Foundations of Data Science I

Jesse Lecy

Latest build on August 01, 2018

Contents

Welcome

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```r
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 1

(PART) Data Programming

Chapter 2

Introduction to R

This lecture introduces you to basic operations when you first start using R such as navigation, the object-oriented framework, loading a package, and creating some data vectors.

2.1 Navigation

You need to know a few operations to help you maneuver the R work environment, such as listing objects (datasets and functions) that are active, changing your working directory, listing available files, and finding help.

2.1.1 Setting Your Working Directory

When you are ready to load data, R needs to know where to look for your files. You can check what is avaiable in the current directory (i.e. folder) by asking to list all of the current files using **dir()**.

```
dir()
```

If the file that you need is located in a different folder, you can change directories easily in R Studio by Session -> Set working director -> Choose directory (or Ctrl + Shift + H).

If you are writing a script, you want to keep track of this step so that it can be reproduced. Use the function **get.wd()** to check your current working directory, and **set.wd()** to change. You need to specify your path as an argument to this function, such as.

```
setwd( "C:/user/projects/financial model" )
```

NOTE! R uses unix style notation with forward slashes, so if you copy and paste from Windows it will look like this, with back slashes:

```
setwd( "C:\user\projects\financial model" )
```

You will need to change them around for it to work.

It is best to save all of your steps in your scripts so that the analysis can be reproduced by yourself or others. In some cases you are doing exploratory or summary work, and you may want to find a file a quickly. You can use the **file.choose()** function to open a GUI to select your file directly. This function is used as an argument inside of a load data function.

```
my.dat <- read.csv( file.choose() )
```

2.2 Commenting Code

Most computer languages have a special character that is used to "comment out" lines so that it is not run by the program. It is used for two important purposes. First, we can add text to document our functions and it will not interfere with the program. And two, we can use it to run a program while ignoring some of the code, often for debugging purposes.

The # hash tag is used for comments in R.

```r
##==============================================
##
##  Here is some documentation for this script
##
##==============================================

x <- 1:10

sum( x )
#> [1] 55

# y <- 1:25     # not run

# sum( y )      # not run
```

2.3 Help

You will use the help functions frequently to figure out what arguments and values are needed for specific functions. Because R is very customizable, you will find that many functions have several or dozens of arguments, and it is difficult to remember the correct syntax and values. But don't worry, to look them up all you need is the function name and a call for help:

> help(dotchart) # opens an external helpfile

If you just need to remind yourself which arguments are defined in a function, you can use the *args()* command:

```r
args( dotchart )
#> function (x, labels = NULL, groups = NULL, gdata = NULL, cex = par("cex"),
#>     pt.cex = cex, pch = 21, gpch = 21, bg = par("bg"), color = par("fg"),
#>     gcolor = par("fg"), lcolor = "gray", xlim = range(x[is.finite(x)]),
#>     main = NULL, xlab = NULL, ylab = NULL, ...)
#> NULL
```

If you can't recall a function name, you can list all of the functions from a specific package as follows:

> help(package="stats") # lists all functions in stats package

2.4 Install Programs (packages)

When you open R by default it will launch a core set of programs, called "packages" in R speak, that are use for most data operations. To see which packages are currently active use the **search()** function.

```
search()
#>  [1] ".GlobalEnv"        "package:bindrcpp"  "package:ggplot2"
#>  [4] "package:scales"    "package:reshape2"  "package:tidyr"
#>  [7] "package:maps"      "package:pander"    "package:dplyr"
#> [10] "package:Lahman"    "package:bookdown"  "package:rmarkdown"
#> [13] "package:stats"     "package:graphics"  "package:grDevices"
#> [16] "package:utils"     "package:datasets"  "package:methods"
#> [19] "Autoloads"         "package:base"
```

These programs manage the basic data operations, run the core graphics engine, and give you basic statistical methods.

The real magic for R comes from the over 7,000 contributed packages available on the CRAN: https://cran.r-project.org/web/views/

A package consists of custom functions and datasets that are generated by users. They are *packaged* together so that they can be shared with others. A package also includes documentation that describes each function, defines all of the arguments, and documents any datasets that are included.

If you know a package name, it is easy to install. In R Studio you can select Tools -> Install Packages and a list of available packages will be generated. But it is easier to use the **install.packages()** command. We will use the Lahman Package in this course, so let's install that now.

Description *This package provides the tables from Sean Lahman's Baseball Database as a set of R data.frames. It uses the data on pitching, hitting and fielding performance and other tables from 1871 through 2013, as recorded in the 2014 version of the database.*

See the documentation here: https://cran.r-project.org/web/packages/Lahman/Lahman.pdf

```
install.packages( "Lahman" )
```

You will be asked to select a "mirror". In R speak this just means the server from which you will download the package (choose anything nearby). R is a community of developers and universities that create code and maintain the infrastructure. A couple of dozen universities around the world host servers that contain copies of the R packages so that they can be easily accessed everywhere.

If the package is successfully installed you will get a message similar to this:

> package 'Lahman' successfully unpacked and MD5 sums checked

Once a new program is installed you can now open ("load" in R speak) the package using the **library()** command:

```
library( "Lahman" )
```

If you now type **search()** you can see that Lahman has been added to the list of active programs. We can now access all of the functions and data that are available in the Lahman package.

2.5 Accessing Built-In Datasets in R

One nice feature of R is that is comes with a bunch of built-in datasets that have been contributed by users are are loaded automatically. You can see the list of available datasets by typing:

```
data()
```

This will list all of the default datasets in core R packages. If you want to see all of the datasets available in installed packages as well use:

```
data( package = .packages(all.available = TRUE) )
```

2.5.1 Basic Data Operations

Let's ignore the underlying data structure right now and look at some ways that we might interact with data.

We will use the **USArrests** dataset available in the core files.

To access the data we need to load it into working memory. Anything that is active in R will be listed in the environment, which you can check using the **ls()** command. We will load the dataset using the **data()** command.

```
remove( list=ls() )
```

```
ls() # nothing currently available
#> character(0)

data( "USArrests" )

ls() # data is now avaible for use
#> [1] "USArrests"
```

Now that we have loaded a dataset, we can start to access the variables and analyze relationships. Let's get to know our dataset.

```
names( USArrests )  # what variables do you have?
#> [1] "Murder"   "Assault"  "UrbanPop" "Rape"

row.names( USArrests )   # what are the obsevations (rows) in our data
#>  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
#>  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
#>  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
#> [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
#> [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
#> [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
#> [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
#> [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
#> [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
#> [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
#> [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
#> [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
#> [49] "Wisconsin"      "Wyoming"

nrow( USArrests )   # how many observations are there?
#> [1] 50

dim( USArrests )    # a quick way to see rows and columns - the dimensions of the dataset
#> [1] 50  4

summary( USArrests )   # summary statistics of variables
#>      Murder          Assault         UrbanPop          Rape
#>  Min.   : 0.800   Min.   : 45.0   Min.   :32.00   Min.   : 7.30
#>  1st Qu.: 4.075   1st Qu.:109.0   1st Qu.:54.50   1st Qu.:15.07
#>  Median : 7.250   Median :159.0   Median :66.00   Median :20.10
#>  Mean   : 7.788   Mean   :170.8   Mean   :65.54   Mean   :21.23
#>  3rd Qu.:11.250   3rd Qu.:249.0   3rd Qu.:77.75   3rd Qu.:26.18
#>  Max.   :17.400   Max.   :337.0   Max.   :91.00   Max.   :46.00
```

We can see that the dataset consists of four variables: Murder, Assault, UrbanPop, and Rape. We also see that our unit of analysis is the state. But where does the data come from, and how are these variables measured?

To see the documentation for a specific dataset you will need to use the **help()** function:

```
help( "USArrests" )
```

We get valuable information about the source and metrics:

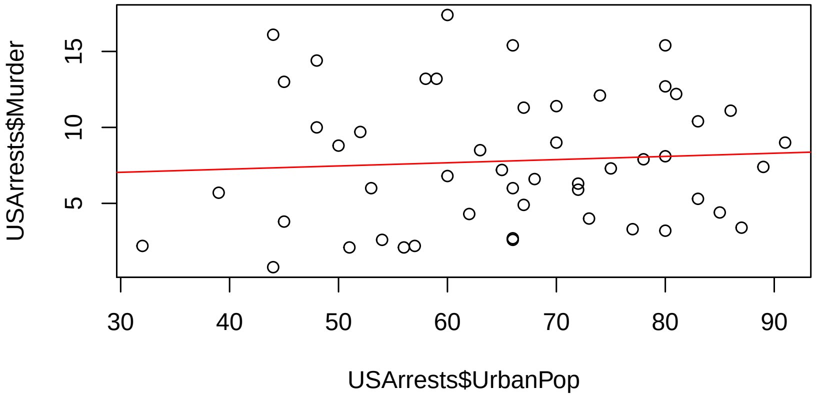
Description *This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.*

Format *A data frame with 50 observations on 4 variables.*

[,1] Murder numeric Murder arrests (per 100,000) [,2] Assault numeric Assault arrests (per 100,000) [,3] UrbanPop numeric Percent urban population [,4] Rape numeric Rape arrests (per 100,000)

To access a specific variable inside of a dataset, you will use the *$* operator between the dataset name and the variable name:

```
summary( USArrests$Murder )
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
#>   0.800   4.075   7.250   7.788  11.250  17.400
summary( USArrests$Assault )
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
#>    45.0   109.0   159.0   170.8   249.0   337.0

# Is there a relationship between urban density and crime?

plot( USArrests$UrbanPop, USArrests$Murder )
abline( lm( USArrests$Murder ~ USArrests$UrbanPop ), col="red" )
```



2.5.2 Using the Lahman Data

Let's take a look at some of the data available in the Lahman package.

```
data( package = "Lahman" )

# Data sets in package ‘Lahman’:
#
# AllstarFull                           AllstarFull table
# Appearances                           Appearances table
# AwardsManagers                        AwardsManagers table
# AwardsPlayers                         AwardsPlayers table
# AwardsShareManagers                   AwardsShareManagers table
# AwardsSharePlayers                    AwardsSharePlayers table
# Batting                               Batting table
# BattingPost                           BattingPost table
# CollegePlaying                        CollegePlaying table
# Fielding                              Fielding table
# FieldingOF                            FieldingOF table
# FieldingPost                          FieldingPost data
# HallOfFame                            Hall of Fame Voting Data
# LahmanData                            Lahman Datasets
# Managers                              Managers table
# ManagersHalf                          ManagersHalf table
# Master                                Master table
# Pitching                              Pitching table
# PitchingPost                          PitchingPost table
# Salaries                              Salaries table
# Schools                               Schools table
# SeriesPost                            SeriesPost table
# Teams                                 Teams table
# TeamsFranchises                       TeamFranchises table
# TeamsHalf                             TeamsHalf table
# battingLabels                         Variable Labels
# fieldingLabels                        Variable Labels
# pitchingLabels                        Variable Labels
```

We see that we have lots of datasets to choose from here. I will use the Master dataset, which is a list of all of the Major League Baseball players over the past century, and their personal information.

```
library( Lahman )

data( Master )

names( Master )
#>  [1] "playerID"     "birthYear"    "birthMonth"   "birthDay"
#>  [5] "birthCountry" "birthState"   "birthCity"    "deathYear"
#>  [9] "deathMonth"   "deathDay"     "deathCountry" "deathState"
#> [13] "deathCity"    "nameFirst"    "nameLast"     "nameGiven"
#> [17] "weight"       "height"       "bats"         "throws"
#> [21] "debut"        "finalGame"    "retroID"      "bbrefID"
#> [25] "deathDate"    "birthDate"

nrow( Master )  # 18,354 players included
#> [1] 19105
```

```
summary( Master )
#>    playerID           birthYear      birthMonth        birthDay
#>  Length:19105       Min.   :1820   Min.   : 1.000   Min.   : 1.00
#>  Class :character   1st Qu.:1895   1st Qu.: 4.000   1st Qu.: 8.00
#>  Mode  :character   Median :1937   Median : 7.000   Median :16.00
#>                     Mean   :1931   Mean   : 6.629   Mean   :15.61
#>                     3rd Qu.:1969   3rd Qu.:10.000   3rd Qu.:23.00
#>                     Max.   :1996   Max.   :12.000   Max.   :31.00
#>                     NA's   :132    NA's   :302      NA's   :449
#>  birthCountry        birthState         birthCity           deathYear
#>  Length:19105       Length:19105       Length:19105       Min.   :1872
#>  Class :character   Class :character   Class :character   1st Qu.:1942
#>  Mode  :character   Mode  :character   Mode  :character   Median :1967
#>                                                           Mean   :1964
#>                                                           3rd Qu.:1990
#>                                                           Max.   :2017
#>                                                           NA's   :9664
#>    deathMonth        deathDay     deathCountry        deathState
#>  Min.   : 1.000   Min.   : 1.00   Length:19105       Length:19105
#>  1st Qu.: 3.000   1st Qu.: 8.00   Class :character   Class :character
#>  Median : 6.000   Median :15.00   Mode  :character   Mode  :character
#>  Mean   : 6.484   Mean   :15.57
#>  3rd Qu.:10.000   3rd Qu.:23.00
#>  Max.   :12.000   Max.   :31.00
#>  NA's   :9665     NA's   :9666
#>   deathCity          nameFirst           nameLast
#>  Length:19105       Length:19105       Length:19105
#>  Class :character   Class :character   Class :character
#>  Mode  :character   Mode  :character   Mode  :character
#>
#>
#>
#>
#>   nameGiven             weight          height        bats
#>  Length:19105       Min.   : 65.0   Min.   :43.00   B   : 1175
#>  Class :character   1st Qu.:170.0   1st Qu.:71.00   L   : 4957
#>  Mode  :character   Median :185.0   Median :72.00   R   :11788
#>                     Mean   :186.4   Mean   :72.27   NA's: 1185
#>                     3rd Qu.:200.0   3rd Qu.:74.00
#>                     Max.   :320.0   Max.   :83.00
#>                     NA's   :854     NA's   :785
#>   throws         debut             finalGame           retroID
#>  L   : 3653   Length:19105       Length:19105       Length:19105
#>  R   :14472   Class :character   Class :character   Class :character
#>  S   :    1   Mode  :character   Mode  :character   Mode  :character
#>  NA's:  979
#>
#>
#>
#>    bbrefID            deathDate            birthDate
#>  Length:19105       Min.   :1872-03-17   Min.   :1820-04-17
#>  Class :character   1st Qu.:1942-06-05   1st Qu.:1896-11-21
#>  Mode  :character   Median :1967-03-06   Median :1939-01-01
#>                        Mean   :1964-10-11   Mean   :1933-03-08  
#>                        3rd Qu.:1990-09-22   3rd Qu.:1970-04-02  
#>                        Max.   :2017-02-19   Max.   :1996-08-12  
#>                        NA's   :9666         NA's   :449
```

We can use **help(Master)** to get information about the dataset, including a data dictionary.

```
help( Master )
```

2.6 MASTER TABLE

Description

Master table - Player names, DOB, and biographical info. This file is to be used to get details about players listed in the Batting, Pitching, and other files where players are identified only by playerID.

Usage

data(Master)

Format

A data frame with 19105 observations on the following 26 variables.

- **playerID**: A unique code asssigned to each player. The playerID links the data in this file with records on players in the other files.
- **birthYear**: Year player was born
- **birthMonth**: Month player was born
- **birthDay**: Day player was born
- **birthCountry**: Country where player was born
- **birthState**: State where player was born
- **birthCity**: City where player was born
- **deathYear**: Year player died
- **deathMonth**: Month player died
- **deathDay**: Day player died
- **deathCountry**: Country where player died
- **deathState**: State where player died
- **deathCity**: City where player died
- **nameFirst**: Player's first name
- **nameLast**: Player's last name
- **nameGiven**: Player's given name (typically first and middle)
- **weight**: Player's weight in pounds
- **height**: Player's height in inches
- **bats**: a factor: Player's batting hand (left (L), right (R), or both (B))
- **throws**: a factor: Player's throwing hand (left(L) or right(R))
- **debut**: Date that player made first major league appearance
- **finalGame**: Date that player made first major league appearance (blank if still active)
- **retroID**: ID used by retrosheet, http://www.retrosheet.org/
- **bbrefID**: ID used by Baseball Reference website, http://www.baseball-reference.com/
- **birthDate**: Player's birthdate, in as.Date format
- **deathDate**: Player's deathdate, in as.Date format

Details

debut, finalGame were converted from character strings with as.Date.

Source

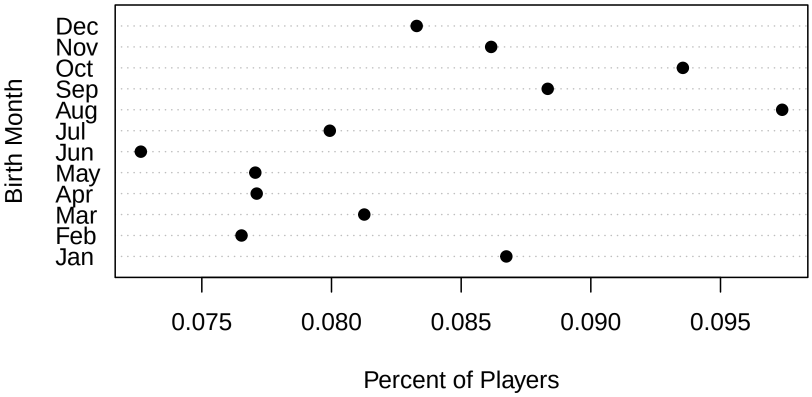
Lahman, S. (2016) Lahman's Baseball Database, 1871-2015, 2015 version, http://www.seanlahman.com/baseball-archive/statistics/

2.6.1 Example Analysis

Perhaps I am curious about some of the data. I see that we have information on the birth month of professional baseball players. If you have read Malcolm Gladwell's book *Outliers* you know there is an interesting cumulative advantage phenomenon that can occur with atheletes as they are young. If you are born near the end of the cutoff, you are on average six months older than other players in your league, and therefore slightly larger physically and more coordinated on average. Six months does not sound like much, but the slight size and coordination advantage means more playing time, which also improves skill. Over time, this small difference accumulates so that those lucky enough to be born near the cutoff become the best players.

Gladwell looked at studies of hockey. Do we see this in baseball?

```
table( Master$birthMonth )
#>
#>    1    2    3    4    5    6    7    8    9   10   11   12
#> 1631 1439 1528 1450 1449 1366 1503 1831 1661 1759 1620 1566

tab <- prop.table( table( Master$birthMonth ) )

names(tab) <- c("Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec")

dotchart( tab, pch=19, xlab = "Percent of Players", ylab = "Birth Month" )
```



Chapter 3

Data Structures

3.1 Vectors

Vectors are the building blocks of data programming in R, so they are extremely important concepts.

Very loosely speaking a vector is a set of numbers or words. For example, *[1, 2, 3]* or *[apple, orange, pear]*. In social science, a vector is a variable in a dataset.

You will spend a lot of time creating data vectors, transforming variables, generating subsets, cleaning data, and adding new observations. These are all accomplished through operators (commands) that act on vectors.

There are four primary vector types ("classes") in R:

Class	Description
numeric	Typical variable of only numbers
character	A vector of letters or words, always enclosed with quotes
factor	Categories which represent groups, like treatment and control
logical	A vector of TRUE and FALSE to designate which observations fit a criteria

Each vector or dataset has a "class" that tells R the data type.

These different vectors can be combined into three different types of datasets (data frames, matrices, and lists), which will be discussed below.

```r
x1 <- c(167,185,119,142)

x2 <- c("adam","jamal","linda","sriti")

x3 <- factor( c("male","male","female","female") )

x4 <- c( "treatment","control","treatment","control" )

x5 <- x4 == "treatment"

dat <- data.frame( name=x2, sex=x3, treat=x4, is.treat=x5, strength=x1 )
```

name	sex	treat	is.treat	strength
adam	male	treatment	TRUE	167
jamal	male	control	FALSE	185
linda	female	treatment	TRUE	119
sriti	female	control	FALSE	142

R keeps track of the data type of each object:

```
class( x1 )   # c(167,185,119,142)
#> [1] "numeric"
class( x2 )   # c("adam","jamal","linda","sriti")
#> [1] "character"
class( x3 )   # factor( c("male","male","female","female") )
#> [1] "factor"
class( x5 )   # c( "treatment","control","treatment","control" )
#> [1] "logical"
class( dat )  # data.frame( name=x2, sex=x3, treat=x4, is.treat=x5, strength=x1 )
#> [1] "data.frame"
```

We often need to know how many elements belong to a vector, which we find with the **length()** function.

```
x1
#> [1] 167 185 119 142

length( x1 )
#> [1] 4
```

3.2 Casting

You can easily move from one data type to another by *casting* a specific type as another type:

```
x <- 1:5

x
#> [1] 1 2 3 4 5

as.character(x)
#> [1] "1" "2" "3" "4" "5"

y <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

y
#> [1]  TRUE FALSE  TRUE  TRUE FALSE

as.numeric( y )
#> [1] 1 0 1 1 0

as.character( y )
#> [1] "TRUE"  "FALSE" "TRUE"  "TRUE"  "FALSE"
```

But in some cases it might not make sense to cast one variable type as another.

```
z <- c("a", "b", "c")

z
#> [1] "a" "b" "c"

as.numeric( z )
#> [1] NA NA NA
```

Casting will often be induced when you try to combine different types of data. For example, when you add a character element to a numeric vector, the whole vector will be cast as a character vector.

```
x10 <- 1:5

x10
#> [1] 1 2 3 4 5

# a vector can only have one data type

x11 <- c( x10, "a")

x11 # all numbers silently recast as characters
#> [1] "1" "2" "3" "4" "5" "a"
```

When you read data in from outside sources, the input functions often will cast numeric vectors as factors if they contain a low number of elements. See the section on factors below for special instrucdtions on moving from factors to numeric vectors.

3.3 The Combine Function

We often need to combine several elements into a single vector, or two vectors to form one. This is done using the **c()** function.

```
c(1,2,3)
#> [1] 1 2 3

c("a","b","c")
#> [1] "a" "b" "c"

x <- 1:5

y <- 10:15

z <- c(x,y)

z
#>  [1]  1  2  3  4  5 10 11 12 13 14 15

c(z,"a")  # combining data types
#>  [1] "1"  "2"  "3"  "4"  "5"  "10" "11" "12" "13" "14" "15" "a"
```

3.4 Numeric Vectors

There are some specific things to note about each vector type.

Math operators will only work on numeric vectors.

```
summary( x1 )
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   119.0   136.2   154.5   153.2   171.5   185.0
```

Note that if we try to run this mathematicl function we get an error:

```
sum( x2 ) # Error in sum(x2) : invalid 'type' (character) of argument
```

Many functions in R are sensitive to the data type of vectors. Mathematical functions, for example, do not make sense when applied to text (character vectors). In many cases R will give an error. In some cases R will silently re-cast the variable, then perform the operation. Be watchful for when silent re-casting occurs because it might have unwanted side effects, such as deleting data or re-coding group levels in the wrong way.

3.4.1 Integers Are Simple Numeric Vectors

The integer vector is a special type of numeric vector. It is used to save memory since integers require less space than numbers that contain decimals points (you need to allocate space for the numbers to the left and the numbers to the right of the decimal). Google "computer memory allocation" if you are interested in the specifics.

If you are doing advanced programming you will be more sensitive to memory allocation and the speed of your code, but in the intro class we will not differentiate between the two types of number vectors. In *most* cases they result in the same results, unless you are doing advanced numerical analysis where rounding errors matter.

```
n <- 1:5

n
#> [1] 1 2 3 4 5

class( n )
#> [1] "integer"

n[ 2 ] <- 2.01

n  # all elements converted to decimals
#> [1] 1.00 2.01 3.00 4.00 5.00

class( n )
#> [1] "numeric"
```

3.5 Character Vectors

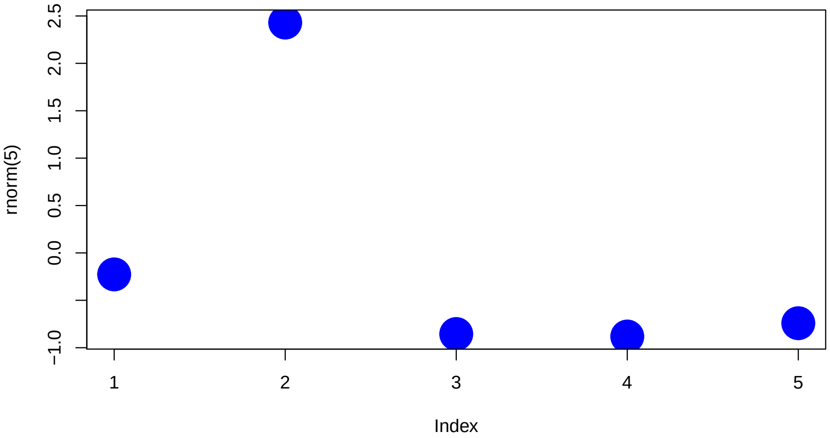
The most important rule to remember with this data type: when creating character vectors, all text must be enclosed by quotation marks.

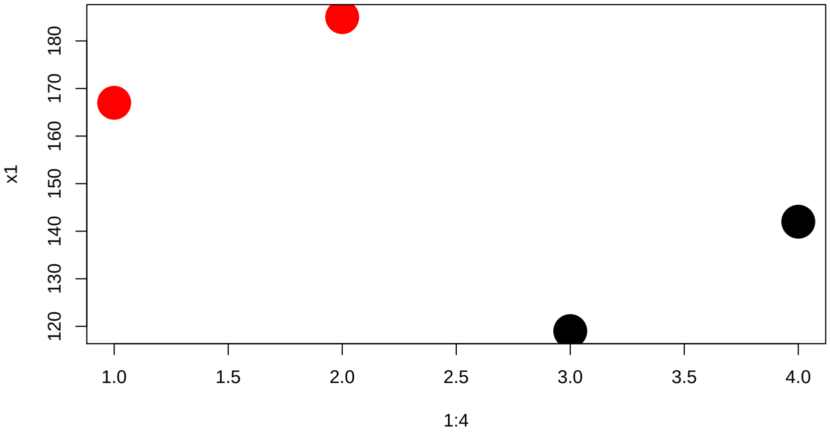
```
c( "a", "b", "c" )    # this works
#> [1] "a" "b" "c"

# c( a, b, c )        # this doesn't work
# Error: object 'a' not found
```

When you type characters surrounded by quotes then R knows you are creating new text ("strings" in programming speak). When you type characters that are not surrounded by quotes, R thinks that you are looking for an object in the environment, like the variables we have already created. It gets confused when it doesn't find the object that you typed.

In generate, you will use quotes when you are creating character vectors, and for arguments in functions. You do not use quotes when you are referencing an active object.

```
ls()   # these are all of the active objects
#>  [1] "dat"         "Master"      "n"           "tab"         "USArrests"
#>  [6] "x"           "x1"          "x10"         "x11"         "x2"
#> [11] "x3"          "x4"          "x5"          "y"           "z"

"x1"
#> [1] "x1"

x1
#> [1] 167 185 119 142

plot( rnorm(5), col="blue", pch=19, cex=4 )    # text used for color argument, use quotes

# what if we want colors that represent groups in our data

plot( 1:4, x1, col=x3, pch=19, cex=4 )        # object used for color argument, no quotes
```





Recall that x3 is the sex of study participants, so the different colors represent the different genders in the study.

3.6 Logical Vectors

Logical vectors are collections of a set of TRUE and FALSE statements.

Logical statements allow us to define groups based upon criteria, then decide whether observations belong to the group. See the section on **operators** below for a complete list of logical statements.

Logical vectors are important because organizing data into these sets is what drives all of the advanced data analytics (set theory is at the basis of mathematics and computer science).

```
dat
#>    name    sex     treat is.treat strength
#> 1  adam   male treatment     TRUE      167
#> 2 jamal   male   control    FALSE      185
#> 3 linda female treatment     TRUE      119
#> 4 sriti female   control    FALSE      142

dat$name == "sriti"
#> [1] FALSE FALSE FALSE  TRUE

dat$sex == "male"
#> [1]  TRUE  TRUE FALSE FALSE

dat$strength > 180
#> [1] FALSE  TRUE FALSE FALSE
```

Typically logical vectors are used in combination with subset operators to identify specific groups in the data.

```
dat
#>    name    sex     treat is.treat strength
#> 1  adam   male treatment     TRUE      167
#> 2 jamal   male   control    FALSE      185
#> 3 linda female treatment     TRUE      119
#> 4 sriti female   control    FALSE      142

# isolate data on all of the females in the dataset

dat[ dat$sex == "female" , ]
#>    name    sex     treat is.treat strength
#> 3 linda female treatment     TRUE      119
#> 4 sriti female   control    FALSE      142
```

When defining logical vectors, you can use the abbreviated versions of T for TRUE and F for FALSE.

```
z1 <- c(T,T,F,T,F,F)

z1
#> [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
```

Note how NAs affect complex logical statements:

```
TRUE & TRUE
#> [1] TRUE

TRUE & FALSE
#> [1] FALSE

TRUE & NA
#> [1] NA

FALSE & NA
#> [1] FALSE
```

If one condition is TRUE, and another is NA, R does not want to throw out the data because the state of the missing value is unclear. As a result, it will preserve the observation, but it will replace all of the data with missing values:

```
dat
#>    name    sex     treat is.treat strength
#> 1  adam   male treatment     TRUE      167
#> 2 jamal   male   control    FALSE      185
#> 3 linda female treatment     TRUE      119
#> 4 sriti female   control    FALSE      142

keep.these <- c(T,F,NA,F)

dat[ keep.these , ]
#>    name  sex     treat is.treat strength
#> 1  adam male treatment     TRUE      167
#> NA <NA> <NA>      <NA>       NA       NA
```

To remove these rows, replace all NAs in your selector vector with FALSE:

```
keep.these[ is.na(keep.these) ] <- FALSE

dat[ keep.these , ]
#>   name  sex     treat is.treat strength
#> 1 adam male treatment     TRUE      167
```

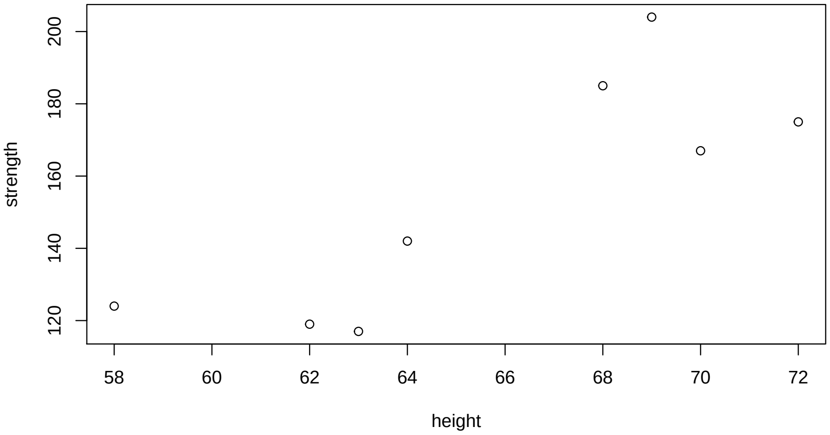
3.7 Factors

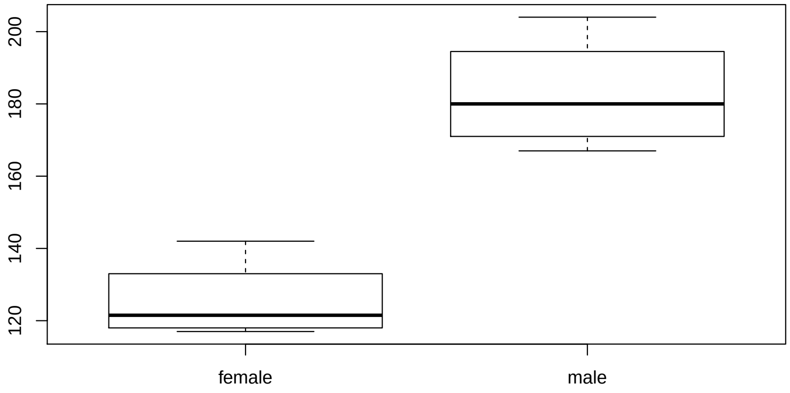
When there are categorical variables within our data, or groups, then we use a special vector to keep track of these groups. We could just use numbers (1=female, 0=male) or characters (“male”,“female”), but factors are useful for two reasons.

First, it saves memory. Text is very “expensive” in terms of memory allocation and processing speed, so using simpler data structure makes R faster.

Second, when a variable is set as a factor, R recognizes that it represents a group and it can deploy object-oriented functionality. When you use a factor in analysis, R knows that you want to split the analysis up by groups.

```
height <- c( 70, 68, 62, 64, 72, 69, 58, 63  )

strength <- c(167,185,119,142,175,204,124,117)

sex <- factor( c("male","male","female","female","male","male","female","female" ) )

plot( height, strength )    # simple scatter plot

plot( sex, strength )       # box and whisker plot
```





Factors are more memory efficient than character vectors because they store the underlying data as a numeric vector instead of a categorical (text) vector. Each group in the data is assigned a number, and when printing items the program only has to remember which group corresponds to which number:

```
sex
#> [1] male   male   female female male   male   female female
#> Levels: female male

as.numeric( sex )
#> [1] 2 2 1 1 2 2 1 1

#  male = 2
#  female = 1
```

If you print a factor, the computer just replaces each category designation with its name (2 would be replaced with "male" in this example). These replacements can be done in real time without clogging the memory of your computer as they don't need to be saved.

In some instances a categorical variable might be represented by numbers. For example, grades 9-12 for high school kids. The **very important** rule to remember with factors is you can't move directly from the factor to numeric using the **as.numeric()** casting function. This will give you the underlying data structure, but will not give you the category names. To get these, you need the **as.character** casting function.

```
grades <- sample( x=9:12, size=10, replace=T )

grades
#>  [1] 10  9 11  9  9 12 12 12  9  9

grades <- as.factor( grades )

grades
#>  [1] 10 9  11 9  9  12 12 12 9  9
#> Levels: 9 10 11 12

as.numeric( grades )
#>  [1] 2 1 3 1 1 4 4 4 1 1

as.character( grades )
```

```
#>  [1] "10" "9"  "11" "9"  "9"  "12" "12" "12" "9"  "9"

# to get back to the original numeric vector

as.numeric( as.character( grades ))
#>  [1] 10  9 11  9  9 12 12 12  9  9
```

Note that when subsetting a factor, it will retain all of the original levels, even when they are not in use.

In this example, there are 37 teams in the Lahman dataset (some of them defunct) and 16 teams in the National League in 2002. But after applying the year and league subsets you will still have 37 levels.

```
# there are only 16 teams in the NL in 2002

sals.2002 <- Salaries [Salaries$yearID=="2002", ]
nl.sals <- sals.2002 [ sals.2002$lgID == "NL",]
levels( nl.sals$teamID )
#>  [1] "ANA" "ARI" "ATL" "BAL" "BOS" "CAL" "CHA" "CHC" "CHN" "CHW" "CIN"
#> [12] "CLE" "COL" "DET" "FLO" "HOU" "KCA" "KCR" "LAA" "LAD" "LAN" "MIA"
#> [23] "MIL" "MIN" "ML4" "MON" "NYA" "NYM" "NYN" "NYY" "OAK" "PHI" "PIT"
#> [34] "SDN" "SDP" "SEA" "SFG" "SFN" "SLN" "STL" "TBA" "TBR" "TEX" "TOR"
#> [45] "WAS" "WSN"
```

After applying a subset, in order to remove the unused factor levels you need to apply either droplevels(), or else recast your factor as a new factor.

For example:

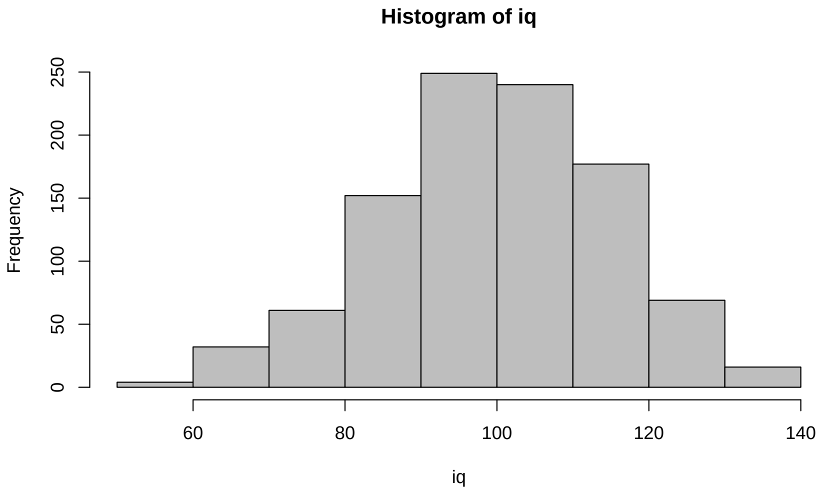
```
sals.2002 <- Salaries [Salaries$yearID=="2002", ]

nl.sals <- sals.2002 [ sals.2002$lgID == "NL",]
levels( nl.sals$teamID )
#>  [1] "ANA" "ARI" "ATL" "BAL" "BOS" "CAL" "CHA" "CHC" "CHN" "CHW" "CIN"
#> [12] "CLE" "COL" "DET" "FLO" "HOU" "KCA" "KCR" "LAA" "LAD" "LAN" "MIA"
#> [23] "MIL" "MIN" "ML4" "MON" "NYA" "NYM" "NYN" "NYY" "OAK" "PHI" "PIT"
#> [34] "SDN" "SDP" "SEA" "SFG" "SFN" "SLN" "STL" "TBA" "TBR" "TEX" "TOR"
#> [45] "WAS" "WSN"

# fix in one of two equivalent ways:
#
# nl.sals$teamID <- droplevels( nl.sals$teamID )
# nl.sals$teamID <- factor( nl.sals$teamID )

levels( nl.sals$teamID )
#>  [1] "ANA" "ARI" "ATL" "BAL" "BOS" "CAL" "CHA" "CHC" "CHN" "CHW" "CIN"
#> [12] "CLE" "COL" "DET" "FLO" "HOU" "KCA" "KCR" "LAA" "LAD" "LAN" "MIA"
#> [23] "MIL" "MIN" "ML4" "MON" "NYA" "NYM" "NYN" "NYY" "OAK" "PHI" "PIT"
#> [34] "SDN" "SDP" "SEA" "SFG" "SFN" "SLN" "STL" "TBA" "TBR" "TEX" "TOR"
#> [45] "WAS" "WSN"
nl.sals$teamID <- droplevels( nl.sals$teamID )
levels( nl.sals$teamID )
#>  [1] "ARI" "ATL" "CHN" "CIN" "COL" "FLO" "HOU" "LAN" "MIL" "MON" "NYN"
#> [12] "PHI" "PIT" "SDN" "SFN" "SLN"
```

3.8 Generating Vectors

You will often need to generate vectors for data transformations or simulations. Here are the most common functions that will be helpful.

```r
# repeat a number, or series of numbers

rep( x=9, times=5 )
#> [1] 9 9 9 9 9

rep( x=c(5,7), times=5 )
#>  [1] 5 7 5 7 5 7 5 7 5 7

rep( x=c(5,7), each=5 )
#>  [1] 5 5 5 5 5 7 7 7 7 7

rep( x=c("treatment","control"), each=5 )  # also works to create categories
#>  [1] "treatment" "treatment" "treatment" "treatment" "treatment"
#>  [6] "control"   "control"   "control"   "control"   "control"


# create a sequence of numbers

seq( from=1, to=15, by=1 )
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

seq( from=1, to=15, by=3 )
#> [1]  1  4  7 10 13

1:15   # shorthand if by=1
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15



# create a random sample

hat <- c("a","b","c","b","f")

sample( x=hat, size=3, replace=FALSE )
#> [1] "c" "a" "b"
sample( x=hat, size=3, replace=FALSE )
#> [1] "b" "b" "a"
sample( x=hat, size=3, replace=FALSE )
#> [1] "b" "a" "c"

# for multiple samples use replacement

sample( x=hat, size=10, replace=TRUE )
#>  [1] "a" "c" "c" "b" "c" "b" "c" "b" "a" "b"
```

```
# create data that follows a normal curve

iq <- rnorm( n=1000, mean=100, sd=15 )

hist( iq, col="gray" )
```



Chapter 4

Operators

Logical operators are the most basic type of data programming and the core of many types of data analysis. Most of the time we are not conducting fancy statistics, we just want to identify members of a group (print all of the females from the study), or describe things that belong to a subset of the data (compare the average price of houses with garages to houses without garages).

In order to accomplish these simple tasks we need to use logic statements. A logic statement answers the question, does an observation belong to a group.

Many times groups are simple. Show me all of the professions that make over $100k a year, for example.

Sometimes groups are complex. Identify the African American children from a specific zip code in Chicago that live in households with single mothers.

You will use nine basic logical operators:

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
x | y	x OR y
x & y	x AND y
!	opposite of

Logical operators create logical vectors, a vector that contains only TRUE or FALSE. The TRUE means that the observation belongs to the group, FALSE means it does not.

```
x1 <- c(7,9,1,2)

x2 <- c("male","male","female","female")

x3 <- c("treatment","control","treatment","control")

x1 > 7
#> [1] FALSE  TRUE FALSE FALSE
```

```
x1 >= 7
#> [1]  TRUE  TRUE FALSE FALSE

x1 == 9
#> [1] FALSE  TRUE FALSE FALSE

x1 = 9  # don't use a single equals operator!  it overwrites your variable

x1
#> [1] 9

x1 <- c(7,9,1,2)

x1 == 9 | x1 == 1
#> [1] FALSE  TRUE  TRUE FALSE

# x2 == male  # this will not work because male is not enclosed with quotes

x2 == "female"
#> [1] FALSE FALSE  TRUE  TRUE

x2 == "female" & x3 == "treatment"
#> [1] FALSE FALSE  TRUE FALSE
```

Note that we use operators to create logical vectors where TRUE designates observation that belong to the defined group, and FALSE designates observations outside the group. We use these logical vectors in three ways:

(1) We can create a selector variable that is used for subsets. When a logical vector is passed to the subset function it will keep all observations with a TRUE value, and drop observations with a FALSE value.

```
x1
#> [1] 7 9 1 2

x1 > 5
#> [1]  TRUE  TRUE FALSE FALSE

keep.these <- x1 > 5

x1[ keep.these ]
#> [1] 7 9

# you can create a selector variable with one variable, and apply it to another

x2[ keep.these ]  # sex of observations where x1 > 5
#> [1] "male" "male"
```

(2) Logical vectors give us an easy way to count things within defined groups.

We can apply a **sum()** function to a logical vector, and the result will be a tally of all of the TRUE cases.

```
# how many females do we have in our study?

sum( x2 == "female" )
#> [1] 2
```

```
# how many females do we have in our treatment group?

sum( x2 == "female" & x3 == "treatment" )
#> [1] 1
```

(3) We use selector variables to replace observations with new values using the assignment operator. This is similar to a find and replace operation.

```
x7 <- c( "mole","mouse","shrew","mouse","rat","shrew")

# the lab assistant incorrectly identified the shrews

x7
#> [1] "mole"  "mouse" "shrew" "mouse" "rat"   "shrew"

x7[ x7 == "shrew" ] <- "possum"

x7
#> [1] "mole"   "mouse"  "possum" "mouse"  "rat"    "possum"


# we don't know if linda received the treatment

x3 <- c("adam","jamal","linda","sriti")

x4 <- c( "treatment","control","treatment","control")

x4[ x3 == "linda" ] <- NA

x4
#> [1] "treatment" "control"   NA          "control"
```

The **!** operator is a special case, where it is not used to define a new logical vector, but rather it swaps the values of an existing logical vector.

```
x1
#> [1] 7 9 1 2

these <- x1 > 5

these
#> [1]  TRUE  TRUE FALSE FALSE

! these
#> [1] FALSE FALSE  TRUE  TRUE

! TRUE
#> [1] FALSE

! FALSE
#> [1] TRUE
```

4.1 Datasets

When we combine multiple vectors together, we now have a dataset. There are three main types that we will use in this class.

Class	Description
data frame	A typical data set comprised of several variables
matrix	A data set comprised of only numbers, used for matrix math
list	The grab bag of data structures - several vectors held together

4.1.1 Data Frames

The most familiar spreadsheet-type data structure is called a data frame in R. It consists of rows, which represent observations, and columns, which represent variables.

```r
data( USArrests )

dim( USArrests )   # number of rows by number of columns
#> [1] 50  4

names( USArrests )   # variable names or column names
#> [1] "Murder"   "Assault"  "UrbanPop" "Rape"

row.names( USArrests )
#>  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
#>  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
#>  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
#> [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
#> [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
#> [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
#> [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
#> [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
#> [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
#> [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
#> [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
#> [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
#> [49] "Wisconsin"      "Wyoming"

head( USArrests )  # print first six rows of the data
#>            Murder Assault UrbanPop Rape
#> Alabama      13.2     236       58 21.2
#> Alaska       10.0     263       48 44.5
#> Arizona       8.1     294       80 31.0
#> Arkansas      8.8     190       50 19.5
#> California    9.0     276       91 40.6
#> Colorado      7.9     204       78 38.7
```

4.1.2 Matrices

A matrix is also a rectangular data object that consists of collections of vectors, but it is special in the sense that it only has numeric vectors and no variable names.

```
mat <- matrix( 1:20, nrow=5 )

mat
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    6   11   16
#> [2,]    2    7   12   17
#> [3,]    3    8   13   18
#> [4,]    4    9   14   19
#> [5,]    5   10   15   20

names( mat )
#> NULL

dim( mat )
#> [1] 5 4

as.data.frame( mat )  # creates variable names
#>   V1 V2 V3 V4
#> 1  1  6 11 16
#> 2  2  7 12 17
#> 3  3  8 13 18
#> 4  4  9 14 19
#> 5  5 10 15 20
```

These are used almost exclusively for matrix algebra operations, which are fundamental to mathematical statistics. We will not use matrices in this course.

4.1.3 Lists

The list is the most flexible data structure. It is created by sticking a bunch of unrelated vectors or datasets together. For example, when you run a regression you generate a bunch of interesting information. This information is saved as a list.

```
x <- 1:100
y <- 2*x + rnorm( 100, 0, 10)

m.01 <- lm( y ~ x )

names( m.01 )
#>  [1] "coefficients"  "residuals"     "effects"       "rank"         
#>  [5] "fitted.values" "assign"        "qr"            "df.residual"  
#>  [9] "xlevels"       "call"          "terms"         "model"        

m.01$coefficients
#> (Intercept)           x 
#>  -0.5043794   1.9985577 

m.01$residuals
#>          1          2          3          4          5          6 
#> -7.3559049 -2.2047624  2.6877683  2.3878862  1.6302649 -7.9711619 
#>          7          8          9         10         11         12 
#> -1.2802661  6.3300627  0.2429119 -4.6427410  5.4029132  1.9405264 
#>         13         14         15         16         17         18 
#>  -4.9317922   1.6882560  -0.2108935  13.0434376  -1.0453205  -2.2810515
#>          19          20          21          22          23          24
#>  -8.5336548  -4.6084283   6.4237313   1.3568726  -5.3062652  -8.3790634
#>          25          26          27          28          29          30
#>  17.4557162   8.4616006   8.1125493  13.2877303   4.3405615 -20.5276203
#>          31          32          33          34          35          36
#>  -1.6807212  -2.6457848  10.5147160   5.8145341   3.8898906   2.6352188
#>          37          38          39          40          41          42
#>  -6.4526023  12.9798312  13.8308778  -5.8406006 -20.1468591  -2.7500585
#>          43          44          45          46          47          48
#> -20.4644534   6.9993682 -19.9155416   1.7284984  -5.8948562   0.5877659
#>          49          50          51          52          53          54
#>   5.8315446   3.7081200   8.6977244 -18.7177596   1.6410328   5.4795727
#>          55          56          57          58          59          60
#>   1.6497880  21.1232521   3.9513602 -14.6290282  -9.3539855 -11.2366011
#>          61          62          63          64          65          66
#>   6.5166020  -5.4553390  -5.2889493  -9.9487999   6.2202358  -4.1928993
#>          67          68          69          70          71          72
#>  -2.9061726   3.1056518  -1.5781122   5.9628746  -9.2307681   8.2549786
#>          73          74          75          76          77          78
#>  14.7068777  15.3941500   6.2179172  -9.9164715   3.0805612  -0.5922107
#>          79          80          81          82          83          84
#>  12.3638649  -8.5165239  -8.2547947   0.3930701  -3.0412042  -8.9960434
#>          85          86          87          88          89          90
#> -16.7984033  12.6843241   8.9197865   8.3100715 -10.3563143  -8.2724792
#>          91          92          93          94          95          96
#>   1.1589511   9.3334340  -6.8427991   7.5781800  -4.6452784  -8.6626880
#>          97          98          99         100
#>   7.7580045   6.6699655  -6.2914860   8.3101291

m.01$call
#> lm(formula = y ~ x)
```

These output are all related to the model we have run, so they are kept organized by the list so they can be used for various further steps like comparing models or checking for model fit.

A data frame is a bit more rigid that a list in that you cannot combine elements that do not have the same dimsions.

```
# new.dataframe <- data.frame( m.01$coefficients, m.01$residuals, m.01$call )
#
# these will fail because the vectors have different lengths
```

4.2 Subsets

The subset operators [ ] are one of the most common you will use in R.

The primary rule of subsets is to use a data operator to create a logical selector vector, and use that to generate subsets. Any observation that corresponds to TRUE will be retained, any observation that corresponds to FALSE will be dropped.

For vectors, you need to specify a single dimension.

```
x1 <- c(167,185,119,142)

x2 <- c("adam","jamal","linda","sriti")

x3 <- factor( c("male","male","female","female") )

x4 <- c( "treatment","control","treatment","control" )

dat <- data.frame( name=x2, sex=x3, treat=x4, strength=x1 )


these <- x1 > 140     # selector vector

these
#> [1]  TRUE  TRUE FALSE  TRUE

x1[ these ]
#> [1] 167 185 142

x2[ these ]
#> [1] "adam"  "jamal" "sriti"
```

For data frames, you need two dimensions (rows and columns). The two dimensions are seperated by a comma, and if you leave one blank you will not drop anything.

```
# dat[  row position ,  column position  ]

dat
#>    name    sex     treat strength
#> 1  adam   male treatment      167
#> 2 jamal   male   control      185
#> 3 linda female treatment      119
#> 4 sriti female   control      142

these <- dat$treat == "treatment"

dat[ these , ]   # all data in the treatment group
#>    name    sex     treat strength
#> 1  adam   male treatment      167
#> 3 linda female treatment      119

dat[ , c("name","sex") ]  # select two columns of data
#>    name    sex
#> 1  adam   male
#> 2 jamal   male
#> 3 linda female
#> 4 sriti female

# to keep a subset as a separate dataset

dat.women <- dat[ dat$sex == "female" , ]

dat.women
```

```
#>    name    sex     treat strength
#> 3 linda female treatment      119
#> 4 sriti female   control      142
```

Note the rules listed above about subsetting factors. After applying a subset, they will retain all of the original levels, even when they are not longer useful. You need to drop the unused levels if you would like them to be omitted from functions that use the factor levels for analysis.

```
df <- data.frame( letters=LETTERS[1:5], numbers=seq(1:5) )

levels( df$letters )
#> [1] "A" "B" "C" "D" "E"

sub.df <- df[ 1:3, ]

sub.df$letters
#> [1] A B C
#> Levels: A B C D E

levels( sub.df$letters )
#> [1] "A" "B" "C" "D" "E"

droplevels( sub.df$letters )
#> [1] A B C
#> Levels: A B C

sub.df$letters <- droplevels( sub.df$letters )
```

4.3 Variable Transformations

When we create a new variable from existing variables, it is called a 'transformation'. This is very common in data science. Crime is measures by the number of assults *per 100,000 people*, for example (crime / pop). A batting average is the number of hits divided by the number of at bats.

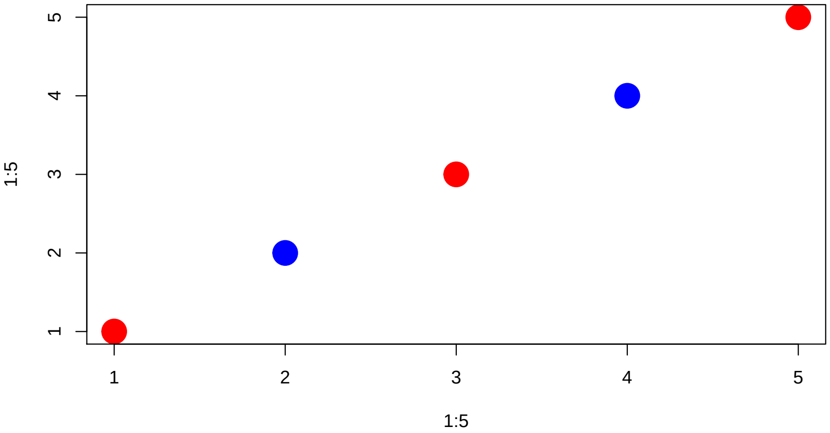
In R, mathematical operations are *vectorized*, which means that operations are performed on the entire vector all at once. This makes transformations fast and easy.

```
x <- 1:10

x + 5
#>  [1]  6  7  8  9 10 11 12 13 14 15

x * 5
#>  [1]  5 10 15 20 25 30 35 40 45 50
```

R uses a convention called "recycling", which means that it will re-use elements of a vector if necessary. In the example below the x vector has 10 elements, but the y vector only has 5 elements. When we run out of y, we just start over from the beginning. This is powerful in some instances, but can be dangerous in others if you don't realize that that it is happening.

```
x <- 1:10

y <- 1:5
```

```
x + y
#>  [1]  2  4  6  8 10  7  9 11 13 15

x * y
#>  [1]  1  4  9 16 25  6 14 24 36 50

# the colors are recycled

plot( 1:5, 1:5, col=c("red","blue"), pch=19, cex=3 )
```



Here is an example of recycling gone wrong:

```
x1 <- c(167,185,119,142)

x2 <- c("adam","jamal","linda","sriti")

x3 <- c("male","male","female","female")

x4 <- c( "treatment","contro","treatment","control" )

dat <- data.frame( name=x2, sex=x3, treat=x4, strength=x1 )


# create a subset of data of all female study participants

dat$sex == "female"
#> [1] FALSE FALSE  TRUE  TRUE

these <- dat$sex == "female"

dat[ these, ]  # correct subset
#>    name    sex     treat strength
#> 3 linda female treatment      119
#> 4 sriti female   control      142
```

```
# same thing, but i mess is up - the female element is recycled in the overwrite

dat$sex = "female"    # whoops just over-wrote my data! should be double equal

these <- dat$sex == "female"

dat[ these , ]
#>    name    sex     treat strength
#> 1  adam female treatment      167
#> 2 jamal female    contro      185
#> 3 linda female treatment      119
#> 4 sriti female   control      142
```

4.4 Missing Values: NA's

Missing values are coded differently in each data analysis program. SPSS uses a period, for example. In R, missing values are coded as "NA".

The important thing to note is that R wants to make sure you know there are missing values if you are conducting analysis. As a result, it will give you the answer of "NA" when you try to do math with a vector that includes a missing value. You have to ask it explicitly to ignore the missing value.

```
x5 <- c( 1, 2, 3, 4 )

x5
#> [1] 1 2 3 4

sum( x5 )
#> [1] 10

mean( x5 )
#> [1] 2.5

x5 <- c( 1, 2, NA, 4 )

x5
#> [1]  1  2 NA  4

# should missing values be treated as zeros or dropped?

sum( x5 )
#> [1] NA

mean( x5 )
#> [1] NA

sum( x5, na.rm=T )   # na.rm=T argument drops missing values
#> [1] 7

mean( x5, na.rm=T )  # na.rm=T argument drops missing values
#> [1] 2.333333
```

You cannot use the == operator to identify missing values in a dataset. There is a special **is.na()** function to locate all of the missing values in a vector.

```
x5
#> [1]  1  2 NA  4

x5 == NA    # this does not do what you want
#> [1] NA NA NA NA

is.na( x5 )    # much better
#> [1] FALSE FALSE  TRUE FALSE

! is.na( x5 )  # if you want to create a selector vector to drop missing values
#> [1]  TRUE  TRUE FALSE  TRUE

x5[ ! is.na(x5) ]
#> [1] 1 2 4

x5[ is.na(x5) ] <- 0  # replace missing values with zero
```

4.5 The ‘attach’ Function

Never Use This!

This is a convenient function for making variable names easily accessible, but it is problematic because of:

scope

conflicting variable names

```
x <- 1:5
y <- 6:10

dat <- data.frame(x,y)

rm(x)
rm(y)


# I want to transform x in my dataset

attach( dat )

2*x
#> [1]  2  4  6  8 10

x <- 2*x

detach( dat )

x
#> [1]  2  4  6  8 10

dat  # whoops! I didn't save my work in the dataset
#>    x  y
#> 1 1  6
#> 2 2  7
#> 3 3  8
#> 4 4  9
#> 5 5 10
```

You will see the **attach()** function used on occassion, and it is tempting because you can write the variable names directly. But in general, try to avoid the **attach()** function and don't form bad habits by using it now because when your scripts become more complicated then can cause problmes.

Chapter 5

Merging Data

5.1 Packages Used in This Chapter

```r
library( pander )
library( dplyr )
library( maps )
```

5.2 Relational Databases

Modern databases are huge - think about the amount of information stored at Amazon in the history of each transation, the database where Google logs every single search from every person around the world, or Twitter's database of all of the tweets (millions each day).

When databases become large, flat spreadsheet style formats are not useful because they create a lot of redundant information, are large to store, and are not efficient to search. Large datasets are instead stored in relational databases - sets of tables that contain unique IDs that allow them to be joined when necessary.

For example, consider a simple customer database. We don't want to store customer info with our transactions because we would be repeating their name and street address every time they make a new purchase. As a result, we store customer information and transaction information separately.

Customer Database

CUSTOMER.ID	FIRST.NAME	LAST.NAME	ADDRESS	ZIP.CODE
178	Alvaro	Jaurez	123 Park Ave	57701
934	Janette	Johnson	456 Candy Ln	57701
269	Latisha	Shane	1600 Penn Ave	20500

Transactions Database

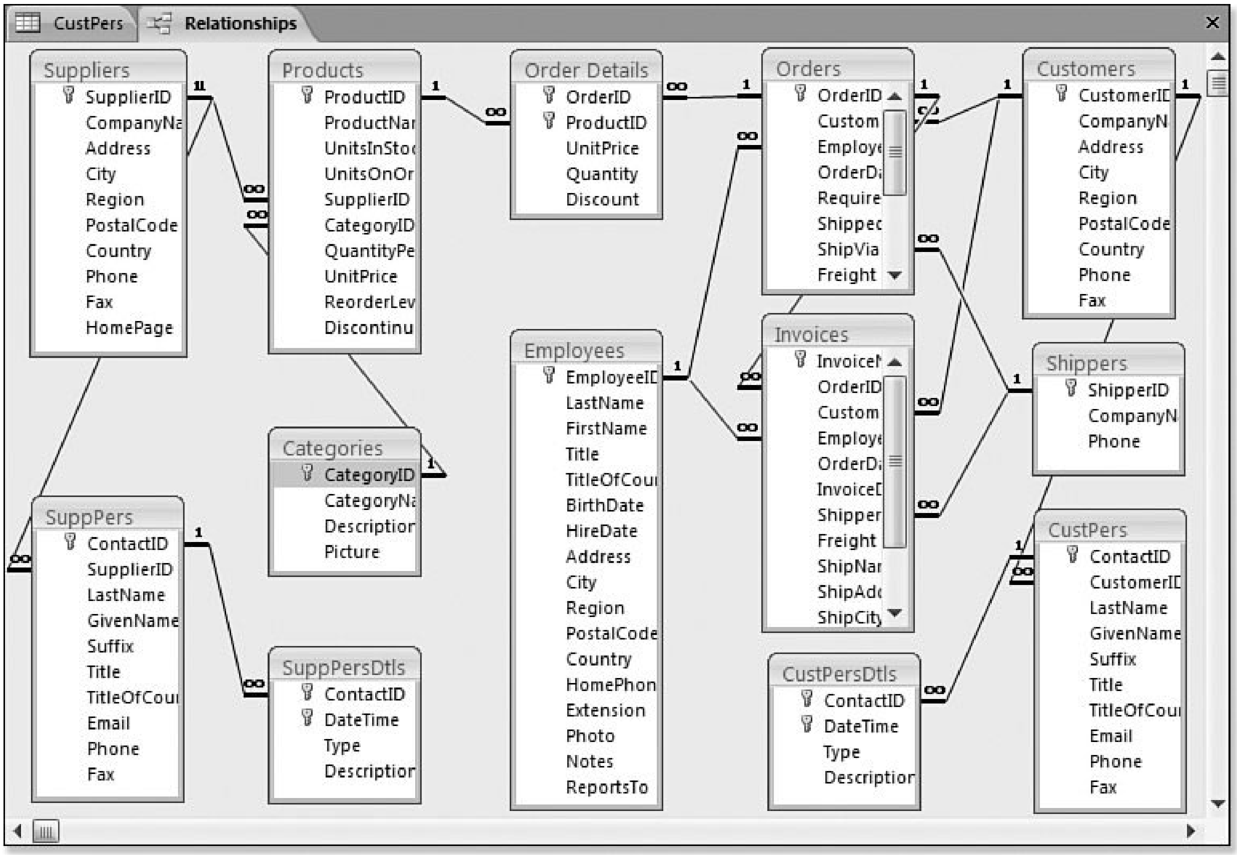
CUSTOMER.ID	PRODUCT	PRICE
178	video	5.38
178	shovel	12
269	book	3.99
269	purse	8
934	mirror	7.64

CUSTOMER.ID	PRODUCT	PRICE

If we want to make the information actionable then we need to combine these datasets. For example, perhaps we want to know the average purchase amount from an individual in the 57701 zip code. We cannot answer that question with either dataset since the zip code is in one dataset, and the price is in another. We need to merge the data.

```
merge( customer.info, purchases )
#>   CUSTOMER.ID FIRST.NAME LAST.NAME       ADDRESS ZIP.CODE PRODUCT PRICE
#> 1         178     Alvaro    Jaurez  123 Park Ave    57701   video  5.38
#> 2         178     Alvaro    Jaurez  123 Park Ave    57701  shovel 12.00
#> 3         269    Latisha     Shane 1600 Penn Ave    20500    book  3.99
#> 4         269    Latisha     Shane 1600 Penn Ave    20500   purse  8.00
#> 5         934    Janette   Johnson  456 Candy Ln    57701  mirror  7.64

full.dat <- merge( customer.info, purchases )

full.dat$PRICE[ full.dat$ZIP.CODE == "57701" ]
#> [1]  5.38 12.00  7.64

mean( full.dat$PRICE[ full.dat$ZIP.CODE == "57701" ] )
#> [1] 8.34
```

In reality, each purchase would have a purchase ID that is linked to shipping addresses, customer complaints, seller ratings, etc. Each seller would have their own data table with info. Each purchase would be tied to a payment type, which has its own data table. The system gets quite complex, which is why it is important to pay attention to the details of putting the data back together again.



We will cover a few details of data merges that will help you avoid common and very subtle mistakes that can lead to incorrect inferences.

5.3 Set Theory

In order to merge data **correctly** you need to understand some very basic principles of set theory.

5.3.1 Set Theory Functions

Let's assume we have two sets: set1=*[A,B]*, set2=*[B,C]*. Each element in this set represents a group of observations that occurs in the dataset. So B represents people that occur in both datasets, A represents people that occur only in the first dataset, and C represents people that only occur in the second dataset.

We can then describe membership through three operations:

Operation	Description
union	The universe of all elements across all both sets: [A,B,C]
intersection	The elements shared by both sets: [B]
difference	The elements in my first set, not in my second [A] or [C]

Let's see how this might work in practice with an example of members of a study:

```
name <- c("frank","wanda","sanjay","nancy")
group <- c("treat","treat","control","control")
gender <- c("male","female","male","female")

data.frame( name, group, gender ) %>% pander
```

name	group	gender
frank	treat	male
wanda	treat	female
sanjay	control	male
nancy	control	female

For this example let's define set 1 as the treatment group, and set 2 as all women in the study. Note that set membership is always defined as binary (you are in the set or out), but it can include multiple criteria (the set of animals can contains cats, dogs, and mice).

```
treated <- name[ group == "treat" ]

treated
#> [1] "frank" "wanda"

females <- name[ gender == "female" ]

females
#> [1] "wanda" "nancy"
```

Now we can specify group belonging using some convenient set theory functions: **union()**, **setdiff()**, and **intersect()**.

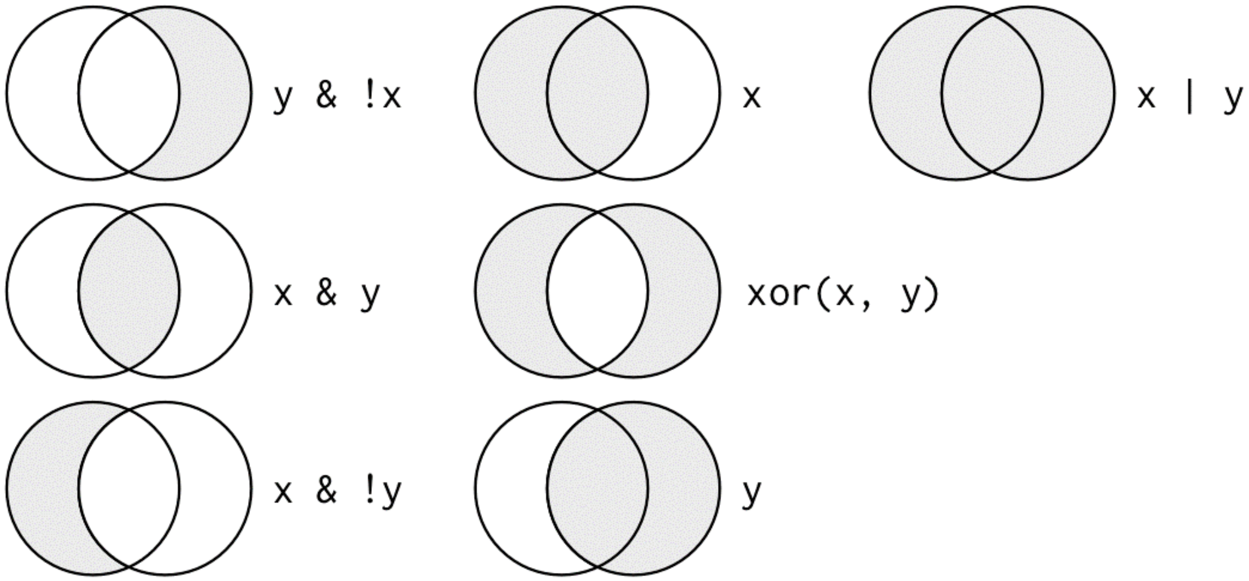
```

union( treated, females )
#> [1] "frank" "wanda" "nancy"

intersect( treated, females )
#> [1] "wanda"

setdiff( treated, females )
#> [1] "frank"

setdiff( females, treated )
#> [1] "nancy"
```

It is very important to note that **union()** and **intersect()** are symmetric functions, meaning *intersect(x,y)* will give you the same result as *intersect(y,x)*. The **setdiff()** function is not symmetric, however.

5.3.2 Set Theory Using Logical Operators

Typically you will define your groups using logical operators, which perform the exact same funciton as set theory functions but are a little more expressive and flexible.

Let's use the same example above where x="treatment" and y="female", then consider these cases:



Who belongs in each group?

```
#   x

name[ group == "treat" ]
#> [1] "frank" "wanda"

#   x & y

name[ group == "treat" & gender == "female" ]
#> [1] "wanda"

#   x & ! y

name[ group == "treat" & gender != "female" ]
#> [1] "frank"

#  x | y

name[ group == "treat" | gender == "female" ]
#> [1] "frank" "wanda" "nancy"
```

Who belongs in these groups?

- !x & !y
- x & ! (x & y)
- (x | y) & ! (x & y)

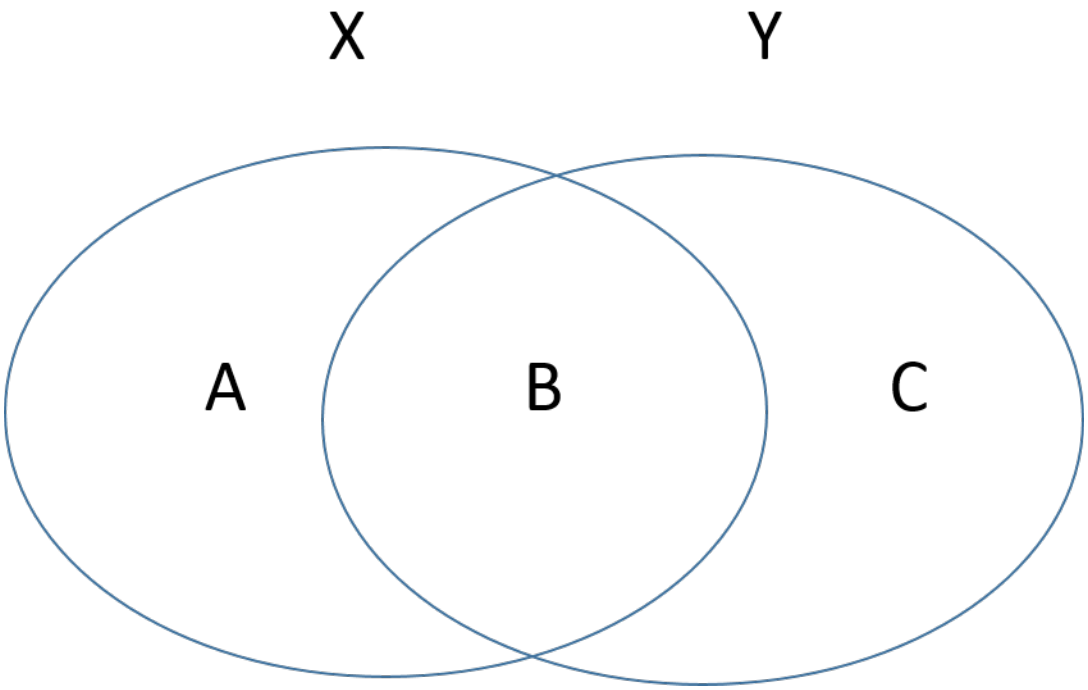
5.4 Merging Data

The Merge Function

The merge function joins two datasets. The function requires two datasets as the arguments, and they need to share a unique ID variable. Recall the example from above:

```
merge( customer.info, purchases )
#>   CUSTOMER.ID FIRST.NAME LAST.NAME       ADDRESS ZIP.CODE PRODUCT PRICE
#> 1         178     Alvaro    Jaurez  123 Park Ave    57701   video  5.38
#> 2         178     Alvaro    Jaurez  123 Park Ave    57701  shovel 12.00
#> 3         269    Latisha     Shane 1600 Penn Ave    20500    book  3.99
#> 4         269    Latisha     Shane 1600 Penn Ave    20500   purse  8.00
#> 5         934    Janette   Johnson  456 Candy Ln    57701  mirror  7.64
```

The important thing to keep in mind is that the default merge operation uses the **intersection** of the two datasets. It will drop all elements that don't occur in both datasets. We may want to fine-tune this as to not lose valuable data and potentially bias our analysis. As an example, no illegal immigrants will have social security numbers, so if you are merging using the SSN, you will drop this group from the data, which could impact your results.



With a little help from the set theory examples above, we can think about which portions of the data we wish to drop and which portions we wish to keep.

Argument	Usage
all=F	DEFAULT - new dataset contains intersection of X and Y (B only)
all=T	New dataset contains union of X and Y (A, B & C)
all.x=T	New dataset contains A and B, not C
all.y=T	New dataset contains B and C, not A

Here is some demonstrations with examples adapted from the R help file.

```
authors
#>   surname nationality deceased
#> 1   Tukey          US      yes
#> 2 Tierney          US       no
#> 3      Ripley          UK       no
#> 4      McNeil   Australia       no
#> 5 Shakespeare     England      yes

books
#>          name                     title
#> 1       Tukey Exploratory Data Analysis
#> 2    Venables Modern Applied Statistics
#> 3      Ripley        Spatial Statistics
#> 4      Ripley     Stochastic Simulation
#> 5      McNeil Interactive Data Analysis
#> 6 R Core Team      An Introduction to R



# adding books to the author bios dataset  ( set B only )

merge(authors, books, by.x = "surname", by.y = "name")
#>   surname nationality deceased                     title
#> 1  McNeil   Australia       no Interactive Data Analysis
#> 2  Ripley          UK       no        Spatial Statistics
#> 3  Ripley          UK       no     Stochastic Simulation
#> 4   Tukey          US      yes Exploratory Data Analysis



# adding author bios to the books dataset  ( set B only )

merge(books, authors, by.x = "name", by.y = "surname")
#>     name                     title nationality deceased
#> 1 McNeil Interactive Data Analysis   Australia       no
#> 2 Ripley        Spatial Statistics          UK       no
#> 3 Ripley     Stochastic Simulation          UK       no
#> 4  Tukey Exploratory Data Analysis          US      yes



# keep books without author bios, lose authors without books  ( sets A and B )

merge( books, authors, by.x = "name", by.y = "surname", all.x=T )
#>          name                     title nationality deceased
#> 1      McNeil Interactive Data Analysis   Australia       no
#> 2 R Core Team      An Introduction to R        <NA>     <NA>
#> 3      Ripley        Spatial Statistics          UK       no
#> 4      Ripley     Stochastic Simulation          UK       no
#> 5       Tukey Exploratory Data Analysis          US      yes
#> 6    Venables Modern Applied Statistics        <NA>     <NA>



# keep authors without book listed, lose books without author bios   ( sets B and C )

merge( books, authors, by.x = "name", by.y = "surname", all.y=T )
```

```
#>          name                     title nationality deceased
#> 1      McNeil Interactive Data Analysis   Australia       no
#> 2      Ripley        Spatial Statistics          UK       no
#> 3      Ripley     Stochastic Simulation          UK       no
#> 4 Shakespeare                      <NA>     England      yes
#> 5     Tierney                      <NA>          US       no
#> 6       Tukey Exploratory Data Analysis          US      yes



# dont' throw out any data   ( sets A and B and C )

merge( books, authors, by.x = "name", by.y = "surname", all=T )
#>          name                     title nationality deceased
#> 1      McNeil Interactive Data Analysis   Australia       no
#> 2 R Core Team      An Introduction to R        <NA>     <NA>
#> 3      Ripley        Spatial Statistics          UK       no
#> 4      Ripley     Stochastic Simulation          UK       no
#> 5 Shakespeare                      <NA>     England      yes
#> 6     Tierney                      <NA>          US       no
#> 7       Tukey Exploratory Data Analysis          US      yes
#> 8    Venables Modern Applied Statistics        <NA>     <NA>
```

Also note that the order of your datasets in the argument list will impact the inclusion or exclusion of elements.

merge(x, y, all=F) EQUALS merge(y, x, all=F)

merge(x, y, all.x=T) DOES NOT EQUAL merge(y, x, all.x=T)

5.4.1 The by.x and by.y Arguments

When you use the default **merge()** function without specifying the variables to merge upon, the function will check for common variable names across the two datasets. If there are multiple, it will join the shared variables to create a new unique key. This might be problematic if that was not the intent.

Take the example of combining fielding and salary data in the Lahman package. If we are not explicit about the merge variable, we may get odd results. Note that they two datasets share four ID variables.

```
library( Lahman )
data( Fielding )
data( Salaries )
```

```


intersect( names(Fielding), names(Salaries) )
#> [1] "playerID" "yearID"   "teamID"   "lgID"

# merge id

int <- intersect( names(Fielding), names(Salaries) )

paste( int[1],int[2],int[3],int[4], sep="." )
#> [1] "playerID.yearID.teamID.lgID"
```

To avoid problems, be explicit using the *by.x* and *by.x* arguments to control which variable is used for the merge.

```
head( merge( Salaries, Fielding ) )
#>   yearID teamID lgID  playerID salary stint POS  G GS InnOuts  PO  A E DP
#> 1   1985    ATL   NL barkele01 870000     1   P 20 18     221   2  9 1  0
#> 2   1985    ATL   NL bedrost01 550000     1   P 37 37     620  13 23 4  3
#> 3   1985    ATL   NL benedbr01 545000     1   C 70 67    1698 314 35 4  1
#> 4   1985    ATL   NL  campri01 633333     1   P 66  2     383   7 13 4  3
#> 5   1985    ATL   NL ceronri01 625000     1   C 91 76    2097 384 48 6  4
#> 6   1985    ATL   NL chambch01 800000     1  1B 39 27     814 299 25 1 31
#>   PB WP SB CS ZR
#> 1 NA NA NA NA NA
#> 2 NA NA NA NA NA
#> 3  1  9 65 24  1
#> 4 NA NA NA NA NA
#> 5  6 20 69 29  1
#> 6 NA NA NA NA NA

head( merge( Salaries, Fielding, by.x="playerID", by.y="playerID" ) )
#>    playerID yearID.x teamID.x lgID.x  salary yearID.y stint teamID.y
#> 1 aardsda01     2010      SEA     AL 2750000     2009     1      SEA
#> 2 aardsda01     2010      SEA     AL 2750000     2015     1      ATL
#> 3 aardsda01     2010      SEA     AL 2750000     2006     1      CHN
#> 4 aardsda01     2010      SEA     AL 2750000     2008     1      BOS
#> 5 aardsda01     2010      SEA     AL 2750000     2013     1      NYN
#> 6 aardsda01     2010      SEA     AL 2750000     2012     1      NYA
#>   lgID.y POS  G GS InnOuts PO A E DP PB WP SB CS ZR
#> 1     AL   P 73  0     214  2 5 0  1 NA NA NA NA NA
#> 2     NL   P 33  0      92  0 1 1  0 NA NA NA NA NA
#> 3     NL   P 45  0     159  1 5 0  1 NA NA NA NA NA
#> 4     AL   P 47  0     146  3 6 0  0 NA NA NA NA NA
#> 5     NL   P 43  0     119  1 5 0  0 NA NA NA NA NA
#> 6     AL   P  1  0       3  0 0 0  0 NA NA NA NA NA
```

5.5 Non-Unique Observations in ID Variables

In some rare instances, you will need to merge to datasets that have non-singular elements in the unique key ID variables, meaning each observation / individual appears more than one time in the data. Note that in this case, for each occurance of an observation / individual in your X dataset, you will merge once with each occurance of the same observation / individual in the Y dataset. The result will be a multiplicative expansion of the size of your dataset.

For example, if John appears on four separate rows of X, and three seperate rows of Y, the new dataset will contain 12 rows of John (4 x 3 = 12).

dataset X contains four separate instances of an individual [X1, X2, X3, X4]

dataset Y contains three separate instances of an individual [Y1, Y2, Y3]

After the merge we have one row for each pair:

X1-Y1
X1-Y2
X1-Y3

X2-Y1
X2-Y2
X2-Y3
X3-Y1
X3-Y2
X3-Y3
X4-Y1
X4-Y2
X4-Y3

For example, perhaps a sales company has a database that keeps track of biographical data, and sales performance. Perhaps we want to see if there is peak age for sales performance. We need to merge these datasets.

```
bio <- data.frame( name=c("John","John","John"),
                   year=c(2000,2001,2002),
                   age=c(43,44,45) )

performance <- data.frame( name=c("John","John","John"),
                           year=c(2000,2001,2002),
                           sales=c("15k","20k","17k") )

# correct merge

merge( bio, performance, by.x=c("name","year"), by.y=c("name","year") )
#>   name year age sales
#> 1 John 2000  43   15k
#> 2 John 2001  44   20k
#> 3 John 2002  45   17k


# incorrect merge

merge( bio, performance, by.x=c("name"), by.y=c("name") )
#>   name year.x age year.y sales
#> 1 John   2000  43   2000   15k
#> 2 John   2000  43   2001   20k
#> 3 John   2000  43   2002   17k
#> 4 John   2001  44   2000   15k
#> 5 John   2001  44   2001   20k
#> 6 John   2001  44   2002   17k
#> 7 John   2002  45   2000   15k
#> 8 John   2002  45   2001   20k
#> 9 John   2002  45   2002   17k
```

It is good practice to check the size (number of rows) of your dataset before and after a merge. If it has expanded, chances are you either used the wrong unique IDs, or your dataset contains duplicates.

5.5.1 Example of Incorrect Merge

Here is a tangible example using the Lahman baseball dataset. Perhaps we want to examine the relationship between fielding position and salary. The *Fielding* dataset contains fielding position information, and the *Salaries* dataset contains salary information. We can merge these two datasets using the *playerID* field.

If we are not thoughtful about this, however, we will end up causing problems. Let's look at an example using Kirby Pucket.

```
kirby.fielding <- Fielding[ Fielding$playerID == "puckeki01" , ]

head( kirby.fielding )
#>        playerID yearID stint teamID lgID POS   G  GS InnOuts  PO  A E DP
#> 83848 puckeki01   1984     1    MIN   AL  OF 128 128    3377 438 16 3  4
#> 85157 puckeki01   1985     1    MIN   AL  OF 161 160    4213 465 19 8  5
#> 86489 puckeki01   1986     1    MIN   AL  OF 160 157    4155 429  8 6  3
#> 87896 puckeki01   1987     1    MIN   AL  OF 147 147    3820 341  8 5  2
#> 89264 puckeki01   1988     1    MIN   AL  OF 158 157    4049 450 12 3  4
#> 90685 puckeki01   1989     1    MIN   AL  OF 157 154    3985 438 13 4  3
#>       PB WP SB CS ZR
#> 83848 NA NA NA NA NA
#> 85157 NA NA NA NA NA
#> 86489 NA NA NA NA NA
#> 87896 NA NA NA NA NA
#> 89264 NA NA NA NA NA
#> 90685 NA NA NA NA NA

nrow( kirby.fielding )
#> [1] 21

kirby.salary <- Salaries[ Salaries$playerID == "puckeki01" , ]

head( kirby.salary )
#>      yearID teamID lgID  playerID  salary
#> 280    1985    MIN   AL puckeki01  130000
#> 917    1986    MIN   AL puckeki01  255000
#> 1610   1987    MIN   AL puckeki01  465000
#> 2244   1988    MIN   AL puckeki01 1090000
#> 2922   1989    MIN   AL puckeki01 2000000
#> 3717   1990    MIN   AL puckeki01 2816667

nrow( kirby.salary )
#> [1] 13

kirby.field.salary <- merge( kirby.fielding, kirby.salary, by.x="playerID", by.y="playerID" )

head( select( kirby.field.salary, yearID.x, yearID.y,   POS,    G,  GS, salary ) )
#>   yearID.x yearID.y POS   G  GS  salary
#> 1     1984     1985  OF 128 128  130000
#> 2     1984     1986  OF 128 128  255000
#> 3     1984     1987  OF 128 128  465000
#> 4     1984     1988  OF 128 128 1090000
#> 5     1984     1989  OF 128 128 2000000
#> 6     1984     1990  OF 128 128 2816667

nrow( kirby.field.salary )
#> [1] 273

21*13
#> [1] 273
```

What we have done here is taken each year of fielding data, and matched it to **every** year of salary data. We can see that we have 21 fielding observations and 13 years of salary data, so our resulting dataset is 273 observation pairs.

This merge also makes it difficult to answer the question of the relationship between fielding position and salary if players change positions over time.

The correct merge in this case would be a merge on a playerID-yearID pair. We can create a unique key by combining playerID and yearID using **paste()**:

```
head( paste( kirby.fielding$playerID, kirby.fielding$yearID, sep=".") )
#> [1] "puckeki01.1984" "puckeki01.1985" "puckeki01.1986" "puckeki01.1987"
#> [5] "puckeki01.1988" "puckeki01.1989"
```

But there is a simple solution as the merge function also allows for multiple variables to be used for a **merge()** command.

```
kirby.field.salary <- merge( kirby.fielding, kirby.salary,
                             by.x=c("playerID","yearID"),
                             by.y=c("playerID","yearID")   )

nrow( kirby.field.salary )
#> [1] 20
```

5.6 The %in% function

Since we are talking about intersections and matches, I want to briefly introduce the **%in%** function. It is a combination of the two.

The **intersect()** function returns a list of unique matches between two vectors.

```
data(Salaries)
data(Fielding)
intersect( names(Salaries), names(Fielding) )
#> [1] "yearID"   "teamID"   "lgID"     "playerID"
```

The **match()** function returns the position of matched elements.

```
x <- c("A","B","C","B")

y <- c("B","D","A","F")

match( x, y )
#> [1]  3  1 NA  1
```

The **%in%** function returns a logical vector, where TRUE signifies that the element in y also occurs in x. In other words, does a specific element in y belong to the intersection of x,y.

This is very useful for creating subsets of data that belong to both sets.

```
x <- c("A","B","C")

y <- c("B","D","A","B","F","B")

y %in% x # does each element of y occur anywhere in x?
#> [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE
```

```
y[ y %in% x] # keep only data that occurs in both
#> [1] "B" "A" "B" "B"
```

5.7 The Match Function

Often times we do not need to merge data, we may just need sort data in one dataset so that it matches the order of another dataset. This is accomplished using the **match()** function.

Note that we can rearrange the order of a dataset by referencing the desired position.

```
x <- c("Second","Third","First")

x
#> [1] "Second" "Third"  "First"

x[ c(3,1,2) ]
#> [1] "First"  "Second" "Third"
```

The **match()** function returns the *positions* of matches of its *first* vector to the *second* vector listed in the arguments. Or in other words, the *order* that vector 2 would need to follow to match vector 1.

```
x <- c("A","B","C")

y <- c("B","D","A")

cbind( x, y )
#>      x   y
#> [1,] "A" "B"
#> [2,] "B" "D"
#> [3,] "C" "A"

match( x, y )
#> [1]  3  1 NA

match( y, x) # not a symmetric operation!
#> [1]  2 NA  1

# In the y vector:
#
#  [3]=A
#  [1]=B
# [NA]=D (no match)

order.y <- match( x, y )

y[ order.y ]
#> [1] "A" "B" NA
```

We can see that **match()** returns the correct order to put y in so that it matches the order of x. In the re-ordered vector, the first element is the original third element A, the second element is the original first element B, and there is no third element because D did not match anything in x.

Note the order of arguments in the function:

match(data I want to match to , data I need to re-order)

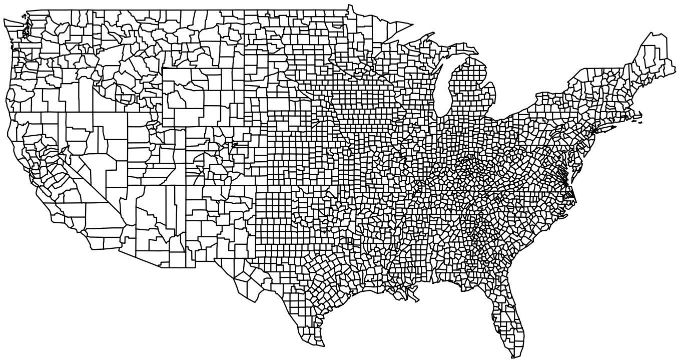
We can use this position information to re-order *y* as follows:

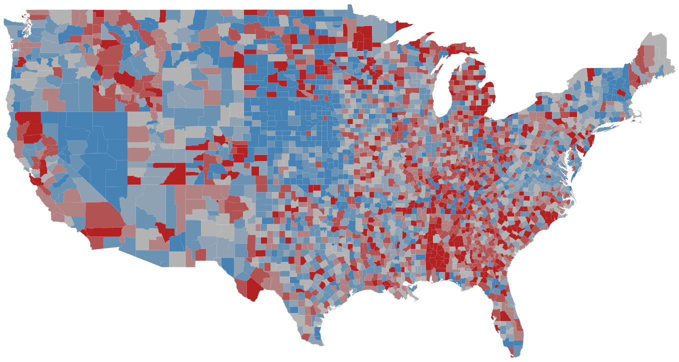
```
x <- sample( LETTERS[1:15], size=10 )

y <- sample( LETTERS[1:15], size=10 )

cbind( x, y )
#>       x   y  
#>  [1,] "E" "J"
#>  [2,] "N" "L"
#>  [3,] "C" "C"
#>  [4,] "B" "N"
#>  [5,] "F" "F"
#>  [6,] "G" "M"
#>  [7,] "A" "I"
#>  [8,] "J" "G"
#>  [9,] "O" "D"
#> [10,] "H" "B"

order.y <- match( x, y )

y.new <- y[ order.y ]

cbind( x, y.new )
#>       x   y.new
#>  [1,] "E" NA   
#>  [2,] "N" "N"  
#>  [3,] "C" "C"  
#>  [4,] "B" "B"  
#>  [5,] "F" "F"  
#>  [6,] "G" "G"  
#>  [7,] "A" NA   
#>  [8,] "J" "J"  
#>  [9,] "O" NA   
#> [10,] "H" NA   


# Note the result if you confuse the order or arguments

order.y <- match( y, x )

y.new <- y[ order.y ]

cbind( x, y.new )
#>       x   y.new
#>  [1,] "E" "G"  
#>  [2,] "N" NA   
#>  [3,] "C" "C"  
#>  [4,] "B" "L"  
#>  [5,] "F" "F"  
#>  [6,] "G" NA   
#>  [7,] "A" NA   
#>  [8,] "J" "M"
#>  [9,] "O" NA
#> [10,] "H" "N"
```

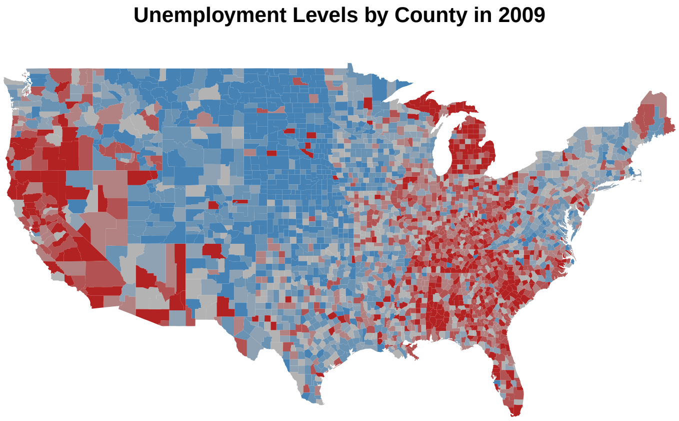
This comes in handy when we are matching information between two tables. For example, in GIS the map regions follow a specific order but your data does not. Create a color scheme for levels of your data, and then re-order the colors so they match the correct region on the map. In this example, we will look at unemployment levels by county.

```
library( maps )
data( county.fips )
data( unemp )

map( database="county" )

# assign a color to each level of unemployment, red = high, gray = medium, blue = low

color.function <- colorRampPalette( c("steelblue", "gray70", "firebrick") )


color.vector <- cut( rank(unemp$unemp), breaks=7, labels=color.function( 7 ) )

color.vector <- as.character( color.vector )

head( color.vector )
#> [1] "#B28282" "#B28282" "#B22222" "#B25252" "#B28282" "#B22222"

# doesn't look quite right

map( database="county", col=color.vector, fill=T, lty=0 )


# what went wrong here?

# our unemployment data (and thus the color vector) follows a different order

cbind( map.id=county.fips$fips, data.id=unemp$fips, color.vector )[ 2500:2510 , ]
#>       map.id  data.id color.vector
#>  [1,] "48011" "47149" "#B28282"
#>  [2,] "48013" "47151" "#B22222"
#>  [3,] "48015" "47153" "#B22222"
#>  [4,] "48017" "47155" "#B28282"
#>  [5,] "48019" "47157" "#B28282"
#>  [6,] "48021" "47159" "#B22222"
#>  [7,] "48023" "47161" "#B25252"
#>  [8,] "48025" "47163" "#B3B3B3"
#>  [9,] "48027" "47165" "#B28282"
#> [10,] "48029" "47167" "#B25252"
#> [11,] "48031" "47169" "#B25252"

# place the color vector in the correct order

this.order <- match( county.fips$fips, unemp$fips )
```

```
color.vec.ordered <- color.vector[ this.order ]

# colors now match their correct counties

map( database="county", col=color.vec.ordered, fill=T, lty=0 )
title( main="Unemployment Levels by County in 2009")
```







Note that elements can be recycled from your *y* vector:

```
x <- c("A","B","C","B")

y <- c("B","D","A","F")

cbind( x, y )
#>      x   y
#> [1,] "A" "B"
#> [2,] "B" "D"
#> [3,] "C" "A"
#> [4,] "B" "F"

match( x, y )
#> [1]  3  1 NA  1

order.y <- match( x, y )

y.new <- y[ order.y ]

cbind( x, y.new )
#>      x   y.new
#> [1,] "A" "A"
#> [2,] "B" "B"
#> [3,] "C" NA
#> [4,] "B" "B"
```

Chapter 6

Analysis with Groups in R

6.0.1 Packages Used in this Chapter

```
library( pander )
library( dplyr )
library( tidyr )
library( reshape2 )
library( scales )
library( ggplot2 )
library( Lahman )
```

6.0.2 Hypothetical Experimental Data

We will demonstrate some functions using this hypothetical dataset:

```
head( d ) %>% pander
```

id	race	blood.type	gender	age	study.group	speed
1	white	A	male	50	treatment	509.6
2	white	B	female	46	treatment	654.7
3	white	A	female	29	treatment	658.5
4	black	A	female	59	treatment	645.8
5	white	A	female	87	treatment	45.22
6	white	A	female	36	treatment	679

6.1 Group Structure

The two most important skills as you first learn a data programming language are:

1. Translating English phrases into computer code using logical statements

2. Organizing your data into groups

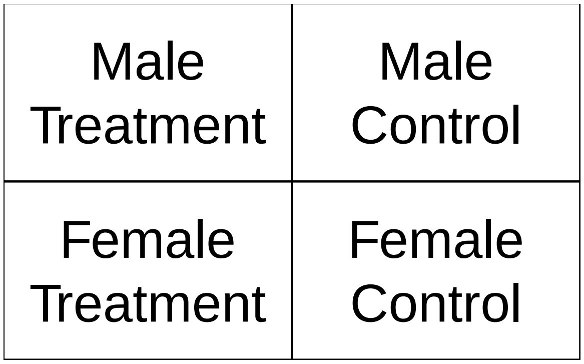
This lecture focuses on efficienty splitting your data into groups, and then analyzing your data by group.

6.1.1 What Are Groups?

A group represents a set of elements with identical characteristics - mice all belong to one group and elephants belong to another. Easy enough, right?

In data analysis, it is a little more complicated because a group is defined by a set of features. Each group still represents a set of elements with identical characteristics, but when we have multiple features there is a unique group for each combination of features.

The simple way to think about this is that the cross-tab of features generates a grid (table), and each cell represents a unique group:



Male Treatment	Male Control
Female Treatment	Female Control

We might be interested in simple groups (treatment cases versus control cases), or complex groups (does the treatment effect women and men differently?).

In previous lectures you have learned to identify a group with a logical statement, and analyze that group discretely.

```r
mean( speed[ study.group == "treatment" & gender=="female" ] )
#> [1] 614.0334
```

In this lecture you will learn to define a group structure, then analyze all of your data using that structure.

```r
tapply( speed, INDEX = list( study.group, gender ), FUN = mean )
```

	female	male
control	462.5	364.1
treatment	614	504.5

6.1.2 Main Take-Away

R has been designed to do efficient data analysis by defining a group structure, then quickly applying a function to all unique members.

The base R packages do this with a set of functions in the **apply()** family. The **tapply()** function allows you to specify an outcome to analyze and a group, then ask for results from a function.

```r
tapply( X=speed, INDEX=list( study.group, gender ), FUN=mean )
```

	female	male
control	462.5	364.1
treatment	614	504.5

The **dplyr** package makes this process easier using some simple verbs and the “pipe” operator.

```
dat  %>%  group_by( study.group, gender )  %>%  summarize( ave.speed = mean(speed) )
```

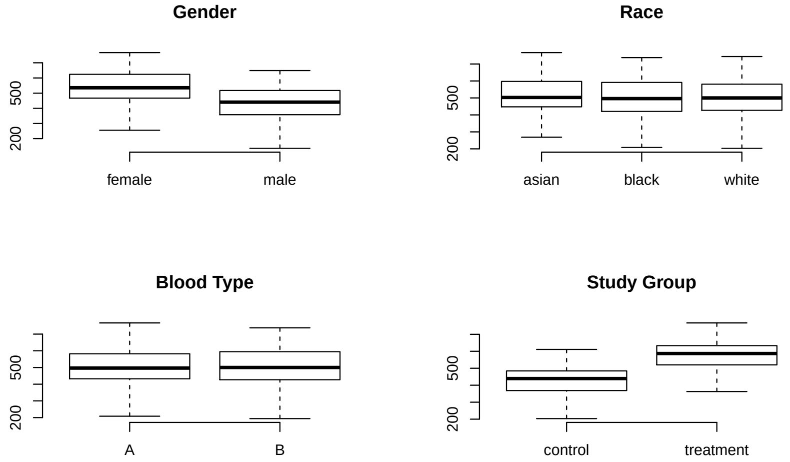
study.group	gender	ave.speed
control	male	364.1
control	female	462.5
treatment	male	504.5
treatment	female	614

6.1.3 Example

Let’s think about a study looking at reading speed. The treatment is a workshop that teaches some speed-reading techniques. In this study we have data on:

- gender (male,female)
- race (black,white,asian)
- blood.type (A,B)
- age (from 18 to 93)

Examining descriptive statistics we can see that reading speed varies by gender and the treatment group, but not by race or blood type:



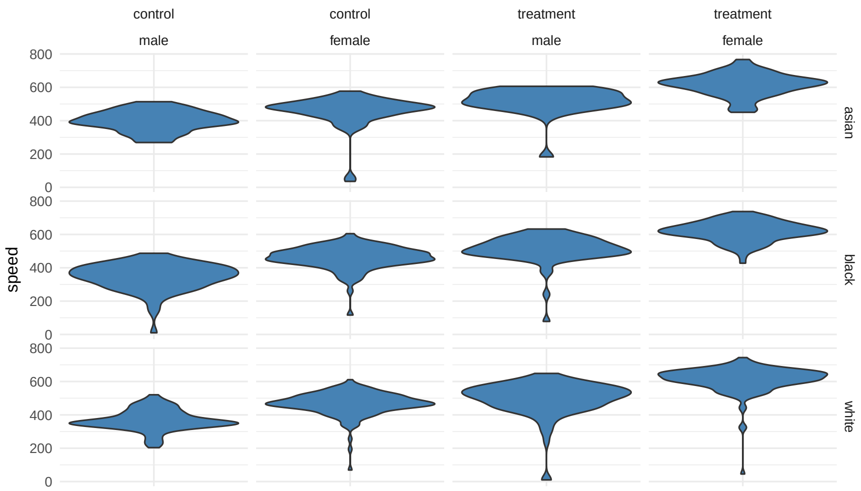
The question is, how many unique groups can we create with these four factors?

Each individual factor contains a small number of levels (only 2 or 3 in this case), which makes the group structure look deceptively simple at first glance. When we start to examine combinations of factors we see that group structure can get complicated pretty quickly.

If we look at gender alone, we have two levels: male and female. So we have two groups. If we look at our study groups alone we have two groups: treatment and control.

If we look at gender and the study groups together, we now have a 2 x 2 grid, or four unique groups.

If the race factor has three levels, how many unique groups will we have considering the study design, gender, and race together?

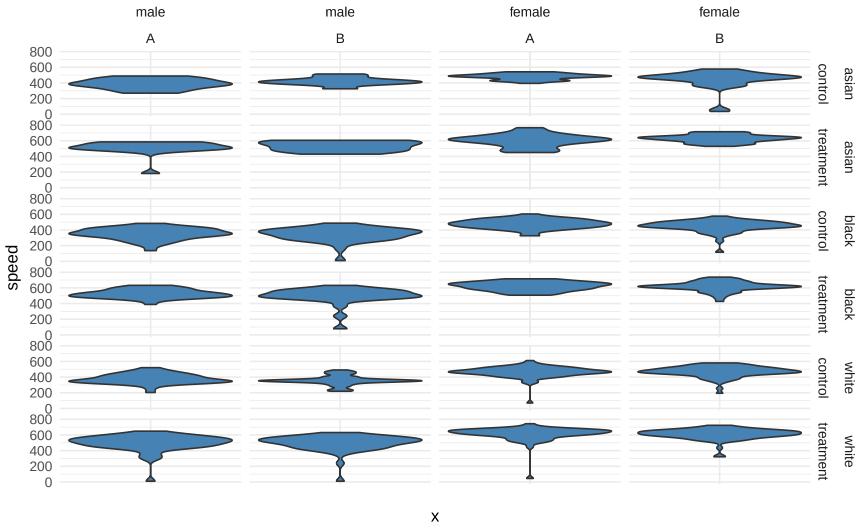


We can calculate the size of the grid by multiplying number of levels for each factor. We see here we have 12 unique groups:

```
nlevels( gender ) * nlevels( study.group ) * nlevels( race )
#> [1] 12
```

If we add blood type, a factor with two levels (A and B), we now have 24 unique groups:

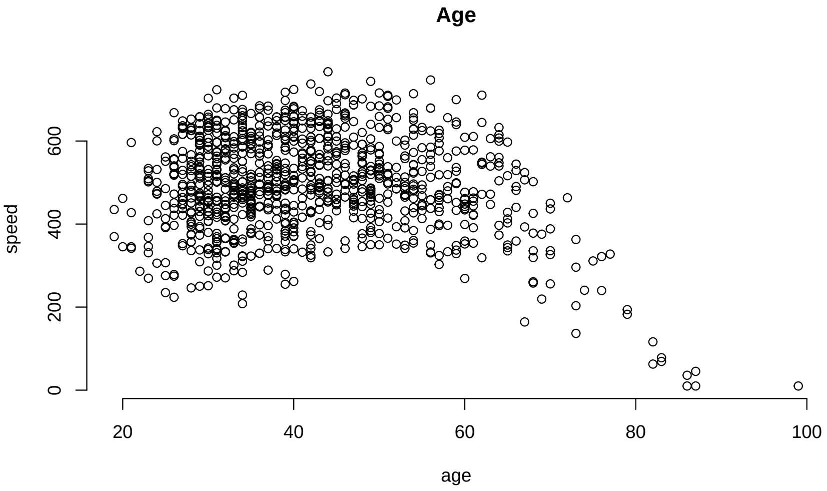
```
p + facet_grid( race + study.group ~ gender + blood.type)
```



What about age? It is a continuous variable, so it's a little more tricky.

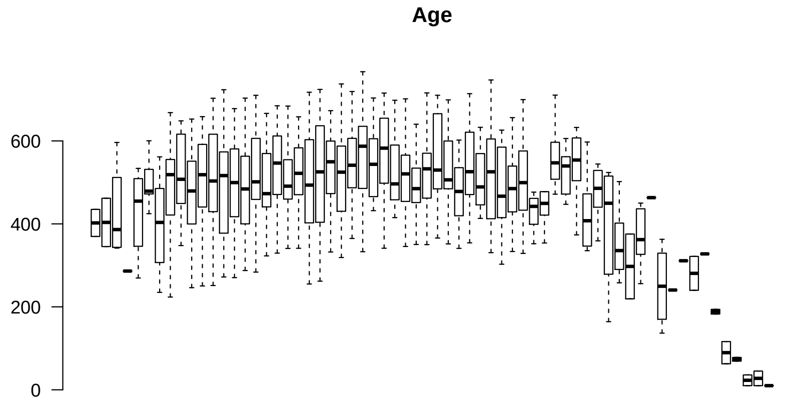
We can certainly analyze the relationship between age and speed using correlation tools.

```
plot( age, speed, bty="n", main="Age" )
```



But we can also incorporate this independent variable into a group structure. We can treat each distinct age as a separate group. The ages in this study range from 18 to 93, so we have 65 distinct ages represented.

```
plot( factor(age), speed, las=2, frame.plot=F, outline=F, main="Age", xaxt="n"  )
```



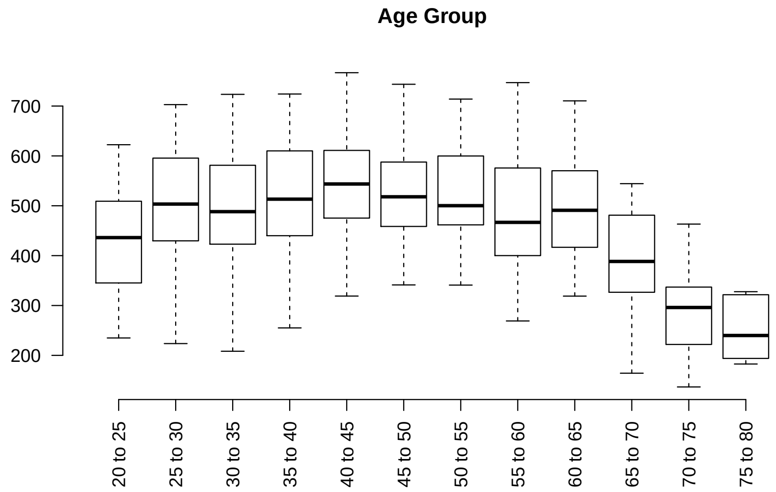
If we think about the overall group structure, then, we have unique groups defined by gender, race, blood type, and study design, and another 65 age groups. So in total we now have 24 x 65 = 1,560 groups! That is getting complicated.

This group design is problematic for two reasons. From a pragmatic standpoint, we can't report results from 1,500 groups in a table. From a more substantive perspective, we although we have 1,500 distinct cells in our grid, many may not include observations that represent the unique combination of all factors. So this group design is not very practical.

A similar problem arises if our data includes time. If our data includes the time of events recorded by hours, days of the week, months, and years, we can have generate complicated group structures if we try to analyze every unique combination.

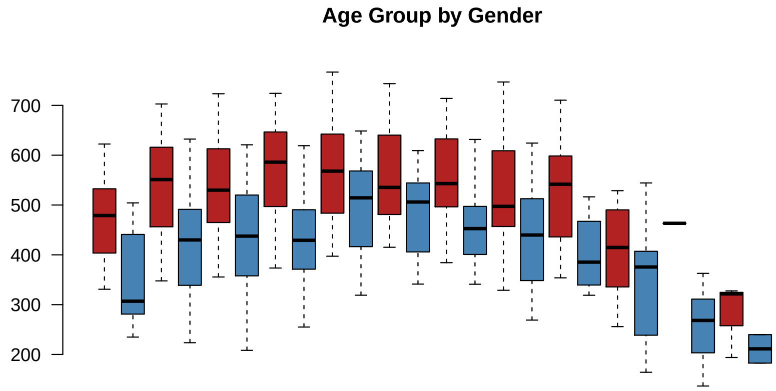
We can simplify our analysis by thinking about age ranges instead of ages, or in other words by binning our continuous data. If we split it into five-year ranges, for example, we have gone from 65 distinct ages to 12 distinct age groups.

```
age.group <- cut( age,
                  breaks=seq(from=20,to=80,by=5),
                  labels=paste( seq(from=20,to=75,by=5), "to", seq(from=25,to=80,by=5) ) )

group.structure <- formula( speed ~ age.group )

boxplot( group.structure, las=2, frame.plot=F, outline=F, main="Age Group"  )
```



We have now simplified our analysis from 1,560 to 288 possible groups. Combinations of groups will also be easier:

```
group.structure <- formula( speed ~ gender * age.group )

boxplot( group.structure,
         las=2, frame.plot=F, outline=F, main="Age Group by Gender",
         col=c("firebrick","steelblue"), xaxt="n" )
```



6.2 Analysis by Group

Let's demonstrate some analysis of groups using the Lahman package and some **dplyr** verbs. Let's do some analysis of player salaries (*Salaries* dataset), and start with a simple group structure - teams in the National League and time.

1. Which team has the highest average player salary?
2. Which team has the most players paid over $5 million a season?
3. Which team has raised it's pay the most over the past decade?

Let's start by thinking about group structure. We have teams, and we have seasons. Teams is stored as a factor, and seasons as a numeric value, so we can consider group for each by counting levels and unique values:

```
nlevels( Salaries$teamID )
#> [1] 46
length( unique( Salaries$yearID ) )
#> [1] 32
```

So we can potentially calculate 32 x 46 = 1,472 average player salaries.

6.2.1 Highest Ave Player Salary

For our first question, we will select only teams from the National League. Let's use the most recent year of data to calculate average pay.

```
Salaries %>% filter( lgID == "NL", yearID == 2016 ) %>%
             group_by( teamID) %>%
             summarize( Ave_Salary = mean(salary) )
#> # A tibble: 15 x 2
#>   teamID Ave_Salary
#>   <fct>       <dbl>
#> 1 ARI      3363041.
#> 2 ATL      2362010.
#> 3 CHC      5312678.
#> 4 CIN      3066899.
#> 5 COL      3413487
#> 6 LAD      6322525.
#> # ... with 9 more rows
```

Since the salaries are large, they are a little hard to read. Let's clean up the table a bit.

```

Salaries  %>%
             filter( lgID == "NL", yearID == 2016 ) %>%
             group_by( teamID ) %>%
             summarize( Ave_Salary=dollar( mean(salary,na.rm=T) ) ) %>%
             arrange( desc(Ave_Salary) ) %>%
             pander()
```

teamID	Ave_Salary
SFG	$6,890,151
LAD	$6,322,525
WSN	$5,448,179
CHC	$5,312,678

teamID	Ave_Salary
NYM	$4,958,857
STL	$4,614,629
SDP	$3,756,475
PIT	$3,706,387
COL	$3,413,487
ARI	$3,363,041
CIN	$3,066,899
MIA	$2,761,222
ATL	$2,362,010
MIL	$2,292,508
PHI	$2,033,793

6.2.2 Most Players Paid Over $5 Million

This question requires you to utilize a logical statement in order to translate from the question to code. We need to inspect each salary, determine whether it is over the $5m threshold, then count all of the cases. The operation will look something like this:

```
sum( Salaries$salary > 5000000 )
#> [1] 3175
```

It gets a little trickier when we want to do the operation simultaneously across groups. Our team group structure is already defined, so let's define our logical vector and count cases that match:

```
dat.NL <- filter( Salaries, yearID == 2010 & lgID == "NL" ) %>% droplevels()

gt.5m <- dat.NL$salary > 5000000

table( dat.NL$teamID, gt.5m )
#>      gt.5m
#>       FALSE TRUE
#>   ARI    23    3
#>   ATL    21    6
#>   CHN    19    8
#>   CIN    21    5
#>   COL    23    6
#>   FLO    23    4
#>   HOU    24    4
#>   LAN    20    7
#>   MIL    25    4
#>   NYN    19    9
#>   PHI    18   10
#>   PIT    27    0
#>   SDN    25    1
#>   SFN    21    7
#>   SLN    19    6
#>   WAS    26    4
```

This solution works, but the table provides too much information. We can use dply to simplify and format the table nicely for our report:

```
Salaries %>%
   filter( yearID == 2010 & lgID == "NL" ) %>%
   group_by( teamID ) %>%
   summarise( gt_five_million = sum( salary > 5000000 ) ) %>%
   arrange( desc(gt_five_million) ) %>%
   pander
```

teamID	gt_five_million
PHI	10
NYN	9
CHN	8
LAN	7
SFN	7
ATL	6
COL	6
SLN	6
CIN	5
FLO	4
HOU	4
MIL	4
WAS	4
ARI	3
SDN	1
PIT	0

6.2.3 Fielding Positions

Which fielding position is the highest paid?

```
merge( Salaries, Fielding ) %>%
  filter( yearID == 2016 ) %>%
  group_by( POS ) %>%
  summarize( Mean_Salary = dollar( mean(salary) ) ) %>%
  pander
```

POS	Mean_Salary
1B	$5,570,032
2B	$3,162,075
3B	$3,579,088
C	$2,521,903
OF	$3,546,115
P	$3,401,676
SS	$2,510,833

6.2.4 Country of Birth

Which country has produced the highest paid baseball players?

```
merge( Salaries, Master ) %>%
  filter( yearID == 2016 ) %>%
  group_by( birthCountry ) %>%
  summarize( Mean_Salary = dollar( mean(salary) ) ) %>%
  pander
```

birthCountry	Mean_Salary
Aruba	$650,000
Australia	$523,400
Brazil	$1,548,792
CAN	$7,854,167
Colombia	$3,125,289
Cuba	$5,532,484
Curacao	$5,724,167
D.R.	$5,102,318
Germany	$511,500
Japan	$8,247,012
Mexico	$4,617,038
Netherlands	$2,425,000
Nicaragua	$2,375,000
P.R.	$3,241,378
Panama	$2,946,550
Saudi Arabia	$522,500
South Korea	$5,326,190
Taiwan	$6,750,000
USA	$4,189,640
V.I.	$507,500
Venezuela	$4,521,051

6.2.5 Pay Raises

To examine pay raises, we will now use more than one year of data. Since the question asks about pay raises over the past decade, we will filter the last ten years of data.

And how since we are looking at patterns over teams and over time, we need to define a group structure with two variables:

```
Salaries %>% filter( yearID > 2006 & lgID == "NL" ) %>%
             group_by( teamID, yearID ) %>%
             summarize( mean= dollar(mean(salary)) ) %>%
             head( 20 ) %>% pander
```

teamID	yearID	mean
ARI	2007	$1,859,555
ARI	2008	$2,364,383
ARI	2009	$2,812,141
ARI	2010	$2,335,314
ARI	2011	$1,986,660
ARI	2012	$2,733,512
ARI	2013	$3,004,400
ARI	2014	$3,763,904

teamID	yearID	mean
ARI	2015	$2,034,250
ARI	2016	$3,363,041
ATL	2007	$3,117,530
ATL	2008	$3,412,189
ATL	2009	$3,335,385
ATL	2010	$3,126,802
ATL	2011	$3,346,257
ATL	2012	$2,856,205
ATL	2013	$3,254,501
ATL	2014	$4,067,042
ATL	2015	$2,990,885
ATL	2016	$2,362,010

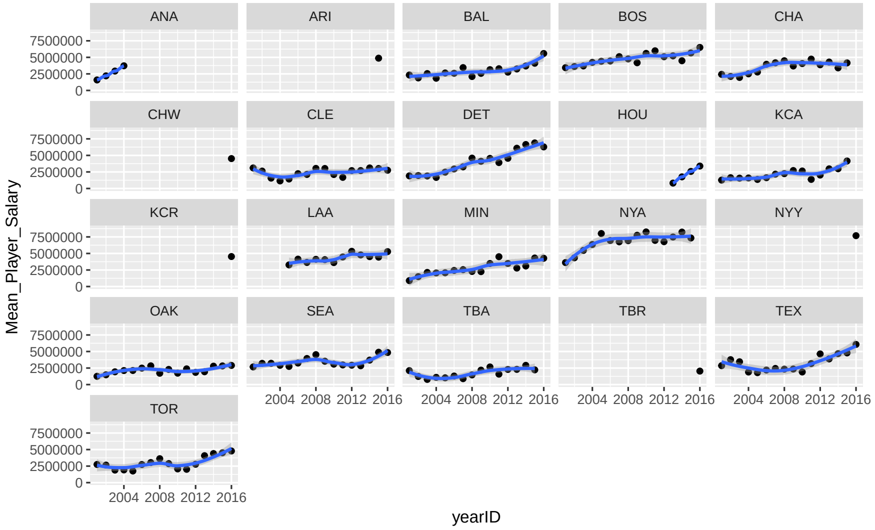
This might seem like an odd format. We might expect something that looks more like our grid structure:

```
dat.NL <- filter( Salaries, yearID > 2010 & lgID == "NL" ) %>% droplevels()

tapply( dat.NL$salary, INDEX=list(dat.NL$teamID, dat.NL$yearID), FUN=mean, na.rm=T ) %>% pander
```

	2011	2012	2013	2014	2015	2016
ARI	1986660	2733512	3004400	3763904	2034250	3363041
ATL	3346257	2856205	3254501	4067042	2990885	2362010
CHC	NA	NA	NA	NA	NA	5312678
CHN	5001893	3392194	3867989	2426759	4138547	NA
CIN	2531571	2935843	4256178	3864911	4187862	3066899
CLE	NA	NA	NA	4500000	NA	NA
COL	3390310	2692054	2976363	3180117	3827544	3413487
FLO	2190154	NA	NA	NA	NA	NA
HOU	2437724	2332731	NA	NA	NA	NA
LAD	NA	NA	NA	NA	NA	6322525
LAN	3472967	3171453	6980069	6781706	7441103	NA
MIA	NA	4373259	1400079	1549515	2835688	2761222
MIL	2849911	3755921	3077881	3748778	3477586	2292508
NYM	NA	NA	NA	NA	NA	4958857
NYN	4401752	3457555	1648278	3168777	3870667	NA
PHI	5765879	5817965	6533200	5654530	4295885	2033793
PIT	1553345	2248286	2752214	2756357	3065259	3706387
SDN	1479650	1973025	2342339	2703061	4555435	NA
SDP	NA	NA	NA	NA	NA	3756475
SFG	NA	NA	NA	NA	NA	6890151
SFN	4377716	3920689	5006441	5839649	6100056	NA
SLN	3904947	3939317	3295004	4310464	4586212	NA
STL	NA	NA	NA	NA	NA	4614629
WAS	2201963	2695171	4548131	4399456	5365085	NA
WSN	NA	NA	NA	NA	NA	5448179

Later on we will look at the benefits of "tidy data", but the basic idea is that you can "facet" your analysis easily when your groups are represented as factors instead of arranged as a table. For example, here is a time series graph that is faceted by teams:

```
Salaries %>% filter( yearID > 2000 & lgID == "AL" ) %>%
             group_by( teamID, yearID ) %>%
             summarize( Mean_Player_Salary=mean(salary) ) -> t1

qplot( data=t1, x=yearID, y=Mean_Player_Salary, geom=c("point", "smooth")  ) + facet_wrap( ~ teamID, nc
```



Now you can quickly see that Detroit is the team that has raised salaries most aggressively.

If we need to, we can easily convert a tidy dataset into something that looks like a table using the **spread()** function:

```
Salaries %>% filter( yearID > 2006 & lgID == "NL" ) %>%
             group_by( teamID, yearID ) %>%
             summarize( mean = dollar(mean(salary)) ) %>%
             spread( key=yearID, value=mean, sep="_" ) %>%
             select( 1:6 ) %>% na.omit() %>%
             pander
```

teamID	yearID_2007	yearID_2008	yearID_2009	yearID_2010	yearID_2011
ARI	$1,859,555	$2,364,383	$2,812,141	$2,335,314	$1,986,660
ATL	$3,117,530	$3,412,189	$3,335,385	$3,126,802	$3,346,257
CHN	$3,691,494	$4,383,179	$5,392,360	$5,429,963	$5,001,893
CIN	$2,210,483	$2,647,061	$3,198,196	$2,760,059	$2,531,571
COL	$2,078,500	$2,640,596	$2,785,222	$2,904,379	$3,390,310
FLO	$984,097	$660,955	$1,315,500	$2,112,212	$2,190,154
HOU	$3,250,333	$3,293,719	$3,814,682	$3,298,411	$2,437,724
LAN	$3,739,811	$4,089,260	$4,016,584	$3,531,778	$3,472,967
MIL	$2,629,130	$2,790,948	$3,083,942	$2,796,837	$2,849,911
NYN	$3,841,055	$4,593,113	$5,334,785	$4,800,819	$4,401,752
PHI	$2,980,940	$3,495,710	$4,185,335	$5,068,871	$5,765,879
PIT	$1,427,327	$1,872,684	$1,872,808	$1,294,185	$1,553,345
SDN	$2,235,022	$2,376,697	$1,604,952	$1,453,819	$1,479,650
SFN	$3,469,964	$2,641,190	$2,965,230	$3,522,905	$4,377,716
SLN	$3,224,529	$3,018,923	$3,278,830	$3,741,630	$3,904,947
WAS	$1,319,554	$1,895,207	$2,140,286	$2,046,667	$2,201,963

```
Salaries %>% filter( yearID > 2006 & lgID == "NL" ) %>%
             group_by( teamID, yearID ) %>%
             summarize( mean = dollar(mean(salary)) ) %>%
             spread( key=teamID, value=mean, sep="_" ) %>%
             select( 1:6 ) %>%
             pander
```

yearID	teamID_ARI	teamID_ATL	teamID_CHC	teamID_CHN	teamID_CIN
2007	$1,859,555	$3,117,530	NA	$3,691,494	$2,210,483
2008	$2,364,383	$3,412,189	NA	$4,383,179	$2,647,061
2009	$2,812,141	$3,335,385	NA	$5,392,360	$3,198,196
2010	$2,335,314	$3,126,802	NA	$5,429,963	$2,760,059
2011	$1,986,660	$3,346,257	NA	$5,001,893	$2,531,571
2012	$2,733,512	$2,856,205	NA	$3,392,194	$2,935,843
2013	$3,004,400	$3,254,501	NA	$3,867,989	$4,256,178
2014	$3,763,904	$4,067,042	NA	$2,426,759	$3,864,911
2015	$2,034,250	$2,990,885	NA	$4,138,547	$4,187,862
2016	$3,363,041	$2,362,010	$5,312,678	NA	$3,066,899