R Programming

Coursera Course by John Hopkins University

INSTRUCTORS: Dr. Jeff Leek, Dr. Roger D. Peng, Dr. Brian Caffo

Contents

Overview of R, R data types and objects, reading and writing data

Installing R & RStudio

- This was covered in the previous course.

R-Markdown reference site

- I found a site that expands on some features of R-Markdown and have been referencing it pretty regularly

Swirl

- swirl teaches you R programming and data science interactively, at your own pace, and right in the R console.

- Start swirl
  - install the package "swirl" if you haven't yet
  - Every time you want to run swirl execute:
    * library("swirl")
    * swirl()
  - You'll then be prompted to install a course
  - Help page for swirl

History of S and R programming

- What is S?
  - R is a dialect of S
  - S was developed by John Chambers and others at Bell Labs

  - Initiated in 1976 as an internal statistical analysis environment, implemented as FORTRAN libraries
    * Early versions did not contain functions for statistical modeling
  - Version 3 was released in 1988, which was rewritten in C and began to resemble the system that we have today.
  - Version 4 was released in 1998 and is the version we use today.
    * This version is documented in *Programming with Data* by John Chambers (the green book)
  - Insightful sells its implementation of the S language under the name *S-PLUS*, which includes a number of fancy features, mostly GUIs.
  - S won the Association for Computing Machinery's Software System Award in '98
  - (More about S)[https://web.archive.org/web/20181014111802/ect.bell-labs.com/sl/S/]
- What is R?
  - R was developed by Ross Ihaka and Robert Gentleman, they documented their experience in a (1996 JCGS paper)[https://amstat.tandfonline.com/doi/abs/10.1080/10618600.1996.10474713].
  - In 1995, R become free software after Martin Machler convinced Ross & Robert to use the GNU (General Public License)
  - Versions
    * R version 1.0.0 was released in 2000
    * R version 3.0.2 is released in Dec. 2013
  - Syntax is similar to S, making it easy for S-PLUS users to switch over
  - Runs on almost any standard computing platform/OS (even on the PS3)
  - Frequent releases; active development and communities
  - Functionality is divided into modular packages as to keep it "lean"
  - It's free!
  - What is free about Free Software?
    * Freedom 0: freedom to run the program, for any purpose
    * Freedom 1: freedom to study how the program works, and adapt it to one's needs. Which implies access to the source code
    * Freedom 2: freedom to redistribute copies
    * Freedom 3: freedom to improve the program, and release your improvements to the public, or to sell them.
    * These are outlined by the (Free Software Foundation)[https://www.fsf.org/]
- Drawbacks of R

  - Essentially based on 40 year old technology,the original S language
  - Little build support for dynamic or 3D graphics. Although there are packages for such
  - Functionality is based on consumer demand and use contributions, if a feature is not present you'll have to build it.

  - Objects that are manipulated in R have to be stored in the physical memory of the computer, as such if an object is bigger than the memory you'll be unable to load it into memory

  - Not ideal for all possible situations, such as calling to order pizza (but this is a drawback of all software packages)

*Design of the R System
+ "base" R system that can be downloaded from (CRAN)[http://cran.r-project.org] (krey-an) which...
- contains the packages: **utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4**.
- and "Recommends" the packages: **boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix**.
+ Packages are available all around the web, but packages on CRAN have to meet a certain level of quality.

- Some Useful Books on S/R
  - Chambers (2008). *Software for Data Analysis*, Springer.

  - Chambers (1998). *Programming with Data*, Springer.

  - Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.

  - Venables & Ripley (2000). *S Programming*, Springer.

  - Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-Plus*, Springer.

  - Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.

  - (Additional Books)[http://www.r-project.org/doc/bib/R-books.html]

Review of getting help

- Covered in previous course

Input and Evaluation: Vocabulary/Syntax

- **Expressions -** The code that is typed into the R prompt.

- **Assignment Operator -** assigns a value to a symbol, Ex:
  `x <- 1`

- *Output a variable*:

```
x <- 36
print(x) ##explicit printing
```

```
## [1] 36
```

```
## or one can just type the variable
x ##auto-printing
```

```
## [1] 36
```

- *Comment:* Use a Hash(#) symbol to make a comment to the right of #

- *[1]* is indicating the following variable is the first element of the vector

```
x <- 1:30 ##Loads x with the numbers 1 to 30
print(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

```
## here, [26] is telling you the next number is the 26th element of the vector
```

- `Inf` - represents infinity and can be used in ordinary calculations (Ex: 1 / `Inf` is 0)

- `Nan` - represents an undefined value ("not a number") (Ex: 0/0 is `NaN`).
  - Can also be thought of as a missing value

- **Attributes -** Some objects in R come with attributes. These attributes can be set or modified with the expression `attributes()`. They are:
  - names, dimnames (dimension names)
  - dimensions (e.g. matrices, arrays) - number of rows & cols, or more depending on dimensions of array
  - class - the data type of the object
  - length - number of elements
  - other user-defined attributes/metadata can be added

- **Coercion -** occurs so that every element of a vector is of the same class (Covered further in Vector section)

Different atomic data types

- R has five basic, or "atomic", classes of objects:
  - character
    - In `R` there is no `string` data type. It is also considered part of the `character` data type
  - numeric (real numbers)
    - R thinks as numbers as these by default

  - integer
    * Must be explicitly declared with the L suffix; `x <- 1` assigns a numeric object, but `x <- 1L` explicitly assigns an integer
  - complex
  - logical (True/False)
- A vector can only contain objects of the same class
  - an empty vector can be created with `vector()`
- However, a **list** is represented as a vector but can contain objects of different classes (as such we usually use these)

Vectors, Lists, and Matrices

- The `c()` function (can be thought to stand for "concatenate")
  - Can be used to create vectors of objects

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

- The `vector()` function
  - Can also be used to create, you guessed it, vectors

```
x <- vector() ## Creates an empty vector
x ## Prints as code that evaluates as FALSE
```

```
## logical(0)
```

```
x <- vector(mode = "numeric", length = 10)
## Creates a vector with length "10" of numeric data type,default value is 0
x
```

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

```
x <- vector("numeric", 5)
##The parameter names are not required, but can easily clarify code
x
```

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

- When different objects are mixed in a vector, **coercion** occurs so all objects are of the same class.
  - R will implicitly create the "Least Common Denominator" of the mixed classes

```
y <- c(1.7, "a") ## character
y
```

```
## [1] "1.7" "a"
```

```
y <- c(TRUE, 2) ## numeric
y
```

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

```
y <- c("a", TRUE) ## character
y
```

```
## [1] "a"    "TRUE"
```

```
y[2] ## "TRUE" is a string stored as a "character" data type
```

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

```
y[3] ## The third element does not exist
```

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

- Objects can be **explicitly coerced** from one class to another using the `as.*` functions, if available.
  - Nonsensical coercion results in `NA`s

```
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```
as.logical(x)
```

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

```
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

```
x
```

```
## [1] 0 1 2 3 4 5 6
```

```
y <- as.character(x)
y
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```
x <- c("a", "b", "c")
as.numeric(x) ##Nonsensical coercion will also show a warning
```

```
## Warning: NAs introduced by coercion
```

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

```
as.logical(x)
```

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

```
as.complex(x)
```

```
## Warning: NAs introduced by coercion
```

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

- Lists (Important data type in R that you should get to know well)
  - Lists are a type of vector that can contain elements of different classes.
  - Doesn't print like a vector because every element is different
    - prints index of element with double brackets bordering it: `[[1]]`

```
x <- list(1, "a", TRUE, 1 + 4i, 16 +18i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
##
## [[5]]
## [1] 16+18i
```

- **Matrices -** a type of vector with a *dimension* attribute.
  - The *dimension* attribute is itself an integer vector of length 2 (numRows, numCols)
  - Constructed *column-wise*, so entries can be thought of starting in the "upper left" corner, then running down the columns
  - Matrices can also be created by adding a *dimension* attribute to an existing vector

```
m <- matrix(nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

```
dim(m)##reports num of rows then cols
```

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

```
attributes(m) ## dim is an attribute of the vector
```

```
## $dim
## [1] 2 3
```

```
m <- matrix(1:6, 2, 3) ## Demonstrating column-wise filling of matrix
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
m <- 1:10 ## m is now just a vector
m
```

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

```
dim(m) <- c(2,5) ## adding the dimension attribute
m
```

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

- Creating a matrix with **cbind** and **rbind**
  - cbind fills the columns with the elements of the vectors that are passed as the respective parameters
  - likewise, rbind fills the rows with the elements of the respective parameters

```
x <- 1:3
y <- 10:12
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Other data types

- Factors
  - Used to represent categorical data
  - can be unordered or ordered
  - Kinda like enumerated data, where it's an integer at heart, and each integer has a *label*
  - Using factors with labels is *better* than using integers because factors are self-describing

        * consider "Male" and "Female" as opposed to just the values 1 and 2
    - Prints differently than a character value, does not include quotations and displays *Levels*

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

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

```
table(x)
```

```
## x
##  no yes
##   2   3
```

```
## displays a frequency table of the factors
unclass(x)
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

```
## strips out the class and displays the underlying integer vector
```

- The order of the levels can be set with the `levels` argument to `factor()`
    - This can be important in linear modelling because the first level is sued as the baseline level.
    - default levels are based alphabetically

```
x <- factor(
        c("yes", "yes", "no", "yes", "no"),
        levels = c("yes", "no")
        )
x
```

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

- Missing Values (`NA` or `NaN`)
    - `NaN` is for undefined mathematical operations
    - `is.na()` and `is.nan()` are logical tests for the respective missing values
    - `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
    - a `NaN` is also a `NA`, however the converse is not true

```
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

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

```
is.nan(x)
```

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

```
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

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

```
is.nan(x)
```

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

- Data Frames
  - Used to store tabular data
  - Special type of list where every element has to have the same length
  - Each element is like a column and the length of each element is the number of rows
  - like lists, Data Frames can store different classes in each column
  - Attribute: `row.names`
    - Useful for annotating data
    - However, often the row names are not interesting and we use "1, 2, 3..."
  - Usually created by calling `read.table()` or `read.csv()`
  - Can be converted to a matrix with `data.matrix()`
    - Forces each object to be coerced

```
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))## cols are named here
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

```
row.names(x)
```

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

- Names Attribute, useful for writing readable code and self-describing objects
  - Any R object can have names

```
x <- 1:3
names(x)## by default there are no names
```

```
## NULL
```

```
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

```r
##Lists can also have names
x <- list(a=1, b=2, c=3)
## here, names are assigned as list is established
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

```r
## Matrices can also have names, called dimnames
m <- matrix(1:4, nrow = 2, ncol = 2)
m
```

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

```r
dimnames(m) <- list(c("a", "b"), c("c", "d"))
#First vector is rownames, second is colnames
m
```

```
##   c d
## a 1 3
## b 2 4
```

Reading Data

Tabular Data

- Functions for **reading** data into R
  - `read.table`,`read.csv` - for reading tabular data
    - most common
    - reads in data that's organized into rows and cols
    - returns a data frame
  - `readLines`, for reading lines of a text file
  - `source`, for reading in R code files (inverse of `dump`)
  - `dget`, for reading in R code files (inverse of `dput`)
  - `load`, for reading in saved workspaces

  - `unserialize`, for reading single R objects in binary form

- Functions for **writing** data from R to files
  - `write.table`
  - `writeLines`
  - `dump`
  - `dput`
  - `save`
  - `serialize`

- Arguments of `read.table` function
  - `file` - the name of a file or connection
  - `header` - logical that indicates if the file has a header line
  - `sep` - a string that indicates how the columns are separated (tokens)
  - `colClasses` - a character vector that indicates the class (Data type) of each column
  - `nrows`
  - `comment.char` - character string that indicates the comment character (default is '#')
  - `skip` - number of lines to skip from the beginning
  - `stringsAsFactors` - (default = TRUE) should character variables be coded as factors?

- Implicit actions `R` takes

```
data <- read.table("foo.txt")
## Header must not have a label for the row labels for R to implicitly determine them
data
```

```
##             Price Num_Sold In_Stock Complex_Num
## Chips&Salsa  2.55     1729     TRUE       1+ 2i
## Drink        1.99     3435     TRUE       5+18i
## Taco         3.49       36    FALSE       3+ 0i
```

- Skips lines that begin with a #
- figures out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.
  - Telling R all these things directly will make it run faster and more efficiently
- `read.csv` is identical to `read.table` except that the default separator is a comma
  - `.csv` files are common output from excel or other spreadsheet programs.

Large Data-sets

- Doing the following things will make your life easier and prevent R from "choking"
  - Read the help page for read.table, which contains many hints
  - Make a rough calculation of the memory required to store your data-set.
    - Say for example, you have a data frame with 1,500,000 rows and 120 columns (not *that* big), all of which are numeric data. To roughly calculate how much memory is required..
    - 1,500,000 * 120 * 8 bytes/numeric = 1440000000 bytes
    - 1440000000 bytes / 2^{20} bytes/MB = 1,373.29 MB
    - 1,373.29 MB = 1.37 GB
    - Rule of thumb is that you'll need twice the amount of RAM to be able to read in the data-set
  - If the data-set is larger than the amount of RAM on your computer you can probably stop right here.
    - Type `free -k` in terminal to return amount of RAM in kilobytes (`-b` for bytes, `-m` for megabytes and `-g` for gigabytes)
  - Set `comment.char = ""` if there are no commented lines in your file.
  - Use the `colClasses` argument.
    - Specifying this option instead of using the default can make `read.table` run *MUCH* faster.
    - To use this option you have to know the class of each column in your data frame.
    - If all of the columns are of the same data type, for example "numeric", then you can just set `colClasses = "numeric"`
    - A quick and dirty way to figure out the classes of each column is to take a small sample and determine it from that.

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

```
##     Index   Boolean
## "integer" "integer"
```

```
##Coded file wrong, I'm not gonna fix it now, Boolean should have said "TRUE" or "FALSE"
tabAll <- read.table("datatable.txt",
                     colClasses = classes)
```

- Set `nrows`
  - This doesn't make R run faster but it helps with memory usage.
  - A mild overestimate is okay.
  - You can type `wc <filename>` in terminal to return the number of: lines, strings, characters; "lines" are the `nrows`.

Useful things to know about your system when using R with larger data-sets

- How much memory is available
  - Type `free -k` into terminal

- What other applications are in use
  - Type `ps aux` in terminal

- Are there other users logged into the same system
  - Type `w` in terminal (Note: `last` will report a history)

- What OS are you using
  - Type `lsb_release -a` into terminal

- Is the OS 32 or 64 bit
  - Type `lscpu`, listed under first two returns
  - On a 64 bit system you'll generally be able to access more memory

Textual Formats

- Contains the metadata, such as classes of columns, making transferring data more efficient as the metadata doesn't need to be determined again.

- Known as `dumping` and `dputing`.
- Edit-able, which in the case of corruption allows for a potential recovery.
- Textual formats can work much better with version control programs.
- Adhere to the "Unix philosophy", which is to store data as text
- *Downside:* The format is not very space-efficient and as such usually requires compression

- `dput` will deparse an R object, and `dget` can read the data back in from a file

```
y <- data.frame(a=1, b="a")
dput(y) ## If file is not specified the output is displayed in the console
```

```
## structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), class = "data.fra
## -1L))
```

```
dput(y, file = "y.R")
new.y <- dget("y.R")##dget retrieves the object from a file
new.y
```

```
##   a b
## 1 1 a
```

- Multiple objects can be deparsed using the `dump` function, then read back in with `source`
  - The parameter for `dump` is a character vector that contains characters for the names of the variables one wishes to dump

```r
x <- "foo"
y <- data.frame(a=1, b="a")
dump(c("x", "y"))
dump(c("x", "y"), file = "data.R")
rm(x, y) ## removes the variables
source("data.R") ## reconstructs y and x objects
y
```

```
##   a b
## 1 1 a
```

```r
x
```

```
## [1] "foo"
```

Connections (Interfaces to the outside world)

- Connections can be made to files or to other, more "exotic" things.
  - `file` - opens a connection to a file
  - `gzfile` - opens a connection to a file compressed with *gzip*.
  - `bzfile` - opens a connection to a file compressed with *bzip2*.
  - `url` - opens a connection to a webpage (in HTML format).
- Arguments
  - `description` is the name of the file
  - `open` indicates how the file is opened
    - **"r" -** read only
    - **"w" -** writing (and initializing a new file)
    - **"a" -** appending
    - **"rb", "wb", "ab" -** reading, writing, or appending in binary mode (Windows)
    - There are other options but they aren't uber important
- Connections are powerful tools that allow you to navigate files or other external objects in a more "sophisticated" way.
  - However, one does not need to deal with the connection interface in many case

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

- ^This is the same as..

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

- As such, the connection was not necessary for this case
- Reading lines of a text file with `con` from a *gzip* file

```
con <- gzfile("words.gz")
x <- readLines(con, 10) ##reads in first 10 lines
x
```

```
##  [1] "1080"     "10-point" "10th"     "11-point" "12-point" "16-point"
##  [7] "18-point" "1st"      "2"        "20-point"
```

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

- `readLines` can be used for reading in lines of webpages.

```
## This might take time
con <- url("http://www.jhsph.edu", "r") ##John Hopkin's School of Public Health
x <- readLines(con)
head(x) ##Displays the header
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
## [3] ""
## [4] "<head>"
## [5] "<meta charset=\"utf-8\" />"
## [6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

Subsetting R objects using the "[", "[[", and "$" operators and logical vectors

Basics

- Operators to extract subsets of R objects
  - `[` always returns an object of the same class as the original
    - subsetting a vector will return a vector, a list will return a list, etc.
    - Can be used to select more than one element (there is one exception, when subsetting a single element from a matrix)
  - `[[` is used to extract elements of a *list* or *data frame*
    - Can only be used to extract a single element
    - The class of the returned object will not necessarily be a list or data frame
  - `$` is used to extract elements of a *list* or *data frame* by name
    - Similar to `[[` as it may not be of the same class
- Numerical Index for subsetting:

```
x <- c("a", "b", "c", "c", "d", "a")
x[1] ## Returns first element
```

```
## [1] "a"
```

```r
x[2] ## Returns second element
```

```
## [1] "b"
```

```r
x[1:4] ## Returns first to fourth elements
```

```
## [1] "a" "b" "c" "c"
```

```r
x[c(2, 5)] ##Returns 2nd and 5th element
```

```
## [1] "b" "d"
```

```r
x[c(-2, -5)]##Returns everything EXCEPT the 2nd and 5th element
```

```
## [1] "a" "c" "c" "a"
```

```r
x[-c(2,5)]##Equivelent since the negative will multiply with every element of c(...)
```

```
## [1] "a" "c" "c" "a"
```

```r
x[2*c(1,3)]##Just like how this will actually be the 2nd and 6th element
```

```
## [1] "b" "a"
```

- Logical Index for subsetting:

```r
x <- c("a", "b", "c", "c", "d", "a")
x[x > "a"]## returns all elements that are greater than "a"
```

```
## [1] "b" "c" "c" "d"
```

```r
u <- x > "a"
## u is a logical vector that indicates which elements of x are greater than "a"
u
```

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

```r
x[u]
```

```
## [1] "b" "c" "c" "d"
```

```r
## subsets all elements of x such that u reports that index as TRUE;
##elements that are > "a"
```

Lists

- Lists can be subsetted with the `[[` or `$` operators

```r
x <- list(foo = 1:4, bar = 0.6)
x[1]
```

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

```r
##Extracts the first element as a list, since the orginal set was a list class
x[[1]]##Extracts the first element as a sequence, not a list
```

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

```
x$bar ##returns the element that is associated with the name "bar"
```

```
## [1] 0.6
```

```
x[["bar"]]##same as x$bar
```

```
## [1] 0.6
```

```
x["bar"]##returns a list with the element "bar" in it
```

```
## $bar
## [1] 0.6
```

- subsetting with the name is helpful when the index isn't known
- To extract multiple elements of a list, one must use the single bracket operator `[`

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]##extracts the first and third element of the list
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

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

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]] ## computed index for 'foo'
```

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

```
x$name ## element 'name' doesn't exist!
```

```
## NULL
```

```
x$foo ## element 'foo' does exist
```

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

- The `[[` can also take an integer sequence instead of a single number

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

```
## [1] 14
```

```
##extracts first element, then the third element of said first element
x[[1]][[3]] ##equivelent
```

```
## [1] 14
```

```
x[[c(2,1)]]##extracts first element of the second element of x
```

```
## [1] 3.14
```

Matrices

- Subsetted as one would expect with (i,j) type indices.

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

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
x[1,2] ##First row, second column
```

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

```
x[2,1] ##Second row, first column
```

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

- Indices can also be missing

```
x[1,] ##Returns first row
```

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

```
x[,2] ##Returns second column
```

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

- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix.
  - This is the exception of the `[` operator always returning the same class
  - This behavior can be turned off with the setting `drop = FALSE`.

```
x <- matrix(1:6, 2, 3)
x[1,2] ##returns vector
```

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

```
x[1,2, drop = FALSE] ##returns a 1x1 matrix
```

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

- This transition of classes also holds when subsetting a single column or row

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

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

```
x[1, , drop = FALSE]
```

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

```
## the second parameter still has to be blank so the row is returned
```

Partial Matching

- Allows one to not type out the full name of an element
  - Works with the [[and $ operators

```
x <- list(aardvark = 1:5, baking = 1:10)
x$a
```

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

```
##$ looks for a name that matches the "a", since aardvark
##starts with an "a" that is returned
x[["a"]]
```

```
## NULL
```

```
x[["a",exact = FALSE]]
```

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

```
## exact parameter has to be set to
##false for the [[ to accept a partial match

y <- list(aardvark = 1:5, apples = 1:3)
y$a
```

```
## NULL
```

```
y[["a", exact = FALSE]]
```

```
## NULL
```

```
## Since there are two names that start with "a" the intended
##element cannot be determined and NULL is returned
```

Removing missing (NA) values from a vector

- A common operation that needs to be done IRL data

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x) ## Creates a logical vector that is TRUE if the
##element is missing, and FLASE if the element is not missing
x[!bad]##Logical is negated to get all the valid elements
```

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

- In the case of multiple things you want to take the subset of with no missing values

```
x <- c(1, 2, NA, 4, NA, 5, NA, 7)
y <- c("a", "b", NA, "d", NA, "f", "g", NA)
good <- complete.cases(x,y)
##Indicates which elements of either vectors are missing
good
```

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

```
##As such, final two elements print FLASE since there is an NA
##in at least one element
x[good]
```

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

```
y[good]
```

```
## [1] "a" "b" "d" "f"
```

```
airquality[1:6, ]## Returns first 6 rows
```

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

```
good <- complete.cases(airquality)
airquality[good, ][1:6, ]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

```
##Returns first 6 rows that have don't have any missing values
```

- Additional note from `swirl()`

```
my_data <- sample(c(rnorm(100), rep(NA,100)), 20)
my_data
```

```
##  [1]         NA  1.1097613  0.9573372         NA  1.5862883         NA
##  [7] -0.6698626  1.1811570         NA         NA         NA  0.5207531
## [13]         NA  0.4521231  0.4680843  0.6077465  1.1135797  0.4790361
## [19]  0.4889550 -0.5295879
```

```r
is.na(my_data)
```

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

```r
## Returns a vector of logicals that indicate what positions of my_data are NA
my_data == NA
```

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

```r
## Returns a vector of NAs because NA is a placeholder for a qty that's
##not available. Therefore the expression is incomplete and returns a
##vector of NAs the same length as my_data
```

Vectorized operations

- A feature of R that makes it easy to use on the command line
- Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```r
x <- 1:4; y <- 6:9
x + y ##Adds vectors by position of elements
```

```
## [1]  7  9 11 13
```

```r
x >=2 ##Returns a logical vector that indicates which vectors are > or = 2
```

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

```r
y == 8
```

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

```r
x * y ##Multiplies each element of x by the respective element of y
```

```
## [1]  6 14 24 36
```

```r
x / y ##Divides by element
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

- Vectorized Matrix Operations

```r
x <- matrix(1:4, 2, 2); y <- matrix(rep(10,4), 2, 2)
x * y ##element-wise multiplication
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```r
x / y
```

```
##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4
```

```
x %*% y ## true matrix multiplication
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```

Stuff for quiz

```
##4
x <- 4L
class(x)#int?
```

```
## [1] "integer"
```

```
##5
x <- c(4,TRUE)
class(x)
```

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

```
##6
x <- c(1,3,5)
y <- c(3,2,10)
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    3    5
## y    3    2   10
```

```
##8
x <- list(2, "a", "b", TRUE)
class(x[[1]])
```

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

```
x[[1]]
```

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

```
##9
x <- 1:4
y <- 2:3
x+y
```

```
## [1] 3 5 5 7
```

```
class(x+y)
```

```
## [1] "integer"
```

```
##10
x <- c(3, 5, 1, 10, 12, 6)
```

```r
ans <- c(0, 0, 0, 10, 12, 6)
x[x %in% 1:5] <- 0
x
```

```
## [1]  0  0  0 10 12  6
```

```r
data <- read.csv("hw1_data.csv")
##First 2 rows
data[1:2,]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
```

```r
##Num rows
nrow(data)
```

```
## [1] 153
```

```r
#Extract final 2 rows
data[(nrow(data)-1):(nrow(data)),]
```

```
##     Ozone Solar.R Wind Temp Month Day
## 152    18     131  8.0   76     9  29
## 153    20     223 11.5   68     9  30
```

```r
#Value of 47th row
data[47,]
```

```
##    Ozone Solar.R Wind Temp Month Day
## 47    21     191 14.9   77     6  16
```

```r
#Number of missing Ozone
bad <-is.na(data[,1])
sum(bad)
```

```
## [1] 37
```

```r
##Mean of ozone without NA
cleanData <- data[!bad,1]
mean(cleanData)##mean of Ozone, ignore NA
```

```
## [1] 42.12931
```

```r
##Find mean of Solar.R where Ozone values are > 31 & Temp values are >90
bigOzone <- (data[,1]>31)
bigTemp <-(data[,4]>90&!is.na(data[,4]))
sOnBig <- data[bigOzone&bigTemp, 2]
bad <- is.na(sOnBig)
cleanSolar <- sOnBig[!bad]
mean(cleanSolar)
```

```
## [1] 212.8
```

```r
##What is the mean of "Temp" when "Month" is equal to 6
wheresSix <- (data[,5]==6)
sixMtemp <- data[wheresSix, 4]
mean(sixMtemp)
```

```
## [1] 79.1
```

```r
##What was the maximum ozone value in month 5
wheresFive <- (data[,5]==5)
fivMonOz <- data[wheresFive, 1]
bad <- is.na(fivMonOz)
cleanFOzone <- fivMonOz[!bad]
max(cleanFOzone)
```

```
## [1] 115
```

Misc Vanilla Functions

`dir.create()` - Creates a directory in current working directory (found with `getwd()`)
`args()` - Returns possible arguments of parameter
`file.<arguments>`
+ exists - checks if parameter exists, returns logical + info - returns info about file; such as: size, if it is a directory, mode, mtime, ctime, atime, uid, gid, username, groupname. `dir.create` - allows to manipulate directories and file permissions * *Sequence of Numbers* + The ':' operator

```r
1:20
```

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

```r
pi:10 ##Increments by 1 until number is > the upper limit, 10
```

```
## [1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
```

```r
15:1 ##Decrementing is cool too
```

```
##  [1] 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- `seq()`

```r
seq(1,20) ##Equivelant to `1:20`
```

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

```r
seq(0,10,by=0.5)
```

```
##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
## [16]  7.5  8.0  8.5  9.0  9.5 10.0
```

```r
my_seq <- seq(5,10,length=30)##sets `by` so the inc is consistent
seq_along(my_seq)#Creates a seq from 1:length(my_seq)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30
```

- `rep()` - creates a vector of a repeated value

```
rep(0, times = 30)
```

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

```
rep(c(0,1,2),times=10)##One can also use a vector as the argument
```

```
##  [1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

```
rep(c(0,1,2),each = 10)##Makes 10 of each
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

- `paste()` - joins elements of a vector
  - `collapse` - argument that tells R what character to add inbetween each element.

```
paste(1:3,c("X", "Y", "Z"), sep ="")#paste can also combine vectors
```

```
## [1] "1X" "2Y" "3Z"
```

```
paste(1:8,c("X", "Y", "Z"), sep ="")#even of diferent length
```

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

- `rnorm()` - draws from a standard normal distribution, number drawn is determined by parameter

```
rnorm(10)
```

```
##  [1]  1.44854790  1.69650664 -0.17136833 -0.05352274  0.86032666  0.32594437
##  [7] -0.86259367  0.05360375 -1.11166115 -0.35939344
```

- `sample()` - takes a sample of the specified size from the elements of x; `replace` is a logical argument that can be included

```
sample(c(1:20),10)
```

```
##  [1]  8  2 11 20  7 19  1 10 18  9
```

```
#`sample(c(1:5),10)`
##^would cause an error since replace defaults to false and n is bigger than N
sample(c(1:5),10,replace=TRUE)
```

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

- `identical()` - logical return; TRUE if the two objects are **exactly** equal

- A note on assigning names

```
my_matrix <- matrix(1:20, 4, 5)##4x5 matrix
patients <- c("Bill", "Gina", "Kelly", "Sean")
cbind(patients, my_matrix)##NOughty!
```

```
##      patients
## [1,] "Bill"   "1" "5" "9"  "13" "17"
```

```
## [2,] "Gina"   "2" "6" "10" "14" "18"
## [3,] "Kelly"  "3" "7" "11" "15" "19"
## [4,] "Sean"   "4" "8" "12" "16" "20"
```

```
my_data <- data.frame(patients, my_matrix)
my_data #The drake meme
```

```
##   patients X1 X2 X3 X4 X5
## 1     Bill  1  5  9 13 17
## 2     Gina  2  6 10 14 18
## 3    Kelly  3  7 11 15 19
## 4     Sean  4  8 12 16 20
```

- `invisible(x)` would be used within a function to prevent implicit printing of a variable when the function is called from the cmd_line

Control structures, functions, scoping rules, dates and times

Learning Objectives

- Write an **if-else** expression
- Write a **for loop**, a **while loop**, and a **repeat loop**
- Define a function in R and specify its return value(see **Functions Part 1** and **Functions Part 2**)
- Describe **how R binds a value to a symbol via the search list**
- Define what **lexical scoping is** with respect to **how the value of free variables are resolved in R**
- Describe the difference between **lexical scoping and dynamic scoping rules**
- Convert a character string representing a date/time into an **R datetime object**.

Control Structures

- Control structures allow you to control the flow of execution of the program
  - `if, else`: testing a condition
  - `for`: execute a loop a fixed number of times
  - `while`:execute a loop *while* a condition is true
  - `repeat`: execute an infinite loop

  - `break`: break the execution of a loop
  - `next`: skip an iteration of a loop
  - `return`: exit a function

- Infinite loops should generally be avoided, even if they are theoretically correct
- For command-line interactive work, the *apply functions are more useful

if-else

- syntax can be just like `cpp`

```
x <- sample(1:6, 1)
if(x>3){
  y <- 10
} else {
  y <- 0
}
vals <- c(x,y)
vals
```

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

- but `R` also excepts other syntax

```
x <- sample(1:6, 1)
y <- if(x>3){
  10
} else {
  0
}
vals <- c(x,y)
vals
```

```
## [1]  4 10
```

- nested `if`s and "else-less" `if`s are also acceptable

for loop

- `for` loops take an iterator variable and assign it to successive values from a sequence or vector.
  - Common for iterating over the elements of an object

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

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

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

- R is flexible in how you can index different objects; the following loops are all equivalent

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

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

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for (i in seq_along(x)) {##seq_along creates an integer sequence that's as long as x
  print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(letter in x) {##letter is assigned to the nth element of x
  print(letter)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(i in 1:4) print(x[i])##{} can be omitted for a single element in the body
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

- Nested for loops are also acceptable

while loop

```
count <- 0
while(count < 10){
  print(count)
  count <- count + 1##Make sure you increment
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

*A while loop with logical operators

```
z <- 5
while(z>=3 & z <= 10){
  print(z)
  coin <- rbinom(1,1,0.5)
  if(coin == 1){ ##random walk
    z <- z+1
  } else {
      z <- z-1
    }
}
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] 4
## [1] 3
## [1] 4
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 6
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 5
```

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

- Conditionals will "**short circuit**" when evaluating `&` or `|`

Repeat, Next, Break

- Not common in statistical applications
- only way to exit a `repeat` loop is to call `break`

```
x0 <- 1
tol <- 1e-8
count <- 0

repeat{
  count <- count + 1
  x1 <- sample(seq(-1,1,length.out = 1000), 1)

  if(abs(x1 - x0) < tol){
    outVect <- c("In ", count," loops we found a x0, ", x0, ", that was within ", tol, " of x1
    output <- paste(outVect, collapse = "")
    print(output)
    break
  } else {
    x0 <- x1
  }
}
```

```
## [1] "In 91 loops we found a x0, -0.917917917917918, that was within 1e-08 of x1, -0.9179179
```

- loops that are not guaranteed to stop ought to have a hard limit on the number of iterations(e.g. using a for loop instead) and then report whether convergence was achieved or not

*`next` - used to skip an iteration of a loop

```
for(i in 1:100){
  if(i <= 20){
    ##Skip the first 20 iterations
    next
  }
  ## Other code could go here
}
```

Functions

Your first R function(s)

```
add2 <- function(x, y) {
  x + y
}

add2(3,5)
```

```
## [1] 8
```

```
above10 <- function(x){
  use <- x > 10
  x[use]
}

above <- function(x, n = 10){##n value is defaulted to 10
  use <- x > n
  x[use]
}

y <- 1:20
above10(y)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
above(y, 12)
```

```
## [1] 13 14 15 16 17 18 19 20
```

```
columnmean <- function(y, removeNA=TRUE) {
  nc <- ncol(y)
  means <- numeric(nc)##numeric vector, length same as value of nc
  for(i in 1:nc) {
    means[i] <-  mean(y[,i], na.rm = removeNA)##many functions have the option to remove NAs
  }
  means
}

columnmean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

```
columnmean(airquality, FALSE)
```

```
## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

Functions pt 1

- created using the `function()` directive

- stored as R objects, of the class "function".
- Function in R are "first class objects", as such..
  - they can be passed as arguments to other functions

  - they can be nested, so that you can define a function inside of another function.
- The return value of a function is the last expression in the function body to be evaluated

- Functions have **named arguments** which potentially have **default values**
  - The **formal arguments** are the arguments included in the function definition
  - The `formals` function returns a list of all the formal arguments of a function
  - Not every function call in R makes use of all the formal arguments, they can be missing or have a default set.

- R function arguments can be matched position-ally or by name.
  - So all the following calls to `sd` are equivalent

```
mydata <- rnorm(100)
sd(mydata)
```

```
## [1] 0.913191
```

```
sd(x = mydata)
```

```
## [1] 0.913191
```

```
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 0.913191
```

```
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 0.913191
```

```
sd(na.rm = FALSE, mydata)## when picking a position R fills to earliest, undeclared argument
```

```
## [1] 0.913191
```

- Messing around with the order of the arguments can lead to confusion, as such it's not recommended

- Positional matching and matching by name can be mixed, which is helpful when functions have many arguments and one doesn't need them all, Ex:

```
args(lm)
```

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

- The following two calls are equivalent
  - `lm(data = mydata, y - x, model = FALSE, subset = 1:100)`
  - `lm( y~x, mydata, 1:100, model = FALSE, x=TRUE)`
- Second option is ideal way to mix and match since first 3 are gonna be specified by position
- Function arguments can also be **partially** matched by the following Order of operations:
  1. Check for exact match for a named argument
  2. Check for a partial match
  3. Check for a positional match

Functions pt 2

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

```
f <-function(a, b = 1, c = 2, d = NULL) {
  ##<super_rad_code.txt>
}
```

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

```
f <- function(a,b) {
  a^2
}
f(2) ##b is not evaluated in the function, so an error does not occur
```

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

```
f <- function(a,b) {
  print(a)
  print(b)
}
tryCatch(f(45), error=function(e){print("Error in print(b) : argument \"b\" is missing, with no
```

```
## [1] 45
## [1] "Error in print(b) : argument \"b\" is missing, with no default"
```

```
  ##The function is evaluated until print(b) tries to execute
```

- The "..." Argument
  - ... indicate a variable number of arguments that are usually passed on to other functions.
  - Often used when extending another function and you don't want to copy the entire argument list of the og function

```
myplot <- function(x,y, type = "1", ...) {
  plot(x,y,type = type, ...)
}
```

- Also used with Generic function so that extra arguments can be passed to methods (more on this later)

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x562ebfb861f0>
## <environment: namespace:base>
```

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

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

```
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

- Arguments that appear **after** `...` on the argument list must be named explicitly and cannot be partially matched

```
paste( "a", "b", sep = ":")
```

```
## [1] "a:b"
```

```
paste( "a", "b", se = ":")##Attempt to partially match 'sep'
```

```
## [1] "a b :"
```

Scoping Rules

Symbol Binding

- How does R decide what value to assign to a symbol in it's body?

```
lm <- function(x) {x*x} ##lm is already a function in the 'stats' package
lm
```

```
## function(x) {x*x}
```

- R searches through different environments when attempting to bind a value to a symbol

1. The global environment - which is your workspace in the Environment pane

- iow, local variables are prioritized

2. Namepaces of each of the packages- in order of the search list

- `search()` displays the search list

```
search()##.GlobalEnv is always [1]
```

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

- *base* package is always last in the search list
- User's can configure which packages get loaded, as such one cannot assume a set list of packages will be available.
  - When a user loads a package with `library` the namespace gets put in `[2]` of the search (list) stack.

- Note that R has separate namespaces for functions and non-functions
  - So it's possible to have an object named `c` and a function named `c`

  - However only once symbol can be named `c` in your `.GlobalEnv`
- Scoping rules for R are the main feature that make it different from the original `S` language
  - R uses *lexical scoping* or *static scoping*. A common alternative is *dynamic scoping*
    * lexical scoping turns out to be particularly useful for simplifying statistical computations

  - These rules determine how a value is associated with a free variable in a function

Free Variables

- Consider the following,

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

- This function has 2 formal arguments `x` and `y`. The additional symbol, `z`, is called a *free variable*.

- A **free variable** is neither a formal argument nor a local variable

Lexical Scoping

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

- What is an environment

  - An *environment* is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.

  - Every environment has a parent environment; thus it is possible for an environment to have multiple "children"

  - the only environment without a parent is the *empty environment*
  - **a closure* or *function closure** - A function that is associated with an environment
    * Key to a lot of interesting operations in `R`

- Searching for the value for a free variable

  - Search starts in the environment in which a function was defined

  - Then the search is continued in the *parent environment*

  - The search continues down the sequence of parent environments until..
  - We hit the *top-level environment*; this is usually the global environment (workspace) or the namespace of a package.

  - After the top-level env. the search continues down the search list until..
  - We hit the *empty environment*, at which point an error is thrown.

- Other languages that support lexical scoping:

  - Scheme

  - Perl

  - Python

  - Common Lisp (**all languages converge to Lisp**)

Scoping Rules (sub)

- R's big sellin' point is that you can functions defined *inside other functions*
  - Languages like `C` don't let you do this

  - In this case, the environment in which a function is defined is the body of another function
  - As such, a function can return a function

```
make.power <-  function(n) {
  pow <-  function(x) {
    x^n
  }
  pow
}
## This function returns another function as is value
```

```
## Which allows us to create functions as follows
cube <- make.power(3)
square <- make.power(2)
cube(3)
```

```
## [1] 27
```

```
square(3)
```

```
## [1] 9
```

- How do you know what a function's environment is?

```
ls(environment(cube)) ##Returns objects within environment that a function was defined
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube)) ##Returns the value of "n" within the 'cube' environment
```

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

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(square))
```

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

- Example of ***dynamic scoping***

```
y <- 10

f <- function(x) {
  y <- 2##assign new value to 2 within function's scope
  y^2 + g(x) ##2^2 + g(x)
}

g <- function(x) {
  x*y ## computes x*10, regardless if y got reassigned elsewhere
}

f(3) ## Computes (2^2 + 2*10) due to dynamic scoping that is simulated here
```

```
## [1] 34
```

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

```
g <- function(x) {
  a <- 3
  x+a+y
}
```

```
y <- 3
g(2)
```

```
## [1] 8
```

- This occurs because GlobalEnv is always first in the search list, as such g is in the same environment as when we call it and `y` exists

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

Aside: Likelihood & Log-likelihood

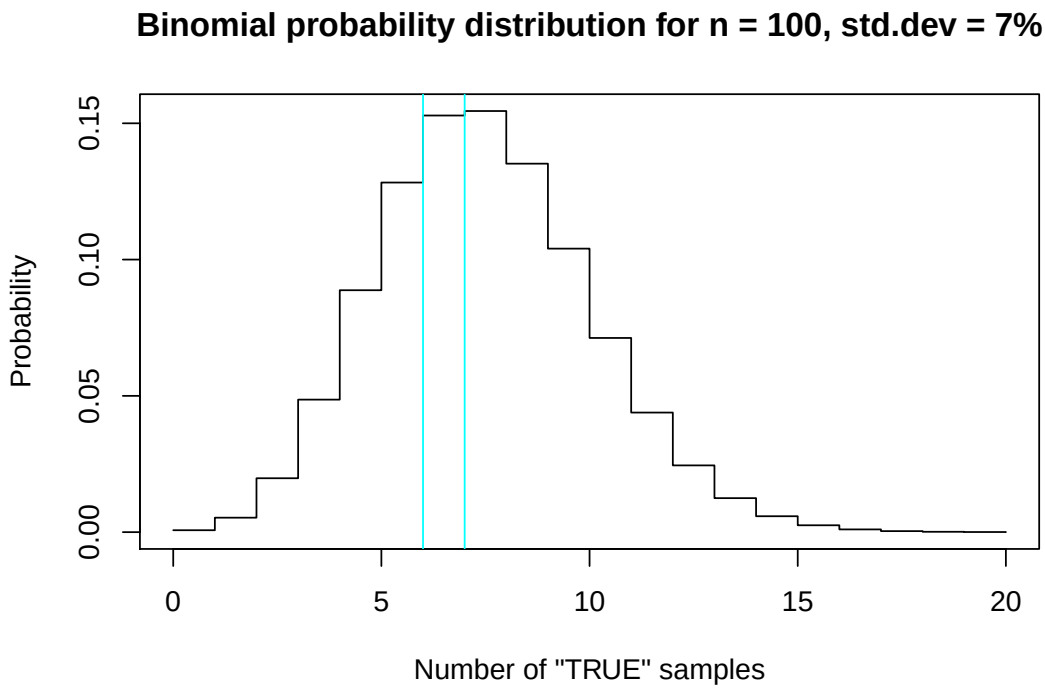
- **Notes are from this video**

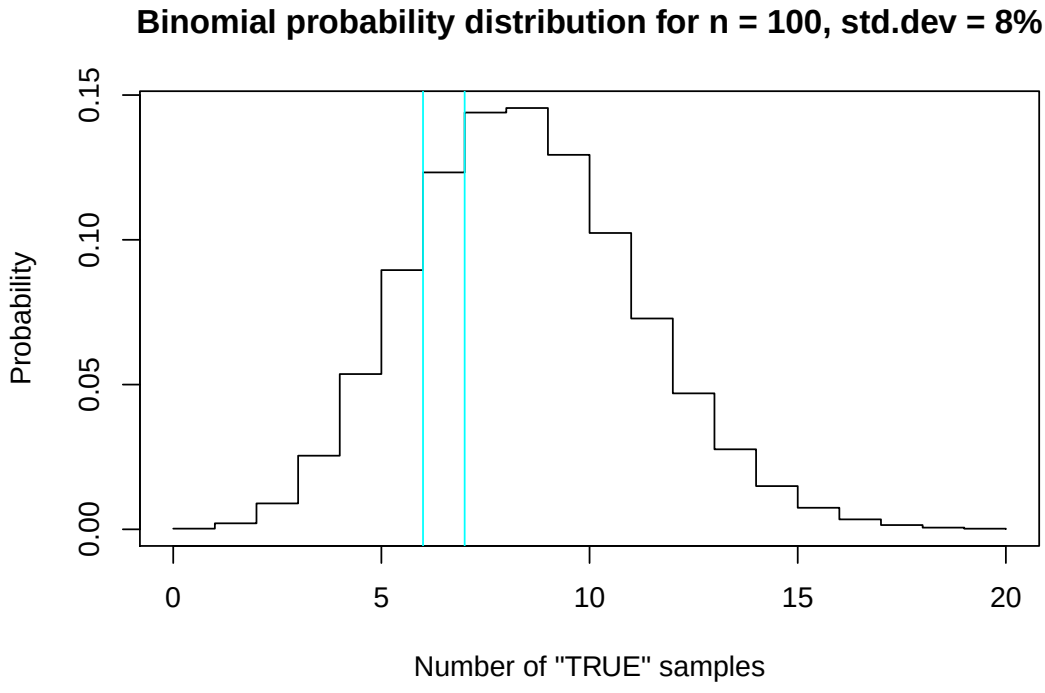
- Probability review
  - proportion of times a given outcome would occur in many trials.
    - Iow, It is the long-term relative frequency

  - P(weight between 32 and 34 grams | mean = 32 and std. deviation = 2.5)
  - Read as: "Probability of a weight between 32 and 34 grams, given that the mean is 32 and the standard deviation is 2.5"

- Likelihood
  - *likelihood* - describes the extent to which the sample provides support for any particular parameter value. Higher support corresponds to a higher value for the likelihood.

  - Likelihood describes probability of distribution factors, such as mean or std. deviation





– If a sample of 100 was taken and 6 people returned TRUE, which of the above models is more ***Likely***?

– Likelihood of either model is the probability at 6.

- A **likelihood function** provides a model for a fixed sample outcome.
- Individual likelihood values are *meaningless*
- Comparing two values, however, is *informative*
- As such we usually discuss a **Likelihood Ratio** - L(theta[1];y) / L(theta[2];y)
  * Suppose the following:

```
L_Of_Theta_0.07 <- 0.152
L_Of_Theta_0.08 <- 0.123
Liklihood_Ratio <- function(a, b) {
  round(a/b, digits = 3)
}
Liklihood_Ratio(L_Of_Theta_0.07, L_Of_Theta_0.08)
```

```
## [1] 1.236
```

```
## [1] "Thus, a population prevalence of 7% has 1.236 times the support of"
## [2] "a population prevalence of 8% (given our sample)"
```

- **likelihood function** - for a given sample, it creates the likelihoods for all possible values of theta
- In summary, in ***likelihood functions***, *the mean* or *the standard deviation* is the parameter

Optimization

- Optimization routines in R, like `optim`, `nlm`, and `optimize` require you to pass a function, whose argument is a vector of parameters (e.g. a **log-likelihood**)
- Optimization tries to find the *min* or *max* of a given function
  - An object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed
- *Note*: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p #Assigns p to whatever variable wasn't fixed
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a+b)
```

```
  }
}

set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals)
nLL
```

```
## function(p) {
##     params[!fixed] <- p #Assigns p to whatever variable wasn't fixed
##     mu <- params[1]
##     sigma <- params[2]
##     a <- -0.5*length(data)*log(2*pi*sigma^2)
##     b <- -0.5*sum((data-mu)^2) / (sigma^2)
##     -(a+b)
##   }
## <bytecode: 0x562ebddc8660>
## <environment: 0x562ec3243568>
```

```
#<environment: ...> tells you the address of the pointer to the defining environment
ls(environment(nLL))##Lists Values in function's environment
```

```
## [1] "data"   "fixed"  "params"
```

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

```
##       mu    sigma
## 1.218239 1.787343
```

```
##Returns estimates for mu&sigma by minimizing neg. liklihood

nLL <- make.NegLogLik(normals, c(FALSE, 2))##Fixing sigma to 2
optimize(nLL, c(-1, 3))$minimum
```

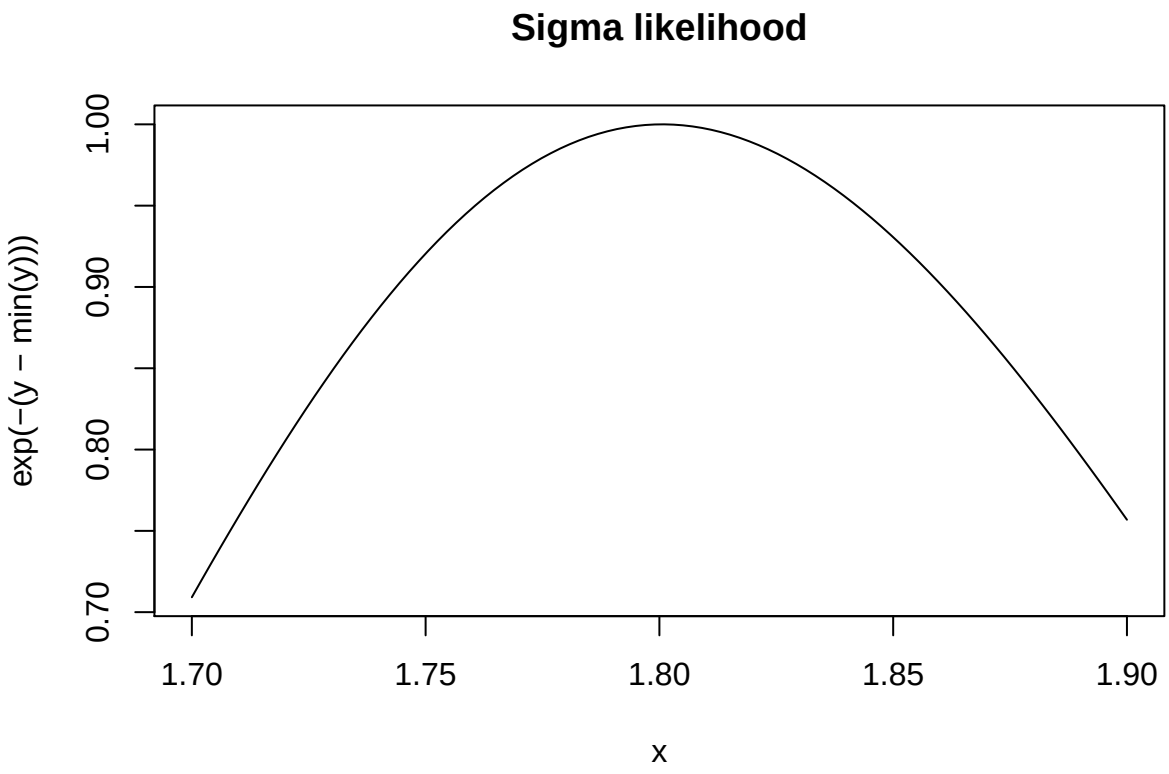
```
## [1] 1.217775
```

```
nLL <- make.NegLogLik(normals, c(1, FALSE))##Fixing mu to 1
optimize(nLL, c(1e-6, 10))$minimum
```

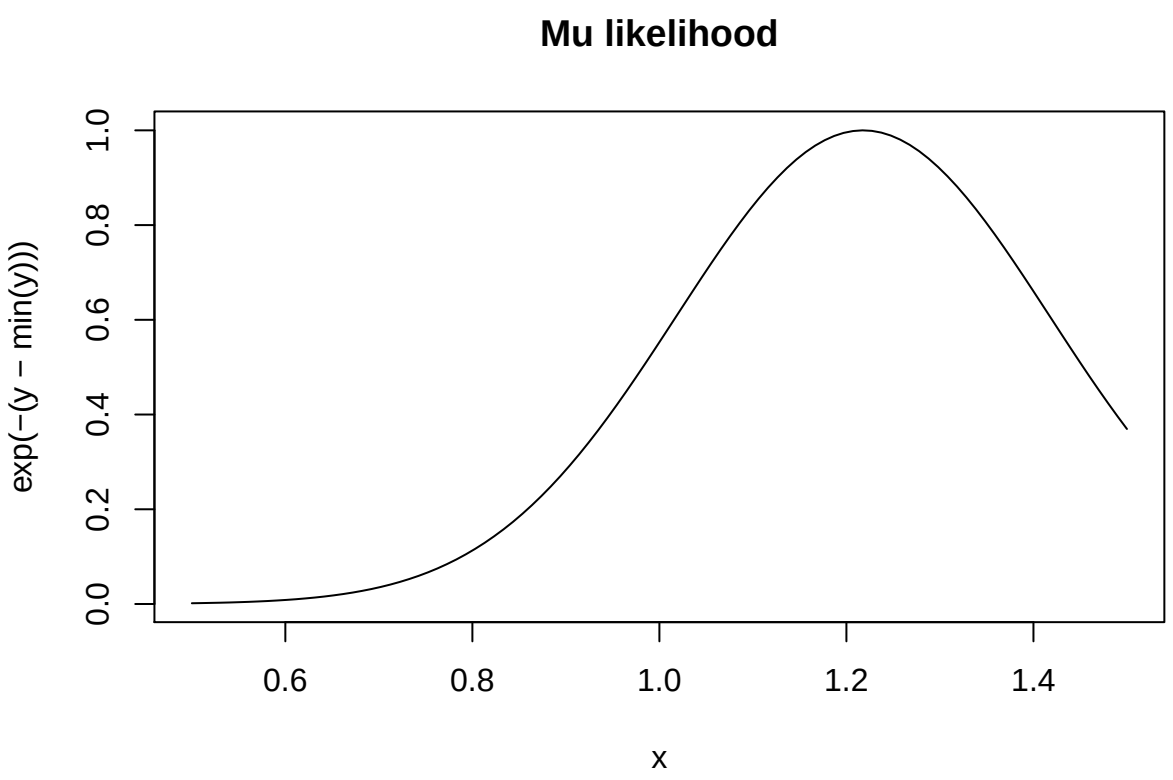
```
## [1] 1.800596
```

- `optimize` will only optimize a function with a single missing variable
  - `optim` can optimize a function with more than one missing variable

```
##The following plots the sigma likelihood
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len=100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l", main = "Sigma likelihood")
```




```
##The following plots the mu likelihood
nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l", main = "Mu likelihood")
```



- Objective function can be "built" which contain all of the necessary data for evaluating the function

- No need to carry around long argument lists
  - useful for interactive and exploratory work

- Code can be simplified and cleaned up

Coding Standards

- "Help make your code readable. . . just like it is with any other *style*, like your clothing, it's hard to get everyone to agree on one set of ideas. But there are some basic/minimal standards for coding `R`"

1. Code should always be written with a text editor and saved as `.txt`

- Can be read by any basic editing program; makes code versatile

- RStudio saves code as text by default

2. Indent your code

3. Limit the width of your code (80 columns)

- indents and column width can be edited in settings of RStudio
- these limits help promote clean code

4. Limit the length of individual functions

- Each function ought to only do one activity
  - `readTheData` should read and return the data, NOT read the data, process it, fit a model, and then print an output.
- Nice to read an entire function that is able to fit on one screen of the editor
- Helps when debugging

Dates and Times

- Dates are represented by the `Date` class
  - Stored internally as the number of days since 1970-01-01
  - Can be coerced from a character string using the `as.Date()` function
  - When printed they default to converting back to a readable character string
  - If `unclass`ed, they'll display the numeric value R is storing them as

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

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

```
unclass(as.Date("1970-01-02"))##One day since origin
```

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

- Times are represented by the `POSIXct` or the `POSIXlt` class
  - Stored internally as the number of seconds since 1970-01-01
    - Negative times are also valid
  - `POSIXct` is just a very large integer; useful when one wants to store times in something like a data frame
    - `c` for "concise"

```
x <- Sys.time()
x ## Already in 'POSIXct' format
```

```
## [1] "2020-02-20 06:09:32 EST"
```

```
unclass(x)
```

```
## [1] 1582196972
```

```
#x$sec
##Throws: "Error in x$sec : $ operator is invalid for atomic vectors"
p <- as.POSIXlt(x) ##After this conversion sec can be extracted
p$sec
```

```
## [1] 32.20062
```

```
str(p)
```

```
##  POSIXlt[1:1], format: "2020-02-20 06:09:32"
```

- `POSIXlt` is a list "underneath" and it stores meta data such as: the day of the week, day of the year, month, day of the month
  - `l` for "list"

```
x <- Sys.time()
x
```

```
## [1] "2020-02-20 06:09:32 EST"
```

```
p <- as.POSIXlt(x)
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```
p$sec
```

```
## [1] 32.22114
```

- `strptime` function - lets you convert dates written in different formats into `POSIXlt`
  - Check `?strptime` for details on the string formatting

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

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

```
class(x)
```

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

*Operations on Dates and Times + Some mathematical operations work (`+`, `-`, logicals (i.e. `==`, `<=`)) + You can't mix classes

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
#x-y ##Throws:
## Incompatible methods ("-.Date", "-.POSIXt") for "-"
## Error: non-numeric argument to binary operator
x <- as.POSIXlt(x)
x-y
```

```
## Time difference of 356.3095 days
```

- Some generic functions that work on dates and times
  - `weekdays`: returns the day of the week
  - `months`: returns the month name
  - `quarters`: returns the quarter number ("Q1", "Q2", "Q3", or "Q4")
- Dates and Times classes keep track of *leap years*, *leap seconds*, *daylight savings*, and *time zones

```
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
x-y
```

```
## Time difference of 2 days
```

```
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y-x
```

```
## Time difference of 1 hours
```

Datetime Object

- Summary
  - Dates and times have special classes in R that allow for numerical and statistical calculations
  - Dates use the `Date` class
  - Times use the `POSIXct` and `POSIXlt` class
  - Character strings can be coerced to date/Time classes using the `strptime`, `as.Date`, `as.POSIXlt`, or `as.POSIXct` functions.
  - A lot of plotting functions will recognize Datetime objects

Misc Vanilla Functions

- `rm(list=ls())` - clears everything from workspace
- `&&` operator will only evaluate the first element of a vector and return a single logical; whereas `&` will evaluate all elements and return a logical vector
  - likewise for the `||` and `|` operators
- All `&` operators are evaluated before `|` operators

- `xor()` evaluates arguments with *exclusive or*
- `which()` returns a vector that indicates which indices are `TRUE`
- `any()` returns true if any element is true in the logical vector passed as an argument
- `all()` returns true if all the elements in the logical vector passed as an argument are `TRUE`
- *Note*: John Chambers, the creator of R once said: "*To understand computations in R, two slogans are helpful:* 1. *Everything that exists is an object.* 2. *Everything that happens is a function call.*"
- This is a strict rule in R programming: all arguments after an ellipses must have default values.
- Let's say I wanted to define a binary operator that multiplied two numbers and then added one to the product. Notice the `%` and `"` surrounding the operator name. An implementation of that operator is below:

```
 "%mult_add_one%" <- function(left, right){ # Notice the quotation marks!
   left * right + 1
 }

4 %mult_add_one% 5
```

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

- `rstudioapi::convertTheme(name)` will let you add a new theme

Testing out additional thangs

```
##2
x <- 1:10
#if(x > 5) { #returns error because only a condition of length 1 can be evaluated in an 'if'
#  x <- 0
#}
```

```
##3
f <- function(x) {
  g <- function(y) {
    y + z
  }
  z <- 4
  x + g(x)
}
```

```
z <- 10
f(3)
```

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

```
##E03
```

Loop functions, debugging tools

Learning Objectives

- Define **an anonymous function** and **describe its use in loop functions.**
- Describe **how to start the R debugger** for an arbitrary R function.
- Describe what **the traceback() function does** and what is **the function stack call**

Loop Functions

- Loop Functions allow you to execute a loop over an object or set of objects and keep compact, as to work on the command line; these functions then to have `apply` in them.

- Overview
  –

lapply

- Loop over a list and evaluate a function on each element

- Takes three arguments: 1) a list `x` (Or it'll be coerced to a list using `as.list`)

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x562ebd8e59a0>
## <environment: namespace:base>
```

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

```
x <- list(a=1:5, b= rnorm(10))
lapply(x,mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.3474802
```

- Each element of the list is a numeric vector and what returns is a vector with a single element

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

```
## $a
## [1] 2.5
##
## $b
## [1] -0.1198232
##
## $c
## [1] 0.782726
##
## $d
## [1] 4.964196
```

- Here `runif` is called, which creates a uniformed list of uniform random variables. The first argument determines how many variables to return (`runif(2)` will return a vector of length 2)

```
x <- 1:4
lapply(x, runif)
```

```
## [[1]]
## [1] 0.7603133
##
## [[2]]
## [1] 0.1554012 0.8494571
##
## [[3]]
## [1] 0.9468178 0.5884192 0.5022508
##
## [[4]]
## [1] 0.189779918 0.001836858 0.877578062 0.134111338
```

```
##Now specify the defaults of runif with lapply
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 0.2274122
##
## [[2]]
## [1] 9.391367 2.929487
##
## [[3]]
## [1] 1.643266 3.991026 4.595754
##
## [[4]]
## [1] 4.3403085 5.1700983 8.4624575 0.5516429
```

###Anonymous Functions

* functions that aren't assigned a name, but are generated "on the fly"

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

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

```
##Use lapply and an Anon Function to extract the first column of each matrix
lapply(x, function(elt) elt[,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

sapply

- Variant of `lapply` that trys to simplify the result
  - If the result is a list where every element's length is 1, then a vector is returned
  - If the result is a list where every element is a vector of the same length (>1), a matrix is returned.
  - `else` case is to just return a list

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

```
## $a
## [1] 2.5
##
## $b
## [1] 0.01869495
##
## $c
## [1] 1.075839
##
## $d
## [1] 5.061851
```

```
sapply(x, mean)##Returns vector with 4 numbers
```

```
##          a          b          c          d
## 2.50000000 0.01869495 1.07583942 5.06185108
```

apply

- Apply a function (often an anon one)over the margins of an array
  - Occasionally you'll hear in the wil' that `apply` is better or faster than a for loop. This ain't true at all in R, but was in ol' versions of S
  - It is most often used to apply a function to the rows or columns of a matrix
  - Involves less typing
    - Less typing is always better because good programmers are alzy
- Arguments

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

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

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean)##Keep second dimension, columns
```

```
##  [1]  0.09021391  0.14723035 -0.22431309 -0.49657847  0.30095015  0.07703985
##  [7] -0.20818099  0.06809774 -0.16083776  0.03672642
```

```
apply(x, 1, sum)##Takes sum over all the rows
```

```
##  [1] -0.2658782  5.4142552 -1.7345767 -1.2493809  5.5245197  1.8382665
##  [7] -8.8367372 -0.2621256 -2.3469654 -2.6130278 -6.2576198  2.6881954
## [13] -2.2629859 -2.7610001  7.3186587 -3.2649161  2.8088937  1.2897966
## [19] -4.4829822  2.0625724
```

- For sums and means of matrix dimensions, we use the following shortcuts:
- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`

- `colMeans = apply(x, 2, mean)`

- These shortcut functions are *much faster*, but one won't notice unless they're using a large matrix.

```
x <- matrix(rnorm(200), 20, 10)
apply(x,1, quantile, probs = c(0.25, 0.75))
```

```
##          [,1]       [,2]        [,3]       [,4]       [,5]        [,6]
## 25% -1.209281 -1.0939584 -0.01333975 -0.1127928 -1.0432010 -1.29228106
## 75%  0.245412  0.8871649  1.50916561  0.6536448  0.1227056 -0.02762272
##           [,7]       [,8]       [,9]      [,10]      [,11]      [,12]
## 25% -1.4571706 -0.3780937 0.08859723 -0.6594755 -0.9903879 -0.7454235
## 75%  0.6964152  0.8198733 1.50727059  1.6560072  0.1385100  0.7853764
##          [,13]      [,14]      [,15]      [,16]      [,17]      [,18]
## 25% -0.9796050 -1.3551031 -1.0500304 -1.8845971 -0.6905687 -0.3767354
## 75%  0.6372398 -0.1741619  0.4519238 -0.4305929  0.8727966  0.5626554
##          [,19]      [,20]
## 25% -0.3517061 -0.5741184
## 75%  0.7766057  0.6684292
```

```
#Returns 25th and 7th percentile for each row of the matrix
```

- Average matrix in an array

```
a <- array(rnorm(2*2*10), c(2,2,10))
apply(a, c(1,2), mean)#Perserve dim 1&2
```

```
##            [,1]       [,2]
## [1,]  0.1471019 -0.4324672
## [2,] -0.9240953  0.5984195
```

```
rowMeans(a,dims = 2)##Equal with 'rowMeans'
```

```
##            [,1]       [,2]
## [1,]  0.1471019 -0.4324672
## [2,] -0.9240953  0.5984195
```

lapply and sapply with *swirl()*

- `cls_list <- lapply(flags, class)` will apply the `class()` function to each column of the flags dataset.

- The following two are equivalent..(But `sapply` gives a named vector)

```
missingHead <- c("name", "landmass", "zone", "area", "population", "language", "religion", "ba
flags <- read.csv(file = "flag.data", header = FALSE)
colnames(flags) <- missingHead

as.character(lapply(flags, class))
```

```
##  [1] "factor"  "integer" "integer" "integer" "integer" "integer" "integer"
##  [8] "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## [15] "integer" "integer" "integer" "factor"  "integer" "integer" "integer"
## [22] "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## [29] "factor"  "factor"
```

```
sapply(flags, class)
```

```
##       name   landmass       zone       area population   language   religion
##   "factor"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##       bars    stripes    colours        red      green       blue       gold
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##      white      black     orange    mainhue    circles    crosses   saltires
##  "integer"  "integer"  "integer"   "factor"  "integer"  "integer"  "integer"
##   quarters   sunstars   crescent   triangle       icon    animate       text
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##    topleft   botright
##   "factor"   "factor"
```

```
flag_colors <- flags[, 11:17]
head(flag_colors)
```

```
##   red green blue gold white black orange
## 1   1     1    0    1     1     1      0
## 2   1     0    0    1     0     1      0
## 3   1     1    0    0     1     0      0
## 4   1     0    1    1     1     0      1
## 5   1     0    1    1     0     0      0
## 6   1     0    0    1     0     1      0
```

```
sapply(flag_colors, sum)
```

```
##    red  green   blue   gold  white  black orange
##    153     91     99     91    146     52     26
```

```
sapply(flag_colors, mean)
```

```
##       red     green      blue      gold     white     black    orange
## 0.7886598 0.4690722 0.5103093 0.4690722 0.7525773 0.2680412 0.1340206
```

- `unique` will return a list of unique elements

```
unique(c(3,4,5,5,5,6,6))
```

```
## [1] 3 4 5 6
```

mapply

- Multivariate version of lapply
- applies a function in parallel over a set of arguments

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

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

*Following two are equal, but mapply is concise

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

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

```
mapply(rep, 1:4, 4:1)
```

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

- "Vectorizing" a function

```
noise <- function(n, mean, sd) {
  rnorm(n,mean,sd)
}
noise(5,1,2)#Works as expected
```

```
## [1] 0.651889 2.922581 1.587653 1.161999 1.367324
```

```
noise(1:5,1:5, 2)#One would expect 1 rand with mean 1, 2 with mean 2, etc...
```

```
## [1]  1.3325101 -0.5391981  7.6989866  1.1759892  4.9660770
```

```
#One is to use mapply to generate the expected behavior
mapply(noise, 1:5, 1:5, 2)
```

```
## [[1]]
## [1] -0.08863871
##
## [[2]]
## [1] 5.600225 4.022880
##
## [[3]]
## [1] 1.872567 3.410842 5.330924
##
## [[4]]
## [1] 8.472646 4.604530 1.914987 2.032915
##
## [[5]]
## [1]  9.011437  0.858857 11.111485  4.477299  4.091213
```

tapply

- Apply a function over subsets of a vector
- Arguments..

```
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

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

```
x = c(rnorm(10), runif(10), rnorm(10,1))
factor <- gl(3, 10)
tapply(x, factor, mean)##apply the mean function to the subsets from factor in the x variable
```

```
##         1         2         3
## -0.282588  0.329488  1.079723
```

- If you don't simplify you get back a list

```
tapply(x, factor, mean, simplify = FALSE)
```

```
## $`1`
## [1] -0.282588
##
## $`2`
## [1] 0.329488
##
## $`3`
## [1] 1.079723
```

- Can also calculate range

```
tapply(x, factor, range)
```

```
## $`1`
## [1] -2.3710229  0.9333887
##
## $`2`
## [1] 0.08510921 0.92167676
##
## $`3`
## [1] -0.6988735  2.3987911
```

split

- Useful in conjunction with `apply` functions.
- `split` takes a vector or other objects and splits it into groups determined by a factor or list of factors

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

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

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
split(x,f)
```

```
## $`1`
##  [1]  1.7724929  0.3434222 -0.6230498 -0.4395223 -0.5052968  0.1860351
##  [7]  0.1764178  0.9158482  0.3201767 -0.3666873
##
## $`2`
##  [1] 0.1734520 0.4932090 0.7371889 0.5110010 0.4750869 0.5878999 0.5725374
```

```
##  [8] 0.6549177 0.8651256 0.3968464
##
## $`3`
##  [1] 2.75203562 0.04618354 2.64408046 0.13326647 1.26635219 1.22237049
##  [7] 0.72309149 2.39425304 0.34108800 1.66053128
```

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

```
lapply(split(x, f), mean)
```

```
## $`1`
## [1] 0.1779837
##
## $`2`
## [1] 0.5467265
##
## $`3`
## [1] 1.318325
```

```
s <- split(airquality, airquality$Month)
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

```
##`na.rm = TRUE` gets rid of nas when calculating mean
```

- Splitting on More than one Level

```
x <- rnorm(10)
f1 <- gl(2,5)
f2 <- gl(5, 2)
f1
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2
## Levels: 1 2
```

```
f2
```

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

```
interaction(f1, f2) #Combines the levels
```

```
##  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

- Interactions can create empty levels

```
str(split(x, list(f1, f2)))
```

```
## List of 10
```

```
##  $ 1.1: num [1:2] -0.0133 -0.9315
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] 1.21 -2.09
##  $ 2.2: num(0)
##  $ 1.3: num -0.526
##  $ 2.3: num -1.54
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] 0.194 0.264
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] -1.119 0.651
```

```
str(split(x, list(f1, f2), drop = TRUE))
```

```
## List of 6
##  $ 1.1: num [1:2] -0.0133 -0.9315
##  $ 1.2: num [1:2] 1.21 -2.09
##  $ 1.3: num -0.526
##  $ 2.3: num -1.54
##  $ 2.4: num [1:2] 0.194 0.264
##  $ 2.5: num [1:2] -1.119 0.651
```

- `drop` = TRUE takes care of empty levels

vapply and tapply with *swirl()*

vapply

- `vapply` allows you to specify the format of the result explicitly.
  - If the default doesn't match the format you specify it'll throw an error
  - Helpful when writing functions

```
sapply(flags, class)#Implicitly returns a simplified result
```

```
##       name   landmass       zone       area population   language   religion
##   "factor"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##       bars    stripes    colours        red      green       blue       gold
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##      white      black     orange    mainhue    circles    crosses   saltires
##  "integer"  "integer"  "integer"   "factor"  "integer"  "integer"  "integer"
##   quarters   sunstars   crescent   triangle       icon    animate       text
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##    topleft   botright
##   "factor"   "factor"
```

```
vapply(flags, class, character(1))#Explicitly returns chars
```

```
##       name   landmass       zone       area population   language   religion
##   "factor"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##       bars    stripes    colours        red      green       blue       gold
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
```

```
##      white      black     orange    mainhue    circles    crosses   saltires
##  "integer"  "integer"  "integer"   "factor"  "integer"  "integer"  "integer"
##   quarters   sunstars   crescent   triangle       icon    animate       text
##  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"  "integer"
##    topleft   botright
##   "factor"   "factor"
```

- faster than `sapply` in large datasets
  - sapply saves typing when doing analysis interactively (at the prompt)

tapply

- `tapply` applies a function to each cell of an array, given by the levels of certain factors.
- The following finds the freq. of animate images on flags, from the flags dataset used in the last `swirl()` section, with respect to each continent.

```
tapply(flags$animate, flags$landmass, mean)
```

```
##         1         2         3         4         5         6
## 0.4193548 0.1764706 0.1142857 0.1346154 0.1538462 0.3000000
```

- Or a stat summary of population (in millions) with respect to red being on the flag

```
tapply(flags$population, flags$red, summary)
```

```
## $`0`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    0.00    3.00   27.63    9.00  684.00
##
## $`1`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     0.0     4.0    22.1    15.0  1008.0
```

Debugging Tools

- These tools come with R

Diagnosing the Problem

- Indications that something's not right
  - `message`: A generic notification/diagnostic message produced by the `message` function; execution of the function continues
  - `warning`: An indication that something is wrong but **not necessarily fatal**; execution of the function continues, warning returns *after execution*; generated by the `warning` function

```
log(-1)
```

```
## Warning in log(-1): NaNs produced
```

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

- `error`: An indication that a fatal problem has occurred; execution stops; produced by the `stop` function

- `condition`: A generic concept for indicating that something unexpected can occur; all of the above are types of conditions; programmers can create their own conditions
- How to evaluate an error
  - Identify your input and how a function was called
  - Identify what you expected as an output, message, etc.
  - Identify what you actual got as an output and how it differs from what you expected
  - Question if your expectation themselves were correct
  - See if you can exactly reproduce the problem as to verifying your understanding of the issue/error

Basic Tools

- Primary tools, these are interactive tools specifically designed to allow you to pick through a function.
  - `traceback`: prints out the function call stack after an error occurs; does nothing if there's no error
  - `debug`: flags a function for "debug" mode which allows you to step through execution of a function one line at a time
    - Anytime the function is called it will suspend the run and open up a debug, which will go through the function line by line
  - `browser`: suspends the execution of a function wherever it is called and puts the function in debug mode
    - Helpful to stop in the middle of a function
  - `trace`: allows you to inset debugging code into a function a specific places.
    - Good when using other's code. Ussually just insert `browser`
  - `recover`: allows you to modify the error behavior so that you can browse the function call stack
    - Normally when you get an error, the execution of the function that threw it will stop and you'll get the error message to display and you're returned to the console, by default. Setting an error handler can change this behavior. `recover` is one of these *error handler*s.

Using the Tools

- Giving the traceback is helpful for communicating an error.

- `traceback()` has to be called right after the error
  - Shows functions between code and where where an error occurred

- `debug`
  - Hit `n` to run next line
  - Ex:
    - `>debug(lm)`
    - `>lm(y ~ x)`
    - *
- `recover`
  - `options(error = recover)` will set a global option for the session so that whenever an error occurs a menu shows up, that is similar to `traceback`
- Summary
  - There are three main indications of a problem/condition: `message`, `warning`, `error`
    - only an `error` is fatal
  - When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs form your expectation.

  - Interactive debugging tools `traceback`, `debug`, `browser`, `trace`, and `recover` can be used to find problematic code in functions
  - Debugging tools are not a substitute for thinking

```
ints <- seq(from = 1.75, length.out = 6, by = 1.5)
sleeptime <- strptime("08:00", "%H:%M")
sleeptime + (ints) * 3600
```

```
## [1] "2020-02-20 09:45:00 EST" "2020-02-20 11:15:00 EST"
## [3] "2020-02-20 12:45:00 EST" "2020-02-20 14:15:00 EST"
## [5] "2020-02-20 15:45:00 EST" "2020-02-20 17:15:00 EST"
```

Simulation & code profiling

Learning Objectives

- Call the **str function on an arbitrary R object.**

- Describe the difference between the **"by.self"** and **"by.total"** output produced by the R profiler.
- Simulate a **random normal variable** with an arbitrary mean and standard deviation.

- Simulate data from a **normal linear model.**

The str Function

"Most useful function in R" * A diagnostic function and an alternative to summary
* Good to display (abbrv.) contents of (possibly nested) lists.
* Displays approx. one line per basic object

```
str(str) #A function( that takes an object, ...)
```

```
## function (object, ...)
```

```
str(lm)#returns the args. for lm fn
```

```
## function (x)
##  - attr(*, "srcref")= 'srcref' int [1:8] 1 7 1 23 7 23 1 1
##   ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x562ebf050ed0>
```

```
x <- rnorm(100, 2, 4)
summary(x)# shows stats summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -9.9878 -0.7047  2.4328  2.1445  4.8830 11.2853
```

```
str(x)#Shows what the data look like
```

```
##  num [1:100] -2.13 4.64 2.95 4.86 -1.75 ...
```

```
f <-  gl(40, 10)##Factor with 40 levels, each is rep 10 times
str(f)##Gives some info on factor
```

```
##  Factor w/ 40 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
summary(f)##Gives num of elements in each of 40 levels, not compact
```

```
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
## 27 28 29 30 31 32 33 34 35 36 37 38 39 40
## 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

```
library(datasets)
head(airquality)
```

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

```
str(airquality)
```

```
## 'data.frame':    153 obs. of  6 variables:
##  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
##  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
```

```
##  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
##  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

```
m <- matrix(rnorm(100), 10, 10)
str(m)
```

```
##  num [1:10, 1:10] -1.663 0.811 -1.912 -1.247 0.998 ...
```

```
m[,-1]
```

```
##             [,1]       [,2]         [,3]        [,4]       [,5]       [,6]
##  [1,] -0.1745469 -1.2523559  0.453733178  0.88127779 -0.2512080 -1.8345276
##  [2,] -0.5914285 -1.0095038 -0.232464148  0.74208177  1.2128894 -0.2690135
##  [3,] -1.3340273  0.7513912  0.870005525  0.14757340 -0.6272581 -1.8339286
##  [4,] -1.0972985 -1.3083535  1.656003734  0.48538856  1.7111585 -0.8144680
##  [5,]  2.0361036  0.5275401 -0.006368929  0.15185604 -0.3943736  0.1635721
##  [6,] -0.3264896 -0.5335396  0.470489453  0.04199875 -2.3214909  0.8555192
##  [7,]  0.7740052 -0.3983760  0.278218649  0.22342231  1.3641192 -0.8199631
##  [8,]  0.7850064 -0.7895695 -0.977902941 -1.01046509  1.1322291 -0.1236028
##  [9,]  0.7632461 -0.2301411 -0.926586142  2.40122210 -0.7743163  0.2549482
## [10,]  0.2948088  0.8771848  1.919770463  0.80196179 -1.4103750  1.7189263
##              [,7]         [,8]       [,9]
##  [1,] -0.95854353 -0.622962660 -2.0908461
##  [2,] -1.60431026  1.262009381  1.6973939
##  [3,] -1.84560942 -0.405774043  1.0638812
##  [4,]  0.55573719  0.666763771 -0.7666166
##  [5,] -0.06011919  0.164639155  0.3820076
##  [6,]  0.77208630  1.781524475  0.2418959
##  [7,] -0.14083939  0.711213964 -1.1327594
##  [8,]  0.39309393 -0.337691156  1.4899074
##  [9,]  0.22421857 -0.009148952 -0.2482471
## [10,]  0.02354199 -0.125309208  0.1835837
```

```
##Create list of airquality split by month
s <- split(airquality, airquality$Month)
str(s)## gives a summary of nested dataframes
```

```
## List of 5
##  $ 5:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
##   ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
##   ..$ Wind   : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##   ..$ Temp   : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
##   ..$ Month  : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 6:'data.frame':   30 obs. of  6 variables:
##   ..$ Ozone  : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
##   ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 323 ...
```

```
##   ..$ Wind   : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 13.8 11.5 ...
##   ..$ Temp   : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
##   ..$ Month  : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
##   ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 7:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
##   ..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 272 175 ...
##   ..$ Wind   : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 5.7 7.4 ...
##   ..$ Temp   : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
##   ..$ Month  : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 8:'data.frame':   31 obs. of  6 variables:
##   ..$ Ozone  : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
##   ..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
##   ..$ Wind   : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 8.6 ...
##   ..$ Temp   : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
##   ..$ Month  : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
##   ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 9:'data.frame':   30 obs. of  6 variables:
##   ..$ Ozone  : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
##   ..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
##   ..$ Wind   : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 10.9 9.7 ...
##   ..$ Temp   : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
##   ..$ Month  : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
##   ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

Simulation - Generating Random Numbers

- Simulation is important to reproduce results
- Functions for simulating probability distributions in R
  - `rnorm`: generate random Normal variate with a given mean and standard deviation
  - `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
  - `pnorm`: evaluate the cumulative distribution function for a Normal distribution
  - `rpois`: generate random Poisson variates with a given rate.
- Probability distribution functions are prefixed with a
  - `d` for density
  - `r` for random number generation
  - `p` for cumulative distribution
  - `q` for quantile function

```r
n <- 100
x <- seq(from = 0, to = 1, length = n)
p <- sample(seq(from = 0, to = 1, length = 100000), size = n)
q <- sample(x)


dnorm(x, mean = 0, sd = 1, log = FALSE)
```

```
##   [1] 0.3989423 0.3989219 0.3988609 0.3987592 0.3986168 0.3984338 0.3982103
##   [8] 0.3979463 0.3976419 0.3972972 0.3969123 0.3964873 0.3960223 0.3955176
##  [15] 0.3949731 0.3943892 0.3937660 0.3931037 0.3924024 0.3916624 0.3908839
##  [22] 0.3900672 0.3892125 0.3883200 0.3873900 0.3864229 0.3854188 0.3843781
##  [29] 0.3833011 0.3821881 0.3810395 0.3798556 0.3786367 0.3773832 0.3760955
##  [36] 0.3747740 0.3734190 0.3720309 0.3706102 0.3691572 0.3676724 0.3661563
##  [43] 0.3646091 0.3630315 0.3614238 0.3597866 0.3581202 0.3564251 0.3547019
##  [50] 0.3529510 0.3511729 0.3493681 0.3475372 0.3456805 0.3437987 0.3418923
##  [57] 0.3399617 0.3380076 0.3360304 0.3340307 0.3320090 0.3299659 0.3279018
##  [64] 0.3258175 0.3237134 0.3215900 0.3194480 0.3172879 0.3151102 0.3129156
##  [71] 0.3107045 0.3084776 0.3062354 0.3039784 0.3017073 0.2994227 0.2971250
##  [78] 0.2948149 0.2924929 0.2901595 0.2878154 0.2854612 0.2830973 0.2807243
##  [85] 0.2783428 0.2759534 0.2735565 0.2711528 0.2687429 0.2663271 0.2639062
##  [92] 0.2614805 0.2590508 0.2566174 0.2541810 0.2517419 0.2493009 0.2468584
##  [99] 0.2444148 0.2419707
```

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

```
##   [1] -0.552358286  0.590461401 -0.527275411  1.341846164  0.243844628
##   [6]  0.195880789 -0.015519729 -0.297577328 -0.167679147  1.156478580
##  [11] -1.338764009  1.086768775 -0.350684197  0.734723245 -0.058414484
##  [16] -1.345812904  0.346905269 -0.638404647  1.124392355 -0.827733031
##  [21] -1.787780138 -0.854010003 -0.572022722 -0.825930679 -0.237584094
##  [26] -0.595326496 -0.410742195  0.327991438 -1.163646612  0.760177665
##  [31] -0.002555836 -0.038227771 -2.036237420 -0.301229079 -0.555646104
##  [36]  0.847295416  1.390820511  1.231246215 -1.113337462 -0.796053893
##  [41] -0.089519877 -1.059656888 -1.604312194  0.791350666  0.068156528
##  [46]  0.614396000 -1.203067753 -0.340454835 -1.171248634 -0.829414768
##  [51]  0.229212042 -0.814992918 -0.993445515 -1.306168459 -1.434221772
##  [56] -0.862847933  1.703799204 -0.655632596 -1.113168626 -1.405505446
##  [61] -0.150643687 -0.394006912  0.252069320 -0.282706438  0.867667613
##  [66]  0.720091121  1.056069601  0.243502060  1.114362662 -0.076552714
##  [71]  0.932925054 -0.157981739 -0.681509515 -1.220101091 -0.708912992
##  [76] -0.748679598  0.576077526 -0.052099464 -0.630337174 -0.898336066
##  [81]  1.512685876 -0.236749293 -1.248803422 -0.430009336  0.093299436
##  [86]  0.833041089 -0.084932478  0.111853039 -1.039435943  0.796422814
##  [91]  0.099903376  0.736959753  1.546881334  0.178920972 -0.282546592
##  [96] -0.767298779 -0.576404240 -0.914855840  0.369910961 -1.467684363
```

```r
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

```
##   [1] 0.6113458 0.6789077 0.5522362 0.6825165 0.8129686 0.8288155 0.6494509
##   [8] 0.5641959 0.5760972 0.7277627 0.5201400 0.6074686 0.7962448 0.7507233
##  [15] 0.8388883 0.7664705 0.5918565 0.6606147 0.7602387 0.5322027 0.7816500
##  [22] 0.7990924 0.8183489 0.6861081 0.7002962 0.7344287 0.8413447 0.7539173
##  [29] 0.7726115 0.6896821 0.7176105 0.5241635 0.6932383 0.8047155 0.6716394
##  [36] 0.6152119 0.7633659 0.6569085 0.5562286 0.6035807 0.7846162 0.7442699
##  [43] 0.5362176 0.5000000 0.8210024 0.7311061 0.7037973 0.7243989 0.6419352
##  [50] 0.5080589 0.8339012 0.5681701 0.7410108 0.5879296 0.8313707 0.6190666
##  [57] 0.6381564 0.7875589 0.8074908 0.5161145 0.8019160 0.8236314 0.6305587
##  [64] 0.5602153 0.8364071 0.8102419 0.7756475 0.7141864 0.6267403 0.7695524
##  [71] 0.7377303 0.6643055 0.5040297 0.6679805 0.7570891 0.5996825 0.6967764
##  [78] 0.5402288 0.6457001 0.5721373 0.5839938 0.5120873 0.7933733 0.8262357
##  [85] 0.5957743 0.7107426 0.7786604 0.6752819 0.7475075 0.7210147 0.7904780
##  [92] 0.6531871 0.7072795 0.5800495 0.5482385 0.6229095 0.6343641 0.5442359
##  [99] 0.5281846 0.8156710
```

```
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

```
##   [1]  2.108123014 -0.122441939  0.811838741 -0.777227822 -0.282046599
##   [6] -1.942963900  0.729027796 -0.940495475 -0.081795483  0.728733557
##  [11] -1.342722663  0.844152872  0.131842397  0.480704817 -0.739860583
##  [16] -1.093070736 -0.854218509  0.842400174 -2.057476674 -1.060311542
##  [21]  0.242118831  2.009157388  0.769351930  2.178883909 -0.093924349
##  [26]  0.709096197  0.379647573  0.264924128  0.233017061  0.437089254
##  [31]  1.688679257 -1.118197540  2.317432111  1.593888058  0.284055611
##  [36] -0.216666464 -0.900933504  0.216050621 -0.766423366 -0.825471263
##  [41]  1.407154454  1.582036946  0.082851851  0.484168511 -0.202292073
##  [46] -1.532166898  0.452947556  0.038398830  1.199889388 -0.109067274
##  [51] -0.624368932 -0.030974654  0.059857006 -1.194295688  0.302819240
##  [56]  0.543877366 -2.312680582 -0.584051607 -0.067216356 -0.368973977
##  [61] -0.914443973 -0.341234133  1.359646741  0.210947383  0.661354366
##  [66]  1.216112054 -0.245837460  1.159377208  1.392663281 -1.223602513
##  [71] -1.818939313 -2.461904703  0.154051948 -0.212921344 -0.796839493
##  [76] -0.095888364 -0.253725028 -0.044018511 -0.100573292 -0.329672524
##  [81]  0.969164090 -0.143305566 -0.861752210  0.270718085  1.244862950
##  [86]  1.478322817  0.061715360  0.179572017  0.641765874 -0.785459304
##  [91] -0.005577333  0.931406139  1.039132780 -0.577997758  0.894183513
##  [96] -0.478567530  1.150877835  0.491707367 -0.399578636  0.835486163
```

- Always set the random number seed when conducting a simulation

```
set.seed(1)
rnorm(5)
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

```
rnorm(5)
```

```
## [1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```

```r
set.seed(1)
rnorm(5)#Will be same as first rnorm
```

```
## [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

- Generating Poisson data

```r
rpois(10, 1)## 10 vars with rate of 1
```

```
##  [1] 0 0 1 1 2 1 1 4 1 2
```

```r
rpois(10, 2)
```

```
##  [1] 4 1 2 0 1 1 0 1 4 1
```

```r
rpois(10, 20)
```

```
##  [1] 19 19 24 23 22 24 23 20 11 22
```

```r
ppois(2, 2) ## Cumulative distribution
```

```
## [1] 0.6766764
```

```r
ppois(4,2)## Pr(x <= 4) with rate of 2
```

```
## [1] 0.947347
```

```r
ppois(6,2)## Pr(x <= 6) with rate of 2
```

```
## [1] 0.9954662
```

Simulation - Simulating a Linear Model

- Suppose we want to simulate from the following linear model:

$$y = B\ 0\ + B\ 1\ x + E$$

where

$$E\ N(0, 2)$$

, Assume

$$x\ N(0, 1)$$

,

$$B\ 0\ = 0.5$$

and

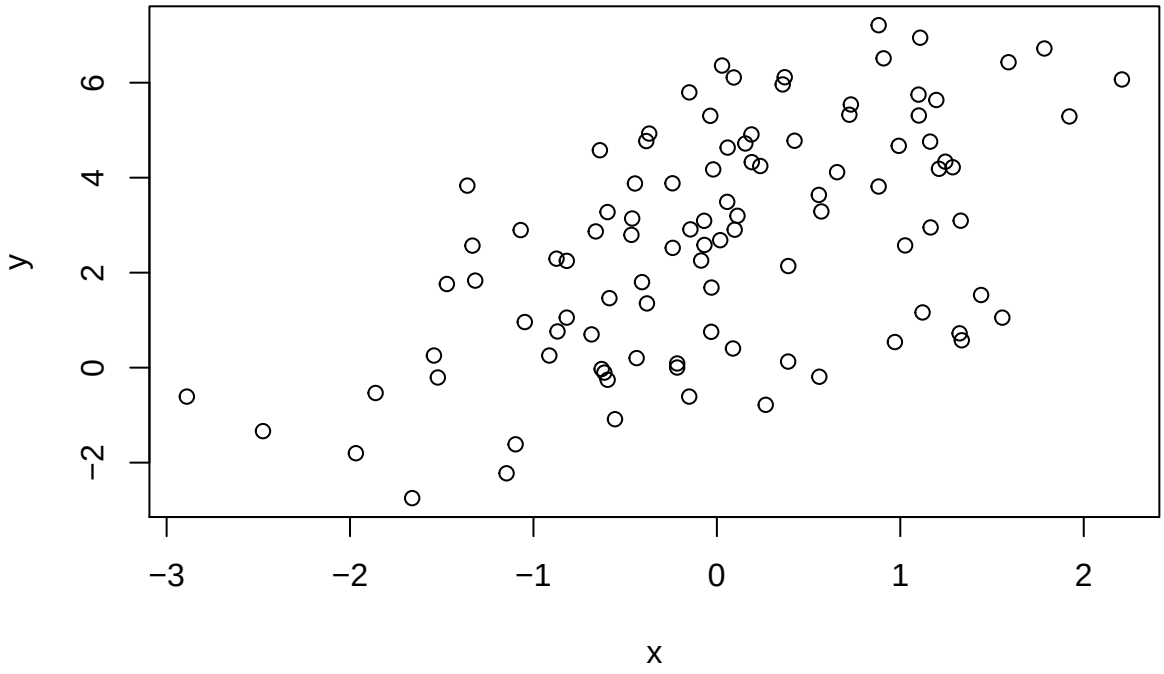
$$B\ 1\ = 2$$

.

```r
set.seed(20)
x <- rnorm(100)# default mean = 0, sd = 2
e <- rnorm(100, 0, 2) #mean = 0, sd = 2
y <- 0.5 + 2 + x + e
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.7472  0.7166  2.8823  2.6844  4.6422  7.2095
```

```
plot(x,y)
```

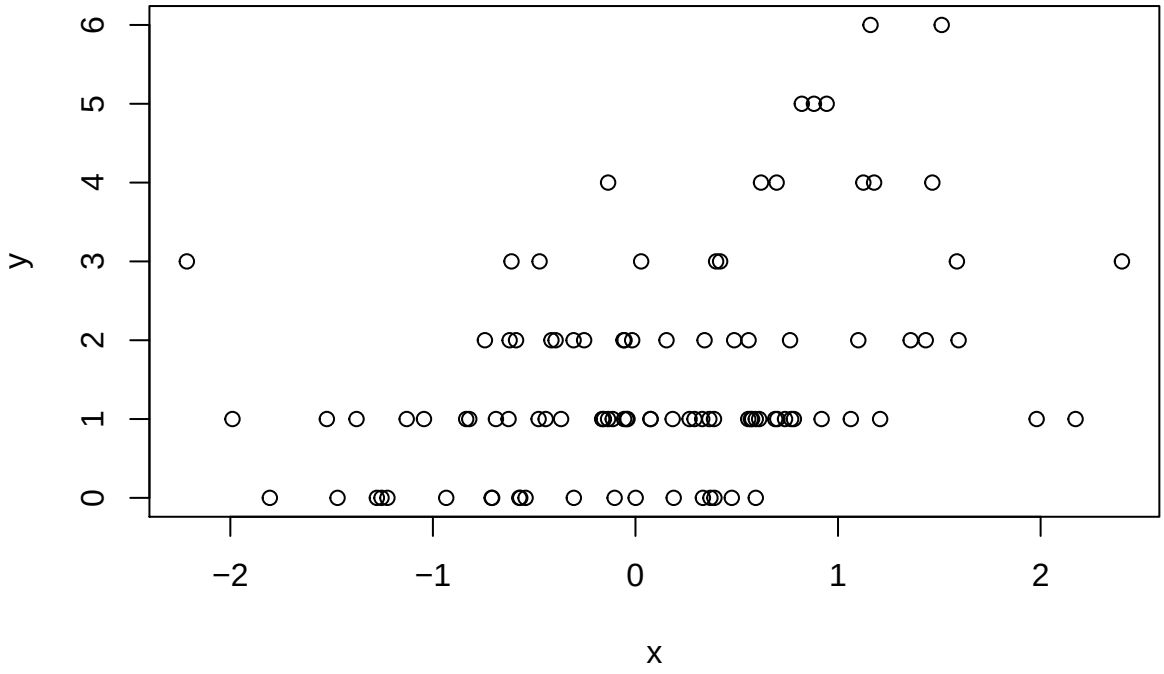


- Simulating a Poisson model where:
  - Y ~ Poisson(mu) +log(mu) = $B\ 0\ + B\ 1\ x$ + $B\ 0\ = 0.5$ and $B\ 1\ = 0.3$.
- We need to use the **rpois** function for this

```
set.seed(1)
x <- rnorm(100)
log.mu <- 0.5 + 0.3 * x
y <- rpois(100, exp(log.mu))
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    1.00    1.00    1.55    2.00    6.00
```

```r
plot(x,y)
```



Simulation - Random Sampling

- `sample` function
  - Allows you to draw randomly from a set
  - `sample(1:10, 4)` will sample 4 random numbers between 1 and 10 without replacement

```r
set.seed(1)
sample(1:10, 4)
```

```
## [1] 9 4 7 1
```

```r
sample(1:10, 4)
```

```
## [1] 2 7 3 6
```

```r
sample(letters, 5)
```

```
## [1] "r" "s" "a" "u" "w"
```

```r
sample(1:10) ## permutation
```

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

```
sample(1:10)
```

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

```
sample(1:10, replace = TRUE) ## with replacement
```

```
##  [1]  3  6 10 10  6  4  4 10  9  7
```

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

Looking at Data in `swirl()`

- When first working with a new dataset, the first thing you should do is look at it!
  - What is the format?
  - What are the dimensions?
  - What are the variable names?
  - How are the variables stored?
  - Are there missing data?
  - Are there any flaws in the data?
- This lesson will be using a dataset constructed from the United States Department of Agriculture's PLANTS Database to show off some ways to look at data

```
plants <- dget("plants.R")
class(plants)
```

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

```
dim(plants)
```

```
## [1] 5166   10
```

```
nrow(plants)
```

```
## [1] 5166
```

```
ncol(plants)
```

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

```
object.size(plants)
```

```
## 686080 bytes
```

```
names(plants)
```

```
##  [1] "Scientific_Name"      "Duration"             "Active_Growth_Period"
##  [4] "Foliage_Color"        "pH_Min"               "pH_Max"
##  [7] "Precip_Min"           "Precip_Max"           "Shade_Tolerance"
## [10] "Temp_Min_F"
```

```
head(plants)
```

```
##                Scientific_Name          Duration Active_Growth_Period
## 1                   Abelmoschus              <NA>                 <NA>
## 2       Abelmoschus esculentus Annual, Perennial                 <NA>
## 3                         Abies              <NA>                 <NA>
## 4                Abies balsamea         Perennial    Spring and Summer
## 5 Abies balsamea var. balsamea         Perennial                 <NA>
## 6                     Abutilon              <NA>                 <NA>
##   Foliage_Color pH_Min pH_Max Precip_Min Precip_Max Shade_Tolerance Temp_Min_F
## 1          <NA>     NA     NA         NA         NA             <NA>         NA
## 2          <NA>     NA     NA         NA         NA             <NA>         NA
## 3          <NA>     NA     NA         NA         NA             <NA>         NA
## 4         Green      4      6         13         60        Tolerant        -43
## 5          <NA>     NA     NA         NA         NA             <NA>         NA
## 6          <NA>     NA     NA         NA         NA             <NA>         NA
```

```
head(plants, 10)
```

```
##                     Scientific_Name          Duration Active_Growth_Period
## 1                         Abelmoschus              <NA>                 <NA>
## 2             Abelmoschus esculentus Annual, Perennial                 <NA>
## 3                               Abies              <NA>                 <NA>
## 4                      Abies balsamea         Perennial    Spring and Summer
## 5       Abies balsamea var. balsamea         Perennial                 <NA>
## 6                            Abutilon              <NA>                 <NA>
## 7               Abutilon theophrasti            Annual                 <NA>
## 8                             Acacia              <NA>                 <NA>
## 9                   Acacia constricta         Perennial    Spring and Summer
## 10 Acacia constricta var. constricta         Perennial                 <NA>
##    Foliage_Color pH_Min pH_Max Precip_Min Precip_Max Shade_Tolerance Temp_Min_F
## 1           <NA>     NA     NA         NA         NA             <NA>         NA
## 2           <NA>     NA     NA         NA         NA             <NA>         NA
## 3           <NA>     NA     NA         NA         NA             <NA>         NA
## 4          Green      4    6.0         13         60        Tolerant        -43
## 5           <NA>     NA     NA         NA         NA             <NA>         NA
## 6           <NA>     NA     NA         NA         NA             <NA>         NA
## 7           <NA>     NA     NA         NA         NA             <NA>         NA
## 8           <NA>     NA     NA         NA         NA             <NA>         NA
```

```
## 9          Green      7    8.5          4         20      Intolerant        -13
## 10          <NA>     NA     NA         NA         NA            <NA>         NA
```

```
tail(plants)
```

```
##       Scientific_Name  Duration Active_Growth_Period Foliage_Color pH_Min
## 5161      Zizia aurea Perennial                 <NA>          <NA>     NA
## 5162 Zizia trifoliata Perennial                 <NA>          <NA>     NA
## 5163          Zostera      <NA>                 <NA>          <NA>     NA
## 5164   Zostera marina Perennial                 <NA>          <NA>     NA
## 5165           Zoysia      <NA>                 <NA>          <NA>     NA
## 5166  Zoysia japonica Perennial                 <NA>          <NA>     NA
##      pH_Max Precip_Min Precip_Max Shade_Tolerance Temp_Min_F
## 5161     NA         NA         NA            <NA>         NA
## 5162     NA         NA         NA            <NA>         NA
## 5163     NA         NA         NA            <NA>         NA
## 5164     NA         NA         NA            <NA>         NA
## 5165     NA         NA         NA            <NA>         NA
## 5166     NA         NA         NA            <NA>         NA
```

```
tail(plants, 15)
```

```
##                       Scientific_Name  Duration Active_Growth_Period
## 5152                          Zizania      <NA>                 <NA>
## 5153                 Zizania aquatica    Annual               Spring
## 5154   Zizania aquatica var. aquatica    Annual                 <NA>
## 5155                Zizania palustris    Annual                 <NA>
## 5156 Zizania palustris var. palustris    Annual                 <NA>
## 5157                       Zizaniopsis      <NA>                 <NA>
## 5158            Zizaniopsis miliacea Perennial    Spring and Summer
## 5159                           Zizia      <NA>                 <NA>
## 5160                    Zizia aptera Perennial                 <NA>
## 5161                     Zizia aurea Perennial                 <NA>
## 5162                Zizia trifoliata Perennial                 <NA>
## 5163                          Zostera      <NA>                 <NA>
## 5164                  Zostera marina Perennial                 <NA>
## 5165                          Zoysia      <NA>                 <NA>
## 5166                 Zoysia japonica Perennial                 <NA>
##      Foliage_Color pH_Min pH_Max Precip_Min Precip_Max Shade_Tolerance
## 5152          <NA>     NA     NA         NA         NA            <NA>
## 5153         Green    6.4    7.4         30         50      Intolerant
## 5154          <NA>     NA     NA         NA         NA            <NA>
## 5155          <NA>     NA     NA         NA         NA            <NA>
## 5156          <NA>     NA     NA         NA         NA            <NA>
## 5157          <NA>     NA     NA         NA         NA            <NA>
## 5158         Green    4.3    9.0         35         70      Intolerant
## 5159          <NA>     NA     NA         NA         NA            <NA>
## 5160          <NA>     NA     NA         NA         NA            <NA>
## 5161          <NA>     NA     NA         NA         NA            <NA>
```

```
## 5162          <NA>     NA     NA          NA          NA            <NA>
## 5163          <NA>     NA     NA          NA          NA            <NA>
## 5164          <NA>     NA     NA          NA          NA            <NA>
## 5165          <NA>     NA     NA          NA          NA            <NA>
## 5166          <NA>     NA     NA          NA          NA            <NA>
##      Temp_Min_F
## 5152         NA
## 5153         32
## 5154         NA
## 5155         NA
## 5156         NA
## 5157         NA
## 5158         12
## 5159         NA
## 5160         NA
## 5161         NA
## 5162         NA
## 5163         NA
## 5164         NA
## 5165         NA
## 5166         NA
```

```
summary(plants)
```

```
##                      Scientific_Name              Duration
##  Abelmoschus                  :   1   Perennial        :3031
##  Abelmoschus esculentus       :   1   Annual           : 682
##  Abies                        :   1   Annual, Perennial: 179
##  Abies balsamea               :   1   Annual, Biennial :  95
##  Abies balsamea var. balsamea:   1   Biennial         :  57
##  Abutilon                     :   1   (Other)          :  92
##  (Other)                      :5160   NA's             :1030
##            Active_Growth_Period      Foliage_Color      pH_Min
##  Spring and Summer   : 447      Dark Green  :  82   Min.   :3.000
##  Spring              : 144      Gray-Green  :  25   1st Qu.:4.500
##  Spring, Summer, Fall:  95      Green       : 692   Median :5.000
##  Summer              :  92      Red         :   4   Mean   :4.997
##  Summer and Fall     :  24      White-Gray  :   9   3rd Qu.:5.500
##  (Other)             :  30      Yellow-Green:  20   Max.   :7.000
##  NA's                :4334      NA's        :4334   NA's   :4327
##      pH_Max         Precip_Min      Precip_Max         Shade_Tolerance
##  Min.   : 5.100   Min.   : 4.00   Min.   : 16.00   Intermediate: 242
##  1st Qu.: 7.000   1st Qu.:16.75   1st Qu.: 55.00   Intolerant  : 349
##  Median : 7.300   Median :28.00   Median : 60.00   Tolerant    : 246
##  Mean   : 7.344   Mean   :25.57   Mean   : 58.73   NA's        :4329
##  3rd Qu.: 7.800   3rd Qu.:32.00   3rd Qu.: 60.00
##  Max.   :10.000   Max.   :60.00   Max.   :200.00
##  NA's   :4327     NA's   :4338    NA's   :4338
```

```
##    Temp_Min_F
##  Min.   :-79.00
##  1st Qu.:-38.00
##  Median :-33.00
##  Mean   :-22.53
##  3rd Qu.:-18.00
##  Max.   : 52.00
##  NA's   :4328
```

```
table(plants$Active_Growth_Period)#Make a table based on a factor
```

```
##
## Fall, Winter and Spring                  Spring         Spring and Fall
##                      15                     144                      10
##       Spring and Summer    Spring, Summer, Fall                  Summer
##                     447                      95                      92
##         Summer and Fall             Year Round
##                      24                       5
```

```
str(plants)##One of the "best" to use
```

```
## 'data.frame':    5166 obs. of  10 variables:
##  $ Scientific_Name     : Factor w/ 5166 levels "Abelmoschus",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Duration            : Factor w/ 8 levels "Annual","Annual, Biennial",..: NA 4 NA 7 7 NA 1
##  $ Active_Growth_Period: Factor w/ 8 levels "Fall, Winter and Spring",..: NA NA NA 4 NA NA N
##  $ Foliage_Color       : Factor w/ 6 levels "Dark Green","Gray-Green",..: NA NA NA 3 NA NA N
##  $ pH_Min              : num  NA NA NA 4 NA NA NA NA 7 NA ...
##  $ pH_Max              : num  NA NA NA 6 NA NA NA NA 8.5 NA ...
##  $ Precip_Min          : int  NA NA NA 13 NA NA NA NA 4 NA ...
##  $ Precip_Max          : int  NA NA NA 60 NA NA NA NA 20 NA ...
##  $ Shade_Tolerance     : Factor w/ 3 levels "Intermediate",..: NA NA NA 3 NA NA NA NA 2 NA
##  $ Temp_Min_F          : int  NA NA NA -43 NA NA NA NA -13 NA ...
```

Simulation in `swirl()`

- This section focuses on the r*** functions for prob. distributions in R
- `sample()`
  - Simulate rolling four six-sided dice:

```
sample(1:6, 4, replace = TRUE)
```

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

- Simulate flipping an unfair coin 100 times

```
flips <- sample(c(0,1), 100, replace = TRUE, c(0.3, 0.7))
sum(flips) #Num of heads
```

```
## [1] 73
```

- `rbinom()`
  - Performs a random binomial sample
  - Demonstrating unfair coin simulation with `rbinorm`
    - This will return the number of successes(1s)

```
rbinom(1, size = 100, prob = 0.7)
```

```
## [1] 70
```

- `rnorm()`
  - std. normal distribution

```
rnorm(10)
```

```
##  [1] -1.4375862 -0.7970895  1.2540831  0.7721422 -0.2195156 -0.4248103
##  [7] -0.4189801  0.9969869 -0.2757780  1.2560188
```

```
rnorm(10, 100, 25)#Default can be changed too
```

```
##  [1] 116.16686 132.48281  78.16845 100.20927  77.97821 114.90648 102.99294
##  [8]  92.94565 136.39971 105.72549
```

- `rpois()`
  - Poisson distribution

```
rpois(n = 5, 10)#mean = 10/(time int)
```

```
## [1] 13 12  7  7  6
```

- `hist()`
  - Histogram function
- All of the standard probability distributions are built into R, including...
  - `rexp()` - exponential
  - `rchisq()` - chi-squared
  - `rgamma()` - gamma

R Profiler

Pt. 1

- Helps you figure out why a program is taking so much time and suggests strats for fixing the problem.
- Profiling is a systematic way to examine how much time is spent in different parts of a program
- Useful when trying to optimize your code

- Profiling is better than guessing when trying to analyze a project
- One should not think of optimization first. You should focus on getting the code to run and be understandable.
  - "*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*" –*Donald Knuth*
  - Difficult to figure out where your program is spending most of it's time without having the program all together.
- *So.. You want optimize your code...*
  - Remember, you are a scientist. Collect data first,
  - collect said data with **profiling**

Using system.time()

- Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time take to evalue the expression
- Comuptes the time (in seconds) needed toe xecute an expression
  - If there's an error, it'll return time 'til the error occured.
- Returns an object of class `proc_time`
  - **user time**: time charged to the CPU(s) for this expression (Time the computer experiences)
  - **elapsed time**: "wall clock" time (Time the user experiences)
- Usually, the two times are relatively close
  - Elapsed time may be *greater than* user time if the CPU spends a lot of time waiting around for external events to happen.

```
system.time(readLines("http://www.jhsph.edu"))
```

```
##    user  system elapsed
##   0.041   0.002   1.406
```

- Elapsed time may be *smaller than* the user time fi your machine has multiple cores/processors (and is capable of using them)
  - Multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS, ACML, MKL)
  - Parallel processing via the **parallel** package

```
hilbert <- function(n) {
  i <- 1:n
  1 / outer(i - 1, i, "+")
}

x <- hilbert(1000)
system.time(svd(x))
```

```
##    user  system elapsed
##   0.563   0.008   0.575
```

- Testing a whole expression:

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

```
##    user  system elapsed
##   0.111   0.000   0.112
```

- `system.time()` allow syou to test cetrain functions or code blocks
  - So this is assuming that you know where to look.
  - Pt. 2 will cover what to do if you don't know where to start

Pt. 2, Using Rprof()

- `Rprof()` function starts the profiler in R
  - R must be compiled with profiler support (but this is usually the case)
  - You should not use both `Rprof()` and `system.time()` together
  - If your code runs very quickly, the profiler is not useful; however you probably don't need it in that case
- `summaryRprof()` function summarizes the output from `Rprof()`
  - keeps track of the function call stack at regualrly sampled intervals and tabulates how much time is spent in each function
  - Default sampling interval is 0.02 seconds
  - There are two methdos for normalizing the data
    - "by.total" divides the time spent in each function by the total run time
    - "by.self" does the same but first subtracts out time spent in fucntions above in the call stack

Using "by total" and "by self"

- `by.total`
  - normalizes by total time spent in a function.
    - so by.total reports how much time is spent in top level function, ussually just calling to helper functions
- `by.self`
  - tells you how much time is being taken in a given function after subtracting out lower level functions

- C or Fortran code is not profiled

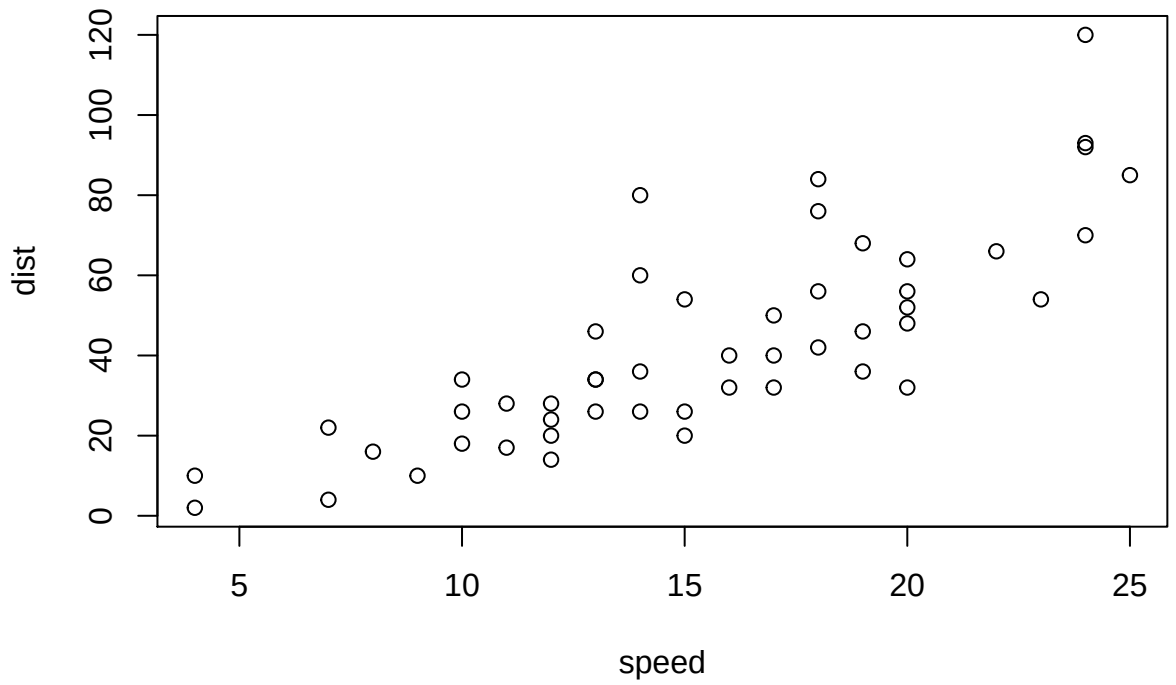
Base Graphics in `swirl()`

- One of the greatest strengths of R is the ease with which it can create publication0quality graphics.

```
data(cars)
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

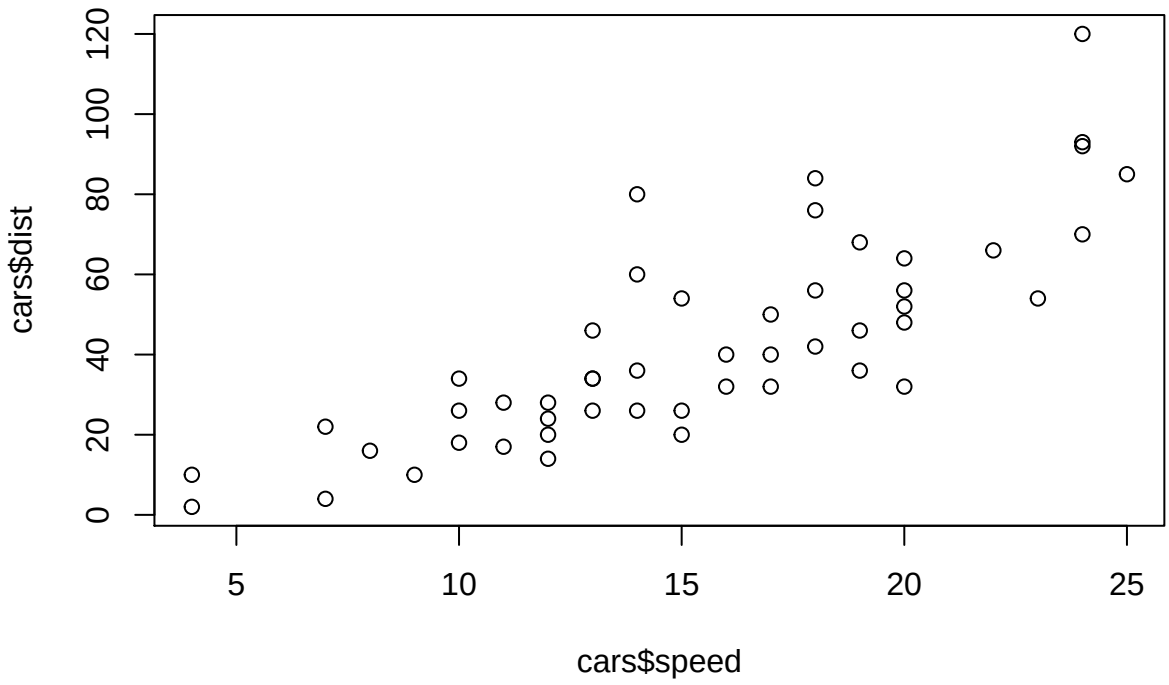
```
plot(cars)
```



- R always trys to output somemthing sensible, as such `plot(cars)` assumes you want the two columns plotted against eachother

- Below R uses the argument names in the first example, then the second corrects this

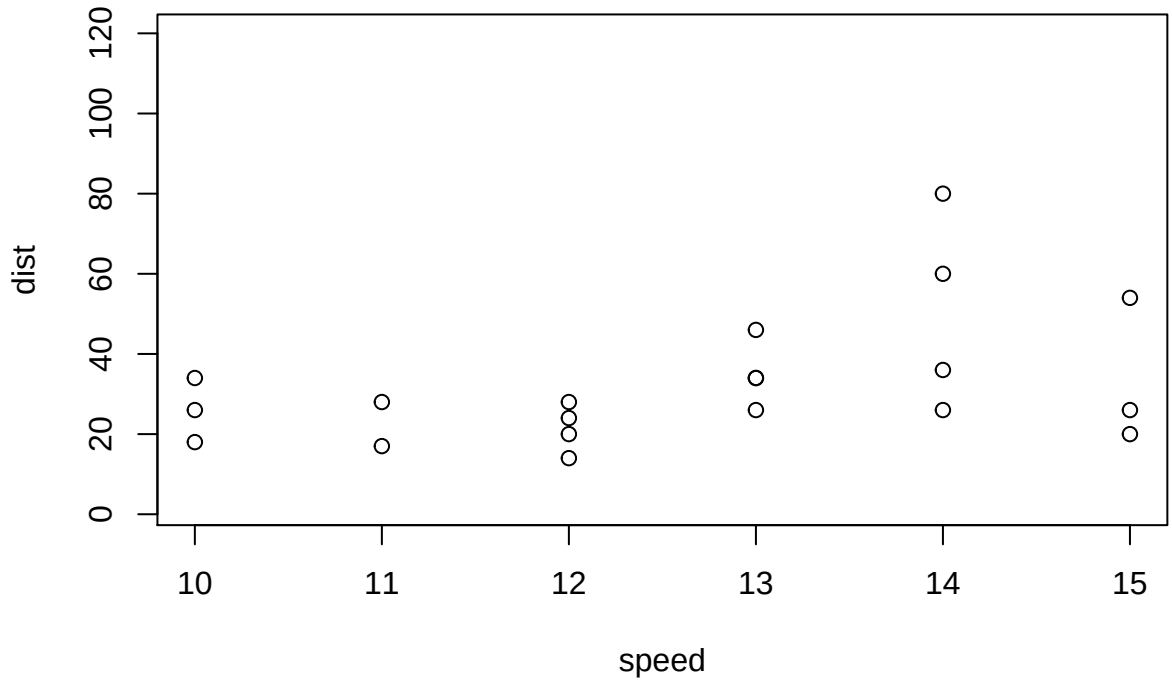
```r
plot(cars$speed, cars$dist)
```



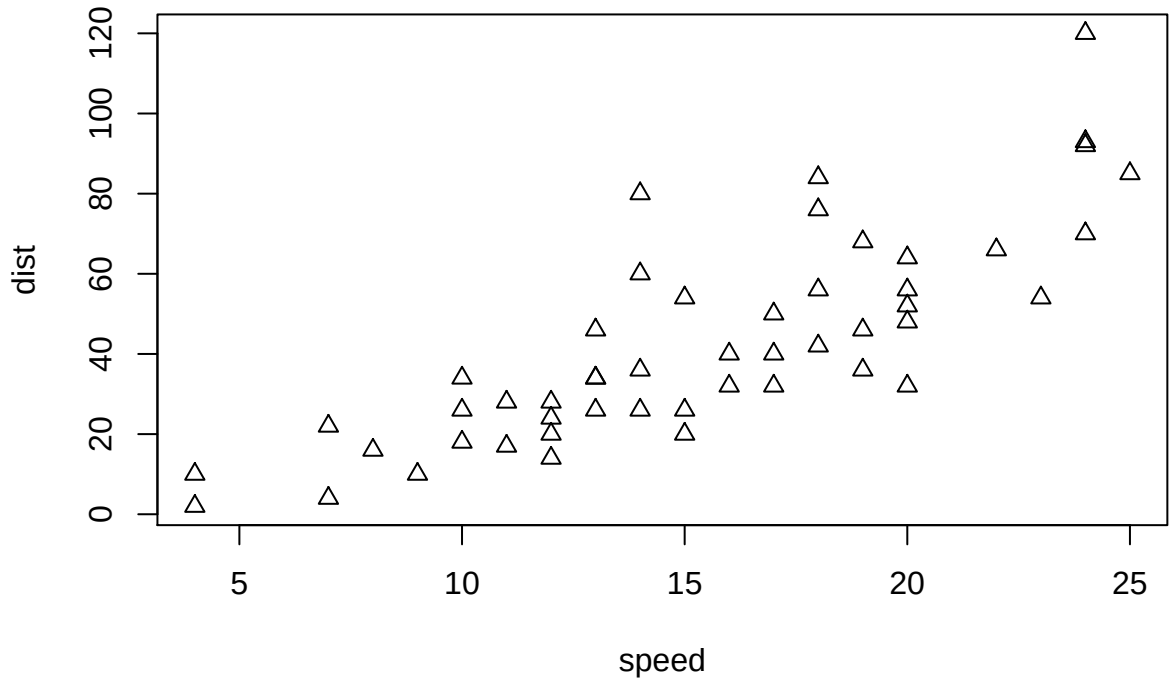

```r
plot(cars$speed, cars$dist,
     xlab = "Speed", ylab = "Stopping Distance",
     main = "My Plot", sub = "My Plot Subtitle")
```

- Limit x range

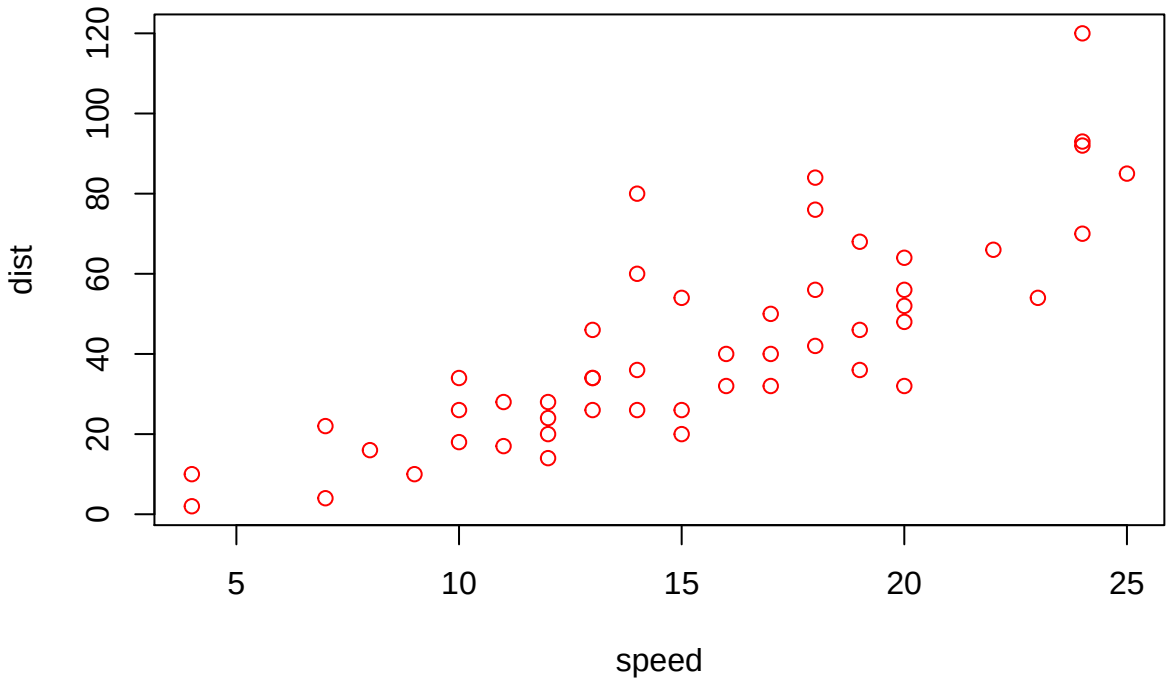
```
plot(cars, xlim = (c(10,15)))
```




```
plot(cars, pch = 2)##Triangles
```

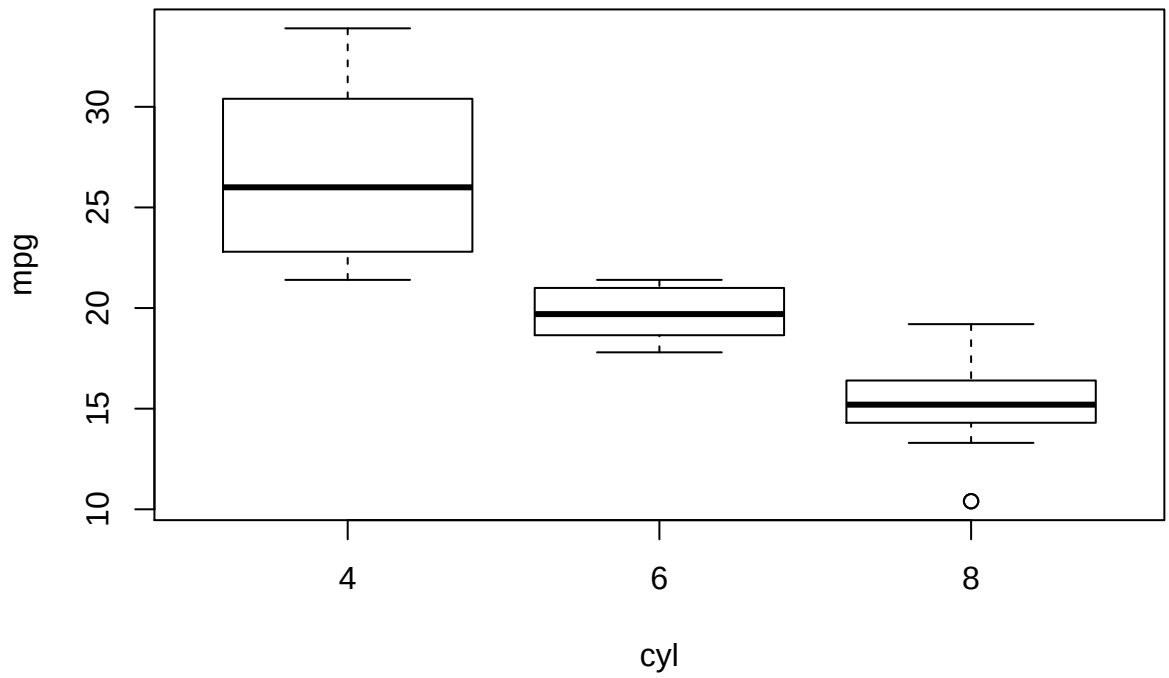



```
plot(cars, col = 2)##Color is red
```



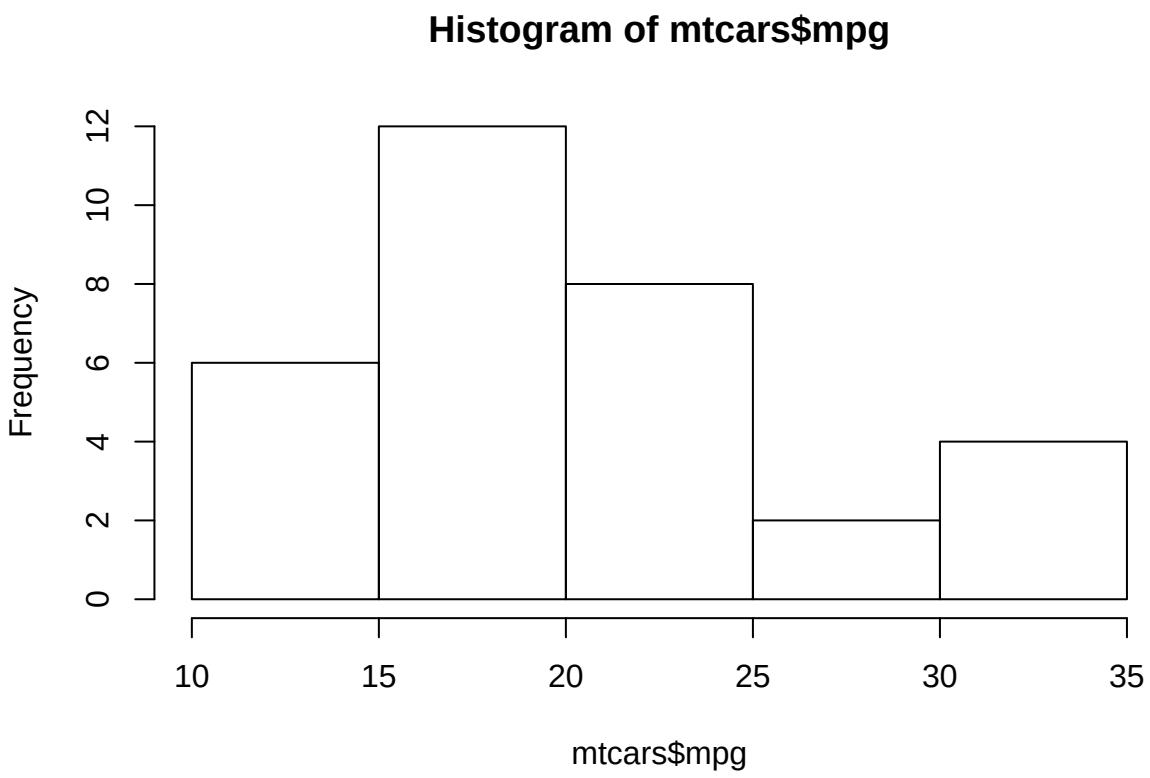
- `boxplot()`

```
data(mtcars)
boxplot(formula = mpg ~ cyl, data = mtcars)
```



- Histograms

```
hist(mtcars$mpg)
```



Scribbles from quiz

```
##1
set.seed(1)
rpois(5,2)
```

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

```
##5
set.seed(10)
x <- rep(0:1, each = 5)
e <- rnorm(10,0,20)
y <- 0.5 + 2 * x + e
```