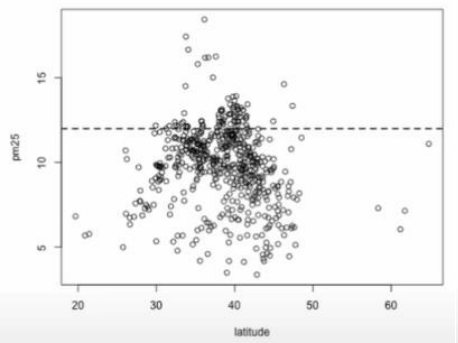



```
par(mfrow = c(1, 2), mar = c(5, 4, 2, 1))
with(subset(pollution, region == "west"), plot(latitude, pm25, main = "West"))
with(subset(pollution, region == "east"), plot(latitude, pm25, main = "East"))
```

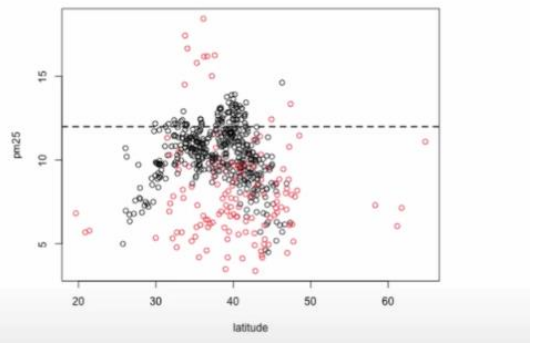


Scatter Plots – Using Color

```
with(pollution, plot(latitude, pm25))
abline(h = 12, lwd = 2, lty = 2)
```




```
with(pollution, plot(latitude, pm25, col = region))
abline(h = 12, lwd = 2, lty = 2)
```



Summary

- Exploratory plots are "quick and dirty"
- Let you summarize the data (usually graphically) and highly any broad features
- Explore basic questions and hypotheses (and perhaps rule them out)
- Suggest modeling strategies for the "next step"

- R Graph Gallery
- R Bloggers

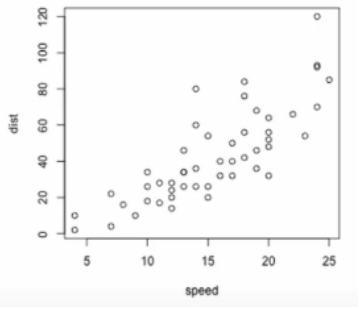
Plotting Systems in R

Base Plotting System

- "Artist's palette" model
- Start with blank canvas and build up from there
- Start with plot function (or similar)
- Use annotation functions to add/modify (`text`, `lines`, `points`, `axis`)

- Convenient, mirrors how we think of building plots and analyzing data
- Can't go back once plot has started (i.e. to adjust margins); need to plan in advance
- Difficult to "translate" to others once a new plot has been created (no graphical "language")
- Plot is just a series of R commands

```
library(datasets)
data(cars)
with(cars, plot(speed, dist))
```

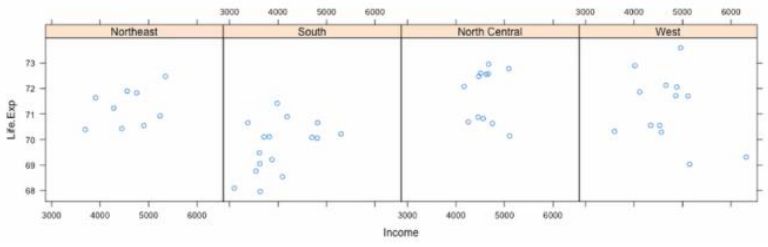


Lattice Plotting System

- Plots are created with a single function call (`xyplot`, `bwplot`, etc.)
- Most useful for conditioning types of plots: Looking at how y changes with x across levels of z
- Things like margins/spacing set automatically because entire plot is specified at once
- Good for puttng many many plots on a screen

- Sometimes awkward to specify an entire plot in a single function call
- Annotation in plot is not especially intuitive
- Use of panel functions and subscripts difficult to wield and requires intense preparation
- Cannot "add" to the plot once it is created

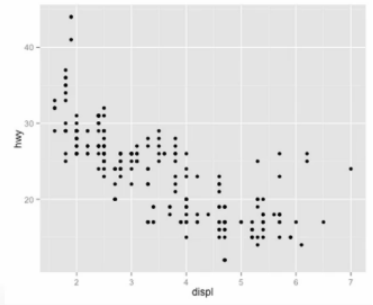
```
library(lattice)
state <- data.frame(state.x77, region = state.region)
xyplot(Life.Exp ~ Income | region, data = state, layout = c(4, 1))
```



GGPlot2 Plotting System

- Splits the difference between base and lattice in a number of ways
- Automatically deals with spacings, text, titles but also allows you to annotate by "adding" to a plot
- Superficial similarity to lattice but generally easier/more intuitive to use
- Default mode makes many choices for you (but you can still customize to your heart's desire)

```
library(ggplot2)
data(mpg)
qplot(displ, hwy, data = mpg)
```



Summary

- Base: "artist's palette" model
- Lattice: Entire plot specified by one function; conditioning
- ggplot2: Mixes elements of Base and Lattice

Base Plotting System

The core plotting and graphics engine in R is encapsulated in the following packages:

- *graphics*: contains plotting functions for the "base" graphing systems, including `plot`, `hist`, `boxplot` and many others.
- *grDevices*: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

The lattice plotting system is implemented using the following packages:

- *lattice*: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`
- *grid*: implements a different graphing system independent of the "base" system; the *lattice* package builds on top of *grid*; we seldom call functions from the *grid* package directly

Process of Making a Plot

When making a plot one must first make a few considerations (not necessarily in this order):

- Where will the plot be made? On the screen? In a file?
- How will the plot be used?
  - Is the plot for viewing temporarily on the screen?
  - Will it be presented in a web browser?
  - Will it eventually end up in a paper that might be printed?
  - Are you using it in a presentation?
- Is there a large amount of data going into the plot? Or is it just a few points?
- Do you need to be able to dynamically resize the graphic?

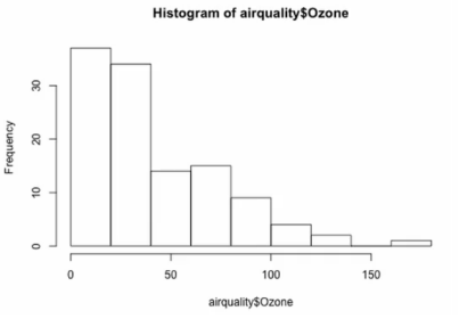
Base Graphics

Base graphics are used most commonly and are a very powerful system for creating 2-D graphics.

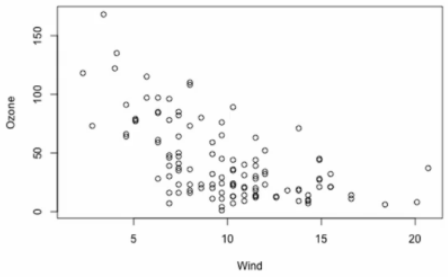
- There are two *phases* to creating a base plot
  - Initializing a new plot
  - Annotating (adding to) an existing plot
- Calling `plot(x, y)` or `hist(x)` will launch a graphics device (if one is not already open) and draw a new plot on the device
- If the arguments to `plot` are not of some special class, then the *default* method for `plot` is called; this function has *many* arguments, letting you set the title, x axis label, y axis label, etc.
- The base graphics system has *many* parameters that can set and tweaked; these parameters are documented in `?par`; it wouldn't hurt to try to memorize this help page!

Basic Graphics Examples

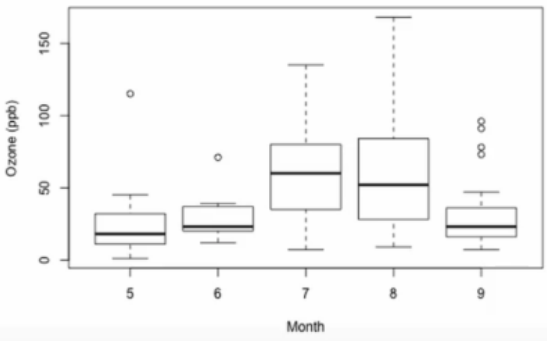
```
library(datasets)
hist(airquality$Ozone)  ## Draw a new plot
```




```
library(datasets)
with(airquality, plot(Wind, Ozone))
```




```
library(datasets)
airquality <- transform(airquality, Month = factor(Month))
boxplot(Ozone ~ Month, airquality, xlab = "Month", ylab = "Ozone (ppb)")
```



Base Graphics Parameters

Many base plotting functions share a set of parameters. Here are a few key ones:

- `pch`: the plotting symbol (default is open circle)
- `lty`: the line type (default is solid line), can be dashed, dotted, etc.
- `lwd`: the line width, specified as an integer multiple
- `col`: the plotting color, specified as a number, string, or hex code; the `colors()` function gives you a vector of colors by name
- `xlab`: character string for the x-axis label
- `ylab`: character string for the y-axis label

The `par()` function is used to specify *global* graphics parameters that affect all plots in an R session. These parameters can be overridden when specified as arguments to specific plotting functions.

- `las`: the orientation of the axis labels on the plot
- `bg`: the background color
- `mar`: the margin size
- `oma`: the outer margin size (default is 0 for all sides)
- `mfrow`: number of plots per row, column (plots are filled row-wise)
- `mfcol`: number of plots per row, column (plots are filled column-wise)

Default values for global graphics parameters

```
par("lty")
```

```
## [1] "solid"
```

```
par("col")
```

```
## [1] "black"
```

```
par("pch")
```

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

```
par("bg")
```

```
## [1] "transparent"
```

```
par("mar")
```

```
## [1] 5.1 4.1 4.1 2.1
```

```
par("mfrow")
```

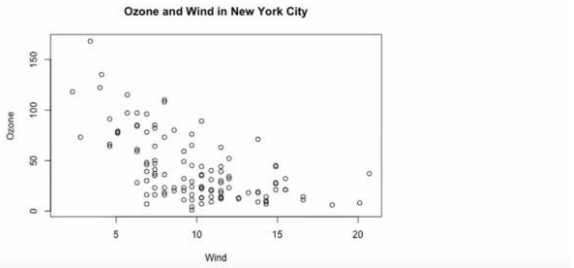
```
## [1] 1 1
```

Base Plotting Functions

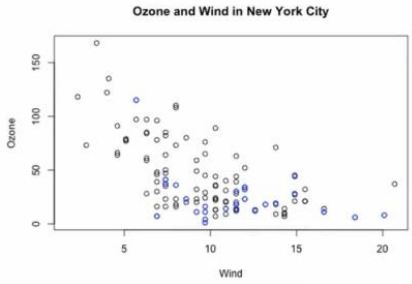
- `plot`: make a scatterplot, or other type of plot depending on the class of the object being plotted
- `lines`: add lines to a plot, given a vector x values and a corresponding vector of y values (or a 2-column matrix); this function just connects the dots
- `points`: add points to a plot
- `text`: add text labels to a plot using specified x, y coordinates
- `title`: add annotations to x, y axis labels, title, subtitle, outer margin
- `mtext`: add arbitrary text to the margins (inner or outer) of the plot
- `axis`: adding axis ticks/labels

With Annotations

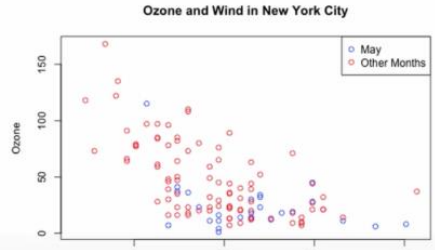
```
library(datasets)
with(airquality, plot(Wind, Ozone))
title(main = "Ozone and Wind in New York City")  ## Add a title
```




```
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City"))
with(subset(airquality, Month == 5), points(Wind, Ozone, col = "blue"))
```

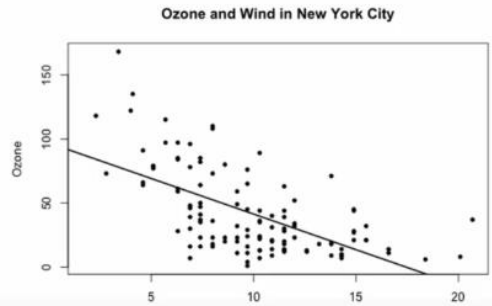



```
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City",
    type = "n"))
with(subset(airquality, Month == 5), points(Wind, Ozone, col = "blue"))
with(subset(airquality, Month != 5), points(Wind, Ozone, col = "red"))
legend("topright", pch = 1, col = c("blue", "red"), legend = c("May", "Other Months"))
```



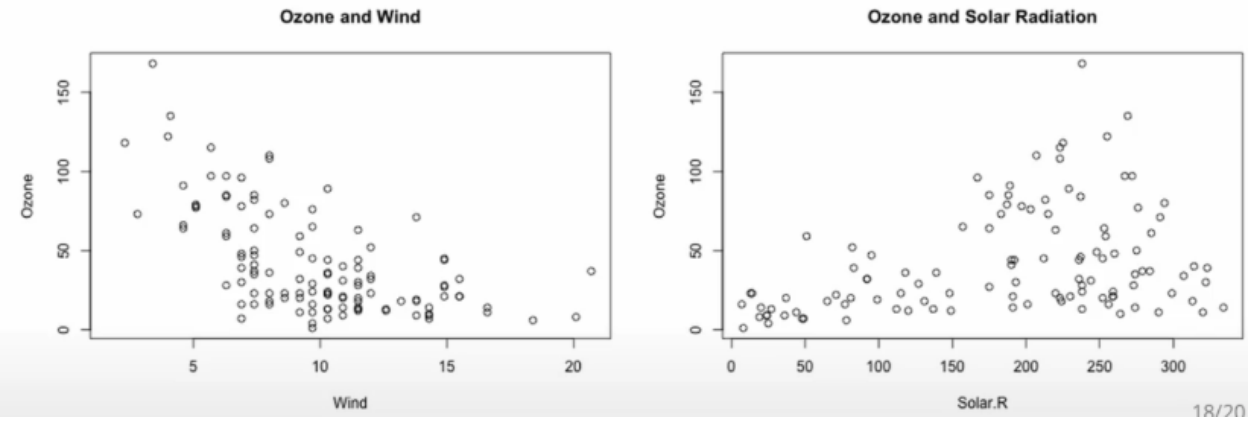
Regression Line

```
with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in New York City",
    pch = 20))
model <- lm(Ozone ~ Wind, airquality)
abline(model, lwd = 2)
```

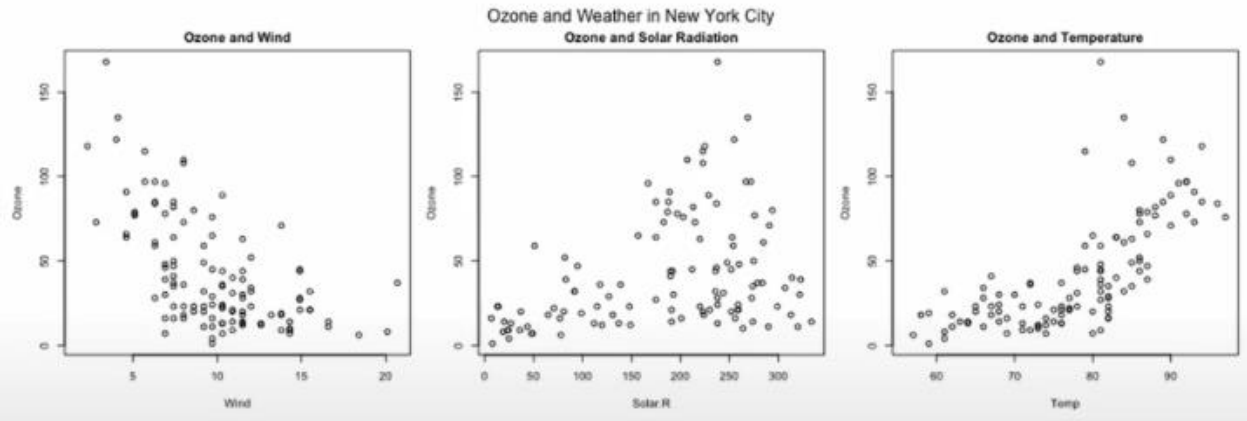


Multiple Base Plots

```r
par(mfrow = c(1, 2))
with(airquality, {
    plot(Wind, Ozone, main = "Ozone and Wind")
    plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
})
```




```r
par(mfrow = c(1, 3), mar = c(4, 4, 2, 1), oma = c(0, 0, 2, 0))
with(airquality, {
    plot(Wind, Ozone, main = "Ozone and Wind")
    plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
    plot(Temp, Ozone, main = "Ozone and Temperature")
    mtext("Ozone and Weather in New York City", outer = TRUE)
})
```



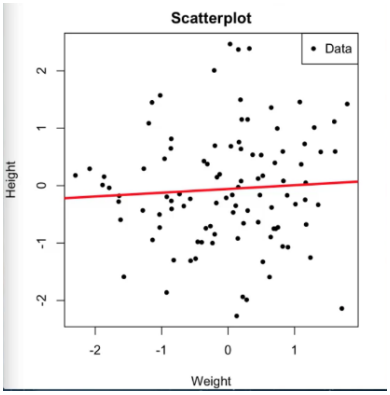
Summary

- Plots in the base plotting system are created by calling successive R functions to "build up" a plot
- Plotting occurs in two stages:
  - Creation of a plot
  - Annotation of a plot (adding lines, points, text, legends)
- The base plotting system is very flexible and offers a high degree of control over plotting

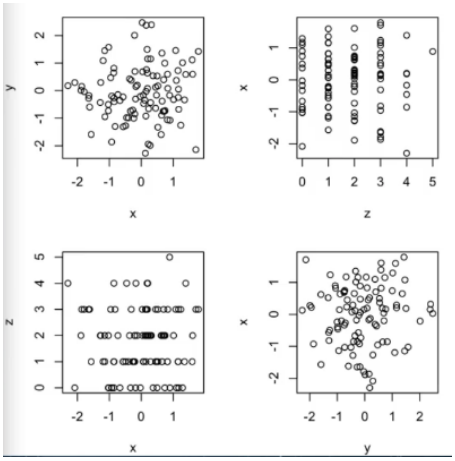
Base Plotting Demonstration

```
> x <- rnorm(100)
> hist(x)
> y <- rnorm(100)
> plot(x, y)
> z <- rnorm(100)
> plot(x, z)
> plot(x, y)
> par(mar = c(2, 2, 2, 2))
> plot(x, y)
> par(mar = c(4, 4, 2, 2))
> plot(x, y)
> plot(x, y, pch = 20)
> plot(x, y, pch = 19)
> plot(x, y, pch = 2)
> plot(x, y, pch = 3)
> plot(x, y, pch = 4)
> example(points)
```

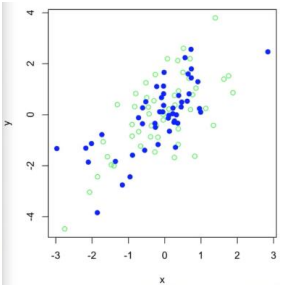
```
> x <- rnorm(100)
> y <- rnorm(100)_
Error: unexpected input in "y <- rnorm(100)_"
> y <- rnorm(100)
> plot(x, y, pch = 20)
> title("Scatterplot")
> text(-2, -2, "Label")
> legend("topleft", legend = "Data")
> legend("topleft", legend = "Data", pch = 20)
> fit <- lm(y ~ x)
> abline(fit)
> abline(fit, lwd = 3)
> abline(fit, lwd = 3, col = "blue")
> plot(x, y, xlab = "Weight", ylab = "Height", main = "Scatterplot", pch = 20)
> legend("topright", legend = "Data", pch = 20)
> fit <- lm(y ~ x)
> abline(fit, lwd = 3, col = "red")
```




```
> z <- rpois(100, 2)
> par(mfrow = c(2, 1))
> plot(x, y, pch = 20)
> plot(x, z, pch = 19)
> par("mar")
[1] 4 4 2 2
> par(mar = c(2, 2, 1, 1))
> plot(x, y, pch = 20)
> plot(x, z, pch = 20)
> par(mfrow = c(1, 2))
> plot(x, y, pch = 20)
> plot(x, z, pch = 20)
> par(mar =c(4, 4, 2, 2))
> plot(x, y, pch = 20)
> plot(x, z, pch = 20)
> par(mfrow = c(2, 2))
> plot(x, y)
> plot(x, z)
> plot(z, x)
> plot(y, x)
> par(mfcol = c(2, 2))
> plot(x, y)
> plot(x, z)
> plot(z, x)
> plot(y, x)
```



```
> par(mfrow = c(1, 1))
> x <- rnorm(100)
> y <- x + rnorm(100)
> g <- gl(2, 50)
> g <- gl(2, 50, labels = c("Male", "Female"))
> str(g)
 Factor w/ 2 levels "Male","Female": 1 1 1 1 1 1 1 1 1 1 ...
> plot(x, y)
> plot(x, y, type = "n")
> points(x[g == "Male"], y[g == "Male"], col = "green")
> points(x[g == "Female"], y[g == "Female"], col = "blue")
> points(x[g == "Female"], y[g == "Female"], col = "blue", pch = 19)
```



Graphics Devices in R

- A graphics device is something where you can make a plot appear
  - A window on your computer (screen device)
  - A PDF file (file device)
  - A PNG or JPEG file (file device)
  - A scalable vector graphics (SVG) file (file device)
- When you make a plot in R, it has to be "sent" to a specific graphics device
- The most common place for a plot to be "sent" is the *screen device*
  - On a Mac the screen device is launched with the `quartz()`
  - On Windows the screen device is launched with `windows()`
  - On Unix/Linux the screen device is launched with `x11()`
- When making a plot, you need to consider how the plot will be used to determine what device the plot should be sent to.
  - The list of devices is found in `?Devices`; there are also devices created by users on CRAN
- For quick visualizations and exploratory analysis, usually you want to use the screen device
  - Functions like `plot` in base, `xyplot` in lattice, or `qplot` in ggplot2 will default to sending a plot to the screen device
  - On a given platform (Mac, Windows, Unix/Linux) there is only one screen device
- For plots that may be printed out or be incorporated into a document (e.g. papers/reports, slide presentations), usually a *file device* is more appropriate
  - There are many different file devices to choose from
- NOTE: Not all graphics devices are available on all platforms (i.e. you cannot launch the `windows()` on a Mac)

How does a plot get created?

There are two basic approaches to plotting. The first is most common:

1. Call a plotting function like `plot`, `xyplot`, or `qplot`
2. The plot appears on the screen device
3. Annotate plot if necessary
4. Enjoy

```
library(datasets)
with(faithful, plot(eruptions, waiting))  ## Make plot appear on screen device
title(main = "Old Faithful Geyser data")  ## Annotate with a title
```

The second approach to plotting is most commonly used for file devices:

1. Explicitly launch a graphics device
2. Call a plotting function to make a plot (Note: if you are using a file device, no plot will appear on the screen)
3. Annotate plot if necessary
4. Explicitly close graphics device with `dev.off()` (this is very important!)

```
pdf(file = "myplot.pdf")  ## Open PDF device; create 'myplot.pdf' in my working directory
## Create plot and send to a file (no plot appears on screen)
with(faithful, plot(eruptions, waiting))
title(main = "Old Faithful Geyser data")  ## Annotate plot; still nothing on screen
dev.off()  ## Close the PDF file device
## Now you can view the file 'myplot.pdf' on your computer
```

Graphics File Devices

There are two basic types of file devices: *vector* and *bitmap* devices

Vector formats:

- `pdf`: useful for line-type graphics, resizes well, usually portable, not efficient if a plot has many objects/points
- `svg`: XML-based scalable vector graphics; supports animation and interactivity, potentially useful for web-based plots
- `win.metafile`: Windows metafile format (only on Windows)
- `postscript`: older format, also resizes well, usually portable, can be used to create encapsulated postscript files; Windows systems often don't have a postscript viewer

Bitmap formats

- `png`: bitmapped format, good for line drawings or images with solid colors, uses lossless compression (like the old GIF format), most web browsers can read this format natively, good for plotting many many many points, does not resize well
- `jpeg`: good for photographs or natural scenes, uses lossy compression, good for plotting many many many points, does not resize well, can be read by almost any computer and any web browser, not great for line drawings
- `tiff`: Creates bitmap files in the TIFF format; supports lossless compression
- `bmp`: a native Windows bitmapped format

Multiple Open Graphics

- It is possible to open multiple graphics devices (screen, file, or both), for example when viewing multiple plots at once
- Plotting can only occur on one graphics device at a time
- The **currently active** graphics device can be found by calling `dev.cur()`
- Every open graphics device is assigned an integer ≥ 2.
- You can change the active graphics device with `dev.set(<integer>)` where `<integer>` is the number associated with the graphics device you want to switch to

Copying Plots

Copying a plot to another device can be useful because some plots require a lot of code and it can be a pain to type all that in again for a different device.

- `dev.copy`: copy a plot from one device to another
- `dev.copy2pdf`: specifically copy a plot to a PDF file

NOTE: Copying a plot is not an exact operation, so the result may not be identical to the original.

```
library(datasets)
with(faithful, plot(eruptions, waiting))  ## Create plot on screen device
title(main = "Old Faithful Geyser data")  ## Add a main title
dev.copy(png, file = "geyserplot.png")  ## Copy my plot to a PNG file
dev.off()  ## Don't forget to close the PNG device!
```

Summary

- Plots must be created on a graphics device
- The default graphics device is almost always the screen device, which is most useful for exploratory analysis
- File devices are useful for creating plots that can be included in other documents or sent to other people
- For file devices, there are vector and bitmap formats
  - Vector formats are good for line drawings and plots with solid colors using a modest number of points
  - Bitmap formats are good for plots with a large number of points, natural scenes or web-based plots

Lattice Plotting System

The lattice plotting system is implemented using the following packages:

- *lattice*: contains code for producing Trellis graphics, which are independent of the "base" graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`
- *grid*: implements a different graphing system independent of the "base" system; the *lattice* package builds on top of *grid*
    - We seldom call functions from the *grid* package directly
- The lattice plotting system does not have a "two-phase" aspect with separate plotting and annotation like in base plotting
- All plotting/annotation is done at once with a single function call

Lattice Functions

- `xyplot`: this is the main function for creating scatterplots
- `bwplot`: box-and-whiskers plots ("boxplots")
- `histogram`: histograms
- `stripplot`: like a boxplot but with actual points
- `dotplot`: plot dots on "violin strings"
- `splom`: scatterplot matrix; like `pairs` in base plotting system
- `levelplot`, `contourplot`: for plotting "image" data

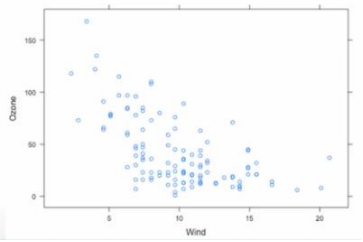
Lattice functions generally take a formula for their first argument, usually of the form

```
xyplot(y ~ x | f * g, data)
```

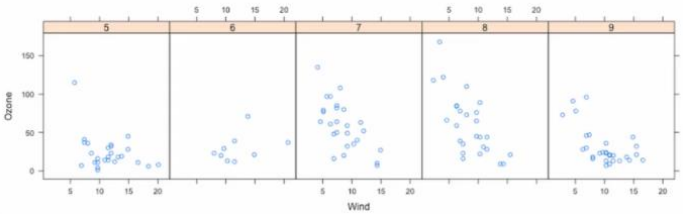
- We use the *formula notation* here, hence the ~.
- On the left of the ~ is the y-axis variable, on the right is the x-axis variable
- f and g are *conditioning variables* — they are optional
    - the * indicates an interaction between two variables
- The second argument is the data frame or list from which the variables in the formula should be looked up
    - If no data frame or list is passed, then the parent frame is used.
- If no other arguments are passed, there are defaults that can be used.

Simple Lattice Plot

```
library(lattice)
library(datasets)
## Simple scatterplot
xyplot(Ozone ~ Wind, data = airquality)
```




```
library(datasets)
library(lattice)
## Convert 'Month' to a factor variable
airquality <- transform(airquality, Month = factor(Month))
xyplot(Ozone ~ Wind | Month, data = airquality, layout = c(5, 1))
```

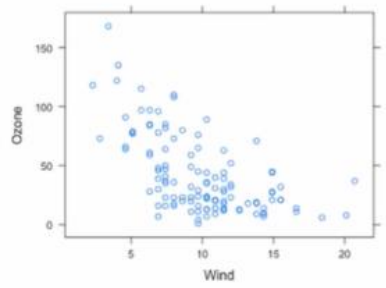


Lattice Behavior

Lattice functions behave differently from base graphics functions in one critical way.

- Base graphics functions plot data directly to the graphics device (screen, PDF file, etc.)
- Lattice graphics functions return an object of class **trellis**
- The print methods for lattice functions actually do the work of plotting the data on the graphics device.
- Lattice functions return "plot objects" that can, in principle, be stored (but it's usually better to just save the code + data).
- On the command line, trellis objects are *auto-printed* so that it appears the function is plotting the data

```
p <- xyplot(Ozone ~ Wind, data = airquality)  ## Nothing happens!
print(p)  ## Plot appears
```



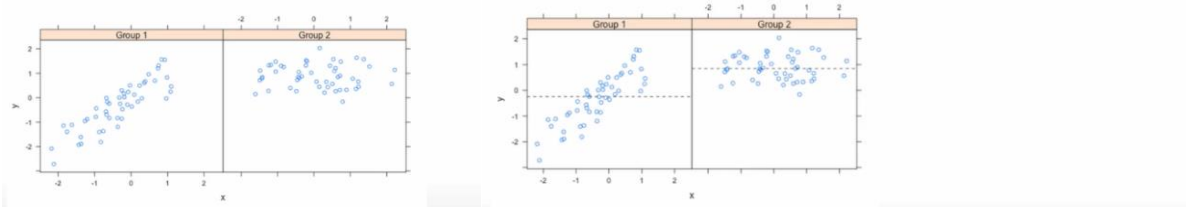

```
xyplot(Ozone ~ Wind, data = airquality)  ## Auto-printing
```

Lattice Panel Functions

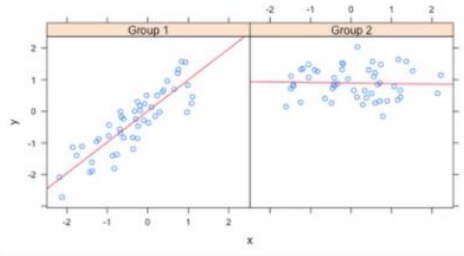
- Lattice functions have a **panel function** which controls what happens inside each panel of the plot.
- The *lattice* package comes with default panel functions, but you can supply your own if you want to customize what happens in each panel
- Panel functions receive the x/y coordinates of the data points in their panel (along with any optional arguments)

```
set.seed(10)
x <- rnorm(100)
f <- rep(0:1, each = 50)
y <- x + f - f * x + rnorm(100, sd = 0.5)
f <- factor(f, labels = c("Group 1", "Group 2"))
xyplot(y ~ x | f, layout = c(2, 1))  ## Plot with 2 panels
```

```
## Custom panel function
xyplot(y ~ x | f, panel = function(x, y, ...) {
    panel.xyplot(x, y, ...)  ## First call the default panel function for 'xyplot'
    panel.abline(h = median(y), lty = 2)  ## Add a horizontal line at the median
})
```



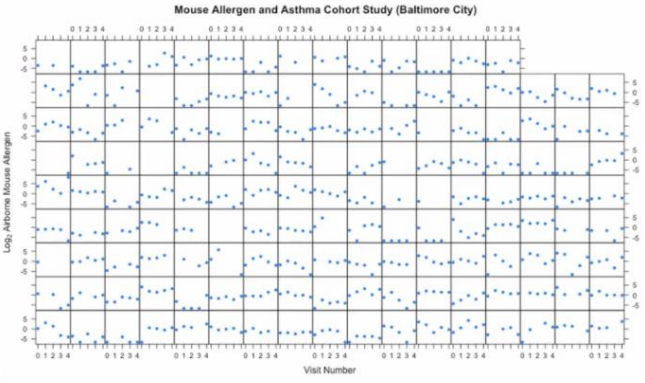

```
## Custom panel function
xyplot(y ~ x | f, panel = function(x, y, ...) {
    panel.xyplot(x, y, ...)  ## First call default panel function
    panel.lmline(x, y, col = 2)  ## Overlay a simple linear regression line
})
```



Example: Many Panel Lattice Plot

- Study: Mouse Allergen and Asthma Cohort Study (MAACS)
- Study subjects: Children with asthma living in Baltimore City, many allergic to mouse allergen
- Design: Observational study, baseline home visit + every 3 months for a year.
- Question: How does indoor airborne mouse allergen vary over time and across subjects?

Ahluwalia et al., *Journal of Allergy and Clinical Immunology*, 2013



Summary

- Lattice plots are constructed with a single function call to a core lattice function (e.g. `xyplot`)
- Aspects like margins and spacing are automatically handled and defaults are usually sufficient
- The lattice system is ideal for creating conditioning plots where you examine the same kind of plot under many different conditions
- Panel functions can be specified/customized to modify what is plotted in each of the plot panels

Ggplot2 Plotting System

- An implementation of the *Grammar of Graphics* by Leland Wilkinson
- Written by Hadley Wickham (while he was a graduate student at Iowa State)
- A "third" graphics system for R (along with **base** and **lattice**)
- Available from CRAN via `install.packages()`
- Web site: http://ggplot2.org (better documentation)

- Grammar of graphics represents and abstraction of graphics ideas/objects
- Think "verb", "noun", "adjective" for graphics
- Allows for a "theory" of graphics on which to build new graphics and graphics objects
- "Shorten the distance from mind to page"

> "In brief, the grammar tells us that a statistical graphic is a **mapping** from data to **aesthetic** attributes (colour, shape, size) of **geometric** objects (points, lines, bars). The plot may also contain statistical transformations of the data and is drawn on a specific coordinate system"

Basics: qplot()

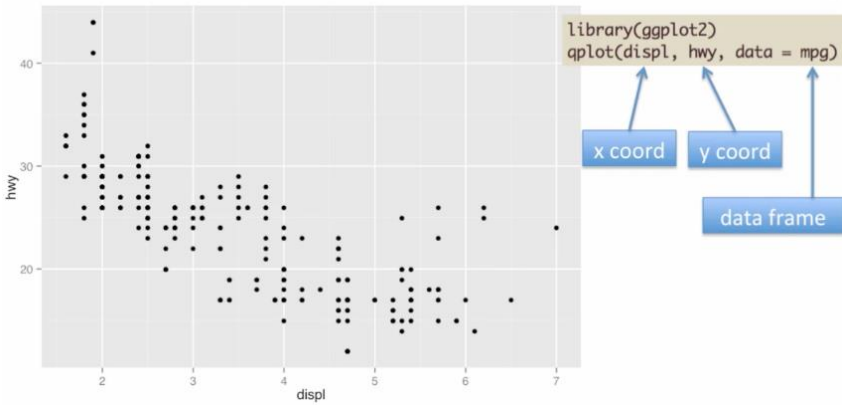
- Works much like the `plot` function in base graphics system
- Looks for data in a data frame, similar to lattice, or in the parent environment
- Plots are made up of *aesthetics* (size, shape, color) and *geoms* (points, lines)

- Factors are important for indicating subsets of the data (if they are to have different properties); they should be **labeled**
- The qplot() hides what goes on underneath, which is okay for most operations
- ggplot() is the core function and very flexible for doing things qplot() cannot do

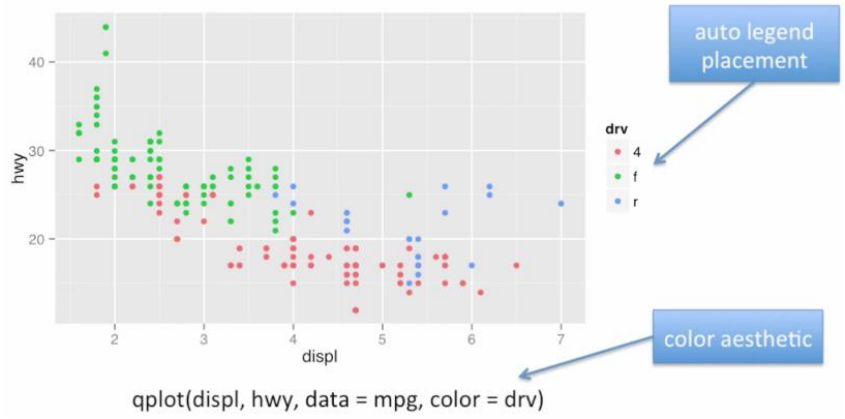
Example Dataset

```
> library(ggplot2)
> str(mpg)
'data.frame':   234 obs. of  11 variables:
 $ manufacturer: Factor w/ 15 levels "audi","chevrolet",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ model       : Factor w/ 38 levels "4runner 4wd",..: 2 2 2 2 2 2 2 3 3 3 ...
 $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
 $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
 $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
 $ trans       : Factor w/ 10 levels "auto(av)","auto(l3)",..: 4 9 10 1 4 9 1 9 4 10
...
 $ drv         : Factor w/ 3 levels "4","f","r": 2 2 2 2 2 2 2 1 1 1 ...
 $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
 $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
 $ fl          : Factor w/ 5 levels "c","d","e","p",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ class       : Factor w/ 7 levels "2seater","compact",..: 2 2 2 2 2 2 2 2 2 2 ...
```

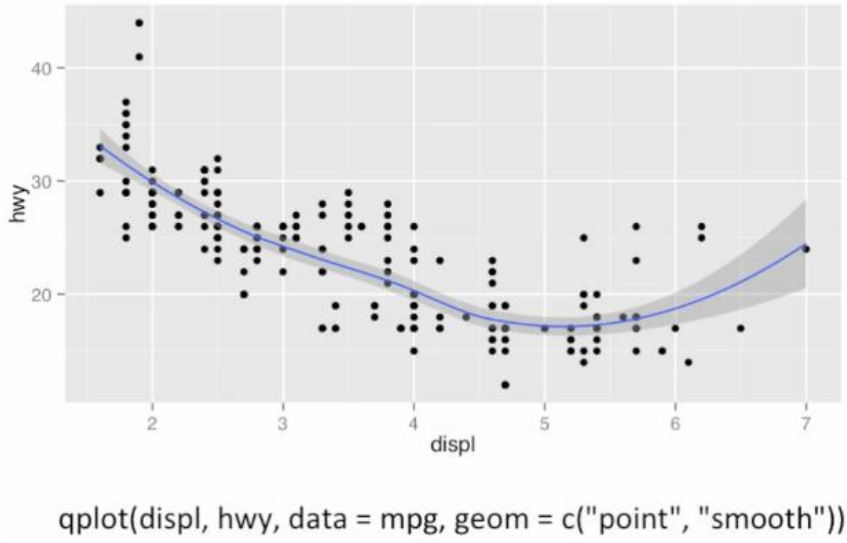
Ggplot2 “Hello, world!”



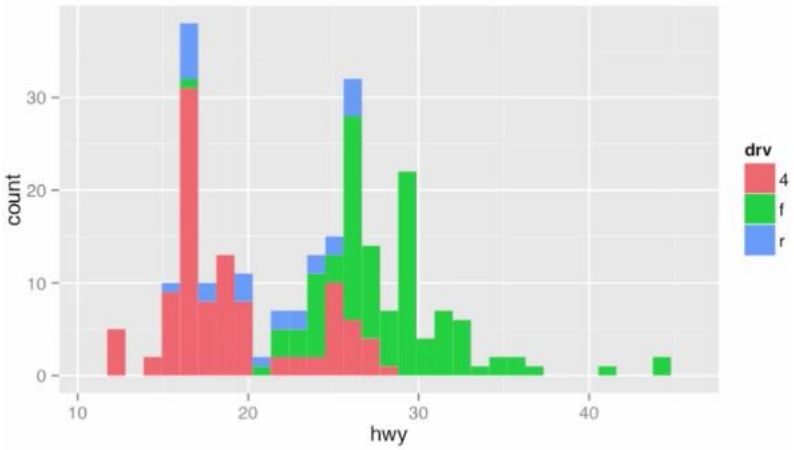
Modifying Aesthetics



Adding a Geom



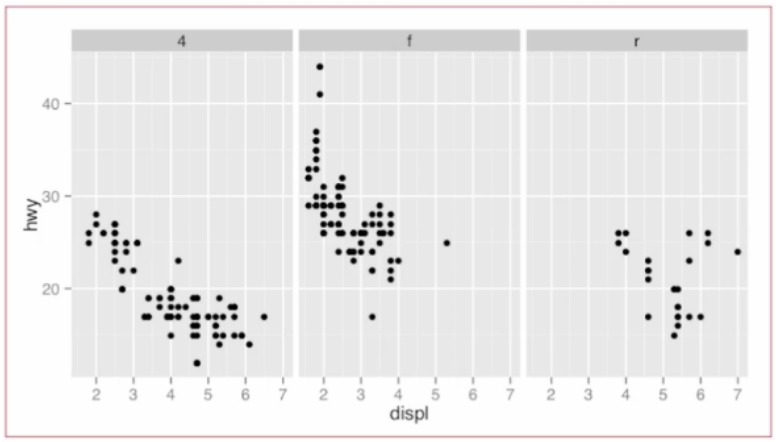
Histograms

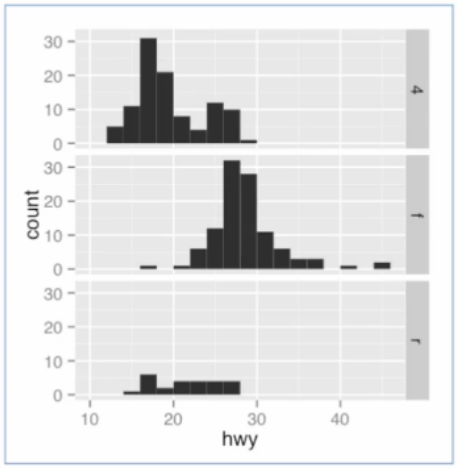


qplot(hwy, data = mpg, fill = drv)

Facets

qplot(displ, hwy, data = mpg, facets = . ~ drv)





qplot(hwy, data = mpg, facets = drv ~ ., binwidth = 2)

MAACS Cohort

- Mouse Allergen and Asthma Cohort Study
- Baltimore children (aged 5—17)
- Persistent asthma, exacerbation in past year
- Study indoor environment and its relationship with asthma morbidity
- Recent publication: http://goo.gl/WqE9j8

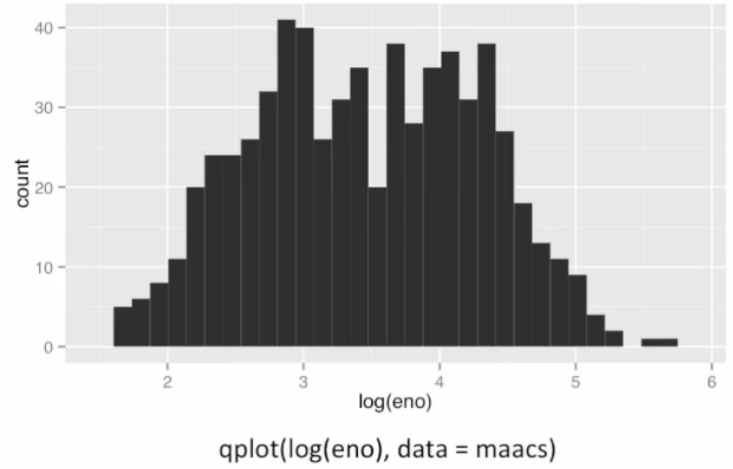
Example MAACS

```
> str(maacs)
'data.frame':   750 obs. of  5 variables:
 $ id       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ eno      : num  141 124 126 164 99 68 41 50 12 30 ...
 $ duBedMusM: num  2423 2793 3055 775 1634 ...
 $ pm25     : num  15.6 34.4 39 33.2 27.1 ...
 $ mopos    : Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...
```

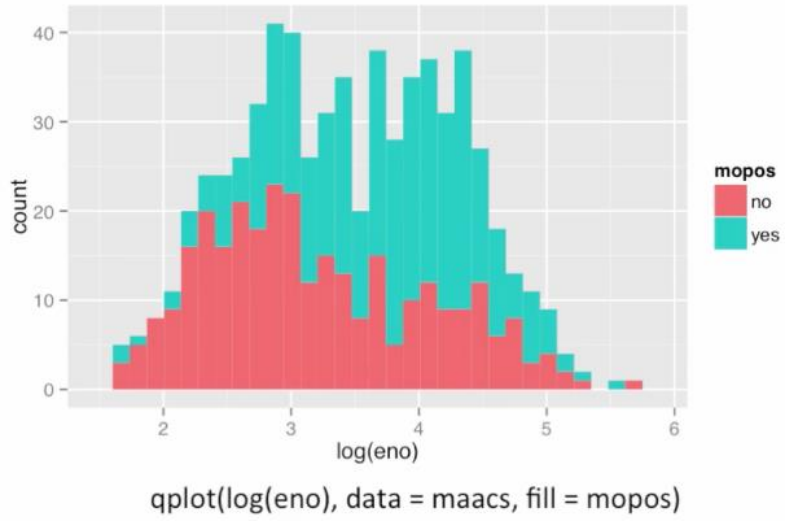
Fine particulate matter

Sensitized to mouse allergen

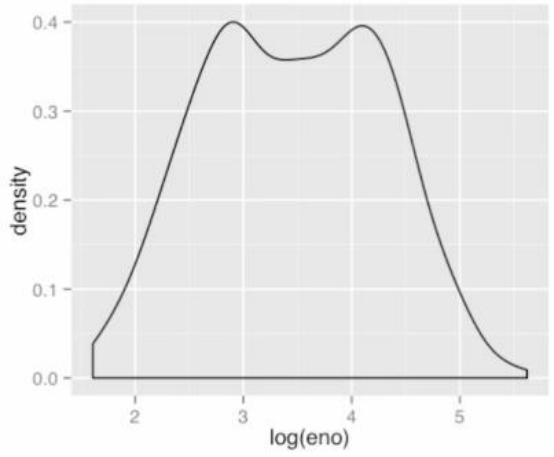
Histogram of eNO



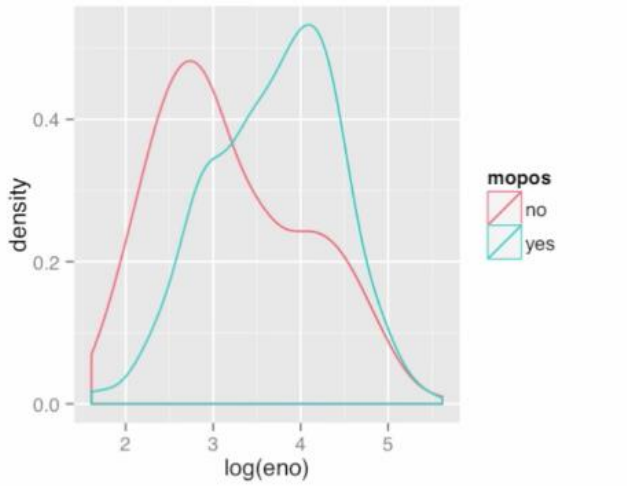
Histogram by Group



Density Smooth

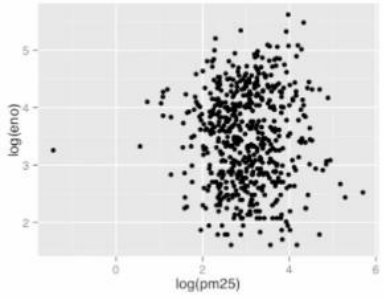


qplot(log(eno), data = maacs, geom = "density")

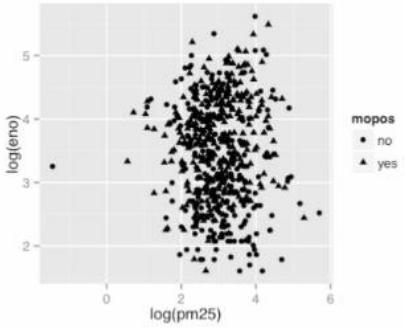


qplot(log(eno), data = maacs, geom = "density", color = mopos)

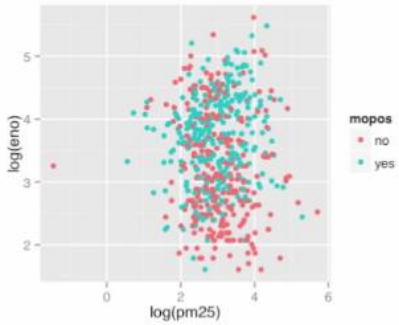
Scatterplots: eNO vs. PM2.5



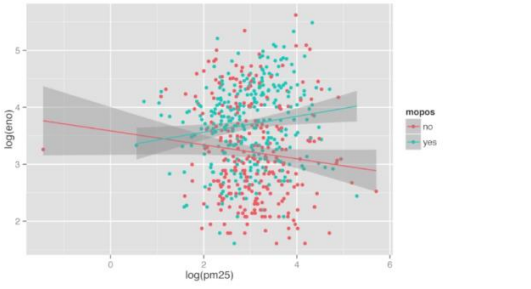
qplot(log(pm25), log(eno), data = maacs)



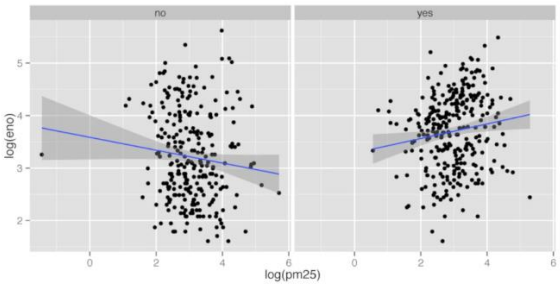
qplot(log(pm25), log(eno), data = maacs, shape = mopos)



qplot(log(pm25), log(eno), data = maacs, color = mopos)



qplot(log(pm25), log(eno), data = maacs, color = mopos) + geom_smooth(method = "lm")



qplot(log(pm25), log(eno), data = maacs, facets = . ~ mopos) + geom_smooth(method = "lm")

Summary of qplot()

- The qplot() function is the analog to plot() but with many built-in features
- Syntax somewhere in between base/lattice
- Produces very nice graphics, essentially publication ready (if you like the design)
- Difficult to go against the grain/customize (don't bother; use full ggplot2 power in that case)

Basic Components of ggplot2 Plot

- A **data frame**
- **aesthetic mappings**: how data are mapped to color, size
- **geoms**: geometric objects like points, lines, shapes.
- **facets**: for conditional plots.
- **stats**: statistical transformations like binning, quantiles, smoothing.
- **scales**: what scale an aesthetic map uses (example: male = red, female = blue).
- **coordinate system**

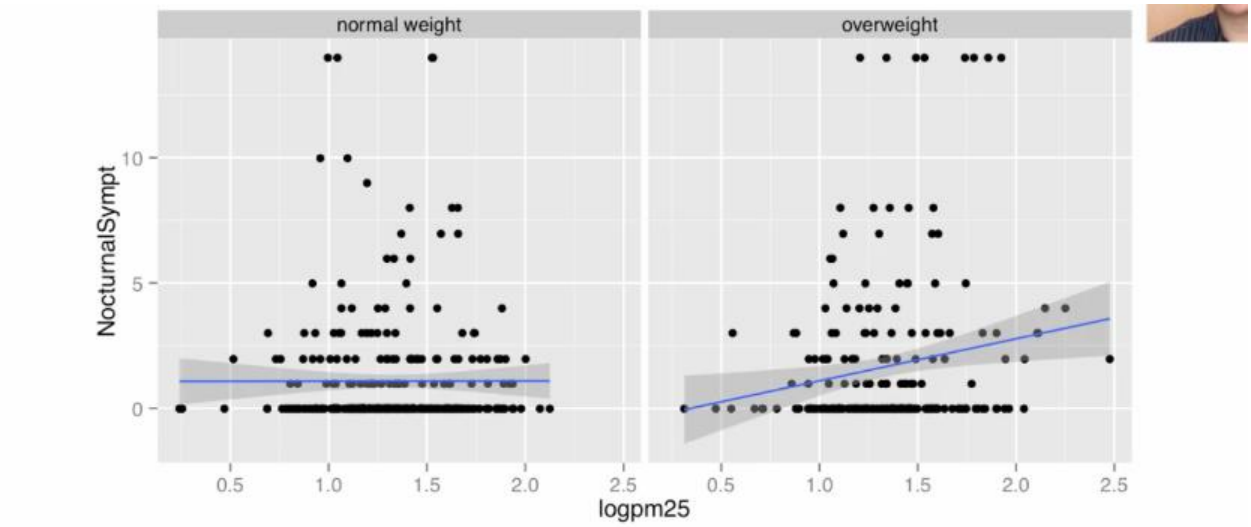
Building Plots with ggplot2

- When building plots in ggplot2 (rather than using qplot) the "artist's palette" model may be the closest analogy
- Plots are built up in layers
  - Plot the data
  - Overlay a summary
  - Metadata and annotation

Example: BMI, PM2.5, Asthma

- Mouse Allergen and Asthma Cohort Study
- Baltimore children (age 5-17)
- Persistent asthma, exacerbation in past year
- Does BMI (normal vs. overweight) modify the relationship between $PM_{2.5}$ and asthma symptoms?

Basic Plot




```
qplot(logpm25, NocturnalSympt, data = maacs, facets = . ~ bmicat, geom =
c("point", "smooth"), method = "lm")
```

Building Up in Layers

```
> head(maacs[, 1:3])
    logpm25        bmicat NocturnalSympt
2 1.5361795 normal weight              1
3 1.5905409 normal weight              0
4 1.5217786 normal weight              0
5 1.4323277 normal weight              0
6 1.2762320    overweight              8
8 0.7139103    overweight              0

> g <- ggplot(maacs, aes(logpm25, NocturnalSympt))

> summary(g)
data: logpm25, bmicat, NocturnalSympt [554x3]
mapping:  x = logpm25, y = NocturnalSympt
faceting: facet_null()
```

Data Frame

Aesthetics

Initial call to ggplot

Summary of ggplot object

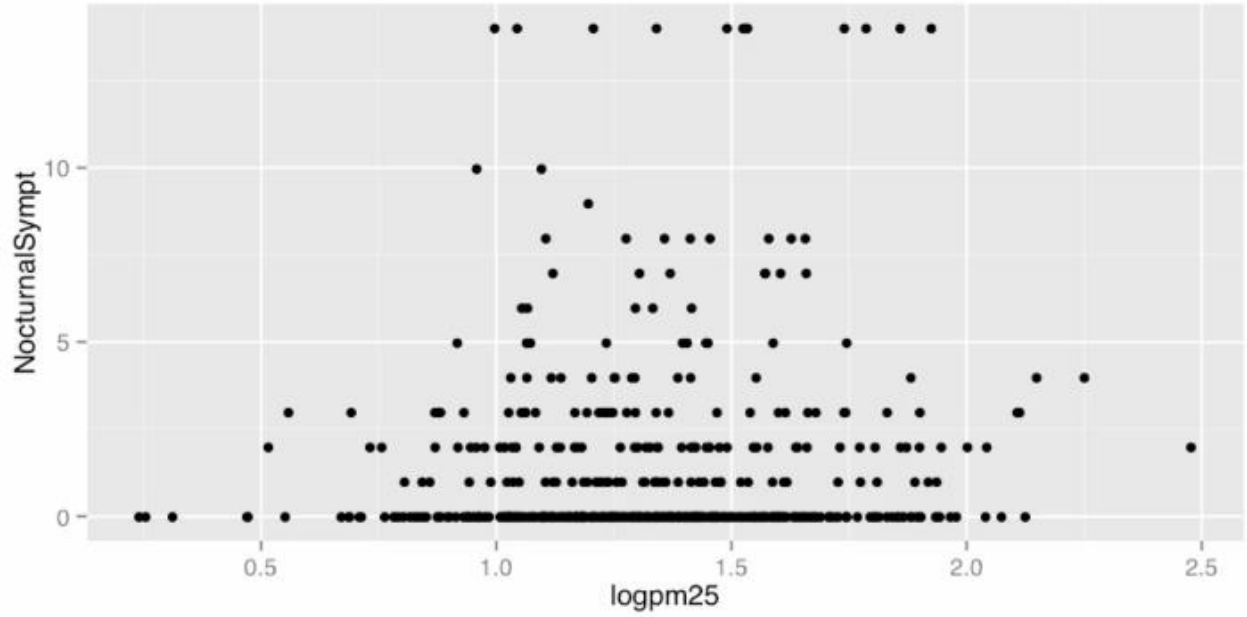
No Plot Yet!

```
> g <- ggplot(maacs, aes(logpm25, NocturnalSympt))
> print(g)
Error: No layers in plot

> p <- g + geom_point()
> print(p)

> g + geom_point()
```

Explicitly save and print ggplot object

Auto-print plot object without saving

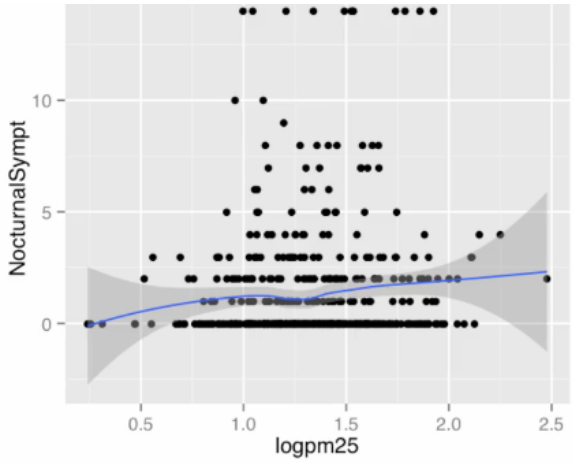
Plot with Point Layer



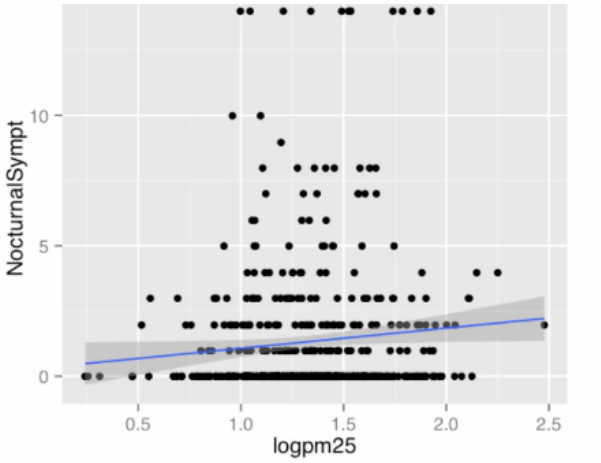

```
g <- ggplot(maacs, aes(logpm25, NocturnalSympt))
g + geom_point()
```

Adding More Layers

Smooth

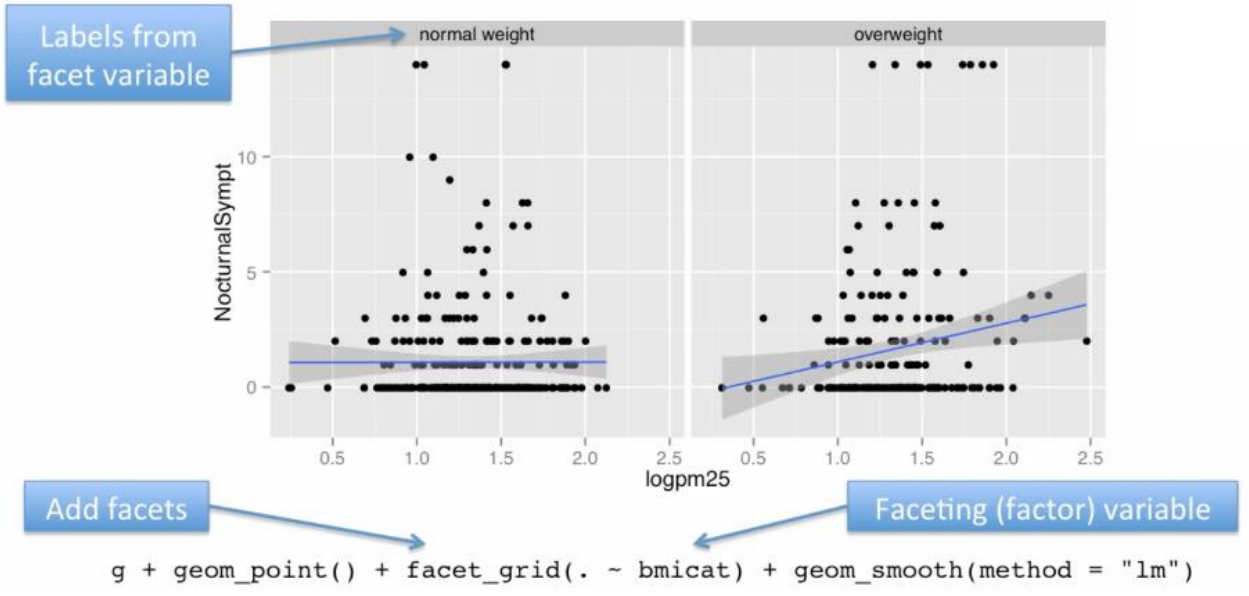



```
g + geom_point() + geom_smooth()
```




```
g + geom_point() + geom_smooth(method = "lm")
```

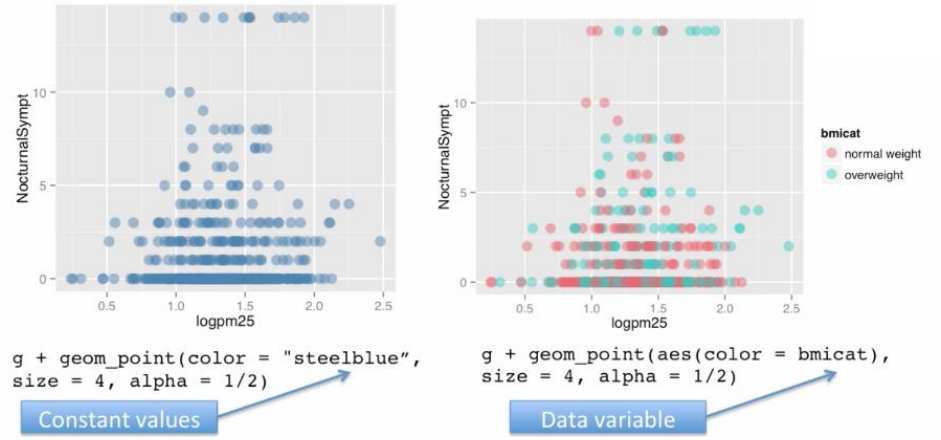
Facets



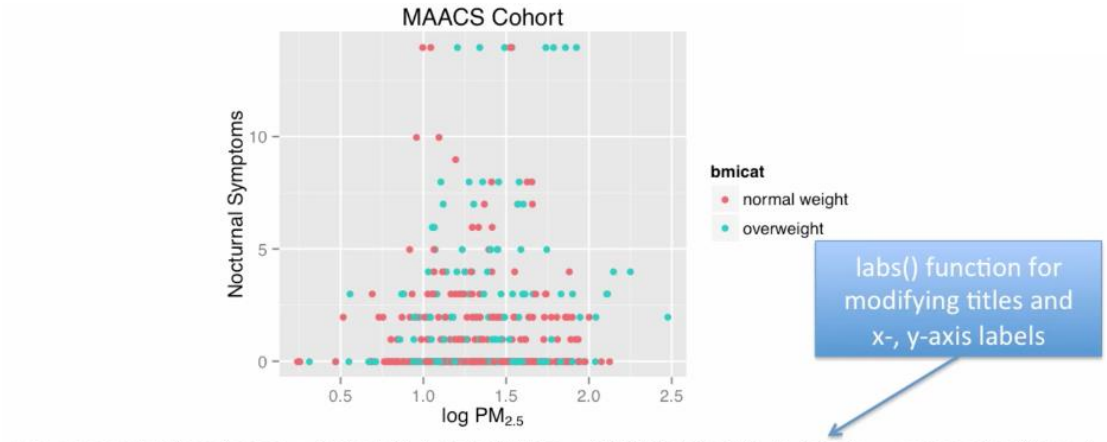
Annotation

- Labels: xlab(), ylab(), labs(), ggtitle()
- Each of the "geom" functions has options to modify
- For things that only make sense globally, use theme()
  - Example: theme(legend.position = "none")
- Two standard appearance themes are included
  - theme_gray(): The default theme (gray background)
  - theme_bw(): More stark/plain

Modifying Aesthetics

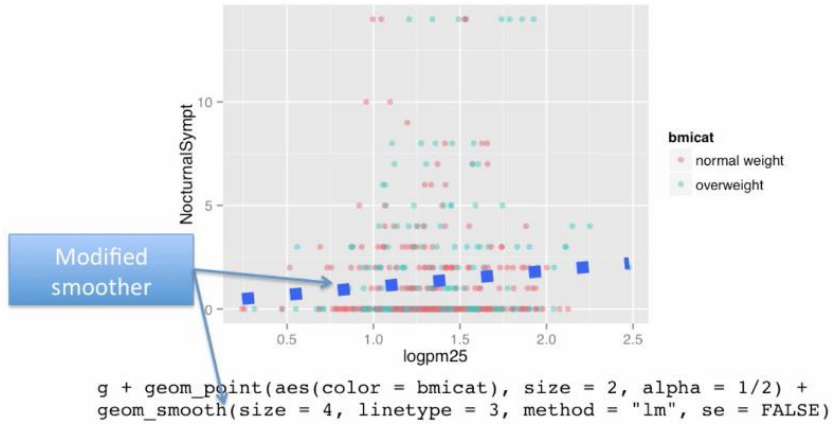


Modifying Labels



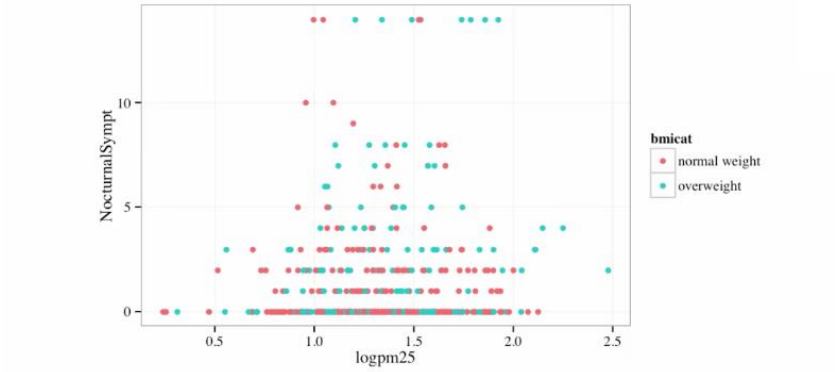

```
g + geom_point(aes(color = bmicat)) + labs(title = "MAACS Cohort") + labs(x = expression("log "
* PM[2.5]), y = "Nocturnal Symptoms")
```

Customizing the Smooth



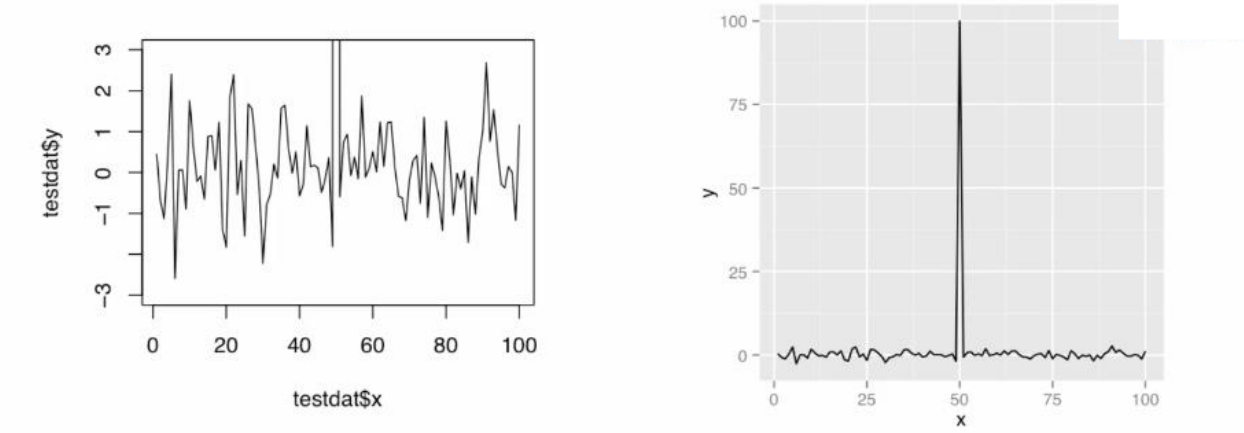

```
g + geom_point(aes(color = bmicat), size = 2, alpha = 1/2) +
geom_smooth(size = 4, linetype = 3, method = "lm", se = FALSE)
```

Changing the Theme



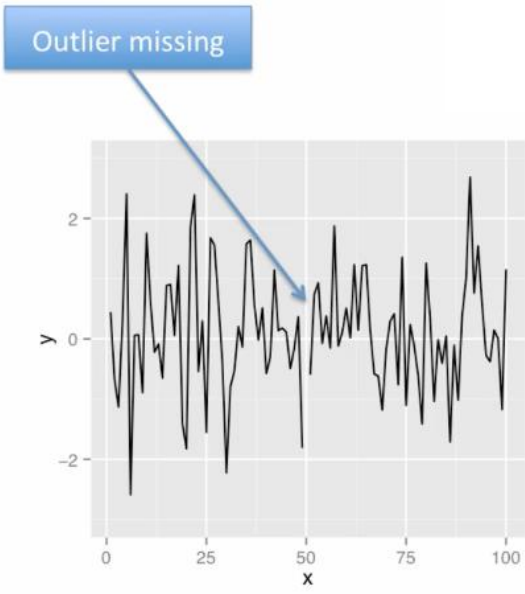

```
g + geom_point(aes(color = bmicat)) + theme_bw(base_family = "Times")
```

Notes about Axis Limits

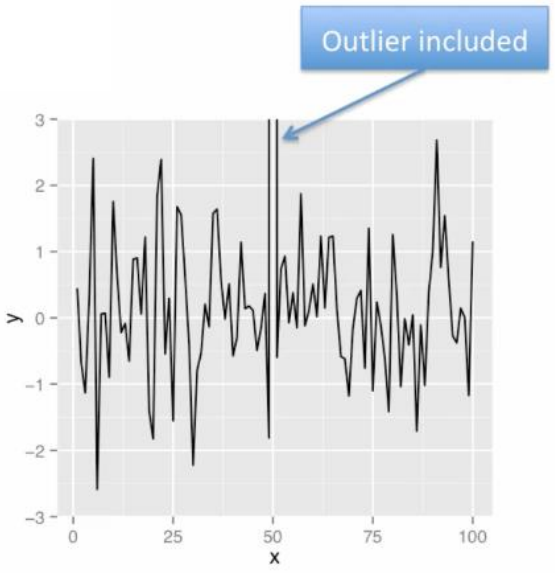



```
testdat <- data.frame(x = 1:100, y = rnorm(100))
testdat[50,2] <- 100  ## Outlier!
plot(testdat$x, testdat$y, type = "l", ylim = c(-3,3))
```

```
g <- ggplot(testdat, aes(x = x, y = y))
g + geom_line()
```




```
g + geom_line() + ylim(-3, 3)
```




```
g + geom_line() + coord_cartesian(ylim = c(-3, 3))
```

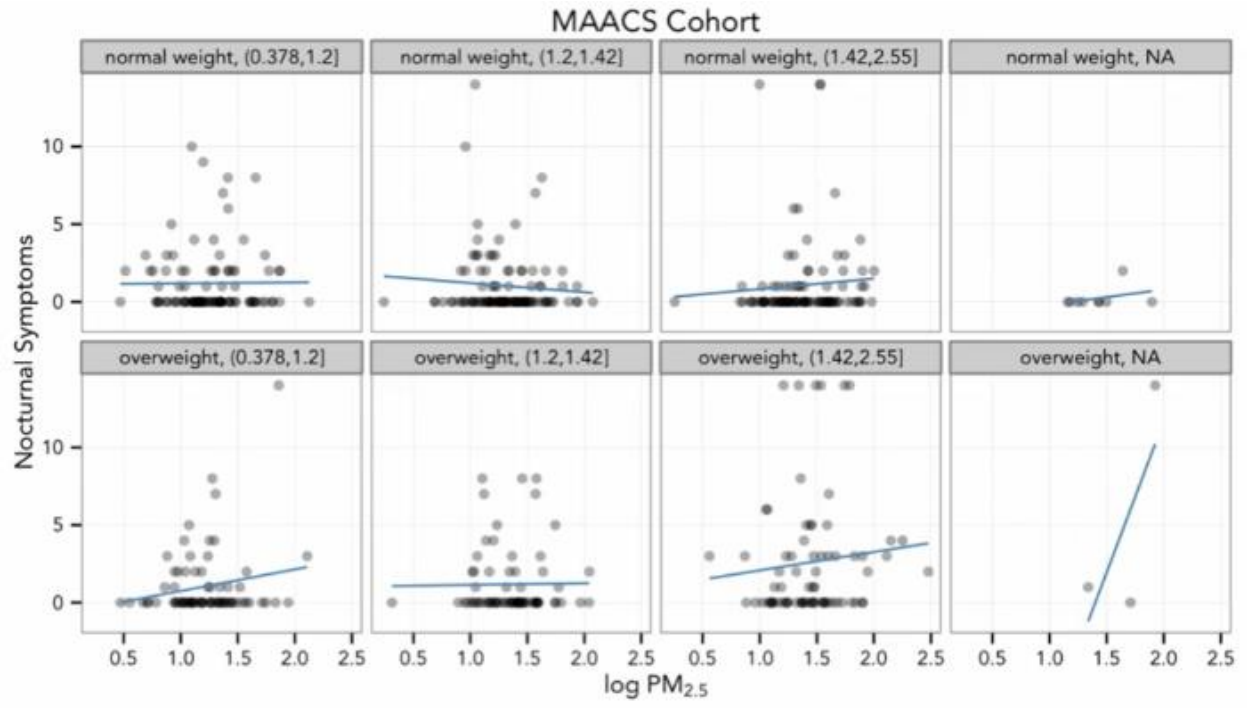
More Complex Example

- How does the relationship between $PM_{2.5}$ and nocturnal symptoms vary by BMI and NO_2?
- Unlike our previous BMI variable, NO_2 is continuous
- We need to make NO2 categorical so we can condition on it in the plotting
  - Use the cut() function for this

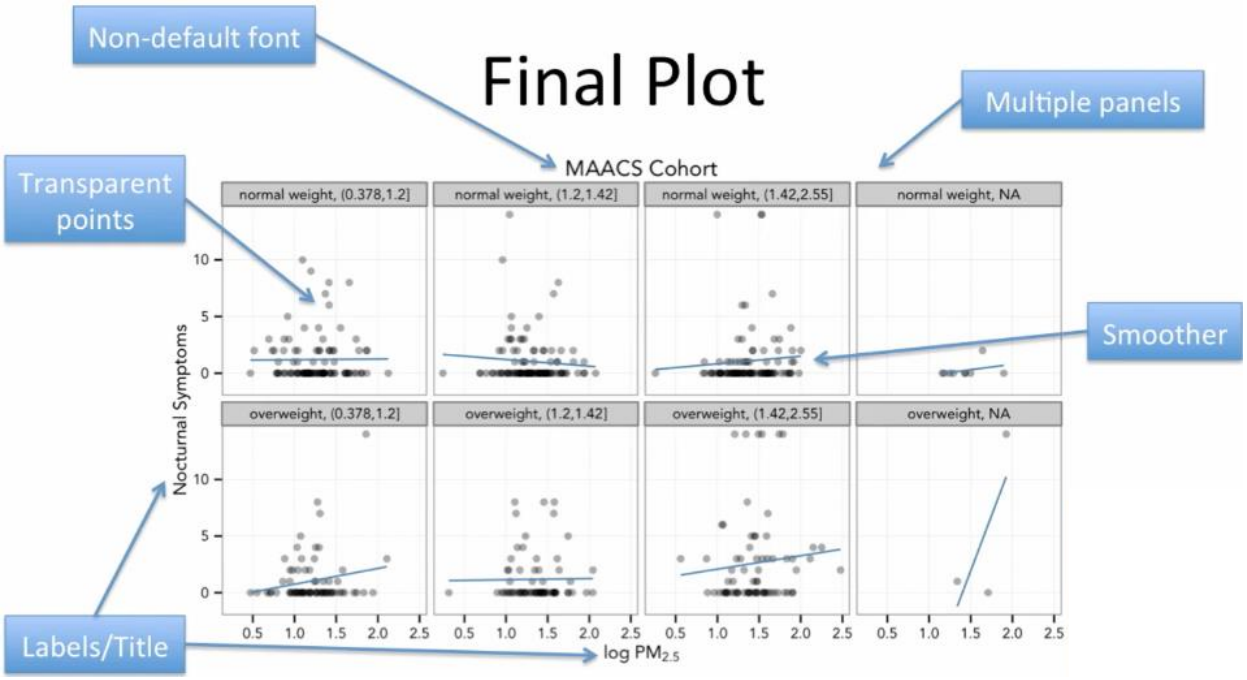
Making NO2 Tertiles

```
## Calculate the deciles of the data
> cutpoints <- quantile(maacs$logno2_new, seq(0, 1, length = 4), na.rm = TRUE)

## Cut the data at the deciles and create a new factor variable
> maacs$no2dec <- cut(maacs$logno2_new, cutpoints)

## See the levels of the newly created factor variable
> levels(maacs$no2dec)
[1] "(0.378,1.2]" "(1.2,1.42]"  "(1.42,2.55]"
```

Final Plot



Code for Final Plot




```
## Setup ggplot with data frame
g <- ggplot(maacs, aes(logpm25, NocturnalSympt))

## Add layers
g + geom_point(alpha = 1/3)
  + facet_wrap(bmicat ~ no2dec, nrow = 2, ncol = 4)
  + geom_smooth(method="lm", se=FALSE, col="steelblue")
  + theme_bw(base_family = "Avenir", base_size = 10)
  + labs(x = expression("log " * PM[2.5]))
  + labs(y = "Nocturnal Symptoms")
  + labs(title = "MAACS Cohort")
```

Add points

Add smoother

Make panels

Change theme

Add labels

Ggplot2 Summary

- ggplot2 is very powerful and flexible if you learn the "grammar" and the various elements that can be tuned/modified
- Many more types of plots can be made; explore and mess around with the package (references mentioned in Part 1 are useful)

Hierarchical Clustering

Clustering organizes things that are **close** into groups

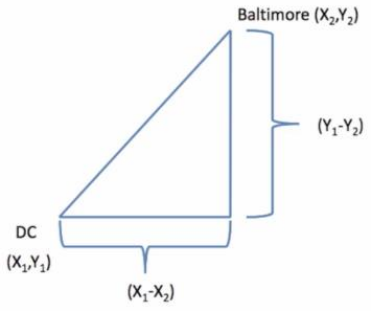
- How do we define close?
- How do we group things?
- How do we visualize the grouping?
- How do we interpret the grouping?

- An agglomerative approach
  - Find closest two things
  - Put them together
  - Find next closest
- Requires
  - A defined distance
  - A merging approach
- Produces
  - A tree showing how close things are to each other

How do we define close?

- Most important step
  - Garbage in -> garbage out
- Distance or similarity
  - Continuous - euclidean distance
  - Continuous - correlation similarity
  - Binary - manhattan distance
- Pick a distance/similarity that makes sense for your problem

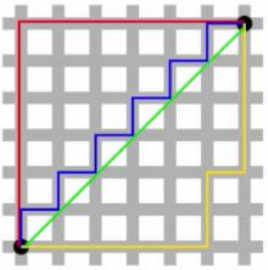
Euclidean Distance



In general:

$$\sqrt{(A_1 - A_2)^2 + (B_1 - B_2)^2 + \ldots + (Z_1 - Z_2)^2}$$

Manhattan (Taxicab) Distance

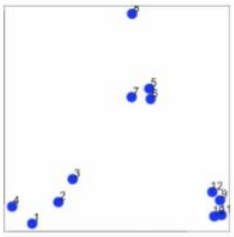


In general:

$$|A_1 - A_2| + |B_1 - B_2| + \ldots + |Z_1 - Z_2|$$

Example: Hierarchical Clustering

```
set.seed(1234)
par(mar = c(0, 0, 0, 0))
x <- rnorm(12, mean = rep(1:3, each = 4), sd = 0.2)
y <- rnorm(12, mean = rep(c(1, 2, 1), each = 4), sd = 0.2)
plot(x, y, col = "blue", pch = 19, cex = 2)
text(x + 0.05, y + 0.05, labels = as.character(1:12))
```



Dist. Function (Euclidean by Default)

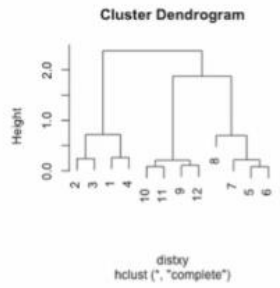
- Important parameters: *x,method*

```
dataFrame <- data.frame(x = x, y = y)
dist(dataFrame)
```

```
##          1       2       3       4       5       6       7       8       9
## 2  0.34121
## 3  0.57494 0.24103
## 4  0.26382 0.52579 0.71862
## 5  1.69425 1.35818 1.11953 1.80667
## 6  1.65813 1.31960 1.08339 1.78081 0.08150
## 7  1.49823 1.16621 0.92569 1.60132 0.21110 0.21667
## 8  1.99149 1.69093 1.45649 2.02849 0.61704 0.69792 0.65063
## 9  2.13630 1.83168 1.67836 2.35676 1.18350 1.11500 1.28583 1.76461
## 10 2.06420 1.76999 1.63110 2.29239 1.23848 1.16550 1.32063 1.83518 0.14090
## 11 2.14702 1.85183 1.71074 2.37462 1.28154 1.21077 1.37370 1.86999 0.11624
## 12 2.05664 1.74663 1.58659 2.27232 1.07701 1.00777 1.17740 1.66224 0.10849
##         10      11
```

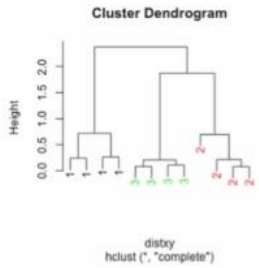
Hclust function- Dendrogram

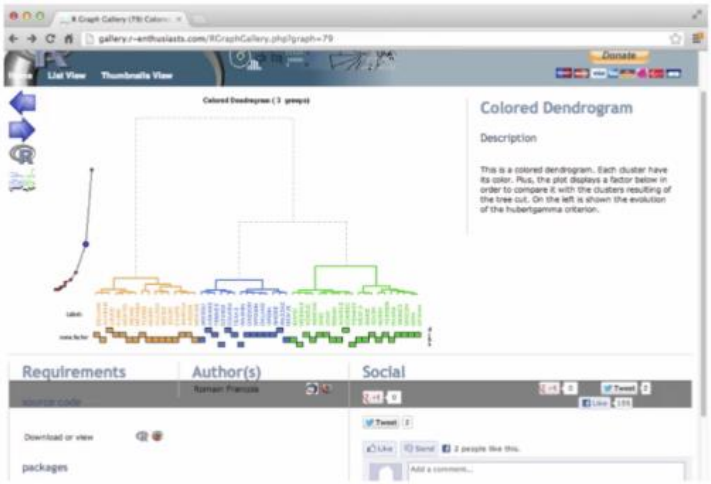
```
dataFrame <- data.frame(x = x, y = y)
distxy <- dist(dataFrame)
hClustering <- hclust(distxy)
plot(hClustering)
```



Pretty Dendrograms

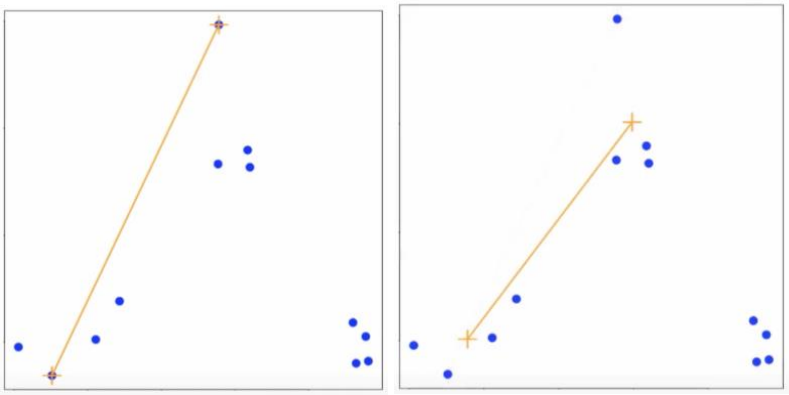
```
dataFrame <- data.frame(x = x, y = y)
distxy <- dist(dataFrame)
hClustering <- hclust(distxy)
myplclust(hClustering, lab = rep(1:3, each = 4), lab.col = rep(1:3, each = 4))
```





http://gallery.r-enthusiasts.com/RGraphGallery.php?graph=79

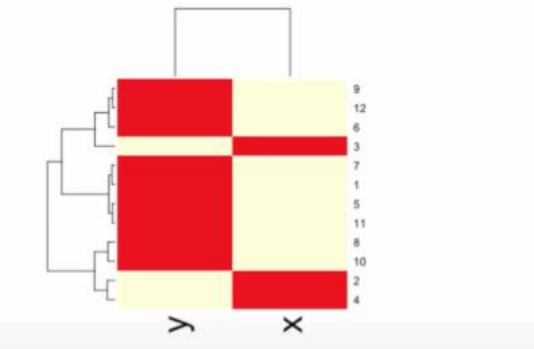
Merging Points



Complete Linkage Average Linkage

Heatmap()

```
dataFrame <- data.frame(x = x, y = y)
set.seed(143)
dataMatrix <- as.matrix(dataFrame)[sample(1:12), ]
heatmap(dataMatrix)
```



Summary

- Gives an idea of the relationships between variables/observations
- The picture may be unstable
  - Change a few points
  - Have different missing values
  - Pick a different distance
  - Change the merging strategy
  - Change the scale of points for one variable
- But it is deterministic
- Choosing where to cut isn't always obvious
- Should be primarily used for exploration
- Rafa's Distances and Clustering Video
- Elements of statistical learning

K-Means Clustering

- How do we define close?
- How do we group things?
- How do we visualize the grouping?
- How do we interpret the grouping?

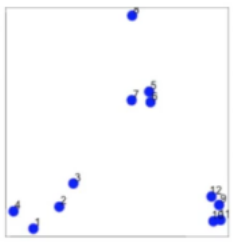
- Most important step
  - Garbage in ⟶ garbage out
- Distance or similarity
  - Continuous - euclidean distance
  - Continous - correlation similarity
  - Binary - manhattan distance
- Pick a distance/similarity that makes sense for your problem

---

- A partioning approach
  - Fix a number of clusters
  - Get "centroids" of each cluster
  - Assign things to closest centroid
  - Reclaculate centroids
- Requires
  - A defined distance metric
  - A number of clusters
  - An initial guess as to cluster centroids
- Produces
  - Final estimate of cluster centroids
  - An assignment of each point to clusters

Example: K-means Clustering

```
set.seed(1234)
par(mar = c(0, 0, 0, 0))
x <- rnorm(12, mean = rep(1:3, each = 4), sd = 0.2)
y <- rnorm(12, mean = rep(c(1, 2, 1), each = 4), sd = 0.2)
plot(x, y, col = "blue", pch = 19, cex = 2)
text(x + 0.05, y + 0.05, labels = as.character(1:12))
```



Kmeans()

- Important parameters: *x*, *centers*, *iter.max*, *nstart*

```
dataFrame <- data.frame(x, y)
kmeansObj <- kmeans(dataFrame, centers = 3)
names(kmeansObj)
```

```
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

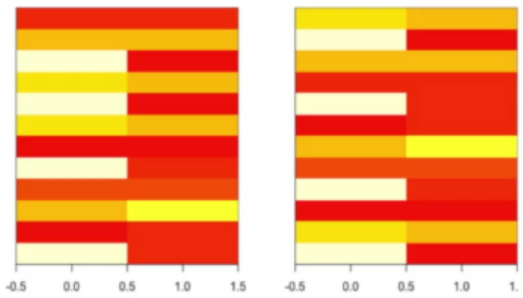
```
kmeansObj$cluster
```

```
##  [1] 3 3 3 3 1 1 1 1 2 2 2 2
```

```
par(mar = rep(0.2, 4))
plot(x, y, col = kmeansObj$cluster, pch = 19, cex = 2)
points(kmeansObj$centers, col = 1:3, pch = 3, cex = 3, lwd = 3)
```

Heatmaps

```
set.seed(1234)
dataMatrix <- as.matrix(dataFrame)[sample(1:12), ]
kmeansObj2 <- kmeans(dataMatrix, centers = 3)
par(mfrow = c(1, 2), mar = c(2, 4, 0.1, 0.1))
image(t(dataMatrix)[, nrow(dataMatrix):1], yaxt = "n")
image(t(dataMatrix)[, order(kmeansObj$cluster)], yaxt = "n")
```

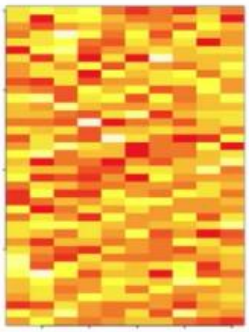


Summary

- K-means requires a number of clusters
  - Pick by eye/intuition
  - Pick by cross validation/information theory, etc.
  - Determining the number of clusters
- K-means is not deterministic
  - Different # of clusters
  - Different number of iterations
- Rafael Irizarry's Distances and Clustering Video
- Elements of statistical learning

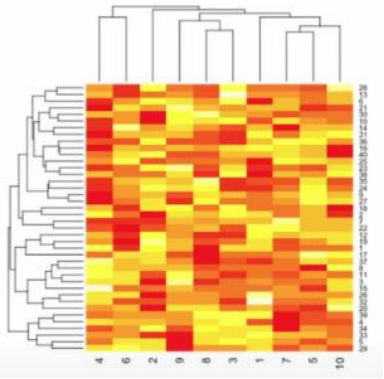
Dimension Reduction (PCA and SVD)

```
set.seed(12345)
par(mar = rep(0.2, 4))
dataMatrix <- matrix(rnorm(400), nrow = 40)
image(1:10, 1:40, t(dataMatrix)[, nrow(dataMatrix):1])
```



Cluster the Data

```
par(mar = rep(0.2, 4))
heatmap(dataMatrix)
```

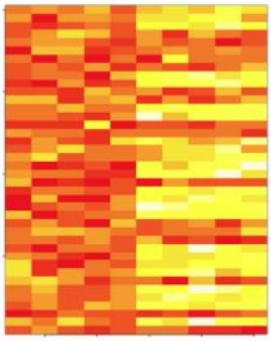


What if we add a pattern?

```
set.seed(678910)
for (i in 1:40) {
    # flip a coin
    coinFlip <- rbinom(1, size = 1, prob = 0.5)
    # if coin is heads add a common pattern to that row
    if (coinFlip) {
        dataMatrix[i, ] <- dataMatrix[i, ] + rep(c(0, 3), each = 5)
    }
}
```

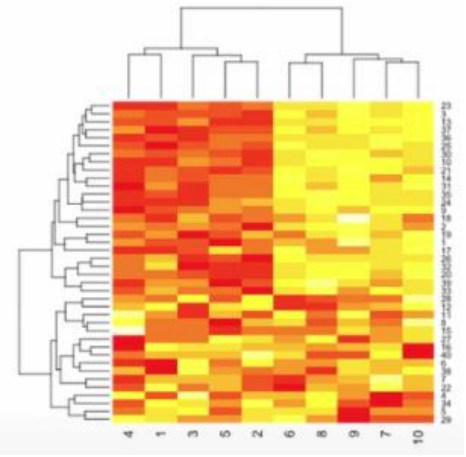
The Data

```
par(mar = rep(0.2, 4))
image(1:10, 1:40, t(dataMatrix)[, nrow(dataMatrix):1])
```



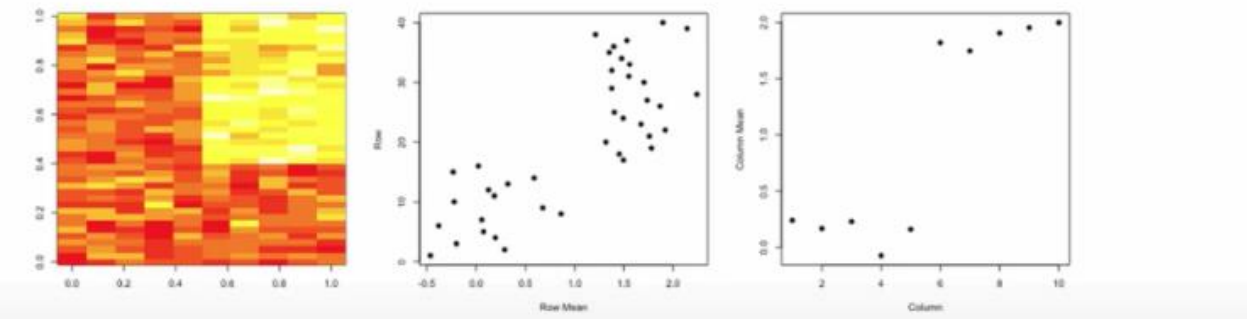
The Clustered Data

```
par(mar = rep(0.2, 4))
heatmap(dataMatrix)
```



Patterns in Rows and Columns

```
hh <- hclust(dist(dataMatrix))
dataMatrixOrdered <- dataMatrix[hh$order, ]
par(mfrow = c(1, 3))
image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
plot(rowMeans(dataMatrixOrdered), 40:1, , xlab = "Row Mean", ylab = "Row", pch = 19)
plot(colMeans(dataMatrixOrdered), xlab = "Column", ylab = "Column Mean", pch = 19)
```



Related Problems

You have multivariate variables $X_1, \ldots, X_n$ so $X_1 = (X_{11}, \ldots, X_{1m})$

- Find a new set of multivariate variables that are uncorrelated and explain as much variance as possible.
- If you put all the variables together in one matrix, find the best matrix created with fewer variables (lower rank) that explains the original data.

The first goal is statistical and the second goal is data compression.

Related Solutions – PCA / SVD

SVD

If X is a matrix with each variable in a column and each observation in a row then the SVD is a "matrix decomposition"

$$X = UDV^T$$

where the columns of U are orthogonal (left singular vectors), the columns of V are orthogonal (right singular vectors) and D is a diagonal matrix (singular values).

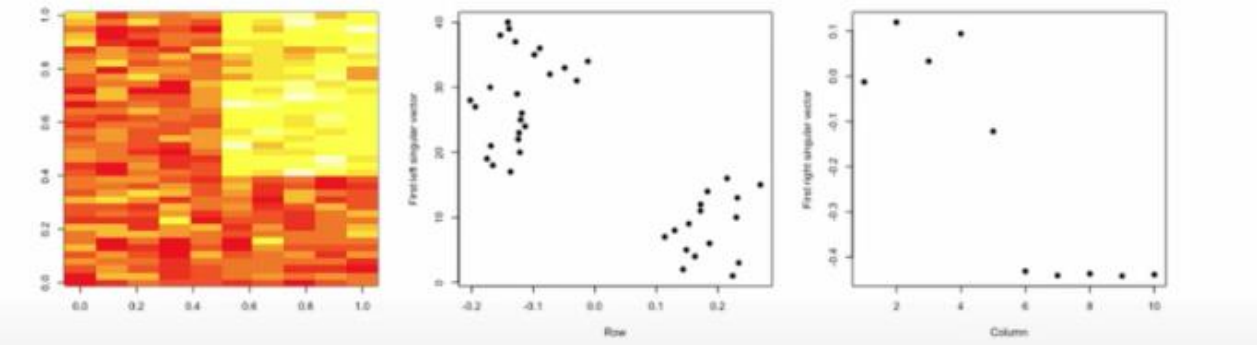
PCA

The principal components are equal to the right singular values if you first scale (subtract the mean, divide by the standard deviation) the variables.

Components of the SVD

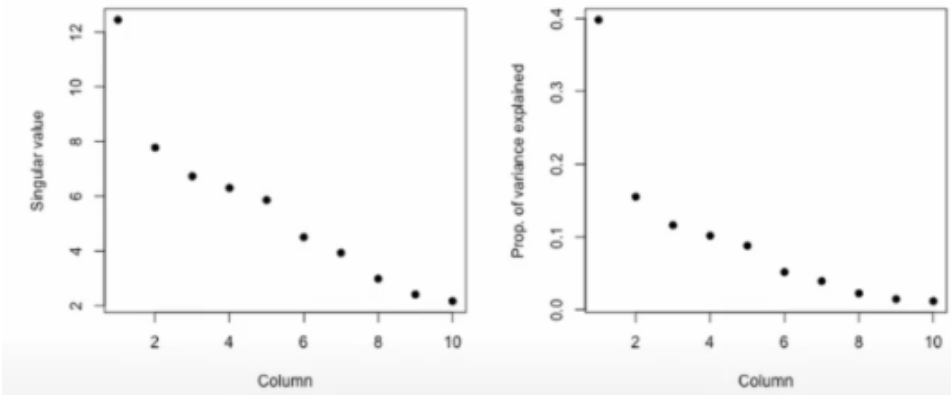
u and v

```
svd1 <- svd(scale(dataMatrixOrdered))
par(mfrow = c(1, 3))
image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
plot(svd1$u[, 1], 40:1, , xlab = "Row", ylab = "First left singular vector",
    pch = 19)
plot(svd1$v[, 1], xlab = "Column", ylab = "First right singular vector", pch = 19)
```

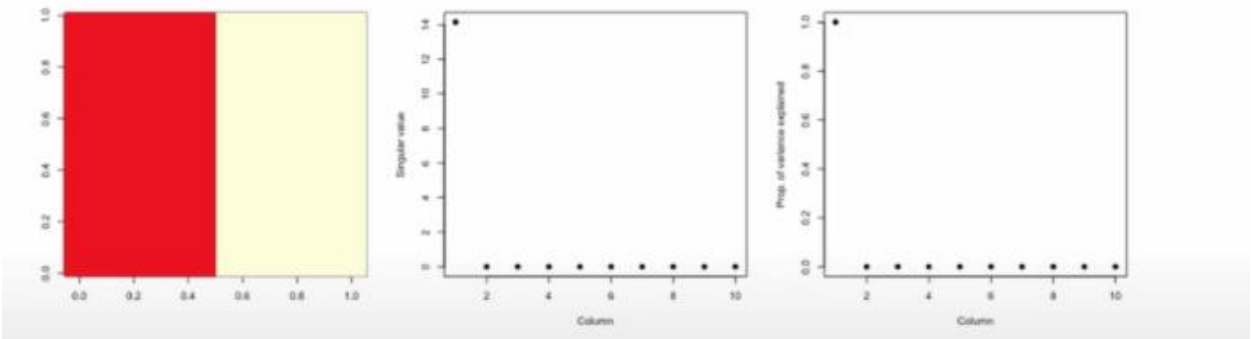


Variance Explained

```
par(mfrow = c(1, 2))
plot(svd1$d, xlab = "Column", ylab = "Singular value", pch = 19)
plot(svd1$d^2/sum(svd1$d^2), xlab = "Column", ylab = "Prop. of variance explained",
     pch = 19)
```

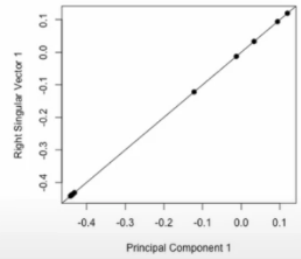



```
constantMatrix <- dataMatrixOrdered*0
for(i in 1:dim(dataMatrixOrdered)[1]){constantMatrix[i,] <- rep(c(0,1),each=5)}
svd1 <- svd(constantMatrix)
par(mfrow=c(1,3))
image(t(constantMatrix)[,nrow(constantMatrix):1])
plot(svd1$d,xlab="Column",ylab="Singular value",pch=19)
plot(svd1$d^2/sum(svd1$d^2),xlab="Column",ylab="Prop. of variance explained",pch=19)
```



Relationship to Principal Components

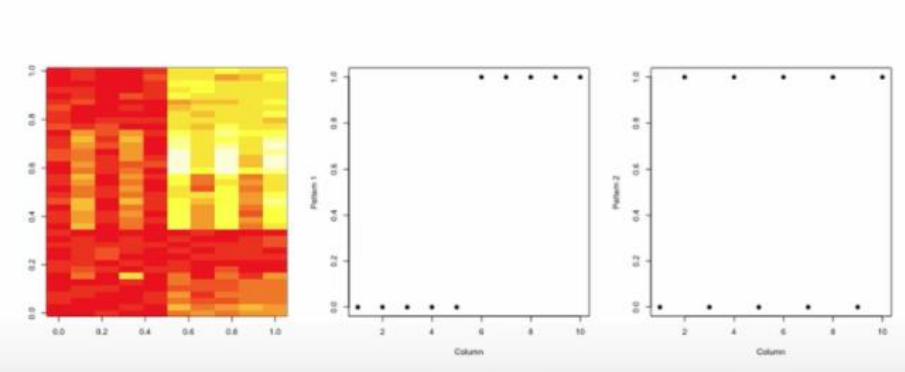
```
svd1 <- svd(scale(dataMatrixOrdered))
pca1 <- prcomp(dataMatrixOrdered, scale = TRUE)
plot(pca1$rotation[, 1], svd1$v[, 1], pch = 19, xlab = "Principal Component 1",
    ylab = "Right Singular Vector 1")
abline(c(0, 1))
```



What if we add a second pattern?

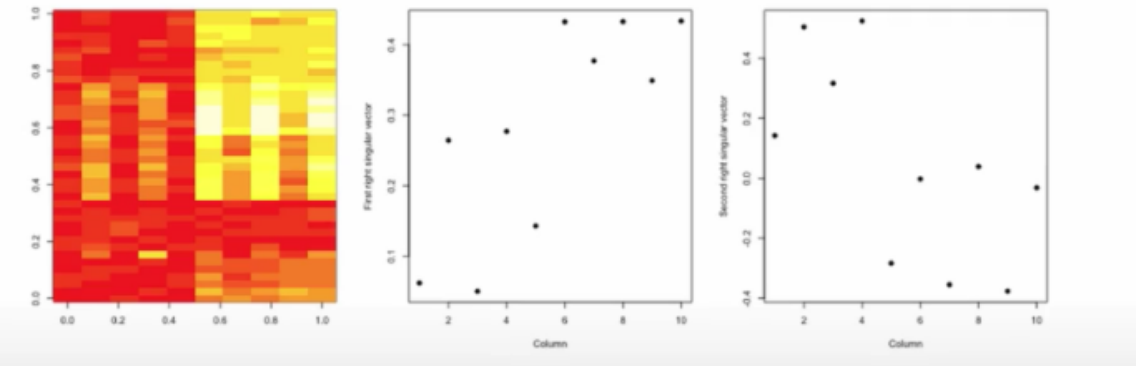
SVD – True Patterns

```
svd2 <- svd(scale(dataMatrixOrdered))
par(mfrow = c(1, 3))
image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
plot(rep(c(0, 1), each = 5), pch = 19, xlab = "Column", ylab = "Pattern 1")
plot(rep(c(0, 1), 5), pch = 19, xlab = "Column", ylab = "Pattern 2")
```



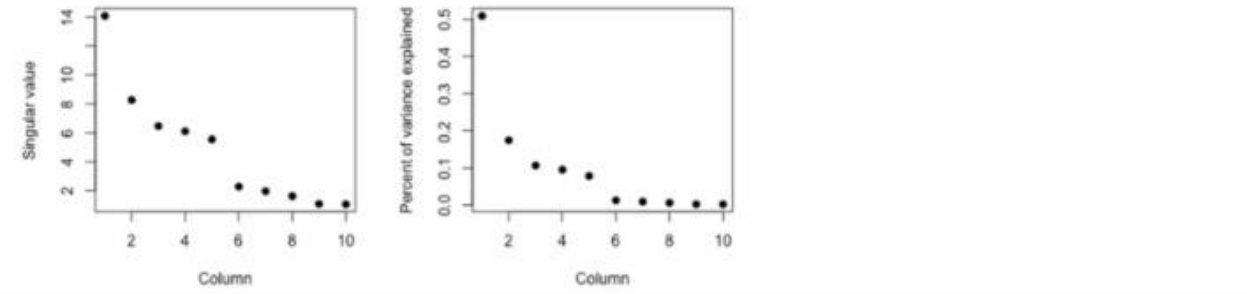
V and patterns of variance in rows

```
svd2 <- svd(scale(dataMatrixOrdered))
par(mfrow = c(1, 3))
image(t(dataMatrixOrdered)[, nrow(dataMatrixOrdered):1])
plot(svd2$v[, 1], pch = 19, xlab = "Column", ylab = "First right singular vector")
plot(svd2$v[, 2], pch = 19, xlab = "Column", ylab = "Second right singular vector")
```



D and variance explained

```
svd1 <- svd(scale(dataMatrixOrdered))
par(mfrow = c(1, 2))
plot(svd1$d, xlab = "Column", ylab = "Singular value", pch = 19)
plot(svd1$d^2/sum(svd1$d^2), xlab = "Column", ylab = "Percent of variance explained",
    pch = 19)
```



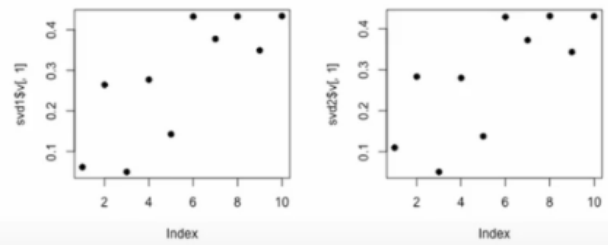
Missing Values

```
dataMatrix2 <- dataMatrixOrdered
## Randomly insert some missing data
dataMatrix2[sample(1:100, size = 40, replace = FALSE)] <- NA
svd1 <- svd(scale(dataMatrix2))  ## Doesn't work!
```

```
## Error: infinite or missing values in 'x'
```

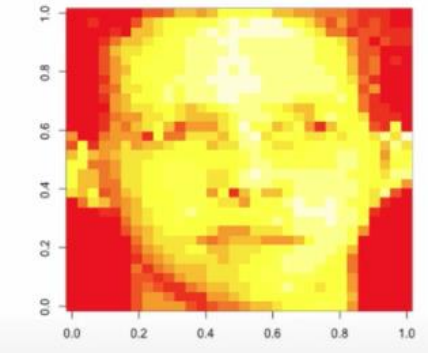
Imputing {impute}

```
library(impute)  ## Available from http://bioconductor.org
dataMatrix2 <- dataMatrixOrdered
dataMatrix2[sample(1:100,size=40,replace=FALSE)] <- NA
dataMatrix2 <- impute.knn(dataMatrix2)$data
svd1 <- svd(scale(dataMatrixOrdered)); svd2 <- svd(scale(dataMatrix2))
par(mfrow=c(1,2)); plot(svd1$v[,1],pch=19); plot(svd2$v[,1],pch=19)
```



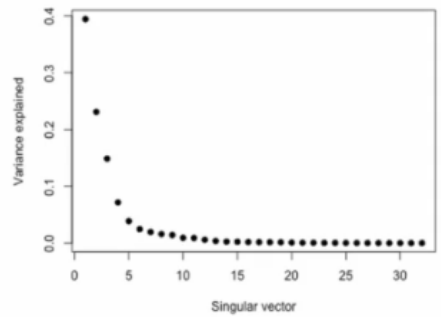
Face Example

```
load("data/face.rda")
image(t(faceData)[, nrow(faceData):1])
```



Variance Explained

```
svd1 <- svd(scale(faceData))
plot(svd1$d^2/sum(svd1$d^2), pch = 19, xlab = "Singular vector", ylab = "Variance explained")
```

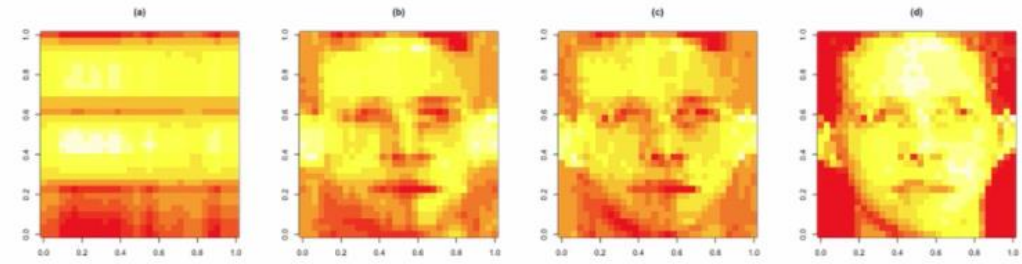


Create Approximations

```
svd1 <- svd(scale(faceData))
## Note that %*% is matrix multiplication

# Here svd1$d[1] is a constant
approx1 <- svd1$u[, 1] %*% t(svd1$v[, 1]) * svd1$d[1]

# In these examples we need to make the diagonal matrix out of d
approx5 <- svd1$u[, 1:5] %*% diag(svd1$d[1:5]) %*% t(svd1$v[, 1:5])
approx10 <- svd1$u[, 1:10] %*% diag(svd1$d[1:10]) %*% t(svd1$v[, 1:10])
```

Plot Approximations

```
par(mfrow = c(1, 4))
image(t(approx1)[, nrow(approx1):1], main = "(a)")
image(t(approx5)[, nrow(approx5):1], main = "(b)")
image(t(approx10)[, nrow(approx10):1], main = "(c)")
image(t(faceData)[, nrow(faceData):1], main = "(d)")  ## Original data
```

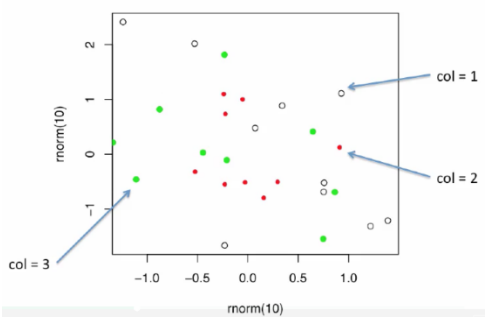


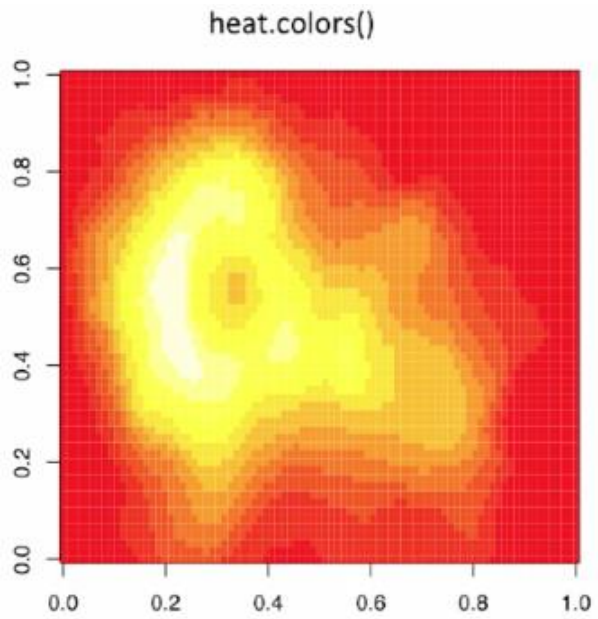
Summary

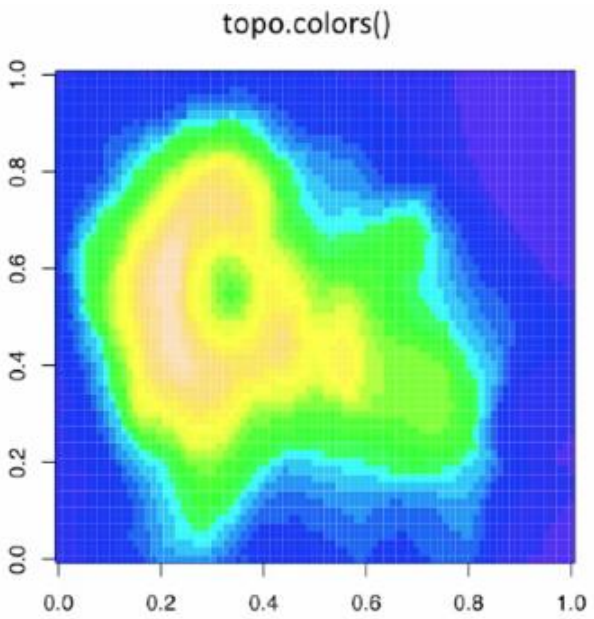
- Scale matters
- PC's/SV's may mix real patterns
- Can be computationally intensive
- Advanced data analysis from an elementary point of view
- Elements of statistical learning
- Alternatives
  - Factor analysis
  - Independent components analysis
  - Latent semantic analysis

Plotting and Color in R

- The default color schemes for most plots in R are horrendous
  - I don't have good taste and even I know that
- Recently there have been developments to improve the handling/specification of colors in plots/graphs/etc.
- There are functions in R and in external packages that are very handy







Color Utilities in R

- The **grDevices** package has two functions
  - `colorRamp`
  - `colorRampPalette`
- These functions take palettes of colors and help to interpolate between the colors
- The function `colors()` lists the names of colors you can use in any plotting function

- `colorRamp`: Take a palette of colors and return a function that takes values between 0 and 1, indicating the extremes of the color palette (e.g. see the 'gray' function)
- `colorRampPalette`: Take a palette of colors and return a function that takes integer arguments and returns a vector of colors interpolating the palette (like `heat.colors` or `topo.colors`)

colorRamp

```
> pal <- colorRamp(c("red", "blue"))

> pal(0)
     [,1] [,2] [,3]
[1,]  255    0    0

> pal(1)
     [,1] [,2] [,3]
[1,]    0    0  255

> pal(0.5)
      [,1] [,2]  [,3]
[1,] 127.5    0 127.5
```

Red

Blue

Green

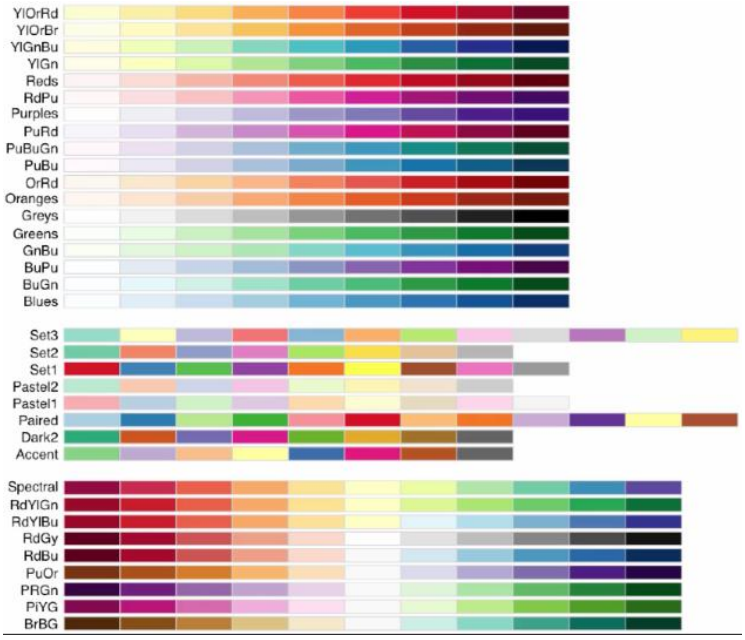
```
> pal(seq(0, 1, len = 10))
           [,1] [,2]      [,3]
 [1,] 255.00000    0   0.00000
 [2,] 226.66667    0  28.33333
 [3,] 198.33333    0  56.66667
 [4,] 170.00000    0  85.00000
 [5,] 141.66667    0 113.33333
 [6,] 113.33333    0 141.66667
 [7,]  85.00000    0 170.00000
 [8,]  56.66667    0 198.33333
 [9,]  28.33333    0 226.66667
[10,]   0.00000    0 255.00000
```

colorRampPalette

```
> pal <- colorRampPalette(c("red", "yellow"))

> pal(2)
[1] "#FF0000" "#FFFF00"

> pal(10)
 [1] "#FF0000" "#FF1C00" "#FF3800" "#FF5500" "#FF7100"
 [6] "#FF8D00" "#FFAA00" "#FFC600" "#FFE200" "#FFFF00"
```

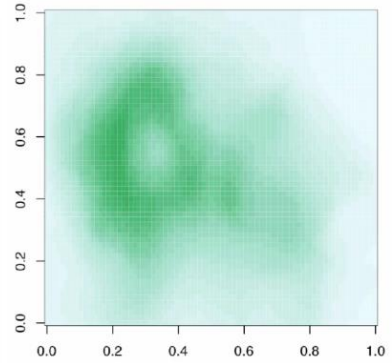
RColorBrewer Package

- One package on CRAN that contains interesting/useful color palettes
- There are 3 types of palettes
  - Sequential
  - Diverging
  - Qualitative
- Palette information can be used in conjunction with the `colorRamp()` and `colorRampPalette()`



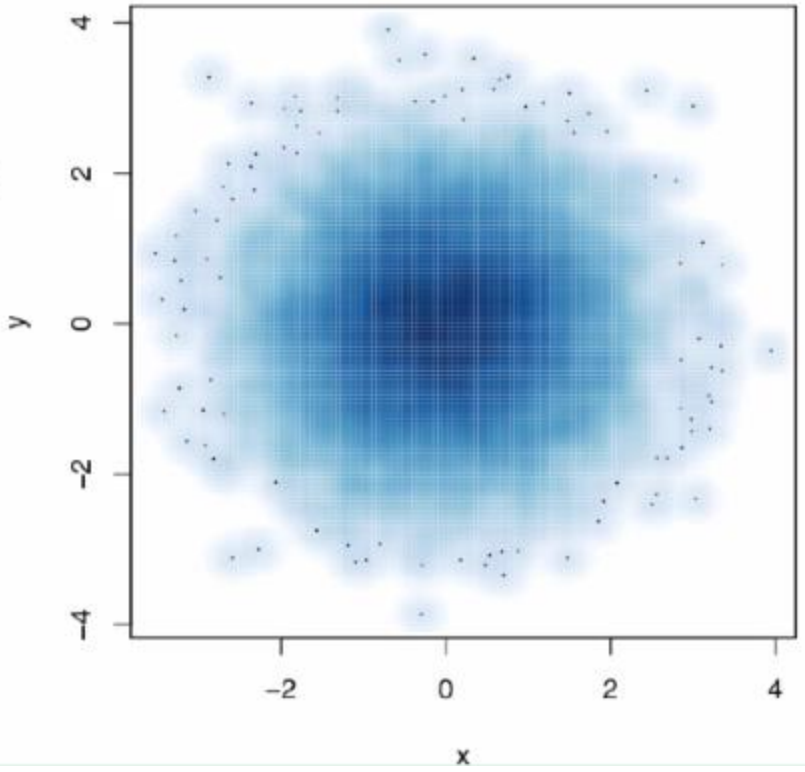
RColorBrewer and colorRampPalette

```
> library(RColorBrewer)

> cols <- brewer.pal(3, "BuGn")

> cols
[1] "#E5F5F9" "#99D8C9" "#2CA25F"

> pal <- colorRampPalette(cols)

> image(volcano, col = pal(20))
```



smoothScatter Function

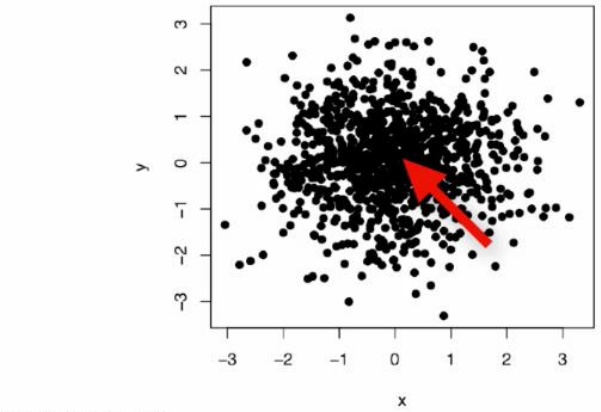
```
x <- rnorm(10000)
y <- rnorm(10000)
smoothScatter(x, y)
```



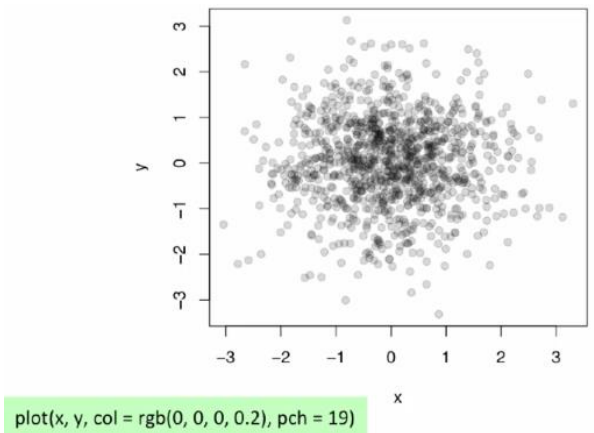
Other Plotting Notes

- The `rgb` function can be used to produce any color via red, green, blue proportions
- Color transparency can be added via the `alpha` parameter to `rgb`
- The **colorspace** package can be used for a different control over colors

Scatterplot with no transparency



plot(x, y, pch = 19)



plot(x, y, col = rgb(0, 0, 0, 0.2), pch = 19)

Summary

- Careful use of colors in plots/maps/etc. can make it easier for the reader to get what you're trying to say (why make it harder?)
- The **RColorBrewer** package is an R package that provides color palettes for sequential, categorical, and diverging data
- The `colorRamp` and `colorRampPalette` functions can be used in conjunction with color palettes to connect data to colors
- Transparency can sometimes be used to clarify plots with many points

Clustering Case Study – Human Activities from Smartphone Data

http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

Slightly Processed Data

```
load("data/samsungData.rda")
names(samsungData)[1:12]
```

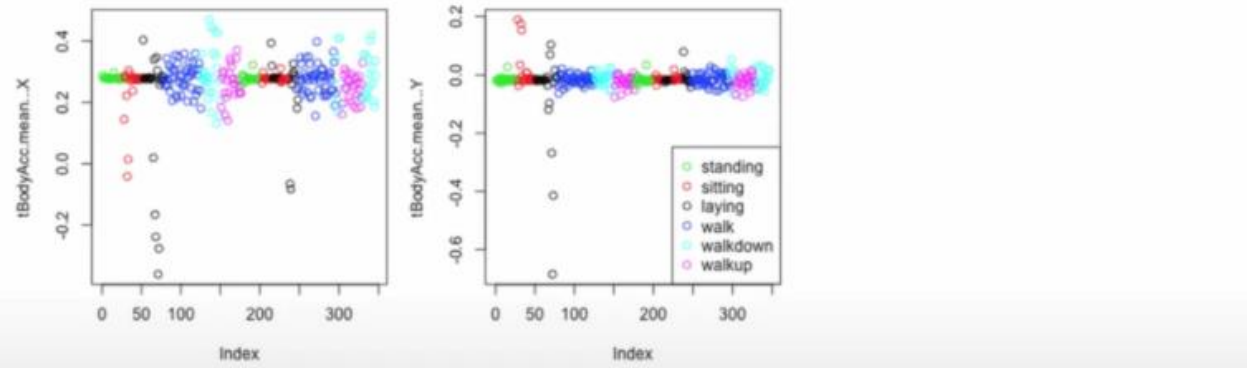
```
##  [1] "tBodyAcc-mean()-X" "tBodyAcc-mean()-Y" "tBodyAcc-mean()-Z"
##  [4] "tBodyAcc-std()-X"  "tBodyAcc-std()-Y"  "tBodyAcc-std()-Z"
##  [7] "tBodyAcc-mad()-X"  "tBodyAcc-mad()-Y"  "tBodyAcc-mad()-Z"
## [10] "tBodyAcc-max()-X"  "tBodyAcc-max()-Y"  "tBodyAcc-max()-Z"
```

```
table(samsungData$activity)
```

```
##
##   laying  sitting standing     walk walkdown   walkup
##     1407     1286     1374     1226      986     1073
```

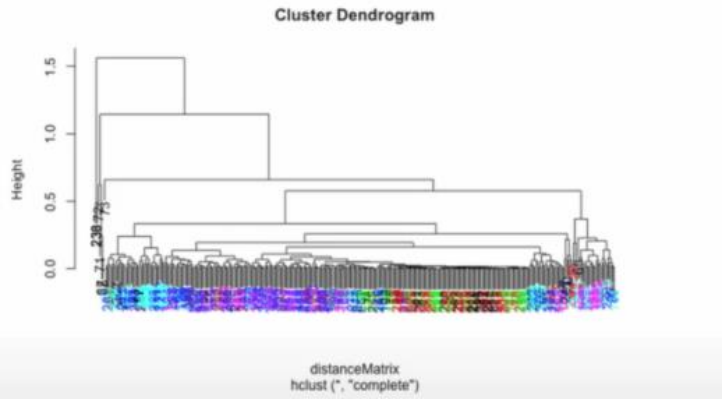
Plotting Average Acceleration- First Subject

```
par(mfrow = c(1, 2), mar = c(5, 4, 1, 1))
samsungData <- transform(samsungData, activity = factor(activity))
sub1 <- subset(samsungData, subject == 1)
plot(sub1[, 1], col = sub1$activity, ylab = names(sub1)[1])
plot(sub1[, 2], col = sub1$activity, ylab = names(sub1)[2])
legend("bottomright", legend = unique(sub1$activity), col = unique(sub1$activity),
    pch = 1)
```



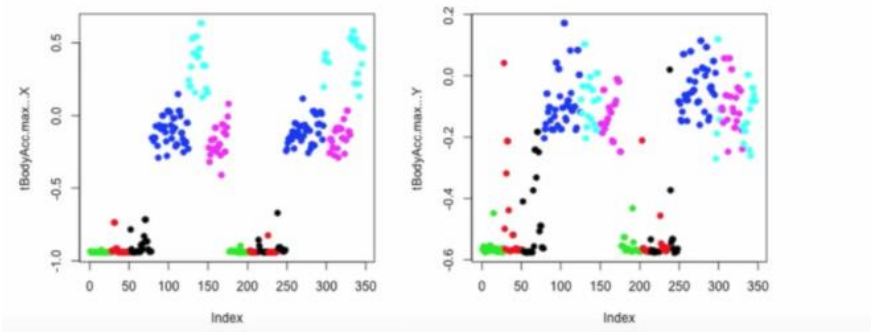
Clustering- Avg. Acceleration

```
source("myplclust.R")
distanceMatrix <- dist(sub1[, 1:3])
hclustering <- hclust(distanceMatrix)
myplclust(hclustering, lab.col = unclass(sub1$activity))
```



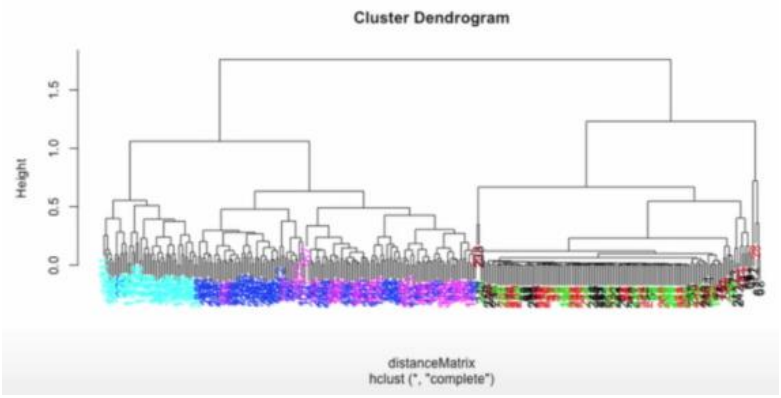
Plotting Max Acceleration – First Subject

```
par(mfrow = c(1, 2))
plot(sub1[, 10], pch = 19, col = sub1$activity, ylab = names(sub1)[10])
plot(sub1[, 11], pch = 19, col = sub1$activity, ylab = names(sub1)[11])
```



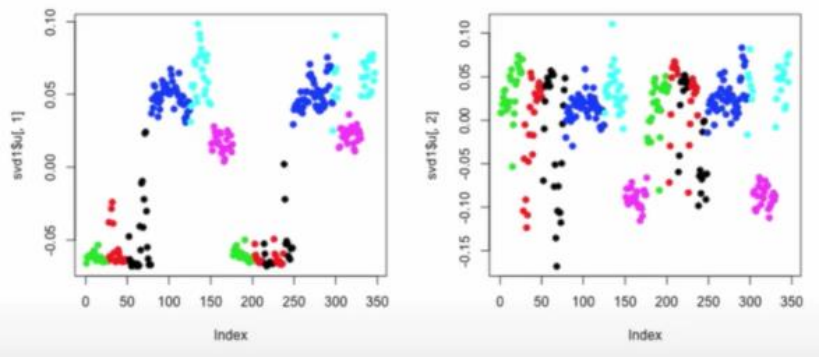
Clustering – Maximum Acceleration

```
source("myplclust.R")
distanceMatrix <- dist(sub1[, 10:12])
hclustering <- hclust(distanceMatrix)
myplclust(hclustering, lab.col = unclass(sub1$activity))
```



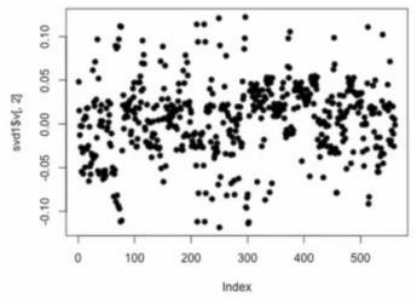
Singular Value Decomposition

```
svd1 = svd(scale(sub1[, -c(562, 563)]))
par(mfrow = c(1, 2))
plot(svd1$u[, 1], col = sub1$activity, pch = 19)
plot(svd1$u[, 2], col = sub1$activity, pch = 19)
```



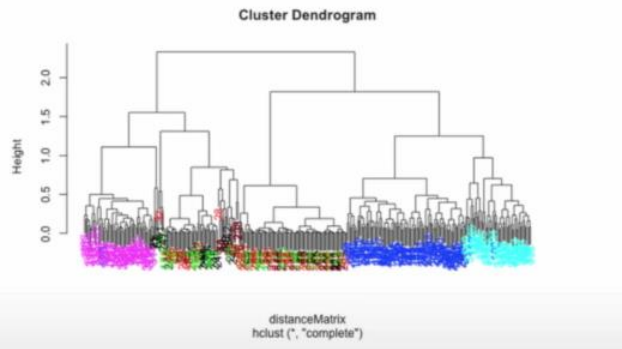
Find Max Contributor

```
plot(svd1$v[, 2], pch = 19)
```



New Clustering w/ Max Contributor

```
maxContrib <- which.max(svd1$v[, 2])
distanceMatrix <- dist(sub1[, c(10:12, maxContrib)])
hclustering <- hclust(distanceMatrix)
myplclust(hclustering, lab.col = unclass(sub1$activity))
```




```
names(samsungData)[maxContrib]
```

```
## [1] "fBodyAcc.meanFreq...Z"
```

K-means Clustering (nstart=1/100)

```
kClust <- kmeans(sub1[, -c(562, 563)], centers = 6)
table(kClust$cluster, sub1$activity)
```

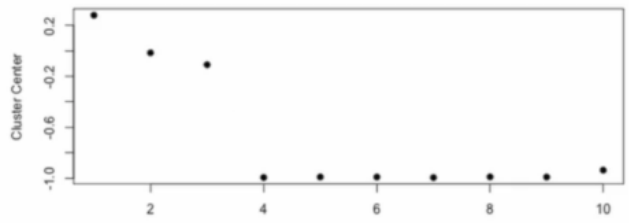
```
##
##     laying sitting standing walk walkdown walkup
##   1      0       0        0   50        1      0
##   2      0       0        0    0       48      0
##   3     27      37       51    0        0      0
##   4      3       0        0    0        0     53
##   5      0       0        0   45        0      0
##   6     20      10        2    0        0      0
```

```
kClust <- kmeans(sub1[, -c(562, 563)], centers = 6, nstart = 100)
table(kClust$cluster, sub1$activity)
```

```
##
##     laying sitting standing walk walkdown walkup
##   1     18      10        2    0        0      0
##   2     29       0        0    0        0      0
##   3      0       0        0   95        0      0
##   4      0       0        0    0       49      0
##   5      3       0        0    0        0     53
##   6      0      37       51    0        0      0
```

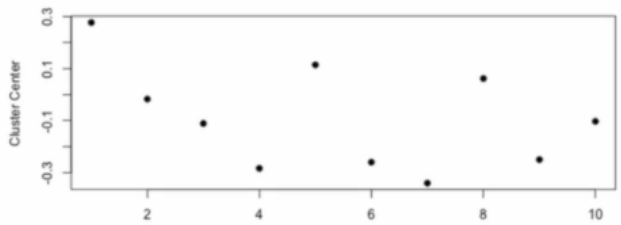
Cluster 1 Variable Centers (Lying)

```
plot(kClust$center[1, 1:10], pch = 19, ylab = "Cluster Center", xlab = "")
```



Cluster 2 Variable Centers (Walking)

```
plot(kClust$center[4, 1:10], pch = 19, ylab = "Cluster Center", xlab = "")
```



Air Pollution Case Study

```
> pm0 <- read.table("RD_501_88101_1999-0.txt", comment.char = "#", header
 = FALSE, sep = "|", na.strings = "")
> dim(pm0)
[1] 117421     28
> head(pm0)
  V1 V2 V3 V4 V5    V6 V7 V8  V9 V10      V11   V12    V13  V14 V15 V16
1 RD  I  1 27  1 88101  1  7 105 120 19990103 00:00     NA   AS   3  NA
2 RD  I  1 27  1 88101  1  7 105 120 19990106 00:00     NA   AS   3  NA
3 RD  I  1 27  1 88101  1  7 105 120 19990109 00:00     NA   AS   3  NA
4 RD  I  1 27  1 88101  1  7 105 120 19990112 00:00  8.841 <NA>   3  NA
5 RD  I  1 27  1 88101  1  7 105 120 19990115 00:00 14.920 <NA>   3  NA
6 RD  I  1 27  1 88101  1  7 105 120 19990118 00:00  3.878 <NA>   3  NA
   V17 V18 V19 V20 V21 V22 V23 V24 V25 V26 V27 V28
1 <NA>  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
2 <NA>  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
3 <NA>  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
4 <NA>  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
5 <NA>  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
6 <NA>  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
```

\

```
> cnames <- readLines("RD_501_88101_1999-0.txt", 1)
> cnames
[1] "# RD|Action Code|State Code|County Code|Site ID|Parameter|POC|Sample
 Duration|Unit|Method|Date|Start Time|Sample Value|Null Data Code|Samplin
g Frequency|Monitor Protocol (MP) ID|Qualifier - 1|Qualifier - 2|Qualifie
r - 3|Qualifier - 4|Qualifier - 5|Qualifier - 6|Qualifier - 7|Qualifier -
 8|Qualifier - 9|Qualifier - 10|Alternate Method Detectable Limit|Uncerta
inty"
```

```
> cnames <- strsplit(cnames, "|", fixed = TRUE)
> cnames
[[1]]
 [1] "# RD"
 [2] "Action Code"
 [3] "State Code"
 [4] "County Code"
 [5] "Site ID"
 [6] "Parameter"
 [7] "POC"
 [8] "Sample Duration"
 [9] "Unit"
[10] "Method"
[11] "Date"
[12] "Start Time"
[13] "Sample Value"
[14] "Null Data Code"
[15] "Sampling Frequency"
[16] "Monitor Protocol (MP) ID"
[17] "Qualifier - 1"
[18] "Qualifier - 2"
[19] "Qualifier - 3"
[20] "Qualifier - 4"
```

```
> names(pm0) <- cnames[[1]]
> head(pm0)
  # RD Action Code State Code County Code Site ID Parameter POC
1   RD           I          1          27       1     88101   1
2   RD           I          1          27       1     88101   1
3   RD           I          1          27       1     88101   1
4   RD           I          1          27       1     88101   1
5   RD           I          1          27       1     88101   1
6   RD           I          1          27       1     88101   1
  Sample Duration Unit Method     Date Start Time Sample Value
1               7  105    120 19990103      00:00           NA
2               7  105    120 19990106      00:00           NA
3               7  105    120 19990109      00:00           NA
4               7  105    120 19990112      00:00        8.841
5               7  105    120 19990115      00:00       14.920
6               7  105    120 19990118      00:00        3.878
  Null Data Code Sampling Frequency Monitor Protocol (MP) ID
1             AS                  3                       NA
2             AS                  3                       NA
```

```
> names(pm0) <- make.names(cnames[[1]])
> head(pm0)
  X..RD Action.Code State.Code County.Code Site.ID Parameter POC
1    RD           I          1          27       1     88101   1
2    RD           I          1          27       1     88101   1
3    RD           I          1          27       1     88101   1
4    RD           I          1          27       1     88101   1
5    RD           I          1          27       1     88101   1
6    RD           I          1          27       1     88101   1
  Sample.Duration Unit Method     Date Start.Time Sample.Value
1               7  105    120 19990103      00:00           NA
2               7  105    120 19990106      00:00           NA
3               7  105    120 19990109      00:00           NA
4               7  105    120 19990112      00:00        8.841
5               7  105    120 19990115      00:00       14.920
6               7  105    120 19990118      00:00        3.878
  Null.Data.Code Sampling.Frequency Monitor.Protocol..MP..ID
1             AS                  3                      NA
2             AS                  3                      NA
3             AS                  3                      NA
4           <NA>                  3                      NA
5           <NA>                  3                      NA
```

```
> x0 <- pm0$Sample.Value
> class(x0)
[1] "numeric"
> str(x0)
 num [1:117421] NA NA NA 8.84 14.92 ...
> summary(x0)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00    7.20   11.50   13.74   17.90  157.10   13217
> mean(is.na(x0))
[1] 0.1125608
```

```
> pm1 <- read.table("RD_501_88101_2012-0.txt", comment.char = "#", header
 = FALSE, sep = "|", na.strings = "")
```

```
> dim(pm1)
[1] 1304287      28
> names(pm1) <- make.names(cnames[[1]])
> head(pm10
+
> head(pm1)
  X..RD Action.Code State.Code County.Code Site.ID Parameter POC
1    RD           I          1           3      10     88101   1
2    RD           I          1           3      10     88101   1
3    RD           I          1           3      10     88101   1
4    RD           I          1           3      10     88101   1
5    RD           I          1           3      10     88101   1
6    RD           I          1           3      10     88101   1
  Sample.Duration Unit Method     Date Start.Time Sample.Value
1               7  105    118 20120101      00:00          6.7
2               7  105    118 20120104      00:00          9.0
3               7  105    118 20120107      00:00          6.5
4               7  105    118 20120110      00:00          7.0
5               7  105    118 20120113      00:00          5.8
6               7  105    118 20120116      00:00          8.0
  Null.Data.Code Sampling.Frequency Monitor.Protocol..MP..ID
1           <NA>                  3                      NA
2           <NA>                  3                      NA
3           <NA>                  3                      NA
4           <NA>                  3                      NA
```

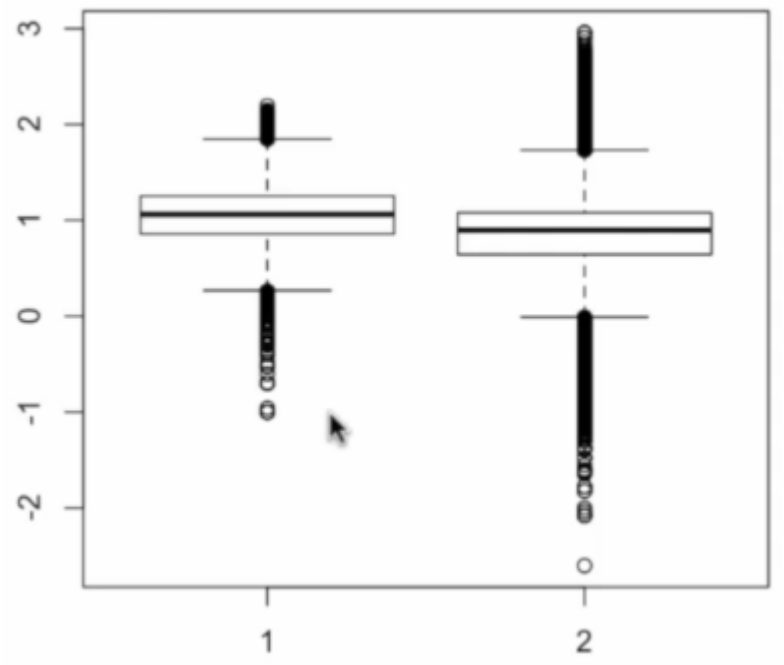
```
> x1 <- pm1$Sample.Value
> str(x1)
 num [1:1304287] 6.7 9 6.5 7 5.8 8 7.9 8 6 9.6 ...
```

```
> summary(x1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 -10.00    4.00    7.63    9.14   12.00  909.00   73133
> summary(x0)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   0.00    7.20   11.50   13.74   17.90  157.10   13217
```

```
> mean(is.na(x1))
[1] 0.05607125
```

```
> boxplot(x0, x1)
```

```
> boxplot(log10(x0), log10(x1))
```




```
> summary(x1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 -10.00    4.00    7.63    9.14   12.00  909.00   73133
> negative <- x1 < 0
```

```
> str(negative)
 logi [1:1304287] FALSE FALSE FALSE FALSE FALSE FALSE ...
> sum(negative, na.rm = TRUE)
[1] 26474
```

```
> dates <- pm1$Date
> str(dates)
 int [1:1304287] 20120101 20120104 20120107 20120110 20120113 20120116 20
120119 20120122 20120125 20120128 ...
```

```
> dates <- as.Date(as.character(dates), "%Y%m%d")
```

```
> str(dates)
 Date[1:1304287], format: "2012-01-01" "2012-01-04" "2012-01-07" "2012-01
-10" ...
```

```
> hist(dates, "month")
> hist(dates[negative])
Error in hist.Date(dates[negative]) :
  Must specify 'breaks' in hist(<Date>)
> hist(dates[negative], "month")
```

```
> site0 <- unique(subset(pm0, State.Code == 36, c(County.Code, Site.ID)))
> site1 <- unique(subset(pm1, State.Code == 36, c(County.Code, Site.ID)))
```

```
> head(site0)
      County.Code Site.ID
65873           1       5
65995           1      12
66056           5      73
66075           5      80
66136           5      83
66197           5     110
> site0 <- paste(site0[,1], site0[,2], sep = ".")
> site1 <- paste(site1[,1], site1[,2], sep = ".")
> str(site0)
 chr [1:33] "1.5" "1.12" "5.73" "5.80" "5.83" "5.110" ...
> str(site1)
 chr [1:18] "1.5" "1.12" "5.80" "5.133" "13.11" "29.5" ...
```

```
> both <- intersect(site0, site1)
> both
 [1] "1.5"     "1.12"    "5.80"    "13.11"   "29.5"    "31.3"
 [7] "63.2008" "67.1015" "85.55"   "101.3"
```

```
> pm0$county.site <- with(pm0, paste(County.Code, Site.ID, sep = "."))
> pm1$county.site <- with(pm1, paste(County.Code, Site.ID, sep = "."))
> cnt0 <- subset(pm0, State.Code == 36 & county.site %in% both)
> cnt1 <- subset(pm1, State.Code == 36 & county.site %in% both)
> head(cnt0)
      X..RD Action.Code State.Code County.Code Site.ID Parameter POC
65873    RD           I         36           1       5     88101   1
65874    RD           I         36           1       5     88101   1
65875    RD           I         36           1       5     88101   1
65876    RD           I         36           1       5     88101   1
65877    RD           I         36           1       5     88101   1
65878    RD           I         36           1       5     88101   1
      Sample.Duration Unit Method     Date Start.Time Sample.Value
65873               7  105    118 19990702      00:00           NA
65874               7  105    118 19990705      00:00           NA
65875               7  105    118 19990708      00:00           NA
65876               7  105    118 19990711      00:00           NA
65877               7  105    118 19990714      00:00         11.8
65878               7  105    118 19990717      00:00         49.4
      Null.Data.Code Sampling.Frequency Monitor.Protocol..MP..ID
```

```
> sapply(split(cnt0, cnt0$county.site), nrow)
   1.12     1.5   101.3   13.11    29.5    31.3    5.80 63.2008 67.1015
     61     122     152      61      61     183      61     122     122
  85.55
      7
```

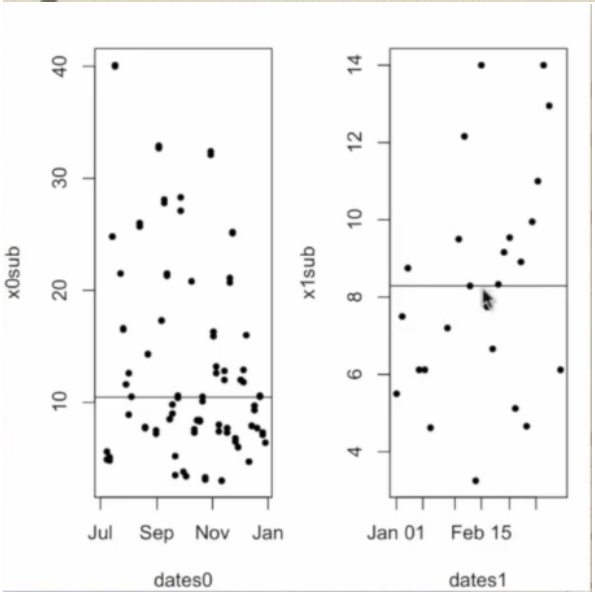
```
> sapply(split(cnt1, cnt1$county.site), nrow)
   1.12     1.5   101.3   13.11    29.5    31.3    5.80 63.2008 67.1015
     31      64      31      31      33      15      31      30      31
  85.55
     31
```

```
> pm1sub <- subset(pm1, State.Code == 36 & County.Code == 63 & Site.ID ==
 2008)
> pm0sub <- subset(pm0, State.Code == 36 & County.Code == 63 & Site.ID ==
 2008)
> dim(pm1sub)
[1] 30 29
> dim(pm0sub)
[1] 122  29
```

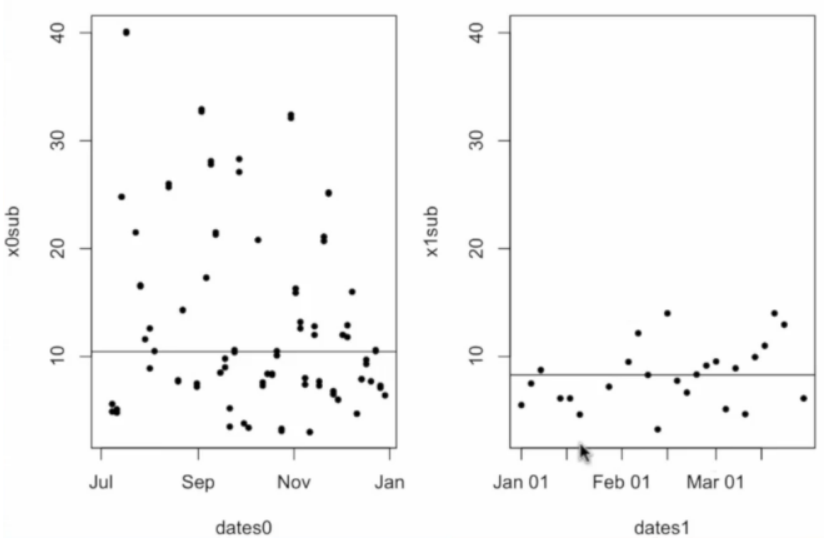
```
> dates1 <- pm1sub$Date
> x1sub <- pm1sub$Sample.Value
> plot(dates1, x1sub)
> dates1 <- as.Date(as.character(dates1), "%Y%m%d")
> str(dates1)
 Date[1:30], format: "2012-01-01" "2012-01-04" "2012-01-07" "2012-01-10"
...
> plot(dates1, x1sub)
```

```
> dates0 <- pm0sub$Date
> dates0 <- as.Date(as.character(dates0), "%Y%m%d")
> x0sub <- pm0sub$Sample.Value
> plot(dates0, x0sub)
```

```
> par(mfrow = c(1, 2), mar = c(4, 4, 2, 1))
> plot(dates0, x0sub, pch = 20)
> abline(h = median(x0sub, na.rm=T))
> plot(dates1, x1sub, pch = 20)
> abline(h = median(x1sub, na.rm=T))
```




```
> range(x0sub, x1sub, na.rm=T)
[1]  3.0 40.1
> rng <- range(x0sub, x1sub, na.rm=T)
> par(mfrow = c(1, 2))
> plot(dates0, x0sub, pch = 20, ylim = rng)
> abline(h = median(x0sub, na.rm=T))
> plot(dates1, x1sub, pch = 20, ylim = rng)
```



Exploring Changes on the State Level

```
> mn0 <- with(pm0, tapply(Sample.Value, State.Code, mean, na.rm = T))
> str(mn0)
 num [1:53(1d)] 19.96 6.67 10.8 15.68 17.66 ...
 - attr(*, "dimnames")=List of 1
  ..$ : chr [1:53] "1" "2" "4" "5" ...
summary(mn0)
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
4.862   9.519  12.310  12.410  15.640  19.960
```

```
> mn1 <- with(pm1, tapply(Sample.Value, State.Code, mean, na.rm = T))
> summary(mn1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.006   7.355   8.729   8.759  10.610  11.990
```

```
> d0 <- data.frame(state = names(mn0), mean = mn0)
> d1 <- data.frame(state = names(mn1), mean = mn1)
> head(d0)
  state      mean
1     1 19.956391
2     2  6.665929
4     4 10.795547
5     5 15.676067
6     6 17.655412
8     8  7.533304
> head(d1)
  state      mean
1     1 10.126190
2     2  4.750389
4     4  8.609956
5     5 10.563636
6     6  9.277373
8     8  4.117144
```

```
> mrg <- merge(d0, d1, by = "state")
> dim(mrg)
[1] 52  3
> head(mrg)
  state    mean.x    mean.y
1     1 19.956391 10.126190
2    10 14.492895 11.236059
3    11 15.786507 11.991697
4    12 11.137139  8.239690
5    13 19.943240 11.321364
6    15  4.861821  8.749336
```

```
> par(mfrow = c(1, 1))
> with(mrg, plot(rep(1999, 52), mrg[, 2], xlim c(1998, 2013))
Error: unexpected symbol in "with(mrg, plot(rep(1999, 52), mrg[, 2], xlim
 c"
> with(mrg, plot(rep(1999, 52), mrg[, 2], xlim = c(1998, 2013))
+
> with(mrg, plot(rep(1999, 52), mrg[, 2], xlim = c(1998, 2013)))
> with(mrg, points(rep(2012, 52), mrg[, 3]))
> segments(rep(1999, 52), mrg[,2], rep(2012, 52), mrg[,3])
```