# Introduction to R (ITR)

*Sean Davis*

*6/29/2017*

## Contents

**1 Why R? 2**
- 1.1 What is R? . . . . . 2
- 1.2 Why use R? . . . . . 2
- 1.3 Why not use R? . . . . . 2
- 1.4 R License and the Open Source Ideal . . . . . 3

**2 R Mechanics 3**
- 2.1 Installing R . . . . . 3
- 2.2 Starting R . . . . . 5

**3 First steps 5**
- 3.1 Interacting with R . . . . . 5
- 3.2 Rules for Names in R . . . . . 5
- 3.3 Resources for Getting Help . . . . . 7

**4 Introduction to R data structures 7**
- 4.1 Vectors . . . . . 7
- 4.2 Rectangular Data . . . . . 14
- 4.3 Lists and Objects . . . . . 18

**5 Basic Textual Input and Output 19**

**6 Plotting and Graphics 20**
- 6.1 Basic Plot Functions . . . . . 20

**7 Control Structures, Looping, and Applying 24**
- 7.1 Control Structures and Looping . . . . . 24
- 7.2 Applying . . . . . 25

**8 Functions 26**

**9 *RStudio*: A Quick Tour 27**

**10 *R*: First Impressions 27**
- 10.1 *R* Data types: vector and list . . . . . 28
- 10.2 Classes: data.frame and beyond . . . . . 30
- 10.3 Help! . . . . . 35

**11 Exercise 1: BRFSS Survey Data 35**

**12 Exercise 2: ALL Phenotypic Data 39**

**13 Exploration and simple univariate measures 45**
- 13.1 Clean data . . . . . 45
- 13.2 Weight in 1990 vs. 2010 Females . . . . . 45
- 13.3 Weight and height in 2010 Males . . . . . 46

**14 Multivariate analysis** **50**
14.1 Input and setup . . . . . . . . . . 50
14.2 Cleaning . . . . . . . . . . 51
14.3 Unsupervised machine learning – multi-dimensional scaling . . . . . . . . . . 52

**15 Using *R* in real life** **52**
15.1 Organizing work . . . . . . . . . . 52
15.2 *R* Packages . . . . . . . . . . 53

**16 Graphics and Visualization** **53**
16.1 Base *R* Graphics . . . . . . . . . . 53
16.2 What makes for a good graphical display? . . . . . . . . . . 55
16.3 Grammar of Graphics: ggplot2 . . . . . . . . . . 55

**A Appendix A – Swirl** **59**

# 1 Why R?

## 1.1 What is R?

R is a number of things, simultaneously. Depending on who is being asked, R is:

- A software package
- A programming language
- A toolkit for developing statistical and analytical tools
- An extensive library of statistical and mathematical software and algorithms
- A scripting language
- much, much more

## 1.2 Why use R?

- R is cross-platform and runs on Windows, Mac, and Linux (as well as more obscure systems).
- R provides a vast number of useful statistical tools, many of which have been painstakingly tested.
- R produces publication-quality graphics in a variety of formats.
- R plays well with FORTRAN, C, and scripts in many languages.
- R scales, making it useful for small and large projects. It is NOT Excel.
- R does not have a meaningfully useful graphical user interface (GUI).

I can develop code for analysis on my Mac laptop. I can then install the *same* code on our 20k core cluster and run it in parallel on 100 samples, monitor the process, and then update a database (for example) with R when complete.

## 1.3 Why not use R?

- R cannot do everything.
- R is not always the "best" tool for the job.
- R will *not* hold your hand. Often, it will *slap* your hand instead.
- The documentation can be opaque (but there is documentation).
- R can drive you crazy (on a good day) or age you prematurely (on a bad one).
- Finding the right package to do the job you want to do can be challenging; worse, some contributed packages are unreliable.]{}
- R does not have a meaningfully useful graphical user interface (GUI).

### 1.4 R License and the Open Source Ideal

R is free (yes, totally free!) and distributed under GNU license. In particular, this license allows one to:

- Download the source code
- Modify the source code to your heart's content
- Distribute the modified source code and even charge money for it, but you must distribute the modified source code under the original GNU license]{}

This license means that R will always be available, will always be open source, and can grow organically without constraint.

## 2 R Mechanics

### 2.1 Installing R

The home page for R is called the Comprehensive R Archive Network (CRAN). The website is not pretty (see figure 1), but it has quite a bit of information on it. It is not the best place to find help on R, although it is one of the best places to get R-related software, tools, and updates.

```
knitr::include_graphics('images/CRAN-screenshot.png')
```

Detailed installation instructions are readily available, but here are abbreviated instructions for convenience.

#### 2.1.1 Windows

NOTE: See Windows installation instructions for more detail. Install *R* and *RStudio* as regular users.

To install *R*, visit the Windows base distribution page. Click on the `Download R-3.4.0 for Windows` link (or use the latest version available). Click on the installer and make the default selection for each option.

To install *RStudio*, visit the RStudio download page. Click on the current RStudio release for Windows link. Click on the installer and follow default instructions.

#### 2.1.2 Mac

NOTE: See R for Mac OS X for more detail.

To install *R*, visit the R for Mac OS X. Click on the the `R-3.4.0.pkg` link (or use the latest version available). Click on the installer and follow default instructions.

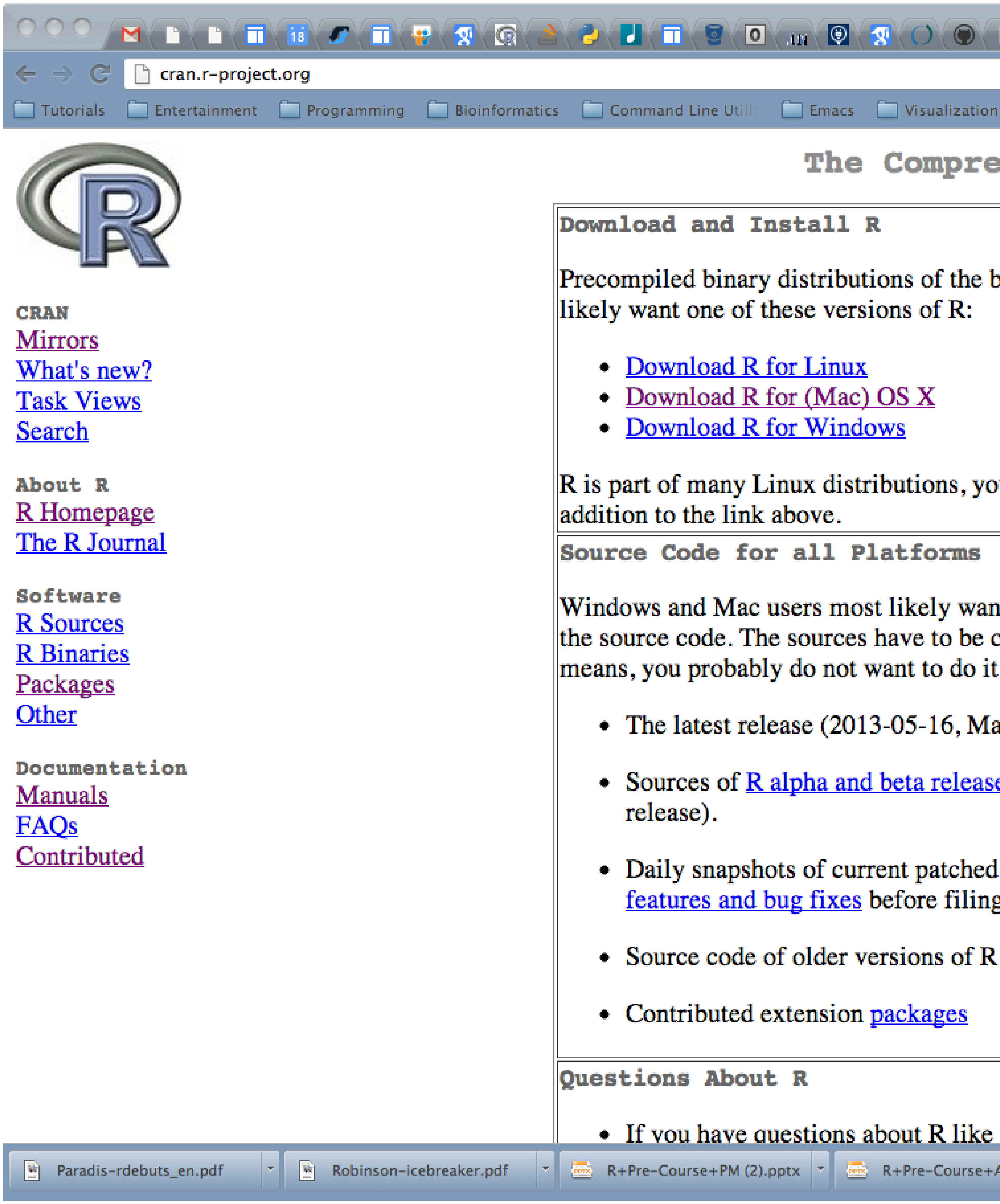
To install *RStudio*, visit the RStudio download page. Click on the current RStudio release for Windows link. Click on the installer and follow default instructions.

#### 2.1.3 Linux

NOTE: See distribution-specific instructions for additional detail.

On debian-based systems, the easiest way to install *R* is through a package manager manager, run under an administrator account. On Linux one usually needs to install *R* packages from source, and *R* package source often contains C, C++, or Fortran code requiring a compiler and `-dev` versions of various system libraries. It is therefore convenient to install the `-dev` version of R.

```
sudo apt-get install r-base r-base-dev
```



The Compre

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Download and Install R

Precompiled binary distributions of the b
likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, yo
addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely wan
the source code. The sources have to be c
means, you probably do not want to do it

- The latest release (2013-05-16, Ma
- Sources of R alpha and beta release
release).
- Daily snapshots of current patched
features and bug fixes before filing
- Source code of older versions of R
- Contributed extension packages

Questions About R

- If you have questions about R like

Figure 1: The Comprehensive R Archive Network (CRAN) website

When installing source packages, it may be necessary to have access to the `-dev` version of various system libraries. Many of these are installed as dependencies of `r-base-dev`; other common examples include the xml and curl libraries

```
sudo apt-get install libxml2-dev
sudo apt-get install libcurl-dev
```

Note in particular the use specification of libraries (the `lib` prefix) and the use of the `-dev` version.

To install *RStudio*, visit the RStudio download page. Download the appropriate archive for your OS. On Ubuntu, install the `.deb` installer with

```
sudo dpkg -i rstudio-1.0.136-amd64.deb
```

## 2.2 Starting R

How to start R depends a bit on the operating system (Mac, Windows, Linux) and interface. In this course, we will largely be using an Integrated Development Environment (IDE) called *RStudio*, but there is nothing to prohibit using R at the command line or in some other interface (and there are a few). A screenshot of the interface is shown in figure 2.

## 3 First steps

### 3.1 Interacting with R

The only meaningful way of interacting with R is by typing into the R console. At the most basic level, anything that we type at the command line will fall into one of two categories:

1. Assignments

   ```r
   x = 1
   y <- 2
   ```

2. Expressions

   ```r
   1 + pi + sin(42)
   ```

   ```
   ## [1] 3.225071
   ```

The assignment type is obvious because either the The "<-" or "=" are used. Note that when we type expressions, R will return a result. In this case, the result of R evaluating `1 + pi + sin(42)` is `3.2250711`.

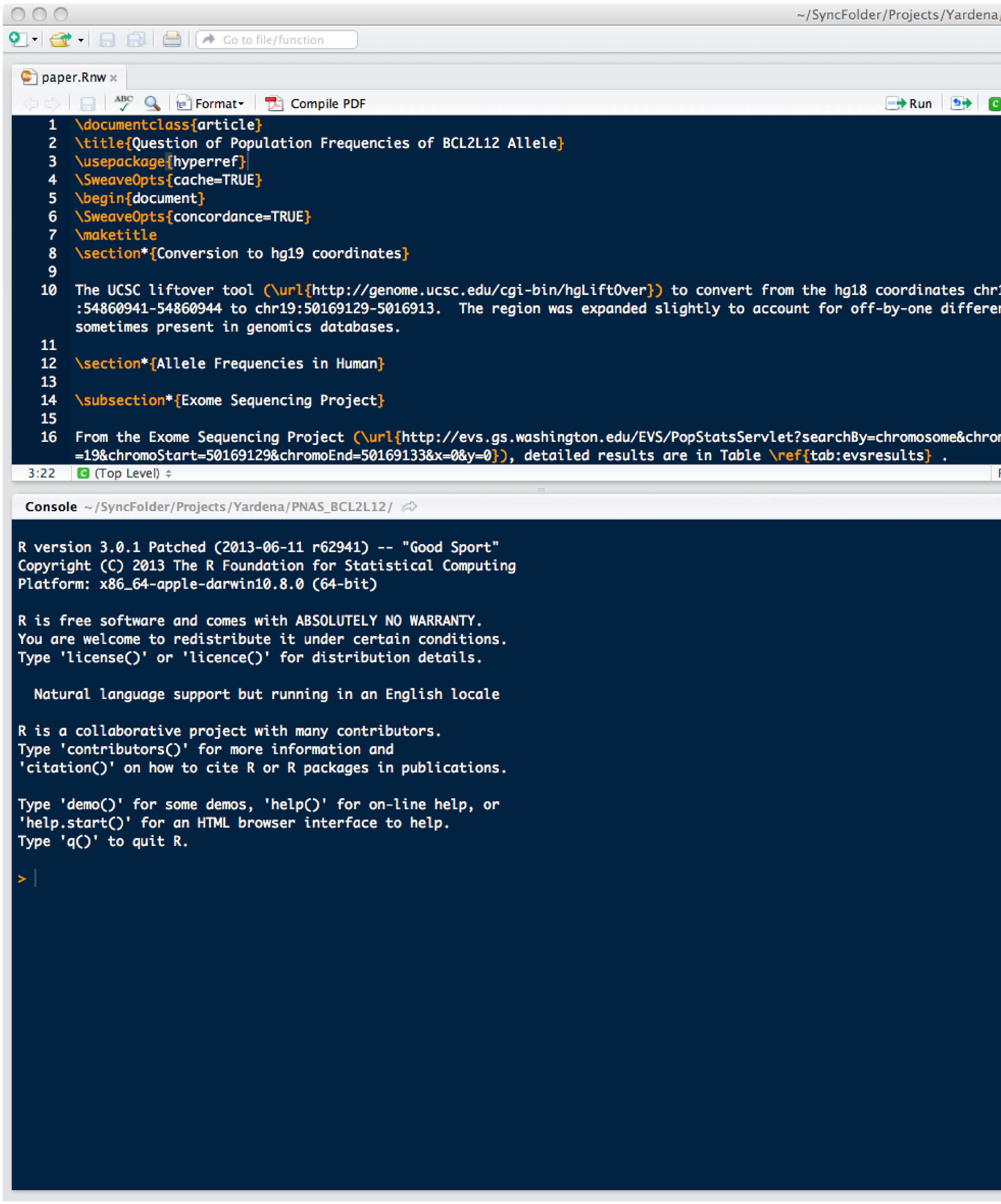
The standard R prompt is a ">" sign. When present, R is waiting for the next expression or assignment. If a line is not a complete R command, R will continue the next line with a "+". For example, typing the fillowing with a "Return" after the second "+" will result in R giving back a "+" on the next line, a prompt to keep typing.

```r
1 + pi +
sin(3.7)
```

```
## [1] 3.611757
```

### 3.2 Rules for Names in R

R allows users to assign names to objects such as variables, functions, and even dimensions of data. However, these names must follow a few rules.


~/SyncFolder/Projects/Yardena/
Go to file/function
paper.Rnw
Format
Compile PDF
Run
3:22
(Top Level)
Console ~/SyncFolder/Projects/Yardena/PNAS_BCL2L12/

```
\documentclass{article}
\title{Question of Population Frequencies of BCL2L12 Allele}
\usepackage{hyperref}
\SweaveOpts{cache=TRUE}
\begin{document}
\SweaveOpts{concordance=TRUE}
\maketitle
\section*{Conversion to hg19 coordinates}

The UCSC liftover tool (\url{http://genome.ucsc.edu/cgi-bin/hgLiftOver}) to convert from the hg18 coordinates chr1
:54860941-54860944 to chr19:50169129-5016913.  The region was expanded slightly to account for off-by-one differen
sometimes present in genomics databases.

\section*{Allele Frequencies in Human}

\subsection*{Exome Sequencing Project}

From the Exome Sequencing Project (\url{http://evs.gs.washington.edu/EVS/PopStatsServlet?searchBy=chromosome&chrom
=19&chromoStart=50169129&chromoEnd=50169133&x=0&y=0}), detailed results are in Table \ref{tab:evsresults} .
```

```
R version 3.0.1 Patched (2013-06-11 r62941) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 2: The Rstudio interface

- Names may contain any combination of letters, numbers, underscore, and "."
- Names may not start with numbers, underscore.
- R names are case-sensitive.

Examples of valid R names include:

```
pi
x
camelCaps
my_stuff
MY_Stuff
this.is.the.name.of.the.man
ABC123
abc1234asdf
.hi
```

## 3.3 Resources for Getting Help

There is extensive built-in help and documentation within R.

If the name of the function or object on which help is sought is known, the following approaches with the name of the function or object will be helpful. For a concrete example, examine the help for the `print` method.

```
help(print)
help('print')
?print
```

If the name of the function or object on which help is sought is *not* known, the following from within R will be helpful.

```
help.search('microarray')
RSiteSearch('microarray')
```

There are also tons of online resources that Google will include in searches if online searching feels more appropriate.

I strongly recommend using `help(newfunction)` for all functions that are new or unfamiliar to you.

# 4 Introduction to R data structures

As in many programming languages, understanding how data are stored and manipulated is important to getting the most out of the experience. In these next few sections, we will introduce some basic R data types and structures as well as some general approaches for working with them.

## 4.1 Vectors

In R, even a single value is a vector with length=1.

```
z = 1
z
```

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

```
length(z)
```

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

In the code above, we "assigned" the value 1 to the variable named `z`. Typing `z` by itself is an "expression" that returns a result which is, in this case, the value that we just assigned. The `length` method takes an R object and returns the R length. There are numerous ways of asking R about what an object represents, and `length` is one of them.

Vectors can contain numbers, strings (character data), or logical values (`TRUE` and `FALSE`) or other "atomic" data types (table 1). *Vectors cannot contain a mix of types!* We will introduce another data structure, the R `list` for situations when we need to store a mix of base R data types.

Table 1: Atomic (simplest) data types in R.

| Data type | Stores |
|---|---|
| numeric | floating point numbers |
| integer | integers |
| complex | complex numbers |
| factor | categorical data |
| character | strings |
| logical | TRUE or FALSE |
| NA | missing |
| NULL | empty |
| function | function type |

### 4.1.1 Creating vectors

Character vectors (also sometimes called "string" vectors) are entered with each value surrounded by single or double quotes; either is acceptable, but they must match. They are always displayed by R with double quotes. Here are some examples of creating vectors:

```r
# examples of vectors
c('hello','world')
```

```
## [1] "hello" "world"
```

```r
c(1,3,4,5,1,2)
```

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

```r
c(1.12341e7,78234.126)
```

```
## [1] 11234100.00    78234.13
```

```r
c(TRUE,FALSE,TRUE,TRUE)
```

```
## [1]  TRUE FALSE  TRUE  TRUE
```

```r
# note how in the next case the TRUE is converted to "TRUE"
# with quotes around it.
c(TRUE,'hello')
```

```
## [1] "TRUE"  "hello"
```

We can also create vectors as "regular sequences" of numbers. For example:

```r
# create a vector of integers from 1 to 10
x = 1:10
```

```
# and backwards
x = 10:1
```

The `seq` function can create more flexible regular sequences.

```
# create a vector of numbers from 1 to 4 skipping by 0.3
y = seq(1,4,0.3)
```

And creating a new vector by concatenating existing vectors is possible, as well.

```
# create a sequence by concatenating two other sequences
z = c(y,x)
z
```

```
##  [1]  1.0  1.3  1.6  1.9  2.2  2.5  2.8  3.1  3.4  3.7  4.0 10.0  9.0  8.0
## [15]  7.0  6.0  5.0  4.0  3.0  2.0  1.0
```

### 4.1.2 Vector Operations

Operations on a single vector are typically done element-by-element. For example, we can add 2 to a vector, 2 is added to each element of the vector and a new vector of the same length is returned.

```
x = 1:10
x + 2
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12
```

If the operation involves two vectors, the following rules apply. If the vectors are the same length: R simply applies the operation to each pair of elements.

```
x + x
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

If the vectors are different lengths, but one length a multiple of the other, R reuses the shorter vector as needed.

```
x = 1:10
y = c(1,2)
x * y
```

```
##  [1]  1  4  3  8  5 12  7 16  9 20
```

If the vectors are different lengths, but one length *not* a multiple of the other, R reuses the shorter vector as needed *and* delivers a warning.

```
x = 1:10
y = c(2,3,4)
x * y
```

```
## Warning in x * y: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  2  6 12  8 15 24 14 24 36 20
```

Typical operations include multiplication ("*"), addition, subtraction, division, exponentiation ("^"), but many operations in R operate on vectors and are then called "vectorized".

### 4.1.3 Logical Vectors

Logical vectors are vectors composed on only the values `TRUE` and `FALSE`. Note the all-upper-case and no quotation marks.

```
a = c(TRUE,FALSE,TRUE)

# we can also create a logical vector from a numeric vector
# 0 = false, everything else is 1
b = c(1,0,217)
d = as.logical(b)
d
```

```
## [1]  TRUE FALSE  TRUE
```

```
# test if a and d are the same at every element
all.equal(a,d)
```

```
## [1] TRUE
```

```
# We can also convert from logical to numeric
as.numeric(a)
```

```
## [1] 1 0 1
```

### 4.1.4 Logical Operators

Some operators like `<, >, ==, >=, <=, !=` can be used to create logical vectors.

```
# create a numeric vector
x = 1:10
# testing whether x > 5 creates a logical vector
x > 5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
x <= 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
x != 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
x == 5
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

We can also assign the results to a variable:

```
y = (x == 5)
y
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

### 4.1.5 Indexing Vectors

In R, an index is used to refer to a specific element or set of elements in an vector (or other data structure). [R uses `[` and `]` to perform indexing, although other approaches to getting subsets of larger data structures are common in R.

```
x = seq(0,1,0.1)
# create a new vector from the 4th element of x
x[4]
```

```
## [1] 0.3
```

We can even use other vectors to perform the "indexing".

```
x[c(3,5,6)]
```

```
## [1] 0.2 0.4 0.5
```

```
y = 3:6
x[y]
```

```
## [1] 0.2 0.3 0.4 0.5
```

Combining the concept of indexing with the concept of logical vectors results in a very power combination.

```
# use help('rnorm') to figure out what is happening next
myvec = rnorm(10)

# create logical vector that is TRUE where myvec is >0.25
gt1 = (myvec > 0.25)
sum(gt1)
```

```
## [1] 6
```

```
# and use our logical vector to create a vector of myvec values that are >0.25
myvec[gt1]
```

```
## [1] 0.3574935 0.5253169 0.4016574 0.7473344 1.4731230 0.5224433
```

```
# or <=0.25 using the logical "not" operator, "!"
myvec[!gt1]
```

```
## [1] -1.97005780  0.01587731 -1.67934480 -0.03437553
```

```
# shorter, one line approach
myvec[myvec > 0.25]
```

```
## [1] 0.3574935 0.5253169 0.4016574 0.7473344 1.4731230 0.5224433
```

### 4.1.6 Character Vectors, A.K.A. Strings

R uses the `paste` function to concatenate strings.

```
paste("abc","def")
```

```
## [1] "abc def"
```

```
paste("abc","def",sep="THISSEP")
```

```
## [1] "abcTHISSEPdef"
```

```
paste0("abc","def")
```

```
## [1] "abcdef"
```

```
## [1] "abcdef"
paste(c("X","Y"),1:10)
```

```
##  [1] "X 1"  "Y 2"  "X 3"  "Y 4"  "X 5"  "Y 6"  "X 7"  "Y 8"  "X 9"  "Y 10"
```

```r
paste(c("X","Y"),1:10,sep="_")
```

```
##  [1] "X_1"  "Y_2"  "X_3"  "Y_4"  "X_5"  "Y_6"  "X_7"  "Y_8"  "X_9"  "Y_10"
```

We can count the number of characters in a string.

```r
nchar('abc')
```

```
## [1] 3
```

```r
nchar(c('abc','d',123456))
```

```
## [1] 3 1 6
```

Pulling out parts of strings is also sometimes useful.

```r
substr('This is a good sentence.',start=10,stop=15)
```

```
## [1] " good "
```

Another common operation is to replace something in a string with something (a find-and-replace).

```r
sub('This','That','This is a good sentence.')
```

```
## [1] "That is a good sentence."
```

When we want to find all strings that match some other string, we can use `grep`, or "grab regular expression".

```r
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'))
```

```
## [1] 1 2 3
```

```r
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'),value=TRUE)
```

```
## [1] "abcdef" "abcd"   "bcde"
```

### 4.1.7 Missing Values, AKA "NA"

R has a special value, "NA", that represents a "missing" value, or *Not Available*, in a vector or other data structure. Here, we just create a vector to experiment.

```r
x = 1:5
x
```

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

```r
length(x)
```

```
## [1] 5
```

```r
is.na(x)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```r
x[2] = NA
x
```

```
## [1]  1 NA  3  4  5
```

The length of `x` is unchanged, but there is one value that is marked as "missing" by virtue of being `NA`.

```r
length(x)
```

```
## [1] 5
```

```
is.na(x)
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE
```

We can remove `NA` values by using indexing. In the following, `is.na(x)` returns a logical vector the length of `x`. The `!` is the logical *NOT* operator and converts `TRUE` to `FALSE` and vice-versa.

```
x[!is.na(x)]
```

```
## [1] 1 3 4 5
```

### 4.1.8 Factors

A factor is a special type of vector, normally used to hold a categorical variable–such as smoker/nonsmoker, state of residency, zipcode–in many statistical functions. Such vectors have class "factor". Factors are primarily used in Analysis of Variance (ANOVA) or other situations when "categories" are needed. When a factor is used as a predictor variable, the corresponding indicator variables are created (more later).

Note of caution that factors in R often *appear* to be character vectors when printed, but you will notice that they do not have double quotes around them. They are stored in R as numbers with a key name, so sometimes you will note that the factor *behaves* like a numeric vector.

```
# create the character vector
citizen<-c("uk","us","no","au","uk","us","us","no","au")

# convert to factor
citizenf<-factor(citizen)
citizen
```

```
## [1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
```

```
citizenf
```

```
## [1] uk us no au uk us us no au
## Levels: au no uk us
```

```
# convert factor back to character vector
as.character(citizenf)
```

```
## [1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
```

```
# convert to numeric vector
as.numeric(citizenf)
```

```
## [1] 3 4 2 1 3 4 4 2 1
```

R stores many data structures as vectors with "attributes" and "class" (just so you have seen this).

```
attributes(citizenf)
```

```
## $levels
## [1] "au" "no" "uk" "us"
##
## $class
## [1] "factor"
```

```
class(citizenf)
```

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

```
# note that after unclassing, we can see the
# underlying numeric structure again
unclass(citizenf)
```

```
## [1] 3 4 2 1 3 4 4 2 1
## attr(,"levels")
## [1] "au" "no" "uk" "us"
```

Tabulating factors is a useful way to get a sense of the “sample” set available.

```
table(citizenf)
```

```
## citizenf
## au no uk us
##  2  2  2  3
```

## 4.2 Rectangular Data

A *matrix* is a rectangular collection of the same data type. It can be viewed as a collection of column vectors all of the same length and the same type (i.e. numeric, character or logical). A *data.frame* is *also* a rectangular array. All of the columns must be the same length, but they may be of *different* types. The rows and columns of a matrix or data frame can be given names. However these are implemented differently in R; many operations will work for one but not both, often a source of confusion.

### 4.2.1 Matrices

We start by building a matrix from parts:

```
x <- 1:10
y <- rnorm(10)

# make a matrix by column binding two numeric vectors
mat<-cbind(x,y)
mat
```

```
##        x           y
##  [1,]  1 -0.01677975
##  [2,]  2 -0.02493741
##  [3,]  3 -0.47394824
##  [4,]  4  0.52438478
##  [5,]  5 -0.29806576
##  [6,]  6 -1.17388828
##  [7,]  7  2.82727812
##  [8,]  8  1.08596128
##  [9,]  9  1.05204720
## [10,] 10 -1.73466401
```

Inspecting the names associated with rows and columns is often useful, particularly if the names have human meaning.

```
rownames(mat)
```

```
## NULL
```

```
colnames(mat)
```

```
## [1] "x" "y"
```

Matrices have dimensions.

```
dim(mat)
```

```
## [1] 10  2
```

```
nrow(mat)
```

```
## [1] 10
```

```
ncol(mat)
```

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

Indexing for matrices works as for vectors except that we now need to include both the row and column (in that order).

```
# The 2nd element of the 1st row of mat
mat[1,2]
```

```
##           y
## -0.01677975
```

```
# The first ROW of mat
mat[1,]
```

```
##           x           y
##  1.00000000 -0.01677975
```

```
# The first COLUMN of mat
mat[,1]
```

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

```
# and all elements of mat that are > 4; note no comma
mat[mat>4]
```

```
## [1]  5  6  7  8  9 10
```

```
## [1]  5  6  7  8  9 10
```

Note that in the last case, there is no ",", so R treats the matrix as a long vector (length=20). This is convenient, sometimes, but it can also be a source of error, as some code may "work" but be doing something unexpected.

In the next example, we create a matrix with 2 columns and 10 rows.

```
m = matrix(rnorm(20),nrow=10)
# multiply all values in the matrix by 20
m = m*20
# and add 100 to the first column of m
m[,1] = m[,1] + 100
# summarize m
summary(m)
```

```
##        V1               V2
##  Min.   : 68.38   Min.   :-39.987
##  1st Qu.: 87.81   1st Qu.: -9.612
##  Median : 93.84   Median : -5.642
##  Mean   : 96.03   Mean   : -2.956
##  3rd Qu.:109.12   3rd Qu.: 13.995
##  Max.   :113.66   Max.   : 18.251
```

### 4.2.2 Data Frames

#### 4.2.2.1 Matrices Versus Data Frames

```r
mat<-cbind(x,y)
head(mat)
```

```
##      x           y
## [1,] 1 -0.01677975
## [2,] 2 -0.02493741
## [3,] 3 -0.47394824
## [4,] 4  0.52438478
## [5,] 5 -0.29806576
## [6,] 6 -1.17388828
```

```r
class(mat[,1])
```

```
## [1] "numeric"
```

```r
z = paste0('a',1:10)
tab<-cbind(x,y,z)
class(tab)
```

```
## [1] "matrix"
```

```r
mode(tab[,1])
```

```
## [1] "character"
```

```r
head(tab,4)
```

```
##      x   y                      z
## [1,] "1" "-0.0167797457321218" "a1"
## [2,] "2" "-0.0249374114211213" "a2"
## [3,] "3" "-0.473948240464736"  "a3"
## [4,] "4" "0.524384778840853"   "a4"
```

```r
tab<-data.frame(x,y,z)
class(tab)
```

```
## [1] "data.frame"
```

```r
head(tab)
```

```
##   x           y  z
## 1 1 -0.01677975 a1
## 2 2 -0.02493741 a2
## 3 3 -0.47394824 a3
## 4 4  0.52438478 a4
## 5 5 -0.29806576 a5
## 6 6 -1.17388828 a6
```

```r
mode(tab[,1])
```

```
## [1] "numeric"
```

```r
class(tab[,3])
```

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

```r
rownames(tab)
```

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

```
rownames(tab)<-paste0("row",1:10)
rownames(tab)
```

```
##  [1] "row1"  "row2"  "row3"  "row4"  "row5"  "row6"  "row7"  "row8"
##  [9] "row9"  "row10"
```

Data frame columns can be refered to by name using the "dollar sign" operator.

```
head(tab)
```

```
##      x           y  z
## row1 1 -0.01677975 a1
## row2 2 -0.02493741 a2
## row3 3 -0.47394824 a3
## row4 4  0.52438478 a4
## row5 5 -0.29806576 a5
## row6 6 -1.17388828 a6
```

```
tab$x
```

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

```
tab$y
```

```
##  [1] -0.01677975 -0.02493741 -0.47394824  0.52438478 -0.29806576
##  [6] -1.17388828  2.82727812  1.08596128  1.05204720 -1.73466401
```

Column names can be set, which can be useful for referring to data later.

```
colnames(tab)
```

```
## [1] "x" "y" "z"
```

```
colnames(tab) = paste0('col',1:3)
```

Data frames have functions to report size and even `summary` functions. Try the following:

```
ncol(tab)
```

```
## [1] 3
```

```
nrow(tab)
```

```
## [1] 10
```

```
dim(tab)
```

```
## [1] 10  3
```

```
summary(tab)
```

```
##       col1            col2                col3
##  Min.   : 1.00   Min.   :-1.73466   a1      :1
##  1st Qu.: 3.25   1st Qu.:-0.42998   a10     :1
##  Median : 5.50   Median :-0.02086   a2      :1
##  Mean   : 5.50   Mean   : 0.17674   a3      :1
##  3rd Qu.: 7.75   3rd Qu.: 0.92013   a4      :1
##  Max.   :10.00   Max.   : 2.82728   a5      :1
##                                     (Other):4
```

Extracting parts of a `data.frame` work as for matrices. Try to think about what each of the following will do before asking R to evaluate the result.

```
tab[1:3,]
```

```
##      col1        col2 col3
## row1    1 -0.01677975   a1
## row2    2 -0.02493741   a2
## row3    3 -0.47394824   a3
```

```
tab[,2:3]
```

```
##              col2 col3
## row1  -0.01677975   a1
## row2  -0.02493741   a2
## row3  -0.47394824   a3
## row4   0.52438478   a4
## row5  -0.29806576   a5
## row6  -1.17388828   a6
## row7   2.82727812   a7
## row8   1.08596128   a8
## row9   1.05204720   a9
## row10 -1.73466401  a10
```

```
tab[,1]>7
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
tab[tab[,1]>7,]
```

```
##       col1      col2 col3
## row8     8  1.085961   a8
## row9     9  1.052047   a9
## row10   10 -1.734664  a10
```

```
tab[tab[,1]>7,3]
```

```
## [1] a8  a9  a10
## Levels: a1 a10 a2 a3 a4 a5 a6 a7 a8 a9
```

```
tab[tab[,1]>7,2:3]
```

```
##            col2 col3
## row8   1.085961   a8
## row9   1.052047   a9
## row10 -1.734664  a10
```

```
tab[tab$x>7,3]
```

```
## factor(0)
## Levels: a1 a10 a2 a3 a4 a5 a6 a7 a8 a9
```

```
tab$z[tab$x>3]
```

```
## NULL
```

## 4.3 Lists and Objects

A list is a collection of objects that may be the same or different types. The objects in a list often have names, and may be indexed either by name (e.g. `my_list$name3`) or component number (e.g. `my_list[[3]]`). As an aside, a data frame is a list of matched column vectors.

We can reate a list, noting the different data types involved.

```
a = list(1,"b",c(1,2,3))
a
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] 1 2 3
```

```
length(a)
```

```
## [1] 3
```

```
class(a)
```

```
## [1] "list"
```

```
a[[3]]
```

```
## [1] 1 2 3
```

A data frame *is* a list.

```
# test if our friend "tab" is a list
is.list(tab)
```

```
## [1] TRUE
```

```
tab[[2]]
```

```
##  [1] -0.01677975 -0.02493741 -0.47394824  0.52438478 -0.29806576
##  [6] -1.17388828  2.82727812  1.08596128  1.05204720 -1.73466401
```

```
names(tab)
```

```
## [1] "col1" "col2" "col3"
```

Table Summary of basic R data structures.

| Data type S | tores |
|---|---|
| vector | one-dimensional data, single data type |
| matrix | two-dimensional data, single data type |
| data frame | two-dimensional data, multiple data types |
| list | list of data types, not all need to be the same type |
| object | a list with attributes and potentially slots and methods |

## 5 Basic Textual Input and Output

The `write.table` function and friends write a data.frame or matrix to disk as a text file.

```
write.table(tab,file='tab.txt',sep="\t",col.names=TRUE)
# remove tab from the workspace
rm(tab)
# make sure it is gone
```

```
ls(pattern="tab")
```

```
## character(0)
```

The `read.table` function and friends read a data.frame or matrix from a text file.

```
tab = read.table('tab.txt',sep="\t",header=TRUE)
head(tab,3)
```

```
##      col1        col2 col3
## row1    1 -0.01677975   a1
## row2    2 -0.02493741   a2
## row3    3 -0.47394824   a3
```

## 6 Plotting and Graphics

### 6.1 Basic Plot Functions

- The command `plot(x,y)` will plot vector x as the independent variable and vector y as the dependent variable.
- Within the command line, you can specify the title of the graph, the name of the x-axis, and the name of the y-axis.
  - main='title'
  - xlab='name of x axis'
  - ylab='name of y axis'
- The command `lines(x,y)` adds a line segment to the plot.
- The command `points(x,y)` adds points to the plot.
- A legend can be created using `legend`.

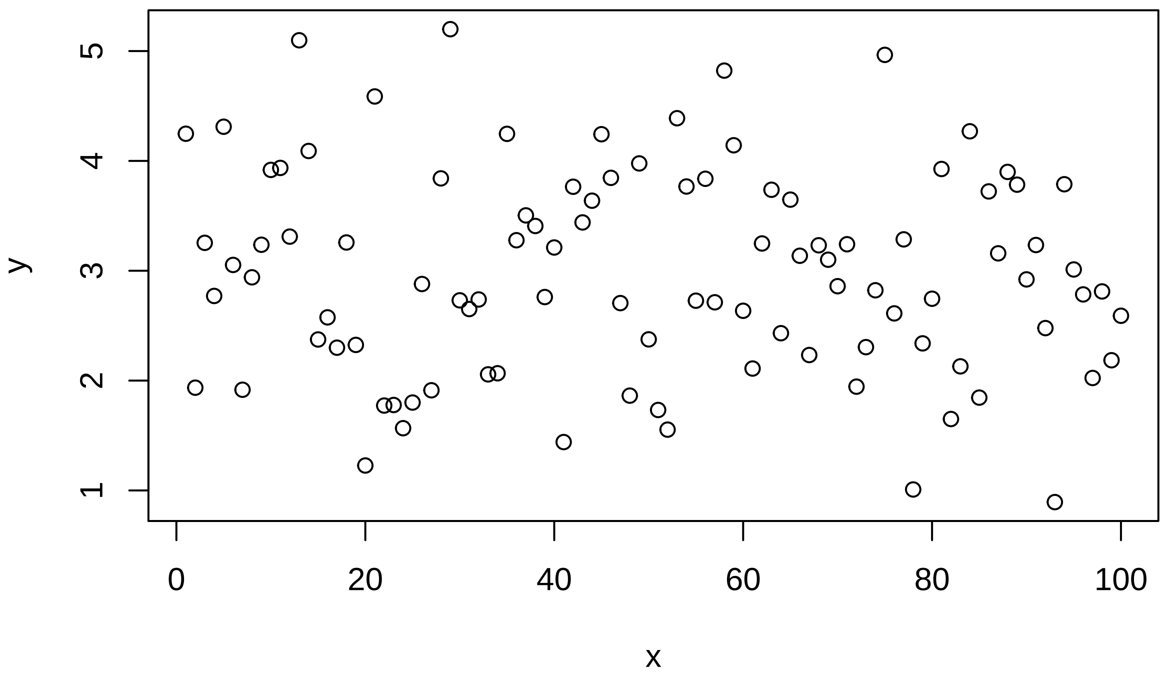
demo

```
demo(graphics)
```

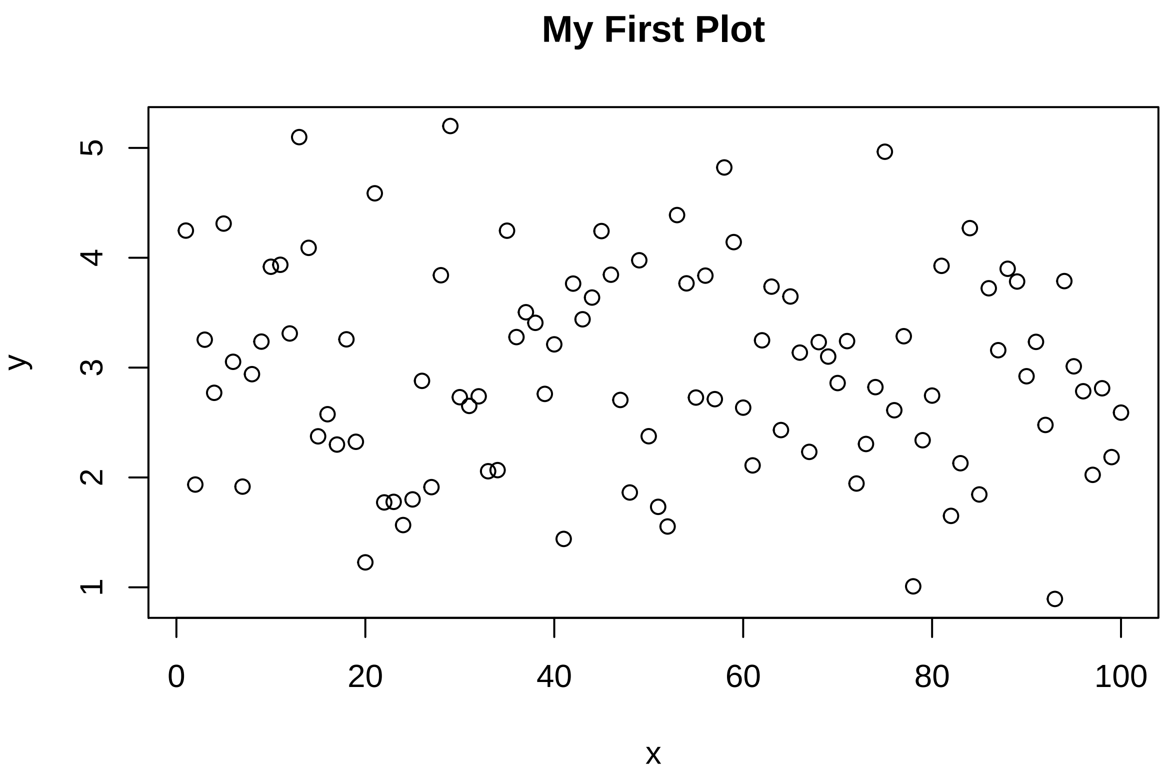
#### 6.1.1 Simple Plotting Example

Try this yourself:

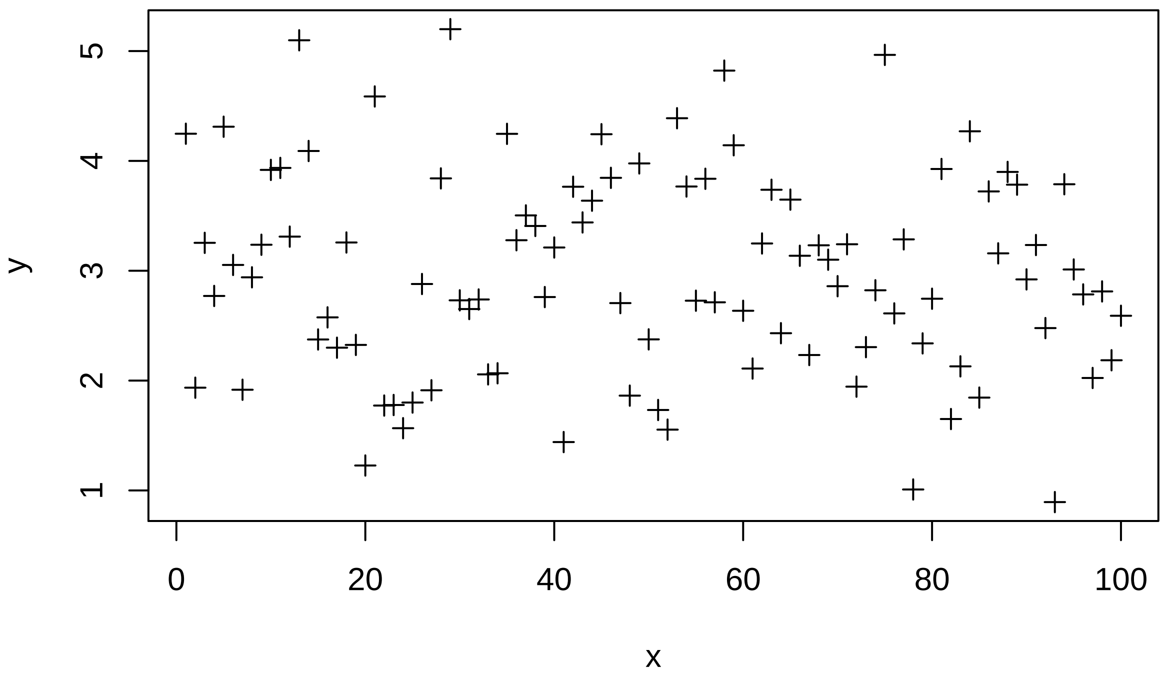
```
    x = 1:100
    y = rnorm(100,3,1) # 100 random normal deviates with mean=3, sd=1
    plot(x,y)
```



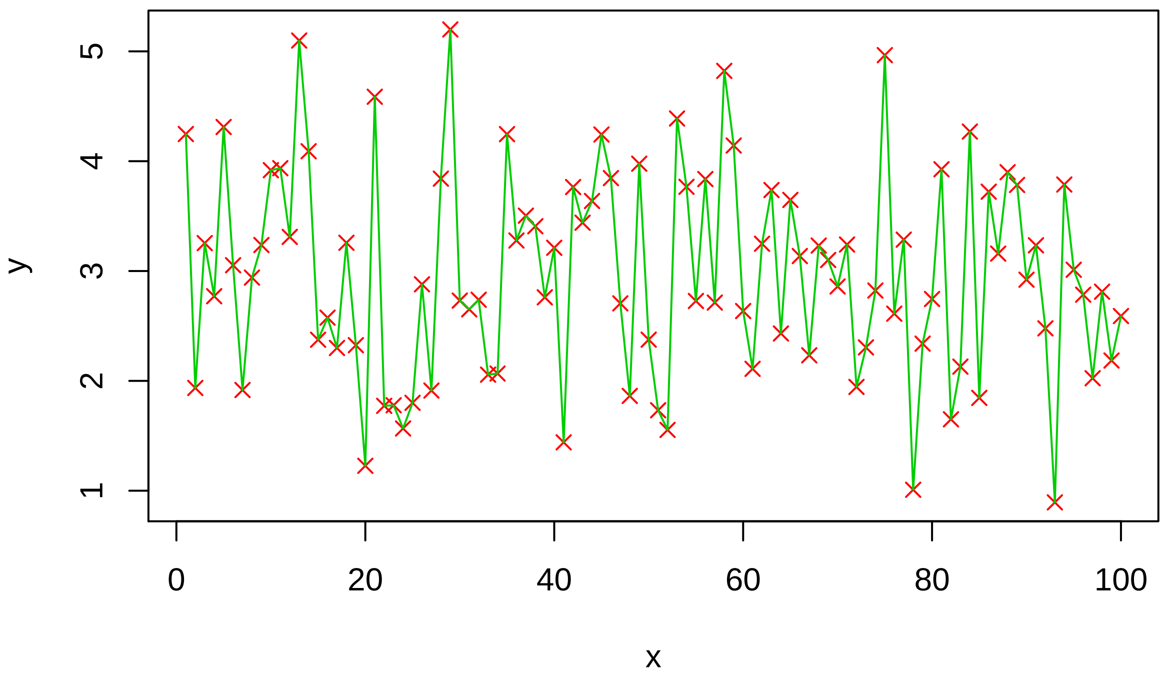
```r
plot(x,y,main='My First Plot')
```



```r
# change point type
plot(x,y,pch=3)
```



```
# change color
plot(x,y,pch=4,col=2)
# draw lines between points
lines(x,y,col=3)
```



### 6.1.2 More Plotting

```
z=sort(y)
# plot a sorted variable vs x
plot(x,z,main='Random Normal Numbers',
xlab='Index',ylab='Random Number')


# another example
plot(-4:4,-4:4)
# and add a point at (0,2) to the plot
points(0,2,pch=6,col=12)
```

### 6.1.3 More Plotting

```
# check margin and outer margin settings
par(c("mar", "oma"))
plot(x,y)
par(oma=c(1,1,1,1))  # set outer margin
plot(x,y)
par(mar=c(2.5,2.1,2.1,1)) # set margin
plot(x,y)


# A basic histogram
hist(z, main="Histogram",
    sub="Random normal")


# A "density" plot
plot(density(z), main="Density plot",
    sub="Random normal")


# A smaller "bandwidth" to capture more detail
plot(density(z, adjust=0.5),
  sub="smaller bandwidth")
```

### 6.1.4 Graphics Devices and Saving Plots

- to make a plot directly to a file use: `png()`, `postscript()`, etc.
- R can have multiple graphics "devices" open.
  - To see a list of active devices: `dev.list()`
  - To close the most recent device: `dev.off()`
  - To close device 5: `dev.off(5)`
  - To use device 5: `dev.set(5)`

### 6.1.5 More Plotting

- Save a png image to a file

  ```
  png(file="myplot.png",width=480,height=480)
  plot(density(z,adjust=2.0),sub="larger bandwidth")
  dev.off()
  ```

- On your own, save a pdf to a file. NOTE: The dimensions in `pdf()` are in *inches*
- Multiple plots on the same page:

  ```
  par(mfrow=c(2,1))
  plot(density(z,adjust=2.0),sub="larger bandwidth")
  hist(z)


  # use dev.off() to turn off the two-row plotting
  ```

#### 6.1.6 R Graphics Galleries and Resources

Visit these sites for some ideas.

- http://www.sr.bham.ac.uk/~ajrs/R/r-gallery.html
- http://gallery.r-enthusiasts.com/
- http://cran.r-project.org/web/views/Graphics.html

# 7 Control Structures, Looping, and Applying

## 7.1 Control Structures and Looping

### 7.1.1 Control Structures in R

- R has multiple types of control structures that allows for sequential evaluation of statements.
- For loops

  ```
  for (x in set) {operations}
  ```

- while loops

  ```
  while (x in condition){operations}
  ```

- If statements (conditional)

  ```
  if (condition) {
  some operations
   } else { other operations }
  ```

### 7.1.2 Control Structure and Looping Examples

```
x<-1:9
length(x)
# a simple conditional then two expressions
if (length(x)<=10) {
   x<-c(x,10:20);print(x)}
# more complex
if (length(x)<5) {
    print(x)
} else {
    print(x[5:20])
}
```

```
# print the values of x, one at a time
for (i in x) print(i)
for(i in x) i   # note R will not echo in a loop
```

#### 7.1.3 Control Structure and Looping Examples

```
# loop over a character vector
y<-c('a','b','hi there')
for (i in y) print(i)

# and a while loop
j<-1
while(j<10) { # do this while j<10
  print(j)
  j<-j+2} # at each iteration, increase j by 2
```

## 7.2 Applying

### 7.2.1 Why Does R Have Apply Functions

- Often we want to apply the same function to all the rows or columns of a matrix, or all the elements of a list.
- We could do this in a loop, but loops take a lot of time in an interpreted language like R.
- R has more efficient built-in operators, the apply functions.

example If mat is a matrix and fun is a function (such as mean, var, lm ...) that takes a vector as its argument, then you can:

```
apply(mat,1,fun) # over rows--second argument is 1
apply(mat,2,fun) # over columns--second argument is 2
```

In either case, the output is a vector.

### 7.2.2 Apply Function Exercise

1. Using the matrix and rnorm functions, create a matrix with 20 rows and 10 columns (200 values total) of random normal deviates.
2. Compute the mean for each row of the matrix.
3. Compute the median for each column.

### 7.2.3 Related Apply Functions

- `lapply(list, function)` applies the function to every element of list
- `sapply(list or vector, function)` applies the function to every element of list or vector, and returns a vector, when possible (easier to process)
- `tapply(x, factor, fun)` uses the factor to split vector x into groups, and then applies fun to each group

#### 7.2.4 Related Apply Function Examples

```
# create a list
my.list <- list(a=1:3,b=5:10,c=11:20)
my.list
# Get the mean for each member of the list
# return a vector
sapply( my.list, mean)
# Get the full summary for each member of
# the list, returned as a list
lapply( my.list, summary)
# Find the mean for each group defined by a factor
my.vector <- 1:10
my.factor <- factor(
  c(1,1,1,2,2,2,3,3,3,3))
tapply(my.vector, my.factor, mean)
```

# 8 Functions

#### 8.0.5 Function Overview

- Functions are objects and are assigned to names, just like data.

  ```
  myFunction = function(argument1,argument2) {
    expression1
    expression2
  }
  ```

- We write functions for anything we need to do again and again.
- You may test your commands interactively at first, and then use the `history()` feature and an editor to create the function.
- It is wise to include a comment at the start of each function to say what it does and to document functions of more than a few lines.

#### 8.0.6 Example Functions

```
add1 = function(x) {
    # this function adds one to the first argument and returns it
    x + 1
}
add1(17)
## [1] 18
add1(c(17,18,19,20))
## [1] 18 19 20 21
```

You can use the `edit()` function to make changes to a function. The following command will open a window, allow you to make changes, and assign the result to a new function, `add2`.

```
add2 = edit(add1)
```

### 8.0.7 Further Reading

The amount of learning material for R is simply astonishing!

- Thomas Girke's R and Bioconductor Manual
- A HUGE collection of contributed R documentation and tutorials
- Bioconductor course materials
- Sean Davis' website
- The Official R Manuals

## 9 *RStudio*: A Quick Tour

Panes

Options

Help

Environment, History, and Files

## 10 *R*: First Impressions

Type values and mathematical formulas into *R*'s command prompt

```
1 + 1
```

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

Assign values to symbols (variables)

```
x = 1
x + x
```

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

Invoke functions such as `c()`, which takes any number of values and returns a single *vector*

```
x = c(1, 2, 3)
x
```

```
## [1] 1 2 3
```

*R* functions, such as `sqrt()`, often operate efficiently on vectors

```
y = sqrt(x)
y
```

```
## [1] 1.000000 1.414214 1.732051
```

There are often several ways to accomplish a task in *R*

```
x = c(1, 2, 3)
x
```

```
## [1] 1 2 3
```

```
x <- c(4, 5, 6)
x
```

```
## [1] 4 5 6
```

```
x <- 7:9
x
```

```
## [1] 7 8 9
```

```
10:12 -> x
x
```

```
## [1] 10 11 12
```

Sometimes *R* does 'surprising' things that can be fun to figure out

```
x <- c(1, 2, 3) -> y
x
```

```
## [1] 1 2 3
```

```
y
```

```
## [1] 1 2 3
```

## 10.1 *R* Data types: vector and list

'Atomic' vectors

- Types include integer, numeric (float-point; real), complex, logical, character, raw (bytes)

  ```
  people <- c("Lori", "Nitesh", "Valerie", "Herve")
  people
  ```

  ```
  ## [1] "Lori"    "Nitesh"  "Valerie" "Herve"
  ```

- Atomic vectors can be named

  ```
  population <- c(Buffalo=259000, Rochester=210000, `New York`=8400000)
  population
  ```

  ```
  ##   Buffalo Rochester  New York
  ##    259000    210000   8400000
  ```

  ```
  log10(population)
  ```

  ```
  ##   Buffalo Rochester  New York
  ##  5.413300  5.322219  6.924279
  ```

- Statistical concepts like `NA` ("not available")

  ```
  truthiness <- c(TRUE, FALSE, NA)
  truthiness
  ```

  ```
  ## [1]  TRUE FALSE    NA
  ```

- Logical concepts like 'and' (`&`), 'or' (`|`), and 'not' (`!`)

  ```
  !truthiness
  ```

  ```
  ## [1] FALSE  TRUE    NA
  ```

  ```
  truthiness | !truthiness
  ```

  ```
  ## [1] TRUE TRUE   NA
  ```

```
truthiness & !truthiness
```

```
## [1] FALSE FALSE    NA
```

- Numerical concepts like infinity (`Inf`) or not-a-number (`NaN`, e.g., 0 / 0)

```
undefined_numeric_values <- c(NA, 0/0, NaN, Inf, -Inf)
undefined_numeric_values
```

```
## [1]   NA  NaN  NaN  Inf -Inf
```

```
sqrt(undefined_numeric_values)
```

```
## Warning in sqrt(undefined_numeric_values): NaNs produced
```

```
## [1]  NA NaN NaN Inf NaN
```

- Common string manipulations

```
toupper(people)
```

```
## [1] "LORI"    "NITESH"  "VALERIE" "HERVE"
```

```
substr(people, 1, 3)
```

```
## [1] "Lor" "Nit" "Val" "Her"
```

- *R* is a green consumer – recycling short vectors to align with long vectors

```
x <- 1:3
x * 2            # '2' (vector of length 1) recycled to c(2, 2, 2)
```

```
## [1] 2 4 6
```

```
truthiness | NA
```

```
## [1] TRUE   NA   NA
```

```
truthiness & NA
```

```
## [1]    NA FALSE    NA
```

- It's very common to nest operations, which can be simultaneously compact, confusing, and expressive (`[`: subset; `<`: less than)

```
substr(tolower(people), 1, 3)
```

```
## [1] "lor" "nit" "val" "her"
```

```
population[population < 1000000]
```

```
##   Buffalo Rochester
##    259000    210000
```

Lists

- The list type can contain other vectors, including other lists

```
frenemies = list(
    friends=c("Larry", "Richard", "Vivian"),
    enemies=c("Dick", "Mike")
)
frenemies
```

```
## $friends
## [1] "Larry"   "Richard" "Vivian"
##
## $enemies
## [1] "Dick" "Mike"
```

- `[` subsets one list to create another list, `[[` extracts a list element

```
frenemies[1]
```

```
## $friends
## [1] "Larry"   "Richard" "Vivian"
```

```
frenemies[c("enemies", "friends")]
```

```
## $enemies
## [1] "Dick" "Mike"
##
## $friends
## [1] "Larry"   "Richard" "Vivian"
```

```
frenemies[["enemies"]]
```

```
## [1] "Dick" "Mike"
```

Factors

- Character-like vectors, but with values restricted to specific levels

```
sex = factor(c("Male", "Male", "Female"),
             levels=c("Female", "Male", "Hermaphrodite"))
sex
```

```
## [1] Male   Male   Female
## Levels: Female Male Hermaphrodite
```

```
sex == "Female"
```

```
## [1] FALSE FALSE  TRUE
```

```
table(sex)
```

```
## sex
##        Female          Male Hermaphrodite
##             1             2             0
```

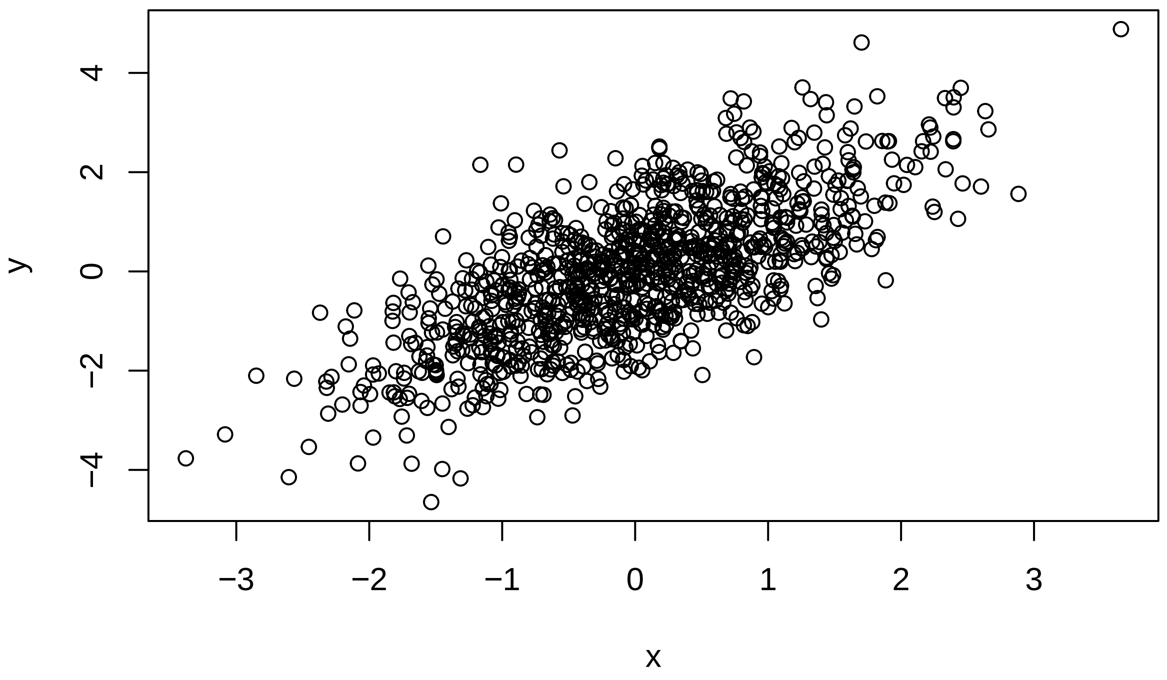
```
sex[sex == "Female"]
```

```
## [1] Female
## Levels: Female Male Hermaphrodite
```

## 10.2 Classes: data.frame and beyond

Variables are often related to one another in a highly structured way, e.g., two 'columns' of data in a spreadsheet

```
x = rnorm(1000)       # 1000 random normal deviates
y = x + rnorm(1000)   # another 1000 deviates, as a function of x
plot(y ~ x)           # relationship between x and y
```



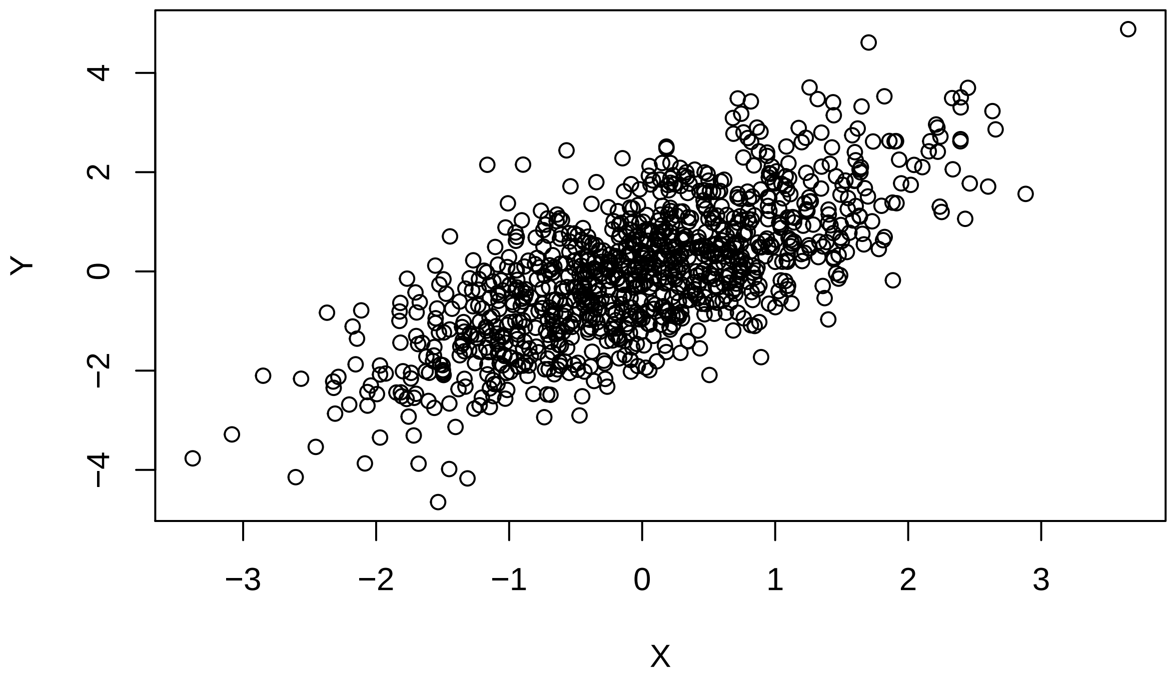
Convenient to manipulate them together

- `data.frame()`: like columns in a spreadsheet

```
df = data.frame(X=x, Y=y)
head(df)            # first 6 rows
```

```
##            X          Y
## 1  0.2412286  1.7552819
## 2  0.6542801 -0.2277761
## 3 -0.6545977 -1.2613852
## 4 -0.2626758 -2.3230579
## 5 -1.9706712 -2.0776550
## 6  1.0008351  0.1921736
```

```
plot(Y ~ X, df)     # same as above
```



- See all data with `View(df)`. Summarize data with `summary(df)`

```
summary(df)
```

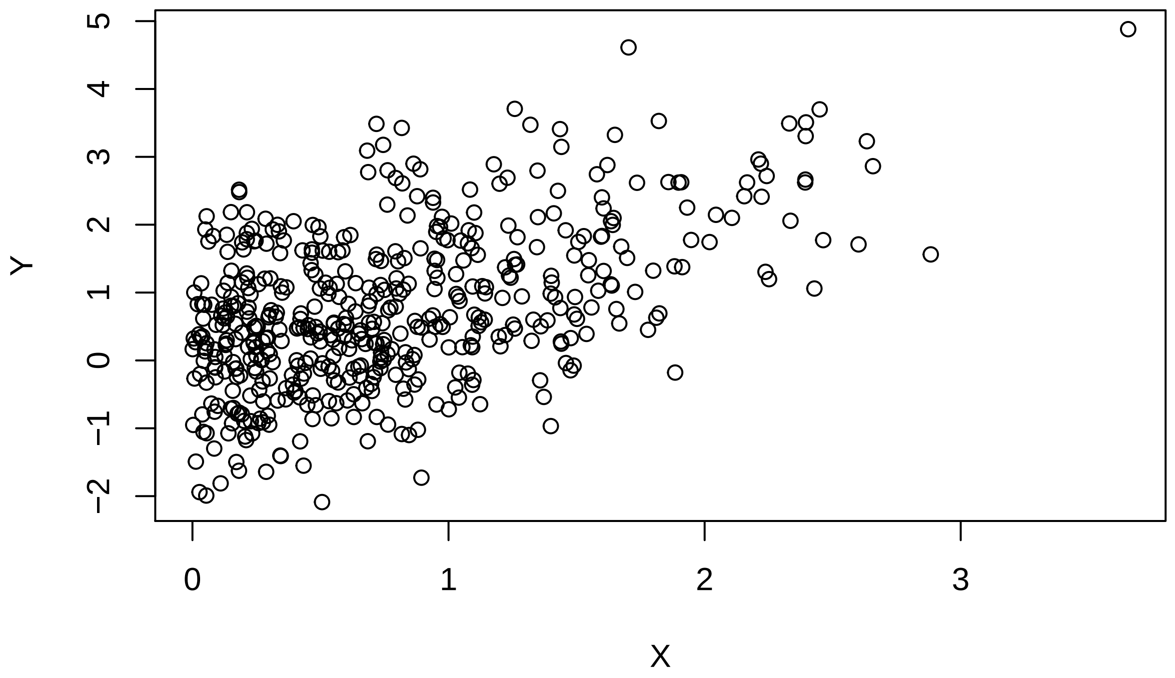
```
##        X                   Y            
##  Min.   :-3.379308   Min.   :-4.648441  
##  1st Qu.:-0.672767   1st Qu.:-0.936939  
##  Median :-0.007956   Median :-0.002262  
##  Mean   :-0.006235   Mean   :-0.014590  
##  3rd Qu.: 0.658513   3rd Qu.: 0.828722  
##  Max.   : 3.653966   Max.   : 4.881259
```

- Easy to manipulate data in a coordinated way, e.g., access column `X` with `$` and subset for just those values greater than 0

```
positiveX = df[df$X > 0,]
head(positiveX)
```

```
##            X          Y
## 1  0.2412286  1.7552819
## 2  0.6542801 -0.2277761
## 6  1.0008351  0.1921736
## 9  0.7185336  3.4859036
## 10 0.2097822 -1.1744343
## 11 0.3971115 -0.4739930
```

```
plot(Y ~ X, positiveX)
```



- *R* is introspective – ask it about itself

```
class(df)
```

```
## [1] "data.frame"
```

```
dim(df)
```

```
## [1] 1000    2
```

```
colnames(df)
```

```
## [1] "X" "Y"
```

- `matrix()` a related class, where all elements have the same type (a `data.frame()` requires elements within a column to be the same type, but elements between columns can be different types).

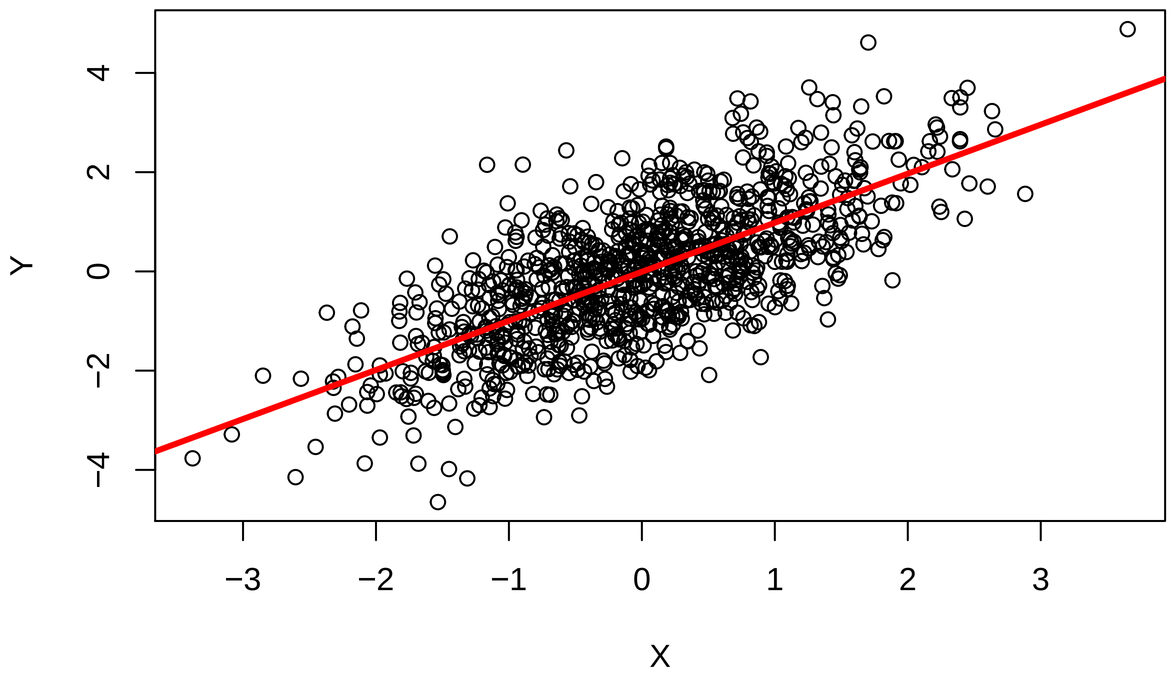
A scatterplot makes one want to fit a linear model (do a regression analysis)

- Use a *formula* to describe the relationship between variables
- Variables found in the second argument

```
fit <- lm(Y ~ X, df)
```

- Visualize the points, and add the regression line

```
plot(Y ~ X, df)
abline(fit, col="red", lwd=3)
```



- Summarize the fit as an ANOVA table

```
anova(fit)
```

```
## Analysis of Variance Table
##
## Response: Y
##            Df Sum Sq Mean Sq F value    Pr(>F)
## X           1 960.93  960.93  999.81 < 2.2e-16 ***
## Residuals 998 959.19    0.96
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

- N.B. – 'Type I' sums-of-squares, so order of independent variables matters; use `drop1()` for 'Type III'. See DataCamp Quick-R
- Introspection – what class is `fit`? What *methods* can I apply to an object of that class?

```
class(fit)
```

```
## [1] "lm"
```

```
methods(class=class(fit))
```

```
##  [1] add1           alias          anova          case.names
##  [5] confint        cooks.distance deviance       dfbeta
##  [9] dfbetas        drop1          dummy.coef     effects
## [13] extractAIC     family         formula        hatvalues
## [17] influence      kappa          labels         logLik
## [21] model.frame    model.matrix   nobs           plot
## [25] predict        print          proj           qr
## [29] residuals      rstandard      rstudent       simulate
## [33] summary        variable.names vcov
## see '?methods' for accessing help and source code
```

### 10.3 Help!

Help available in *Rstudio* or interactively

- Check out the help page for `rnorm()`

  ```
  ?rnorm
  ```

- 'Usage' section describes how the function can be used

  ```
  rnorm(n, mean = 0, sd = 1)
  ```

- Arguments, some with default values. Arguments matched first by name, then position
- 'Arguments' section describes what the arguments are supposed to be
- 'Value' section describes return value
- 'Examples' section illustrates use
- Often include citations to relevant technical documentation, reference to related functions, obscure details
- Can be intimidating, but in the end actually *very* useful

## 11 Exercise 1: BRFSS Survey Data

We will explore a subset of data collected by the CDC through its extensive Behavioral Risk Factor Surveillance System (BRFSS) telephone survey. Check out the link for more information. We'll look at a subset of the data.

1. Use `file.choose()` to find the path to the file 'BRFSS-subset.csv'

   ```
   path <- file.choose()
   ```

2. Input the data using `read.csv()`, assigning to a variable `brfss`

   ```
   brfss <- read.csv(path)
   ```

3. Use command like `class()`, `head()`, `dim()`, `summary()` to explore the data.

   - What variables have been measured?
   - Can you guess at the units used for, e.g., Weight and Height?

   ```
   class(brfss)
   head(brfss)
   dim(brfss)
   summary(brfss)
   ```

4. Use the `$` operator to extract the 'Sex' column, and summarize the number of males and females in the survey using `table()`. Do the same for 'Year', and for both `Sex` and `Year`

   ```
   table(brfss$Sex)
   ```

   ```
   ##
   ## Female   Male
   ##  12039   7961
   ```

   ```
   table(brfss$Year)
   ```

```
##
##  1990  2010
## 10000 10000
```

```
table(brfss$Sex, brfss$Year)
```

```
##
##          1990 2010
##   Female 5718 6321
##   Male   4282 3679
```

```
with(brfss, table(Sex, Year))                 # same, but easier
```

```
##         Year
## Sex      1990 2010
##   Female 5718 6321
##   Male   4282 3679
```

5. Use `aggregate()` to summarize the mean weight of each group. What about the median weight of each group? What about the *number* of observations in each group?

```
with(brfss, aggregate(Weight, list(Year, Sex), mean, na.rm=TRUE))
```

```
##   Group.1 Group.2        x
## 1    1990  Female 64.81838
## 2    2010  Female 72.95424
## 3    1990    Male 81.17999
## 4    2010    Male 88.84657
```

```
with(brfss, aggregate(Weight, list(Year=Year, Sex=Sex), mean, na.rm=TRUE))
```

```
##   Year    Sex        x
## 1 1990 Female 64.81838
## 2 2010 Female 72.95424
## 3 1990   Male 81.17999
## 4 2010   Male 88.84657
```

6. Use a `formula` and the `aggregate()` function to describe the relationship between Year, Sex, and Weight

```
aggregate(Weight ~ Year + Sex, brfss, mean)  # same, but more informative
```

```
##   Year    Sex   Weight
## 1 1990 Female 64.81838
## 2 2010 Female 72.95424
## 3 1990   Male 81.17999
## 4 2010   Male 88.84657
```

```
aggregate(. ~ Year + Sex, brfss, mean)       # all variables
```

```
##   Year    Sex      Age   Weight   Height
## 1 1990 Female 46.09153 64.84333 163.2914
## 2 2010 Female 57.07807 73.03178 163.2469
## 3 1990   Male 43.87574 81.19496 178.2242
## 4 2010   Male 56.25465 88.91136 178.0139
```

7. Create a subset of the data consisting of only the 1990 observations. Perform a t-test comparing the weight of males and females ("'Weight' as a function of 'Sex'", `Weight ~ Sex`)

```r
brfss_1990 = brfss[brfss$Year == 1990,]
t.test(Weight ~ Sex, brfss_1990)
```

```
##
##  Welch Two Sample t-test
##
## data:  Weight by Sex
## t = -58.734, df = 9214, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -16.90767 -15.81554
## sample estimates:
## mean in group Female   mean in group Male
##             64.81838             81.17999
```

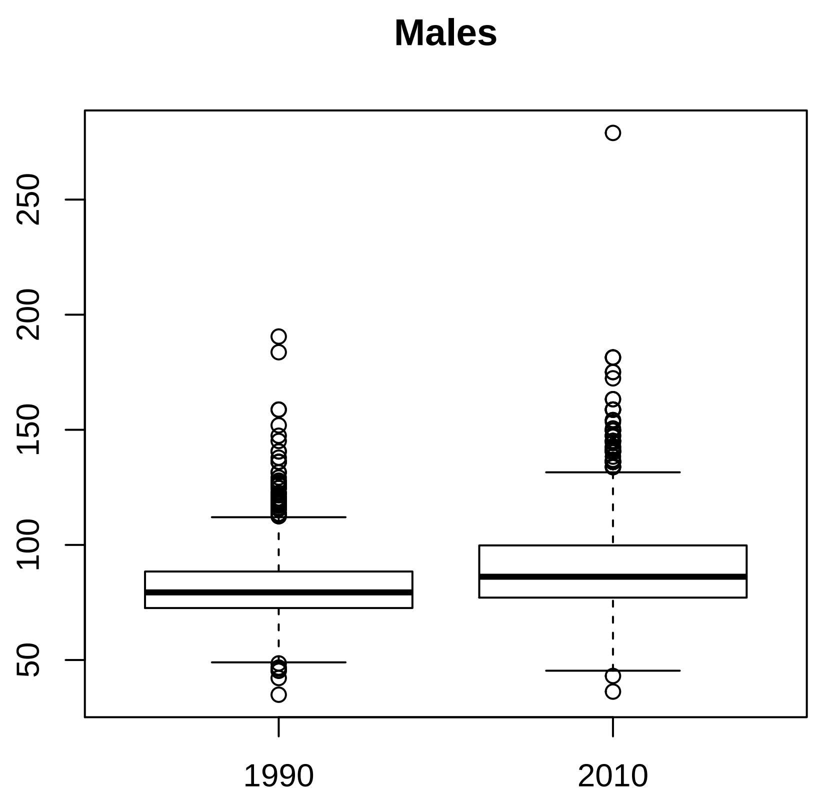
```r
t.test(Weight ~ Sex, brfss, subset = Year == 1990)
```

```
##
##  Welch Two Sample t-test
##
## data:  Weight by Sex
## t = -58.734, df = 9214, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -16.90767 -15.81554
## sample estimates:
## mean in group Female   mean in group Male
##             64.81838             81.17999
```

What about differences between weights of males (or females) in 1990 versus 2010? Check out the help page `?t.test.formula`. Is there a way of performing a t-test on `brfss` without explicitly creating the object `brfss_1990`?

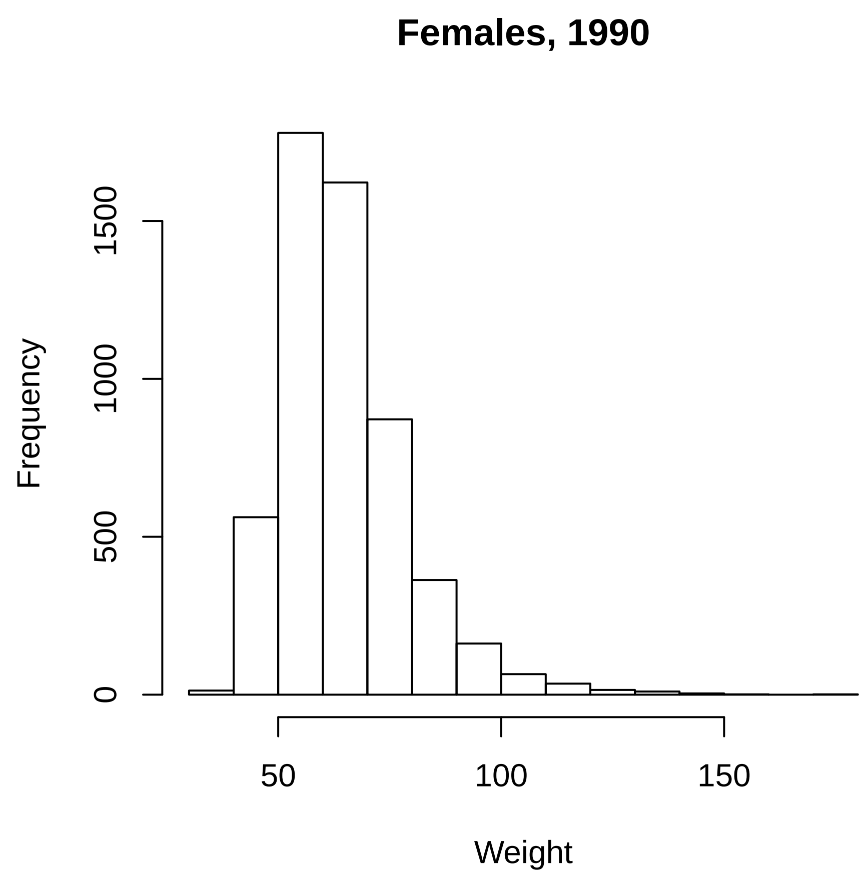
8. Use `boxplot()` to plot the weights of the Male individuals. Can you transform weight, e.g., `sqrt(Weight) ~ Year`? Interpret the results. Do similar boxplots for the t-tests of the previous question.

```r
boxplot(Weight ~ Year, brfss, subset = Sex == "Male",
        main="Males")
```



9. Use `hist()` to plot a histogram of weights of the 1990 Female individuals.

```
hist(brfss_1990[brfss_1990$Sex == "Female", "Weight"],
     main="Females, 1990", xlab="Weight" )
```



## 12 Exercise 2: ALL Phenotypic Data

This data comes from an (old) Acute Lymphoid Leukemia microarray data set.

Choose the file that contains ALL (acute lymphoblastic leukemia) patient information and input the date using `read.csv()`; for `read.csv()`, use `row.names=1` to indicate that the first column contains row names.

```
path <- file.choose()    # look for ALL-phenoData.csv
```

```
stopifnot(file.exists(path))
pdata <- read.csv(path, row.names=1)
```

Check out the help page `?read.delim` for input options. The exercises use `?read.csv`; Can you guess why? Explore basic properties of the object you've created, for instance...

```
class(pdata)
```

```
## [1] "data.frame"
```

```
colnames(pdata)
```

```
##  [1] "cod"            "diagnosis"      "sex"            "age"
##  [5] "BT"             "remission"      "CR"             "date.cr"
##  [9] "t.4.11."        "t.9.22."        "cyto.normal"    "citog"
## [13] "mol.biol"       "fusion.protein" "mdr"            "kinet"
## [17] "ccr"            "relapse"        "transplant"     "f.u"
## [21] "date.last.seen"
```

```r
dim(pdata)
```

```
## [1] 128  21
```

```r
head(pdata)
```

```
##        cod diagnosis sex age BT remission CR   date.cr t.4.11. t.9.22.
## 01005 1005 5/21/1997   M  53 B2        CR CR  8/6/1997   FALSE    TRUE
## 01010 1010 3/29/2000   M  19 B2        CR CR 6/27/2000   FALSE   FALSE
## 03002 3002 6/24/1998   F  52 B4        CR CR 8/17/1998      NA      NA
## 04006 4006 7/17/1997   M  38 B1        CR CR  9/8/1997    TRUE   FALSE
## 04007 4007 7/22/1997   M  57 B2        CR CR 9/17/1997   FALSE   FALSE
## 04008 4008 7/30/1997   M  17 B1        CR CR 9/27/1997   FALSE   FALSE
##       cyto.normal        citog mol.biol fusion.protein mdr   kinet   ccr
## 01005       FALSE      t(9;22)  BCR/ABL           p210 NEG dyploid FALSE
## 01010       FALSE  simple alt.      NEG           <NA> POS dyploid FALSE
## 03002          NA         <NA>  BCR/ABL           p190 NEG dyploid FALSE
## 04006       FALSE      t(4;11) ALL1/AF4           <NA> NEG dyploid FALSE
## 04007       FALSE      del(6q)      NEG           <NA> NEG dyploid FALSE
## 04008       FALSE complex alt.      NEG           <NA> NEG hyperd. FALSE
##       relapse transplant               f.u date.last.seen
## 01005   FALSE       TRUE BMT / DEATH IN CR           <NA>
## 01010    TRUE      FALSE               REL      8/28/2000
## 03002    TRUE      FALSE               REL     10/15/1999
## 04006    TRUE      FALSE               REL      1/23/1998
## 04007    TRUE      FALSE               REL      11/4/1997
## 04008    TRUE      FALSE               REL     12/15/1997
```

```r
summary(pdata$sex)
```

```
##    F    M NA's
##   42   83    3
```

```r
summary(pdata$cyto.normal)
```

```
##    Mode   FALSE    TRUE    NA's
## logical      69      24      35
```

Remind yourselves about various ways to subset and access columns of a data.frame

```r
pdata[1:5, 3:4]
```

```
##       sex age
## 01005   M  53
## 01010   M  19
## 03002   F  52
## 04006   M  38
## 04007   M  57
```

```r
pdata[1:5, ]
```

```
##        cod diagnosis sex age BT remission CR   date.cr t.4.11. t.9.22.
## 01005 1005 5/21/1997   M  53 B2        CR CR  8/6/1997   FALSE    TRUE
## 01010 1010 3/29/2000   M  19 B2        CR CR 6/27/2000   FALSE   FALSE
## 03002 3002 6/24/1998   F  52 B4        CR CR 8/17/1998      NA      NA
## 04006 4006 7/17/1997   M  38 B1        CR CR  9/8/1997    TRUE   FALSE
## 04007 4007 7/22/1997   M  57 B2        CR CR 9/17/1997   FALSE   FALSE
##       cyto.normal       citog mol.biol fusion.protein mdr   kinet   ccr
## 01005       FALSE     t(9;22)  BCR/ABL           p210 NEG dyploid FALSE
## 01010       FALSE simple alt.      NEG           <NA> POS dyploid FALSE
## 03002          NA        <NA>  BCR/ABL           p190 NEG dyploid FALSE
## 04006       FALSE     t(4;11) ALL1/AF4           <NA> NEG dyploid FALSE
## 04007       FALSE     del(6q)      NEG           <NA> NEG dyploid FALSE
##       relapse transplant               f.u date.last.seen
## 01005   FALSE       TRUE BMT / DEATH IN CR            <NA>
## 01010    TRUE      FALSE               REL       8/28/2000
## 03002    TRUE      FALSE               REL      10/15/1999
## 04006    TRUE      FALSE               REL       1/23/1998
## 04007    TRUE      FALSE               REL       11/4/1997
```

```
head(pdata[, 3:5])
```

```
##       sex age BT
## 01005   M  53 B2
## 01010   M  19 B2
## 03002   F  52 B4
## 04006   M  38 B1
## 04007   M  57 B2
## 04008   M  17 B1
```

```
tail(pdata[, 3:5], 3)
```

```
##        sex age BT
## 65003    M  30 T3
## 83001    M  29 T2
## LAL4  <NA>  NA  T
```

```
head(pdata$age)
```

```
## [1] 53 19 52 38 57 17
```

```
head(pdata$sex)
```

```
## [1] M M F M M M
## Levels: F M
```

```
head(pdata[pdata$age > 21,])
```

```
##        cod diagnosis sex age BT remission CR   date.cr t.4.11. t.9.22.
## 01005 1005 5/21/1997   M  53 B2        CR CR  8/6/1997   FALSE    TRUE
## 03002 3002 6/24/1998   F  52 B4        CR CR 8/17/1998      NA      NA
## 04006 4006 7/17/1997   M  38 B1        CR CR  9/8/1997    TRUE   FALSE
## 04007 4007 7/22/1997   M  57 B2        CR CR 9/17/1997   FALSE   FALSE
## 08001 8001 1/15/1997   M  40 B2        CR CR 3/26/1997   FALSE   FALSE
## 08011 8011 8/21/1998   M  33 B3        CR CR 10/8/1998   FALSE   FALSE
##       cyto.normal        citog mol.biol fusion.protein mdr   kinet   ccr
## 01005       FALSE      t(9;22)  BCR/ABL           p210 NEG dyploid FALSE
## 03002          NA         <NA>  BCR/ABL           p190 NEG dyploid FALSE
## 04006       FALSE      t(4;11) ALL1/AF4           <NA> NEG dyploid FALSE
## 04007       FALSE      del(6q)      NEG           <NA> NEG dyploid FALSE
## 08001       FALSE     del(p15)  BCR/ABL           p190 NEG    <NA> FALSE
## 08011       FALSE del(p15/p16)  BCR/ABL      p190/p210 NEG dyploid FALSE
##       relapse transplant               f.u date.last.seen
## 01005   FALSE       TRUE BMT / DEATH IN CR            <NA>
## 03002    TRUE      FALSE               REL      10/15/1999
## 04006    TRUE      FALSE               REL       1/23/1998
## 04007    TRUE      FALSE                REL      11/4/1997
## 08001    TRUE      FALSE                REL      7/11/1997
## 08011   FALSE       TRUE BMT / DEATH IN CR           <NA>
```

It seems from below that there are 17 females over 40 in the data set. However, some individuals have `NA` for the age and / or sex, and these `NA` values propagate through some computations. Use `table()` to summarize the number of females over 40, and the number of samples for which this classification cannot be determined. When *R* encounters an `NA` value in a subscript index, it introduces an `NA` into the result. Observe this (rows of `NA` values introduced into the result) when subsetting using `[` versus using the `subset()` function.

```
idx <- pdata$sex == "F" & pdata$age > 40
table(idx, useNA="ifany")
```

```
## idx
## FALSE  TRUE  <NA>
##   108    17     3
```

```
dim(pdata[idx,])            # WARNING: 'NA' rows introduced
```

```
## [1] 20 21
```

```
tail(pdata[idx,])
```

```
##         cod  diagnosis  sex age   BT remission                 CR
## 49006 49006  8/12/1998    F  43   B2        CR                 CR
## 57001 57001  1/29/1997    F  53   B3      <NA> DEATH IN INDUCTION
## 62001 62001 11/11/1997    F  50   B4       REF                REF
## NA.1   <NA>       <NA> <NA>  NA <NA>      <NA>               <NA>
## 02020  2020  3/23/2000    F  48   T2      <NA> DEATH IN INDUCTION
## NA.2   <NA>       <NA> <NA>  NA <NA>      <NA>               <NA>
##          date.cr t.4.11. t.9.22. cyto.normal         citog mol.biol
## 49006 11/19/1998      NA      NA          NA          <NA>  BCR/ABL
## 57001       <NA>   FALSE   FALSE        TRUE        normal      NEG
## 62001       <NA>   FALSE    TRUE       FALSE t(9;22)+other  BCR/ABL
## NA.1        <NA>      NA      NA          NA          <NA>     <NA>
## 02020       <NA>   FALSE   FALSE       FALSE  complex alt.      NEG
## NA.2        <NA>      NA      NA          NA          <NA>     <NA>
##       fusion.protein  mdr   kinet   ccr relapse transplant  f.u
## 49006           p210  NEG diploid FALSE    TRUE      FALSE  REL
## 57001           <NA>  NEG hyperd.    NA      NA         NA <NA>
## 62001           <NA>  NEG hyperd.    NA      NA         NA <NA>
## NA.1            <NA> <NA>    <NA>    NA      NA         NA <NA>
## 02020           <NA>  NEG diploid    NA      NA         NA <NA>
## NA.2            <NA> <NA>    <NA>    NA      NA         NA <NA>
##       date.last.seen
## 49006      4/26/1999
## 57001           <NA>
## 62001           <NA>
## NA.1            <NA>
## 02020           <NA>
## NA.2            <NA>
```

```
dim(subset(pdata, idx))     # BETTER: no NA rows
```

```
## [1] 17 21
```

```
dim(subset(pdata, (sex == "F") & (age > 40)))  # alternative
```

```
## [1] 17 21
```

```r
tail(subset(pdata,idx))
```

```
##         cod  diagnosis sex age BT remission                 CR    date.cr
## 28032 28032  9/26/1998   F  52 B1        CR                 CR 10/30/1998
## 30001 30001  1/16/1997   F  54 B3      <NA> DEATH IN INDUCTION       <NA>
## 49006 49006  8/12/1998   F  43 B2        CR                 CR 11/19/1998
## 57001 57001  1/29/1997   F  53 B3      <NA> DEATH IN INDUCTION       <NA>
## 62001 62001 11/11/1997   F  50 B4       REF                REF       <NA>
## 02020  2020  3/23/2000   F  48 T2      <NA> DEATH IN INDUCTION       <NA>
##       t.4.11. t.9.22. cyto.normal         citog mol.biol fusion.protein
## 28032    TRUE   FALSE       FALSE       t(4;11) ALL1/AF4           <NA>
## 30001   FALSE    TRUE       FALSE t(9;22)+other  BCR/ABL           p190
## 49006      NA      NA          NA          <NA>  BCR/ABL           p210
## 57001   FALSE   FALSE        TRUE        normal      NEG           <NA>
## 62001   FALSE    TRUE       FALSE t(9;22)+other  BCR/ABL           <NA>
## 02020   FALSE   FALSE       FALSE  complex alt.      NEG           <NA>
##       mdr   kinet   ccr relapse transplant  f.u date.last.seen
## 28032 NEG dyploid  TRUE   FALSE      FALSE  CCR      5/16/2002
## 30001 NEG hyperd.    NA      NA         NA <NA>           <NA>
## 49006 NEG dyploid FALSE    TRUE      FALSE  REL      4/26/1999
## 57001 NEG hyperd.    NA      NA         NA <NA>           <NA>
## 62001 NEG hyperd.    NA      NA         NA <NA>           <NA>
## 02020 NEG dyploid    NA      NA         NA <NA>           <NA>
```

```r
## robust `[`: exclude NA values
dim(pdata[idx & !is.na(idx),])
```

```
## [1] 17 21
```

Use the `mol.biol` column to subset the data to contain just individuals with 'BCR/ABL' or 'NEG', e.g.,

```r
bcrabl <- subset(pdata, mol.biol %in% c("BCR/ABL", "NEG"))
```

The `mol.biol` column is a factor, and retains all levels even after subsetting. It is sometimes convenient to retain factor levels, but in our case we use `droplevels()` to removed unused levels

```r
bcrabl$mol.biol <- droplevels(bcrabl$mol.biol)
```

The `BT` column is a factor describing B- and T-cell subtypes

```r
levels(bcrabl$BT)
```

```
##  [1] "B"  "B1" "B2" "B3" "B4" "T"  "T1" "T2" "T3" "T4"
```

How might one collapse B1, B2, ... to a single type B, and likewise for T1, T2, ..., so there are only two subtypes, B and T? One strategy is to replace two-letter level (e.g., `B1`) with the single-letter level (e.g., `B`). Do this using `substring()` to select the first letter of level, and update the previous levels with the new value using `levels<-`.

```r
table(bcrabl$BT)
```

```
## 
##  B B1 B2 B3 B4  T T1 T2 T3 T4
##  4  9 35 22  9  5  1 15  9  2
```

```r
levels(bcrabl$BT) <- substring(levels(bcrabl$BT), 1, 1)
table(bcrabl$BT)
```

```
##
##  B  T
## 79 32
```

Use `aggregate()` to count the number of samples with B- and T-cell types in each of the BCR/ABL and NEG groups

```
aggregate(rownames(bcrabl) ~ BT + mol.biol, bcrabl, length)
```

```
##   BT mol.biol rownames(bcrabl)
## 1  B  BCR/ABL               37
## 2  B      NEG               42
## 3  T      NEG               32
```

Use `aggregate()` to calculate the average age of males and females in the BCR/ABL and NEG treatment groups.

```
aggregate(age ~ mol.biol + sex, bcrabl, mean)
```

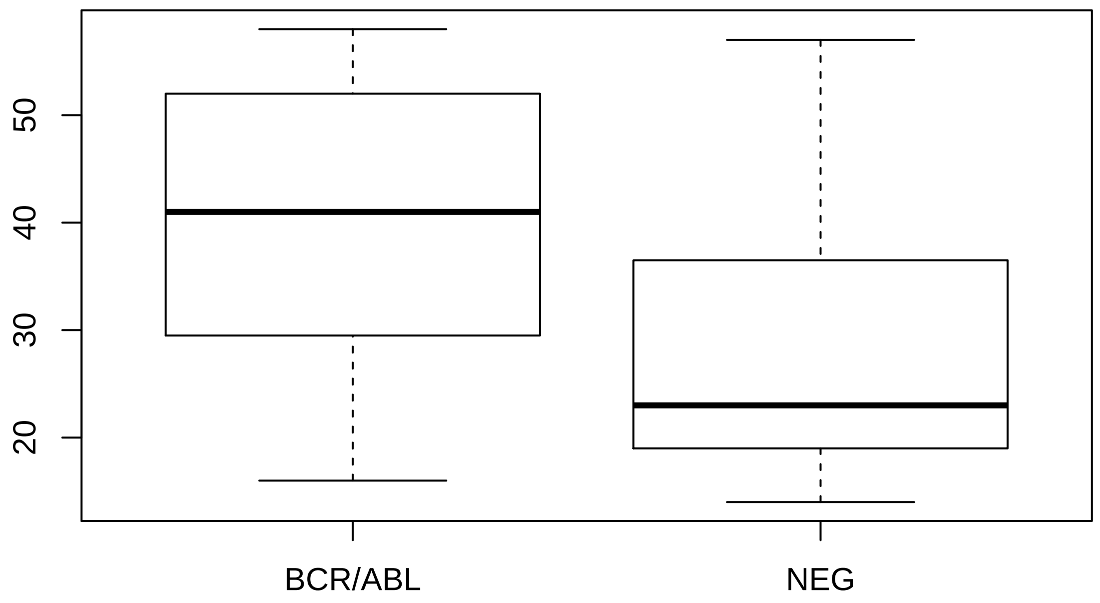
```
##   mol.biol sex      age
## 1  BCR/ABL   F 39.93750
## 2      NEG   F 30.42105
## 3  BCR/ABL   M 40.50000
## 4      NEG   M 27.21154
```

Use `t.test()` to compare the age of individuals in the BCR/ABL versus NEG groups; visualize the results using `boxplot()`. In both cases, use the `formula` interface. Consult the help page `?t.test` and re-do the test assuming that variance of ages in the two groups is identical. What parts of the test output change?

```
t.test(age ~ mol.biol, bcrabl)
```

```
##
##  Welch Two Sample t-test
##
## data:  age by mol.biol
## t = 4.8172, df = 68.529, p-value = 8.401e-06
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   7.13507 17.22408
## sample estimates:
## mean in group BCR/ABL     mean in group NEG
##              40.25000              28.07042
```

```
boxplot(age ~ mol.biol, bcrabl)
```



## 13 Exploration and simple univariate measures

```
path <- file.choose()    # look for BRFSS-subset.csv
```

```
stopifnot(file.exists(path))
brfss <- read.csv(path)
```

### 13.1 Clean data

*R* read `Year` as an integer value, but it's really a `factor`

```
brfss$Year <- factor(brfss$Year)
```

### 13.2 Weight in 1990 vs. 2010 Females

Create a subset of the data

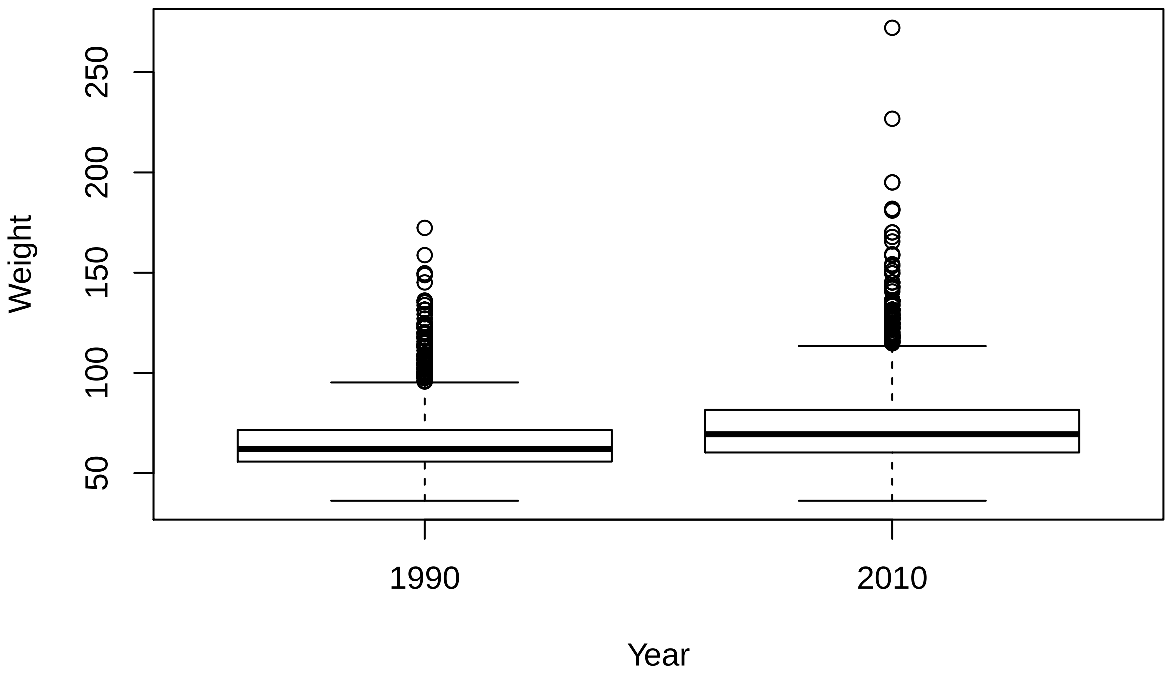
```
brfssFemale <- brfss[brfss$Sex == "Female",]
summary(brfssFemale)
```

```
##       Age            Weight            Sex            Height     
##  Min.   :18.00   Min.   : 36.29   Female:12039   Min.   :105.0  
##  1st Qu.:37.00   1st Qu.: 57.61   Male  :    0   1st Qu.:157.5  
##  Median :52.00   Median : 65.77                  Median :163.0  
##  Mean   :51.92   Mean   : 69.05                  Mean   :163.3  
##  3rd Qu.:67.00   3rd Qu.: 77.11                  3rd Qu.:168.0  
##  Max.   :99.00   Max.   :272.16                  Max.   :200.7  
##  NA's   :103     NA's   :560                     NA's   :140    
##    Year     
##  1990:5718  
##  2010:6321  
## 
```

```
##
##
##
##
```

Visualize

```r
plot(Weight ~ Year, brfssFemale)
```



Statistical test

```r
t.test(Weight ~ Year, brfssFemale)
```

```
##
##  Welch Two Sample t-test
##
## data:  Weight by Year
## t = -27.133, df = 11079, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -8.723607 -7.548102
## sample estimates:
## mean in group 1990 mean in group 2010
##           64.81838           72.95424
```

## 13.3 Weight and height in 2010 Males

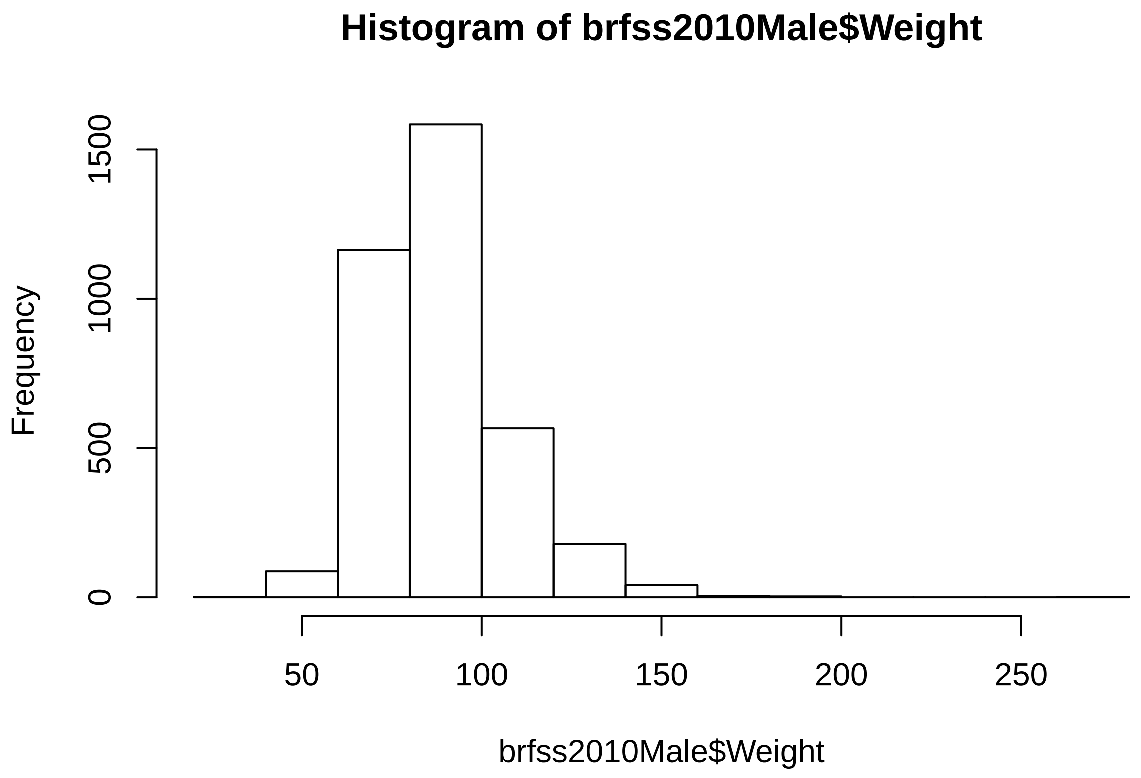
Create a subset of the data

```r
brfss2010Male <- subset(brfss,  Year == 2010 & Sex == "Male")
summary(brfss2010Male)
```

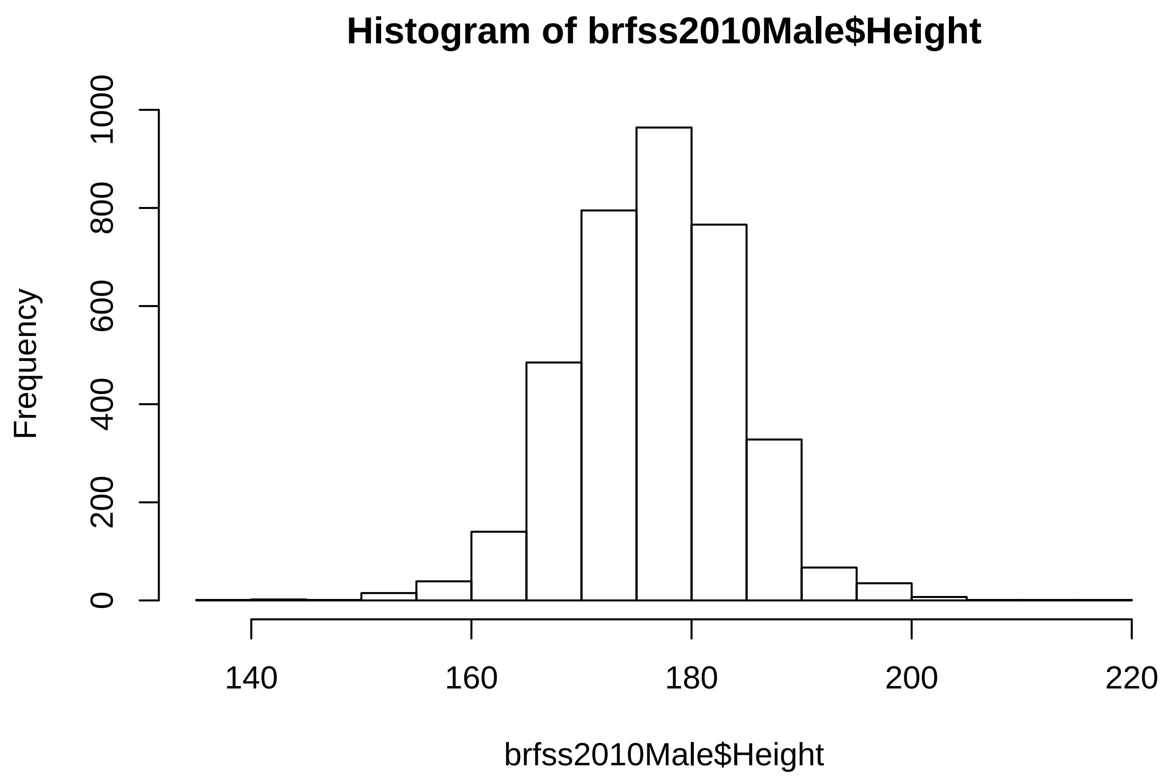
```
##       Age              Weight            Sex             Height      Year
##  Min.   :18.00   Min.   : 36.29   Female:    0   Min.   :135   1990:    0
##  1st Qu.:45.00   1st Qu.: 77.11   Male  :3679   1st Qu.:173   2010:3679
##  Median :57.00   Median : 86.18                 Median :178
##  Mean   :56.25   Mean   : 88.85                 Mean   :178
##  3rd Qu.:68.00   3rd Qu.: 99.79                 3rd Qu.:183
##  Max.   :99.00   Max.   :278.96                 Max.   :218
##  NA's   :30      NA's   :49                     NA's   :31
```

Visualize the relationship

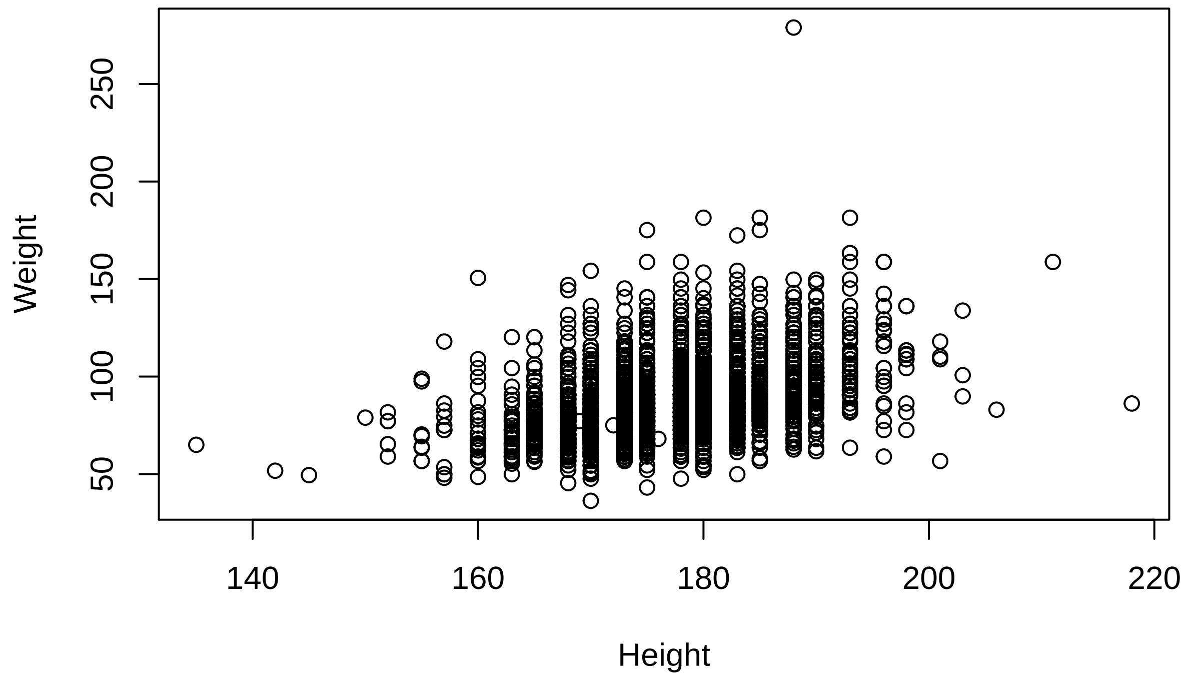
```r
hist(brfss2010Male$Weight)
```



```r
hist(brfss2010Male$Height)
```



```r
plot(Weight ~ Height, brfss2010Male)
```



Fit a linear model (regression)

```r
fit <- lm(Weight ~ Height, brfss2010Male)
fit
```

```
##
## Call:
```

```
## lm(formula = Weight ~ Height, data = brfss2010Male)
##
## Coefficients:
## (Intercept)       Height
##    -86.8747       0.9873
```

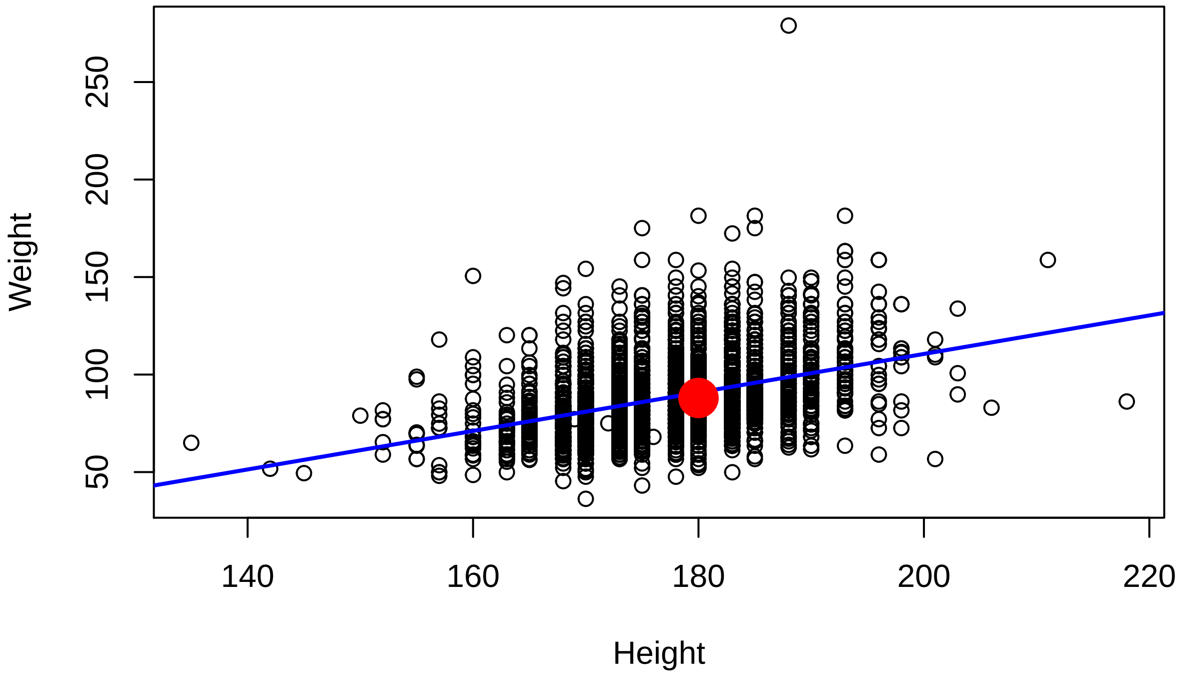
Summarize as ANOVA table

```
anova(fit)
```

```
## Analysis of Variance Table
##
## Response: Weight
##             Df  Sum Sq Mean Sq F value    Pr(>F)
## Height       1  197664  197664   693.8 < 2.2e-16 ***
## Residuals 3617 1030484     285
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Plot points, superpose fitted regression line; where am I?

```
plot(Weight ~ Height, brfss2010Male)
abline(fit, col="blue", lwd=2)
points(180, 88, col="red", cex=4, pch=20)
```



Class and available ‘methods’

```
class(fit)                # 'noun'
methods(class=class(fit)) # 'verb'
```

Diagnostics

```
plot(fit)
?plot.lm
```

# 14 Multivariate analysis

This is a classic microarray experiment. Microarrays consist of 'probesets' that interogate genes for their level of expression. In the experiment we're looking at, there are 12625 probesets measured on each of the 128 samples. The raw expression levels estimated by microarray assays require considerable pre-processing, the data we'll work with has been pre-processed.

## 14.1 Input and setup

Start by finding the expression data file on disk.

```
path <- file.choose()          # look for ALL-expression.csv
stopifnot(file.exists(path))
```

The data is stored in 'comma-separate value' format, with each probeset occupying a line, and the expression value for each sample in that probeset separated by a comma. Input the data using `read.csv()`. There are three challenges:

1. The row names are present in the first column of the data. Tell *R* this by adding the argument `row.names=1` to `read.csv()`.
2. By default, *R* checks that column names do not look like numbers, but our column names *do* look like numbers. Use the argument `check.colnames=FALSE` to over-ride *R*'s default.
3. `read.csv()` returns a `data.frame`. We could use a `data.frame` to work with our data, but really it is a `matrix()` – the columns are of the same type and measure the same thing. Use `as.matrix()` to coerce the `data.frame` we input to a `matrix`.

```
exprs <- read.csv(path, row.names=1, check.names=FALSE)
exprs <- as.matrix(exprs)
class(exprs)
```

```
## [1] "matrix"
```

```
dim(exprs)
```

```
## [1] 12625   128
```

```
exprs[1:6, 1:10]
```

```
##              01005     01010    03002    04006    04007     04008
## 1000_at   7.597323  7.479445 7.567593 7.384684 7.905312  7.065914
## 1001_at   5.046194  4.932537 4.799294 4.922627 4.844565  5.147762
## 1002_f_at 3.900466  4.208155 3.886169 4.206798 3.416923  3.945869
## 1003_s_at 5.903856  6.169024 5.860459 6.116890 5.687997  6.208061
## 1004_at   5.925260  5.912780 5.893209 6.170245 5.615210  5.923487
## 1005_at   8.570990 10.428299 9.616713 9.937155 9.983809 10.063484
##               04010     04016    06002     08001
## 1000_at    7.474537  7.536119 7.183331  7.735545
## 1001_at    5.122518  5.016132 5.288943  4.633217
## 1002_f_at  4.150506  3.576360 3.900935  3.630190
## 1003_s_at  6.292713  5.665991 5.842326  5.875375
## 1004_at    6.046607  5.738218 5.994515  5.748350
## 1005_at   10.662059 11.269115 8.812869 10.165159
```

```
range(exprs)
```

```
## [1]  1.984919 14.126571
```

We'll make use of the data describing the samples

```
path <- file.choose()         # look for ALL-phenoData.csv
stopifnot(file.exists(path))
```

```
pdata <- read.csv(path, row.names=1)
class(pdata)
```

```
## [1] "data.frame"
```

```
dim(pdata)
```

```
## [1] 128  21
```

```
head(pdata)
```

```
##        cod diagnosis sex age BT remission CR   date.cr t.4.11. t.9.22.
## 01005 1005 5/21/1997   M  53 B2        CR CR  8/6/1997   FALSE    TRUE
## 01010 1010 3/29/2000   M  19 B2        CR CR 6/27/2000   FALSE   FALSE
## 03002 3002 6/24/1998   F  52 B4        CR CR 8/17/1998      NA      NA
## 04006 4006 7/17/1997   M  38 B1        CR CR  9/8/1997    TRUE   FALSE
## 04007 4007 7/22/1997   M  57 B2        CR CR 9/17/1997   FALSE   FALSE
## 04008 4008 7/30/1997   M  17 B1        CR CR 9/27/1997   FALSE   FALSE
##       cyto.normal        citog mol.biol fusion.protein mdr   kinet   ccr
## 01005       FALSE      t(9;22)  BCR/ABL           p210 NEG dyploid FALSE
## 01010       FALSE  simple alt.      NEG           <NA> POS dyploid FALSE
## 03002          NA         <NA>  BCR/ABL           p190 NEG dyploid FALSE
## 04006       FALSE      t(4;11) ALL1/AF4           <NA> NEG dyploid FALSE
## 04007       FALSE      del(6q)      NEG           <NA> NEG dyploid FALSE
## 04008       FALSE complex alt.      NEG           <NA> NEG hyperd. FALSE
##       relapse transplant               f.u date.last.seen
## 01005   FALSE       TRUE BMT / DEATH IN CR           <NA>
## 01010    TRUE      FALSE               REL      8/28/2000
## 03002    TRUE      FALSE               REL     10/15/1999
## 04006    TRUE      FALSE               REL      1/23/1998
## 04007    TRUE      FALSE               REL      11/4/1997
## 04008    TRUE      FALSE               REL     12/15/1997
```

Some of the results below involve plots, and it's convenient to choose pretty and functional colors. We use the RColorBrewer package; see colorbrewer.org

```
library(RColorBrewer)  ## not available? install package via RStudio
highlight <- brewer.pal(3, "Set2")[1:2]
```

'highlight' is a vector of length 2, light and dark green.

For more options see `?RColorBrewer` and to view the predefined palettes `display.brewer.all()`

## 14.2 Cleaning

We'll add a column to `pdata`, derived from the `BT` column, to indicate whether the sample is B-cell or T-cell ALL.

```
pdata$BorT <- factor(substr(pdata$BT, 1, 1))
```

Microarray expression data is usually represented as a matrix of genes as rows and samples as columns. Statisticians usually think of their data as samples as rows, features as columns. So we'll transpose the expression values

```
exprs <- t(exprs)
```

Confirm that the `pdata` rows correspond to the `exprs` rows.

```
stopifnot(identical(rownames(pdata), rownames(exprs)))
```

### 14.3 Unsupervised machine learning – multi-dimensional scaling

Reduce high-dimensional data to lower dimension for visualization.

Calculate distance between *samples* (requires that the expression matrix be transposed).

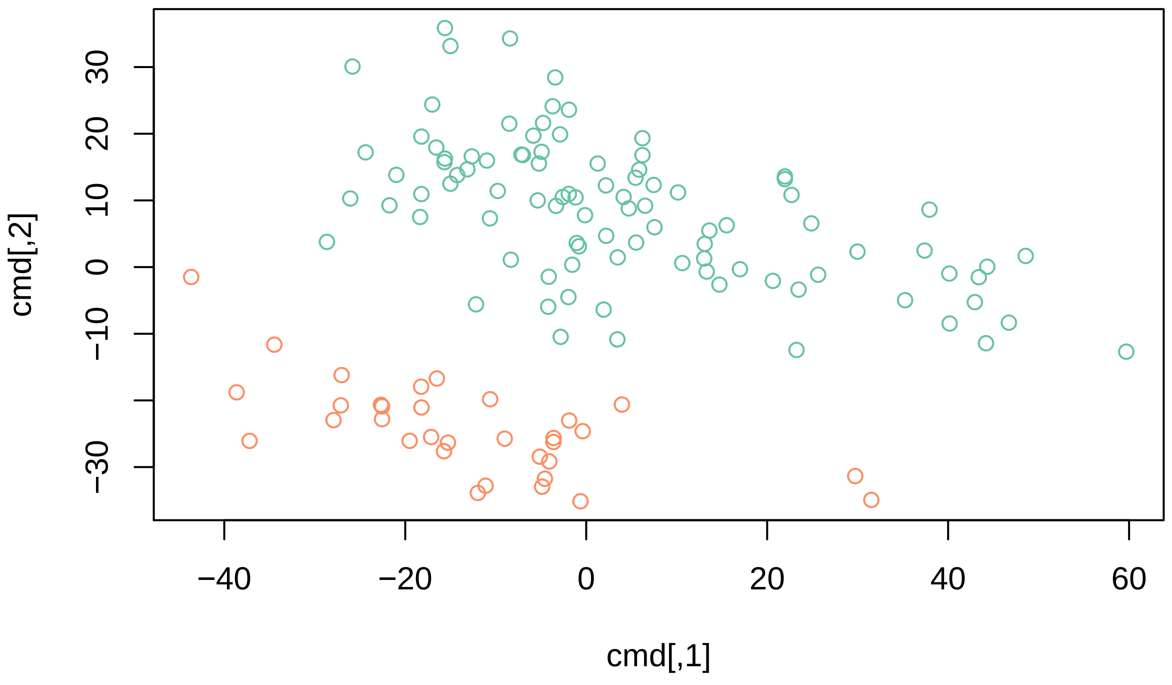
```
d <- dist(exprs)
```

Use the `cmdscale()` function to summarize the distance matrix into two points in two dimensions.

```
cmd <- cmdscale(d)
```

Visualize the result, coloring points by B- or T-cell status

```
plot(cmd, col=highlight[pdata$BorT])
```



## 15 Using *R* in real life

### 15.1 Organizing work

Usually, work is organized into a directory with:

- A folder containing *R* scripts (`scripts/BRFSS-visualize.R`)
- 'External' data like the csv files that we've been working with, usually in a separate folder (`extdata/BRFSS-subset.csv`)

- (sometimes) *R* objects written to disk using `saveRDS()` (`.rds` files) that represent final results or intermediate 'checkpoints' (`extdata/ALL-cleaned.rds`). Read the data into an *R* session using `readRDS()`.
- Use `setwd()` to navigate to folder containing scripts/, extdata/ folder
- Source an entire script with `source("scripts/BRFSS-visualization.R")`.

*R* can also save the state of the current session (prompt when choosing to `quit()` *R*), and to view and save the `history()` of the the current session; I do not find these to be helpful in my own work flows.

## 15.2 *R* Packages

All the functionality we have been using comes from *packages* that are automatically *loaded* when *R* starts. Loaded packages are on the `search()` path.

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "Autoloads"         "package:base"
```

Additional packages may be *installed* in *R*'s libraries. Use 'installed.packages() or the *RStudio* interface to see installed packages. To use these packages, it is necessary to attach them to the search path, e.g., for survival analysis

```
library("survival")
```

There are many thousands of *R* packages, and not all of them are installed in a single installation. Important repositories are

- CRAN: https://cran.r-project.org/
- Bioconductor: https://bioconductor.org/packages

Packages can be discovered in various ways, including CRAN Task Views and the *Bioconductor* web and *Bioconductor* support sites.

To install a package, use `install.packages()` or, for *Bioconductor* packages, instructions on the package landing page, e.g., for GenomicRanges. Here we install the ggplot2 package.

```
install.packages("ggplot2", repos="https://cran.r-project.org")
```

A package needs to be installed once, and then can be used in any *R* session.

# 16 Graphics and Visualization

Load the BRFSS-subset.csv data

```
path <- "BRFSS-subset.csv"   # or file.choose()
brfss <- read.csv(path)
```

Clean it by coercing `Year` to factor

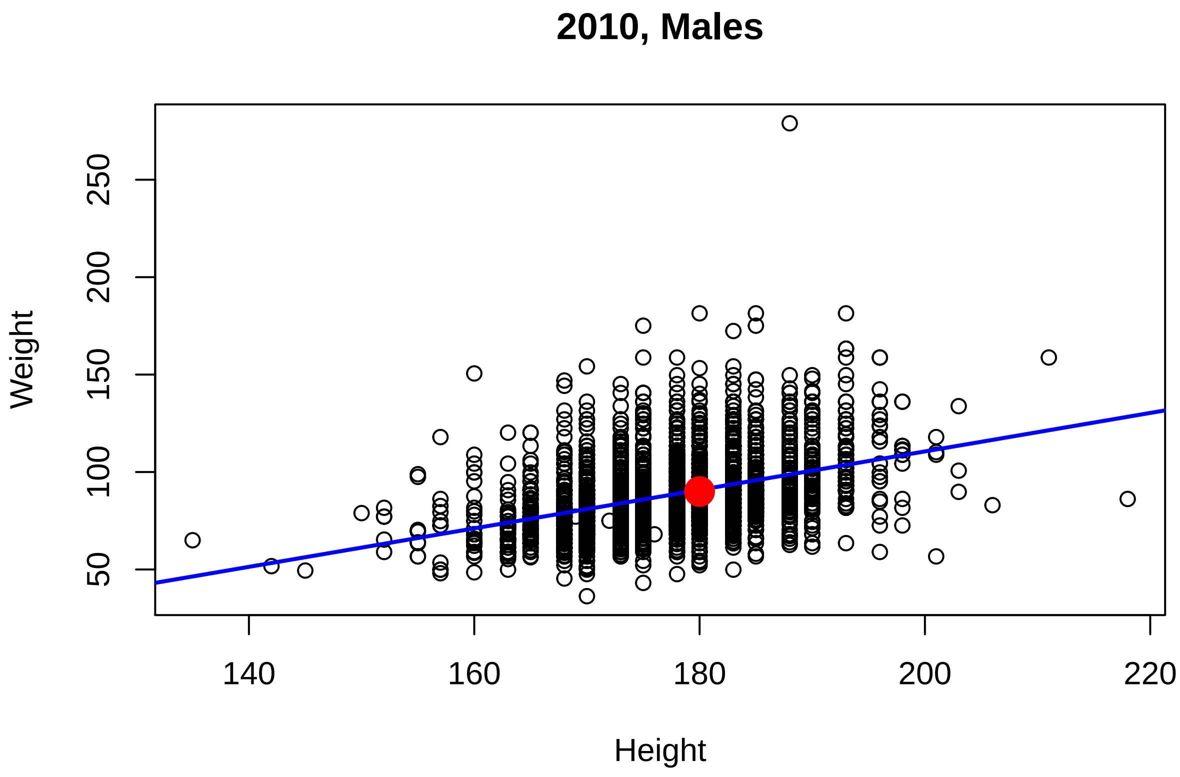
```
brfss$Year <- factor(brfss$Year)
```

## 16.1 Base *R* Graphics

Useful for quick exploration during a normal work flow.

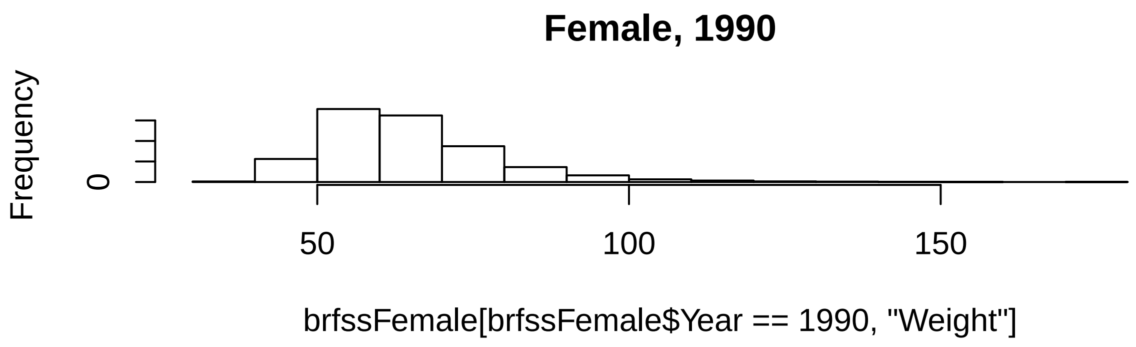
- Main functions: `plot()`, `hist()`, `boxplot()`, ...
- Graphical parameters – see `?par`, but often provided as arguments to `plot()`, etc.
- Construct complicated plots by layering information, e.g., points, regression line, annotation.

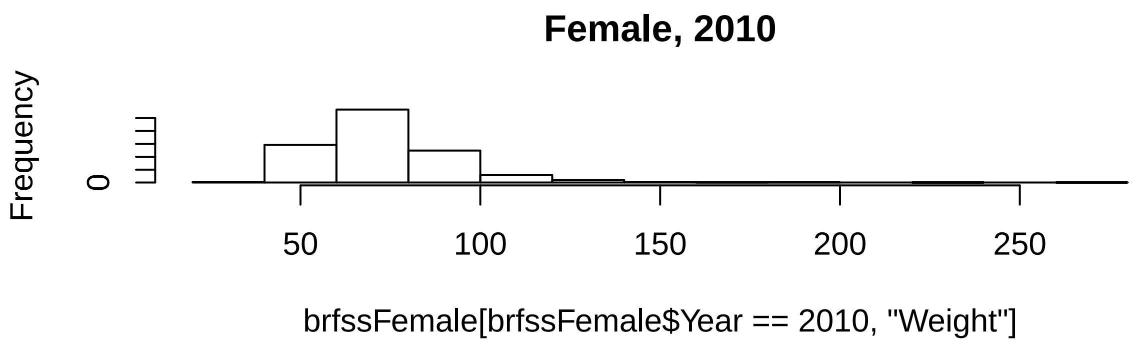
```
brfss2010Male <- subset(brfss, (Year == 2010) & (Sex == "Male"))
fit <- lm(Weight ~ Height, brfss2010Male)

plot(Weight ~ Height, brfss2010Male, main="2010, Males")
abline(fit, lwd=2, col="blue")
points(180, 90, pch=20, cex=3, col="red")
```



- Approach to complicated graphics: create a grid of panels (e.g., `par(mfrows=c(1, 2))`, populate with plots, restore original layout.

```
brfssFemale <- subset(brfss, Sex=="Female")

opar = par(mfrow=c(2, 1))      # layout: 2 'rows' and 1 'column'
hist(                          # first panel -- 1990
    brfssFemale[ brfssFemale$Year == 1990, "Weight" ],
    main = "Female, 1990")
hist(                          # second panel -- 2010
    brfssFemale[ brfssFemale$Year == 2010, "Weight" ],
    main = "Female, 2010")
```





```r
par(opar)                         # restore original layout
```

## 16.2 What makes for a good graphical display?

- Common scales for comparison
- Efficient use of space
- Careful color choice – qualitative, gradient, divergent schemes; color blind aware; . . .
- Emphasis on data rather than labels
- Convey statistical uncertainty

## 16.3 Grammar of Graphics: ggplot2

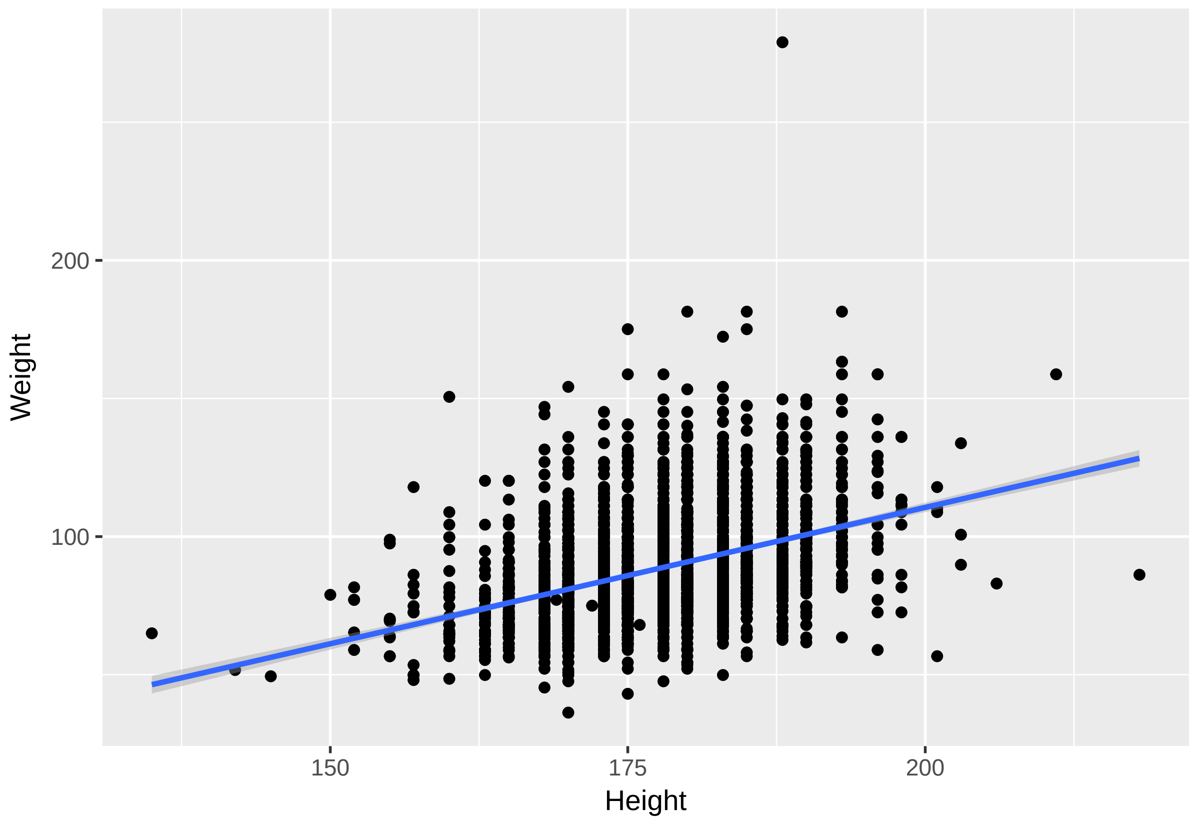
```r
library(ggplot2)
```

- http://docs.ggplot2.org

'Grammar of graphics'

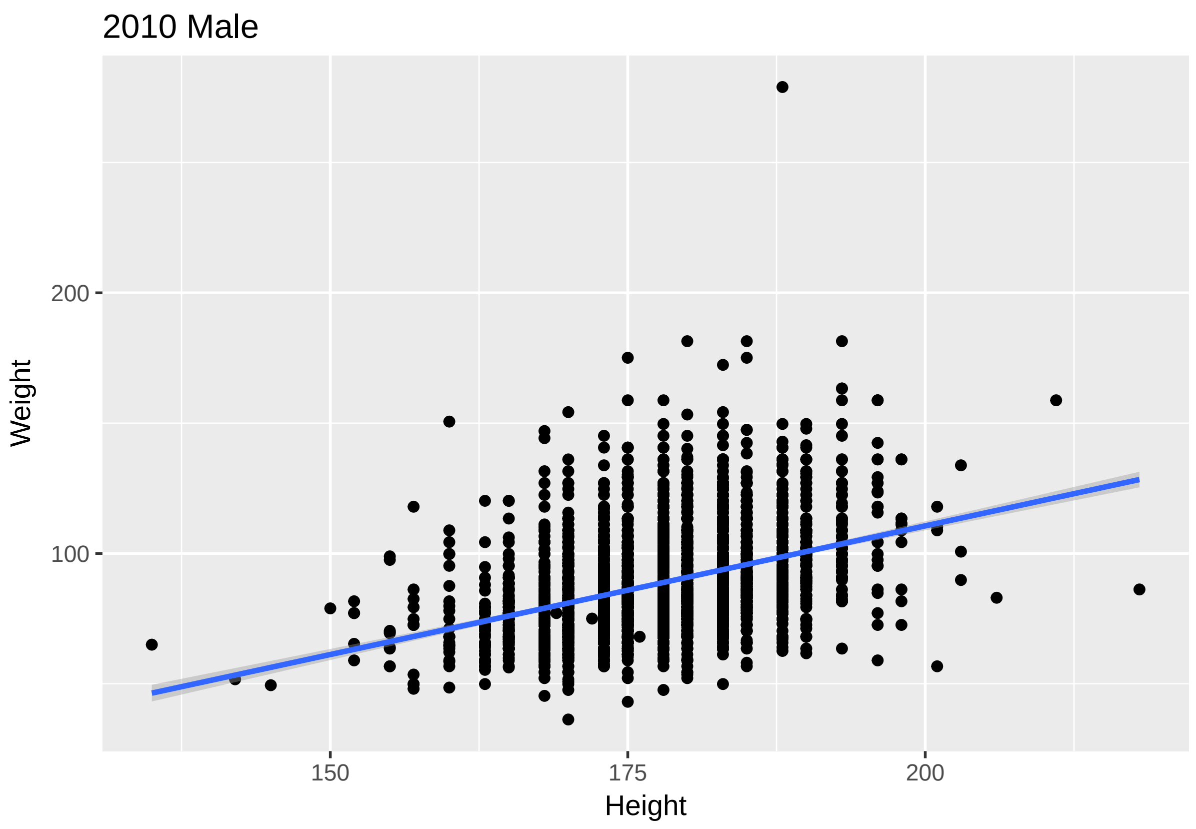
- Specify data and 'aesthetics' (`aes()`) to be plotted
- Add layers (`geom_*()`) of information

  ```r
  ggplot(brfss2010Male, aes(x=Height, y=Weight)) +
      geom_point() +
      geom_smooth(method="lm")
  ```



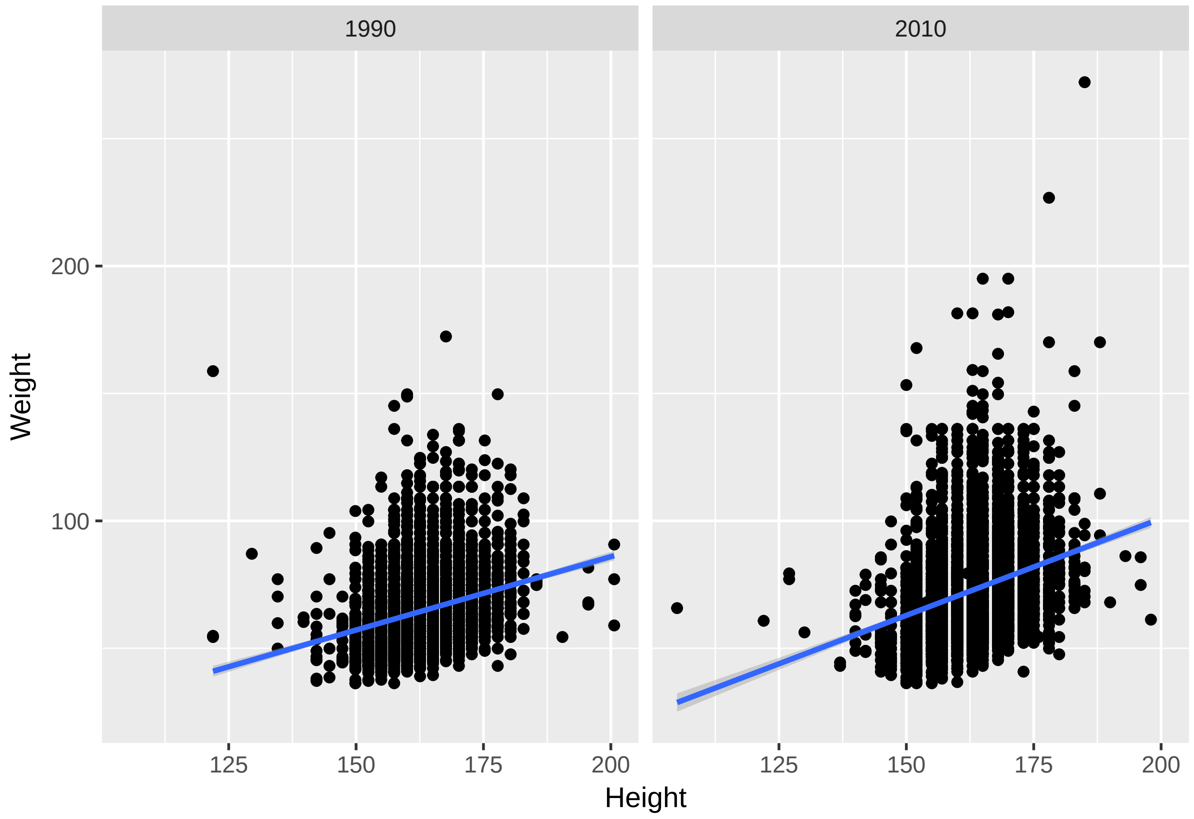
- Capture a plot and augment it

```
plt <- ggplot(brfss2010Male, aes(x=Height, y=Weight)) +
    geom_point() +
    geom_smooth(method="lm")
plt + labs(title = "2010 Male")
```



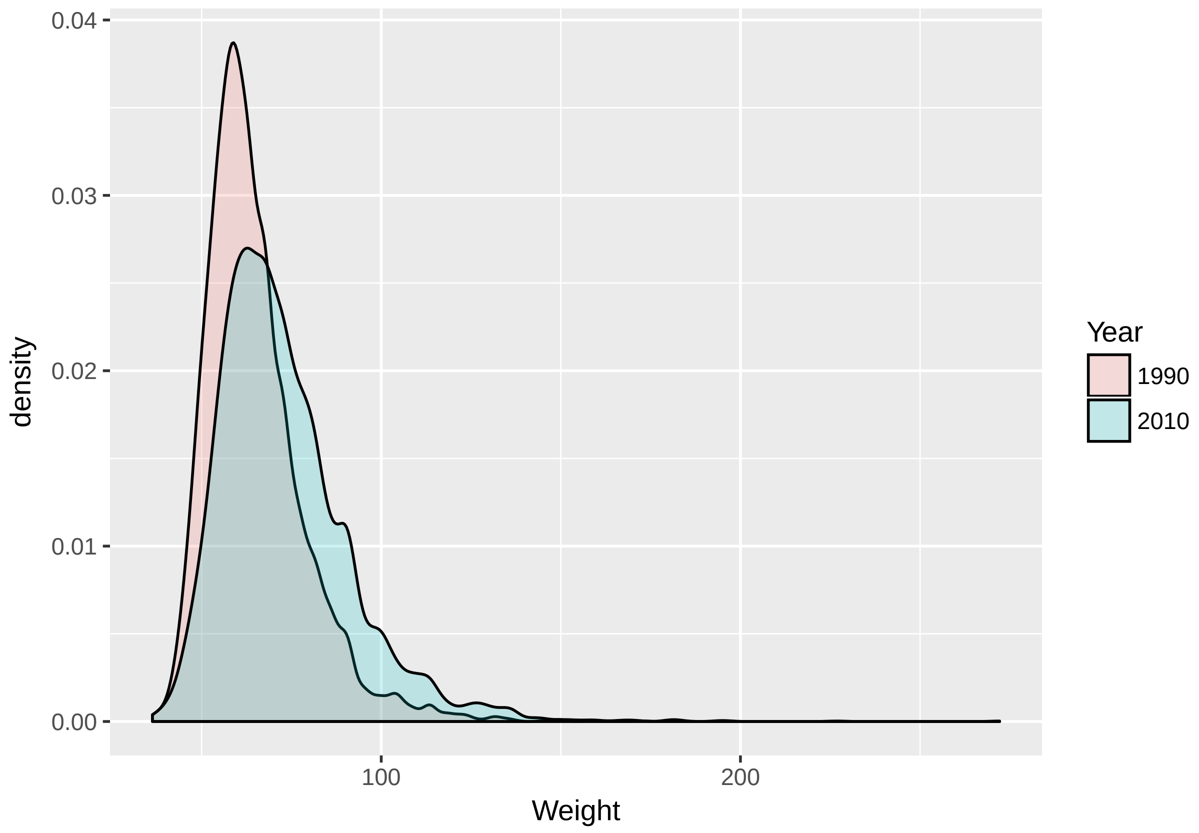
- Use `facet_*()` for layouts

```
ggplot(brfssFemale, aes(x=Height, y=Weight)) +
    geom_point() + geom_smooth(method="lm") +
    facet_grid(. ~ Year)
```



- Choose display to emphasize relevant aspects of data

```
ggplot(brfssFemale, aes(Weight, fill=Year)) +
    geom_density(alpha=.2)
```



lsls— author: "Sean Davis" date: "6/29/2017" output: html_document: default pdf_document: default —

## A Appendix A – Swirl

The following is from the swirl website.

> The swirl R package makes it fun and easy to learn R programming and data science. If you are new to R, have no fear.

To get started, we need to install a new package into R.

```
install.packages('swirl')
```

Once installed, we want to load it into the R workspace so we can use it.

```
library('swirl')
```

Finally, to get going, start swirl and follow the instructions.

```
swirl()
```