R

R is a statistical computing language. It has lots of libraries for uploading and cleaning data sets, running statistical procedures, and making graphs. You can also run `R` commands within a LaTeX document.

```
# Comments start with number symbols.

# You can't make multi-line comments,
# but you can stack multiple comments like so.

# in Windows you can use CTRL-ENTER to execute a line.
# on Mac it is COMMAND-ENTER



################################################################################
# Stuff you can do without understanding anything about programming
################################################################################

# In this section, we show off some of the cool stuff you can do in
# R without understanding anything about programming. Do not worry
# about understanding everything the code does. Just enjoy!

data()          # browse pre-loaded data sets
data(rivers)    # get this one: "Lengths of Major North American Rivers"
ls()            # notice that "rivers" now appears in the workspace
head(rivers)    # peek at the data set
# 735 320 325 392 524 450

length(rivers)  # how many rivers were measured?
# 141
summary(rivers) # what are some summary statistics?
#   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  135.0   310.0   425.0   591.2   680.0  3710.0

# make a stem-and-leaf plot (a histogram-like data visualization)
stem(rivers)

#  The decimal point is 2 digit(s) to the right of the |
#
#   0 | 4
#   2 | 011223334555566667778888899900001111223333344455555666688888999
#   4 | 1111222333445566779001233344567
#   6 | 000112233578012234468
#   8 | 045790018
#  10 | 04507
#  12 | 1471
#  14 | 56
#  16 | 7
#  18 | 9
#  20 |
#  22 | 25
#  24 | 3
```

```
#  26 |
#  28 |
#  30 |
#  32 |
#  34 |
#  36 | 1

stem(log(rivers)) # Notice that the data are neither normal nor log-normal!
# Take that, Bell curve fundamentalists.

#  The decimal point is 1 digit(s) to the left of the |
#
#  48 | 1
#  50 |
#  52 | 15578
#  54 | 44571222466689
#  56 | 023334677000124455789
#  58 | 00122366666999933445777
#  60 | 122445567800133459
#  62 | 112666799035
#  64 | 00011334581257889
#  66 | 003683579
#  68 | 0019156
#  70 | 079357
#  72 | 89
#  74 | 84
#  76 | 56
#  78 | 4
#  80 |
#  82 | 2

# make a histogram:
hist(rivers, col="#333333", border="white", breaks=25) # play around with these parameters
hist(log(rivers), col="#333333", border="white", breaks=25) # you'll do more plotting later

# Here's another neat data set that comes pre-loaded. R has tons of these.
data(discoveries)
plot(discoveries, col="#333333", lwd=3, xlab="Year",
     main="Number of important discoveries per year")
plot(discoveries, col="#333333", lwd=3, type = "h", xlab="Year",
     main="Number of important discoveries per year")

# Rather than leaving the default ordering (by year),
# we could also sort to see what's typical:
sort(discoveries)
#  [1]  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  2  2  2  2
# [26]  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  3  3  3
# [51]  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4
# [76]  4  4  4  4  5  5  5  5  5  5  5  6  6  6  6  6  6  7  7  7  7  8  9 10 12

stem(discoveries, scale=2)
#
#  The decimal point is at the |
#
```

```
#   0 | 000000000
#   1 | 000000000000
#   2 | 00000000000000000000000000
#   3 | 0000000000000000000
#   4 | 000000000000
#   5 | 0000000
#   6 | 000000
#   7 | 0000
#   8 | 0
#   9 | 0
#  10 | 0
#  11 |
#  12 | 0

max(discoveries)
# 12
summary(discoveries)
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#     0.0     2.0     3.0     3.1     4.0    12.0

# Roll a die a few times
round(runif(7, min=.5, max=6.5))
# 1 4 6 1 4 6 4
# Your numbers will differ from mine unless we set the same random.seed(31337)

# Draw from a standard Gaussian 9 times
rnorm(9)
# [1]  0.07528471  1.03499859  1.34809556 -0.82356087  0.61638975 -1.88757271
# [7] -0.59975593  0.57629164  1.08455362



######################################################
# Data types and basic arithmetic
######################################################

# Now for the programming-oriented part of the tutorial.
# In this section you will meet the important data types of R:
# integers, numerics, characters, logicals, and factors.
# There are others, but these are the bare minimum you need to
# get started.

# INTEGERS
# Long-storage integers are written with L
5L # 5
class(5L) # "integer"
# (Try ?class for more information on the class() function.)
# In R, every single value, like 5L, is considered a vector of length 1
length(5L) # 1
# You can have an integer vector with length > 1 too:
c(4L, 5L, 8L, 3L) # 4 5 8 3
length(c(4L, 5L, 8L, 3L)) # 4
class(c(4L, 5L, 8L, 3L)) # "integer"
```

```r
# NUMERICS
# A "numeric" is a double-precision floating-point number
5 # 5
class(5) # "numeric"
# Again, everything in R is a vector;
# you can make a numeric vector with more than one element
c(3,3,3,2,2,1) # 3 3 3 2 2 1
# You can use scientific notation too
5e4 # 50000
6.02e23 # Avogadro's number
1.6e-35 # Planck length
# You can also have infinitely large or small numbers
class(Inf)  # "numeric"
class(-Inf) # "numeric"
# You might use "Inf", for example, in integrate(dnorm, 3, Inf);
# this obviates Z-score tables.

# BASIC ARITHMETIC
# You can do arithmetic with numbers
# Doing arithmetic on a mix of integers and numerics gives you another numeric
10L + 66L # 76      # integer plus integer gives integer
53.2 - 4  # 49.2    # numeric minus numeric gives numeric
2.0 * 2L  # 4       # numeric times integer gives numeric
3L / 4    # 0.75    # integer over numeric gives numeric
3 %% 2    # 1       # the remainder of two numerics is another numeric
# Illegal arithmetic yeilds you a "not-a-number":
0 / 0 # NaN
class(NaN) # "numeric"
# You can do arithmetic on two vectors with length greater than 1,
# so long as the larger vector's length is an integer multiple of the smaller
c(1,2,3) + c(1,2,3) # 2 4 6
# Since a single number is a vector of length one, scalars are applied
# elementwise to vectors
(4 * c(1,2,3) - 2) / 2 # 1 3 5
# Except for scalars, use caution when performing arithmetic on vectors with
# different lengths. Although it can be done,
c(1,2,3,1,2,3) * c(1,2) # 1 4 3 2 2 6
# Matching lengths is better practice and easier to read
c(1,2,3,1,2,3) * c(1,2,1,2,1,2)

# CHARACTERS
# There's no difference between strings and characters in R
"Horatio" # "Horatio"
class("Horatio") # "character"
class('H') # "character"
# Those were both character vectors of length 1
# Here is a longer one:
c('alef', 'bet', 'gimmel', 'dalet', 'he')
# =>
# "alef"   "bet"    "gimmel" "dalet"  "he"
length(c("Call","me","Ishmael")) # 3
# You can do regex operations on character vectors:
substr("Fortuna multis dat nimis, nulli satis.", 9, 15) # "multis "
gsub('u', 'ø', "Fortuna multis dat nimis, nulli satis.") # "Fortøna møltis dat nimis, nølli satis."
```

```r
# R has several built-in character vectors:
letters
# =>
#  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
# [20] "t" "u" "v" "w" "x" "y" "z"
month.abb # "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

# LOGICALS
# In R, a "logical" is a boolean
class(TRUE) # "logical"
class(FALSE)    # "logical"
# Their behavior is normal
TRUE == TRUE    # TRUE
TRUE == FALSE   # FALSE
FALSE != FALSE  # FALSE
FALSE != TRUE   # TRUE
# Missing data (NA) is logical, too
class(NA)   # "logical"
# Use | and & for logic operations.
# OR
TRUE | FALSE    # TRUE
# AND
TRUE & FALSE    # FALSE
# Applying | and & to vectors returns elementwise logic operations
c(TRUE,FALSE,FALSE) | c(FALSE,TRUE,FALSE) # TRUE TRUE FALSE
c(TRUE,FALSE,TRUE) & c(FALSE,TRUE,TRUE) # FALSE FALSE TRUE
# You can test if x is TRUE
isTRUE(TRUE)    # TRUE
# Here we get a logical vector with many elements:
c('Z', 'o', 'r', 'r', 'o') == "Zorro" # FALSE FALSE FALSE FALSE FALSE
c('Z', 'o', 'r', 'r', 'o') == "Z" # TRUE FALSE FALSE FALSE FALSE

# FACTORS
# The factor class is for categorical data
# Factors can be ordered (like childrens' grade levels) or unordered (like gender)
factor(c("female", "female", "male", NA, "female"))
#  female female male   <NA>   female
# Levels: female male
# The "levels" are the values the categorical data can take
# Note that missing data does not enter the levels
levels(factor(c("male", "male", "female", NA, "female"))) # "female" "male"
# If a factor vector has length 1, its levels will have length 1, too
length(factor("male")) # 1
length(levels(factor("male"))) # 1
# Factors are commonly seen in data frames, a data structure we will cover later
data(infert) # "Infertility after Spontaneous and Induced Abortion"
levels(infert$education) # "0-5yrs"  "6-11yrs" "12+ yrs"

# NULL
# "NULL" is a weird one; use it to "blank out" a vector
class(NULL) # NULL
parakeet = c("beak", "feathers", "wings", "eyes")
parakeet
# =>
```

```r
# [1] "beak"     "feathers" "wings"    "eyes"
parakeet <- NULL
parakeet
# =>
# NULL

# TYPE COERCION
# Type-coercion is when you force a value to take on a different type
as.character(c(6, 8)) # "6" "8"
as.logical(c(1,0,1,1)) # TRUE FALSE  TRUE  TRUE
# If you put elements of different types into a vector, weird coercions happen:
c(TRUE, 4) # 1 4
c("dog", TRUE, 4) # "dog"  "TRUE" "4"
as.numeric("Bilbo")
# =>
# [1] NA
# Warning message:
# NAs introduced by coercion

# Also note: those were just the basic data types
# There are many more data types, such as for dates, time series, etc.



##################################################
# Variables, loops, if/else
##################################################

# A variable is like a box you store a value in for later use.
# We call this "assigning" the value to the variable.
# Having variables lets us write loops, functions, and if/else statements

# VARIABLES
# Lots of way to assign stuff:
x = 5 # this is possible
y <- "1" # this is preferred
TRUE -> z # this works but is weird

# LOOPS
# We've got for loops
for (i in 1:4) {
  print(i)
}
# We've got while loops
a <- 10
while (a > 4) {
    cat(a, "...", sep = "")
    a <- a - 1
}
# Keep in mind that for and while loops run slowly in R
# Operations on entire vectors (i.e. a whole row, a whole column)
# or apply()-type functions (we'll discuss later) are preferred

# IF/ELSE
```

```r
# Again, pretty standard
if (4 > 3) {
    print("4 is greater than 3")
} else {
    print("4 is not greater than 3")
}
# =>
# [1] "4 is greater than 3"

# FUNCTIONS
# Defined like so:
jiggle <- function(x) {
    x = x + rnorm(1, sd=.1) #add in a bit of (controlled) noise
    return(x)
}
# Called like any other R function:
jiggle(5)   # 5±. After set.seed(2716057), jiggle(5)==5.005043



################################################################################
# Data structures: Vectors, matrices, data frames, and arrays
################################################################################

# ONE-DIMENSIONAL

# Let's start from the very beginning, and with something you already know: vectors.
vec <- c(8, 9, 10, 11)
vec #  8  9 10 11
# We ask for specific elements by subsetting with square brackets
# (Note that R starts counting from 1)
vec[1]      # 8
letters[18] # "r"
LETTERS[13] # "M"
month.name[9]   # "September"
c(6, 8, 7, 5, 3, 0, 9)[3]   # 7
# We can also search for the indices of specific components,
which(vec %% 2 == 0)    # 1 3
# grab just the first or last few entries in the vector,
head(vec, 1)    # 8
tail(vec, 2)    # 10 11
# or figure out if a certain value is in the vector
any(vec == 10) # TRUE
# If an index "goes over" you'll get NA:
vec[6]  # NA
# You can find the length of your vector with length()
length(vec) # 4
# You can perform operations on entire vectors or subsets of vectors
vec * 4 # 16 20 24 28
vec[2:3] * 5    # 25 30
any(vec[2:3] == 8) # FALSE
# and R has many built-in functions to summarize vectors
mean(vec)   # 9.5
var(vec)    # 1.666667
```

```r
sd(vec)     # 1.290994
max(vec)    # 11
min(vec)    # 8
sum(vec)    # 38
# Some more nice built-ins:
5:15    # 5  6  7  8  9 10 11 12 13 14 15
seq(from=0, to=31337, by=1337)
# =>
#  [1]     0  1337  2674  4011  5348  6685  8022  9359 10696 12033 13370 14707
# [13] 16044 17381 18718 20055 21392 22729 24066 25403 26740 28077 29414 30751

# TWO-DIMENSIONAL (ALL ONE CLASS)

# You can make a matrix out of entries all of the same type like so:
mat <- matrix(nrow = 3, ncol = 2, c(1,2,3,4,5,6))
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
# Unlike a vector, the class of a matrix is "matrix", no matter what's in it
class(mat) # => "matrix"
# Ask for the first row
mat[1,] # 1 4
# Perform operation on the first column
3 * mat[,1] # 3 6 9
# Ask for a specific cell
mat[3,2]    # 6

# Transpose the whole matrix
t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6

# Matrix multiplication
mat %*% t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]   17   22   27
# [2,]   22   29   36
# [3,]   27   36   45

# cbind() sticks vectors together column-wise to make a matrix
mat2 <- cbind(1:4, c("dog", "cat", "bird", "dog"))
mat2
# =>
#      [,1] [,2]
# [1,] "1"  "dog"
# [2,] "2"  "cat"
# [3,] "3"  "bird"
# [4,] "4"  "dog"
```

```r
class(mat2) # matrix
# Again, note what happened!
# Because matrices must contain entries all of the same class,
# everything got converted to the character class
c(class(mat2[,1]), class(mat2[,2]))

# rbind() sticks vectors together row-wise to make a matrix
mat3 <- rbind(c(1,2,4,5), c(6,7,0,4))
mat3
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    4    5
# [2,]    6    7    0    4
# Ah, everything of the same class. No coercions. Much better.

# TWO-DIMENSIONAL (DIFFERENT CLASSES)

# For columns of different types, use a data frame
# This data structure is so useful for statistical programming,
# a version of it was added to Python in the package "pandas".

students <- data.frame(c("Cedric","Fred","George","Cho","Draco","Ginny"),
                       c(3,2,2,1,0,-1),
                       c("H", "G", "G", "R", "S", "G"))
names(students) <- c("name", "year", "house") # name the columns
class(students) # "data.frame"
students
# =>
#     name year house
# 1 Cedric    3     H
# 2   Fred    2     G
# 3 George    2     G
# 4    Cho    1     R
# 5  Draco    0     S
# 6  Ginny   -1     G
class(students$year)    # "numeric"
class(students[,3]) # "factor"
# find the dimensions
nrow(students)  # 6
ncol(students)  # 3
dim(students)   # 6 3
# The data.frame() function converts character vectors to factor vectors
# by default; turn this off by setting stringsAsFactors = FALSE when
# you create the data.frame
?data.frame

# There are many twisty ways to subset data frames, all subtly unalike
students$year   # 3  2  2  1  0 -1
students[,2]    # 3  2  2  1  0 -1
students["year"]   # 3  2  2  1  0 -1

# An augmented version of the data.frame structure is the data.table
# If you're working with huge or panel data, or need to merge a few data
# sets, data.table can be a good choice. Here's a whirlwind tour:
```

```
install.packages("data.table") # download the package from CRAN
require(data.table) # load it
students <- as.data.table(students)
students # note the slightly different print-out
# =>
#      name year house
# 1: Cedric    3     H
# 2:   Fred    2     G
# 3: George    2     G
# 4:    Cho    1     R
# 5:  Draco    0     S
# 6:  Ginny   -1     G
students[name=="Ginny"] # get rows with name == "Ginny"
# =>
#     name year house
# 1: Ginny   -1     G
students[year==2] # get rows with year == 2
# =>
#      name year house
# 1:   Fred    2     G
# 2: George    2     G
# data.table makes merging two data sets easy
# let's make another data.table to merge with students
founders <- data.table(house=c("G","H","R","S"),
                       founder=c("Godric","Helga","Rowena","Salazar"))
founders
# =>
#    house founder
# 1:     G  Godric
# 2:     H   Helga
# 3:     R  Rowena
# 4:     S Salazar
setkey(students, house)
setkey(founders, house)
students <- founders[students] # merge the two data sets by matching "house"
setnames(students, c("house","houseFounderName","studentName","year"))
students[,order(c("name","year","house","houseFounderName")), with=F]
# =>
#    studentName year house houseFounderName
# 1:        Fred    2     G           Godric
# 2:      George    2     G           Godric
# 3:       Ginny   -1     G           Godric
# 4:      Cedric    3     H            Helga
# 5:         Cho    1     R           Rowena
# 6:       Draco    0     S          Salazar

# data.table makes summary tables easy
students[,sum(year),by=house]
# =>
#    house V1
# 1:     G  3
# 2:     H  3
# 3:     R  1
# 4:     S  0
```

```
# To drop a column from a data.frame or data.table,
# assign it the NULL value
students$houseFounderName <- NULL
students
# =>
#    studentName year house
# 1:        Fred    2     G
# 2:      George    2     G
# 3:       Ginny   -1     G
# 4:      Cedric    3     H
# 5:         Cho    1     R
# 6:       Draco    0     S

# Drop a row by subsetting
# Using data.table:
students[studentName != "Draco"]
# =>
#    house studentName year
# 1:     G        Fred    2
# 2:     G      George    2
# 3:     G       Ginny   -1
# 4:     H      Cedric    3
# 5:     R         Cho    1
# Using data.frame:
students <- as.data.frame(students)
students[students$house != "G",]
# =>
#   house houseFounderName studentName year
# 4     H            Helga      Cedric    3
# 5     R           Rowena         Cho    1
# 6     S          Salazar       Draco    0

# MULTI-DIMENSIONAL (ALL ELEMENTS OF ONE TYPE)

# Arrays creates n-dimensional tables
# All elements must be of the same type
# You can make a two-dimensional table (sort of like a matrix)
array(c(c(1,2,4,5),c(8,9,3,6)), dim=c(2,4))
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    4    8    3
# [2,]    2    5    9    6
# You can use array to make three-dimensional matrices too
array(c(c(c(2,300,4),c(8,9,0)),c(c(5,60,0),c(66,7,847))), dim=c(3,2,2))
# =>
# , , 1
#
#      [,1] [,2]
# [1,]    2    8
# [2,]  300    9
# [3,]    4    0
#
# , , 2
```

```
#
#      [,1] [,2]
# [1,]    5   66
# [2,]   60    7
# [3,]    0  847

# LISTS (MULTI-DIMENSIONAL, POSSIBLY RAGGED, OF DIFFERENT TYPES)

# Finally, R has lists (of vectors)
list1 <- list(time = 1:40)
list1$price = c(rnorm(40,.5*list1$time,4)) # random
list1
# You can get items in the list like so
list1$time # one way
list1[["time"]] # another way
list1[[1]] # yet another way
# =>
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
# [34] 34 35 36 37 38 39 40
# You can subset list items like any other vector
list1$price[4]

# Lists are not the most efficient data structure to work with in R;
# unless you have a very good reason, you should stick to data.frames
# Lists are often returned by functions that perform linear regressions

##################################################
# The apply() family of functions
##################################################

# Remember mat?
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
# Use apply(X, MARGIN, FUN) to apply function FUN to a matrix X
# over rows (MAR = 1) or columns (MAR = 2)
# That is, R does FUN to each row (or column) of X, much faster than a
# for or while loop would do
apply(mat, MAR = 2, jiggle)
# =>
#      [,1] [,2]
# [1,]    3   15
# [2,]    7   19
# [3,]   11   23
# Other functions: ?lapply, ?sapply

# Don't feel too intimidated; everyone agrees they are rather confusing

# The plyr package aims to replace (and improve upon!) the *apply() family.
install.packages("plyr")
require(plyr)
```

```
?plyr



#########################
# Loading data
#########################

# "pets.csv" is a file on the internet
# (but it could just as easily be be a file on your own computer)
pets <- read.csv("http://learnxinyminutes.com/docs/pets.csv")
pets
head(pets, 2) # first two rows
tail(pets, 1) # last row

# To save a data frame or matrix as a .csv file
write.csv(pets, "pets2.csv") # to make a new .csv file
# set working directory with setwd(), look it up with getwd()

# Try ?read.csv and ?write.csv for more information



#########################
# Statistical Analysis
#########################

# Linear regression!
linearModel <- lm(price  ~ time, data = list1)
linearModel # outputs result of regression
# =>
# Call:
# lm(formula = price ~ time, data = list1)
#
# Coefficients:
# (Intercept)         time
#      0.1453       0.4943
summary(linearModel) # more verbose output from the regression
# =>
# Call:
# lm(formula = price ~ time, data = list1)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -8.3134 -3.0131 -0.3606  2.8016 10.3992
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  0.14527    1.50084   0.097    0.923
# time         0.49435    0.06379   7.749 2.44e-09 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 4.657 on 38 degrees of freedom
```

```
# Multiple R-squared:  0.6124,  Adjusted R-squared:  0.6022
# F-statistic: 60.05 on 1 and 38 DF,  p-value: 2.44e-09
coef(linearModel) # extract estimated parameters
# =>
# (Intercept)        time
#   0.1452662   0.4943490
summary(linearModel)$coefficients # another way to extract results
# =>
#              Estimate Std. Error    t value     Pr(>|t|)
# (Intercept) 0.1452662 1.50084246 0.09678975 9.234021e-01
# time        0.4943490 0.06379348 7.74920901 2.440008e-09
summary(linearModel)$coefficients[,4] # the p-values
# =>
#  (Intercept)         time
# 9.234021e-01 2.440008e-09

# GENERAL LINEAR MODELS
# Logistic regression
set.seed(1)
list1$success = rbinom(length(list1$time), 1, .5) # random binary
glModel <- glm(success  ~ time, data = list1,
    family=binomial(link="logit"))
glModel # outputs result of logistic regression
# =>
# Call:  glm(formula = success ~ time,
#   family = binomial(link = "logit"), data = list1)
#
# Coefficients:
# (Intercept)         time
#     0.17018     -0.01321
#
# Degrees of Freedom: 39 Total (i.e. Null);  38 Residual
# Null Deviance:        55.35
# Residual Deviance: 55.12   AIC: 59.12
summary(glModel) # more verbose output from the regression
# =>
# Call:
# glm(formula = success ~ time,
#   family = binomial(link = "logit"), data = list1)

# Deviance Residuals:
#    Min      1Q  Median      3Q     Max
# -1.245  -1.118  -1.035   1.202   1.327
#
# Coefficients:
#             Estimate Std. Error z value Pr(>|z|)
# (Intercept)  0.17018    0.64621   0.263    0.792
# time        -0.01321    0.02757  -0.479    0.632
#
# (Dispersion parameter for binomial family taken to be 1)
#
#     Null deviance: 55.352  on 39  degrees of freedom
# Residual deviance: 55.121  on 38  degrees of freedom
# AIC: 59.121
```

```
#
# Number of Fisher Scoring iterations: 3


#########################
# Plots
#########################

# BUILT-IN PLOTTING FUNCTIONS
# Scatterplots!
plot(list1$time, list1$price, main = "fake data")
# Plot regression line on existing plot
abline(linearModel, col = "red")
# Get a variety of nice diagnostics
plot(linearModel)
# Histograms!
hist(rpois(n = 10000, lambda = 5), col = "thistle")
# Barplots!
barplot(c(1,4,5,1,2), names.arg = c("red","blue","purple","green","yellow"))

# GGPLOT2
# But these are not even the prettiest of R's plots
# Try the ggplot2 package for more and better graphics
install.packages("ggplot2")
require(ggplot2)
?ggplot2
pp <- ggplot(students, aes(x=house))
pp + geom_histogram()
ll <- as.data.table(list1)
pp <- ggplot(ll, aes(x=time,price))
pp + geom_point()
# ggplot2 has excellent documentation (available http://docs.ggplot2.org/current/)
```

How do I get R?

- Get R and the R GUI from http://www.r-project.org/
- RStudio is another GUI