Discovering Econometrics with R: Day 1

Richard Bluhm

August 5, 2013

What is *R*?

- *R* is a statistical programming language based on *S*
- It's open source and completely *free*! Yes, *free*!
- *R* `1.0` was released in 2000, now version `3.0.1` in Jun 2013
- Very quickly becoming a popular alternative to *expensive* proprietary software like SAS, Stata, EViews and Matlab
- Massive online user base contributing new programs *every day*
- Heavily used in Biostatistics, Medicine and Computing
- Quickly becoming more popular in Econometrics and the Social Sciences (particularly in the US)
- Somewhat dated, but extremely flexible language, ability to interface with most major languages (C^{++}, Python, etc.) and database types (SQL, Hana, Hadoop, etc.).

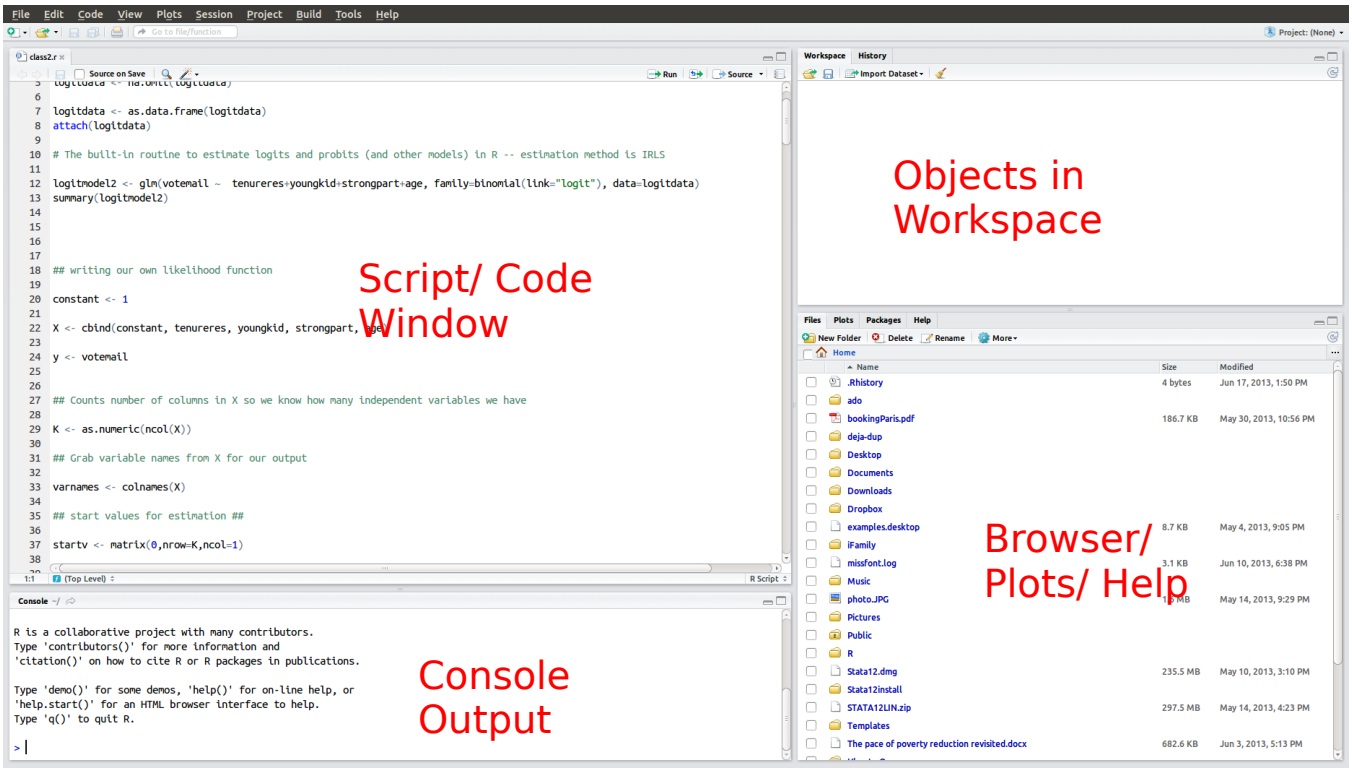
What can you do with *R*?

- Load and manipulate data from almost any source
- Make descriptive statistics and graphs
- Fit all sorts of statistical and econometric models (including our favorite regression models)
- Make advanced graphs of statistical results
- Easily write simulations for statistical or other types of models
- Load user-written packages that implement new things
- Use a fully fledged matrix/ vector language
- Write your own functions/ programs and share them
- *R* is possibly the most flexible fully-developed statistical language existing today (but new ones coming *e.g. Julia*)
- *R* is *open source* so you can learn from other people's code

How to install R?

- Get the latest R version for your operating system (runs on all major platforms) from: `http://www.r-project.org`
- Install the software, now you have *R* for the console without a Graphical User Interface (GUI)/ Integrated Development Environment (IDE)
- Get the latest version of RStudio for your operating system (runs on all major platforms and is our recommended IDE) from: `http://www.rstudio.com`
- Install RStudio, now you have a pretty GUI and very sleek development platform
- Note to Linux users: you may have *R* available directly from your package manager. On Ubuntu type `sudo apt-get install r-base` at the terminal.

The look and feel of R and RStudio



A few more points before we get started

- *R* is an object-oriented programming language build around specific and generic functions. It relies on the functional programming paradigm.
- For example, the function `lm()` which we will use throughout the course estimates a linear model and then saves lots of objects that other functions can use afterwards
- Most *R* functions are *polymorphic* generic functions: they change depending on what objects they are being called on
- For example, `summary()` gives very different output depending on what you ask it to summarize
- Every operation is a function. Even simple math (e.g. 1+1) and matrix calculations (e.g. X') are in fact functions.

A first look at using R

- Open RStudio and just type simple math commands at the console prompt
- Try `1+1`
- Try `a <- 1`, `b <- 2`, and `a + b`
- Now use the code/ scripting window and type the same things on new lines
- Two important keyboard shortcuts:
  - CTRL + SHIFT + ENTER: run all lines
  - CTRL + ALT + B: run until current line
  - More here `http://www.rstudio.com/ide/docs/using/keyboard_shortcuts`
- Verify that the console output is the same, look at the two new objects in the workspace

Basic R objects

- R knows five types of atomic objects:
  - numeric [e.g. 1.23456]
  - integer [e.g. 1, typed as 1L]
  - complex [e.g. a + bi, real + imaginary]
  - boolean: logical [e.g. TRUE or FALSE, T or F]
  - strings: character [e.g. "Hello World!"]
- The data types collecting these atomic objects are:
  - vectors: several elements of a single atomic type (*R* does not have scalars, they are 1-element vectors)
  - matrices: collections of equal-length vectors
  - factors: categorical data (ordered, unordered)
  - data frames: a data set, collections of equal-length vectors of *different types*
  - lists: collections of unequal-length vectors of *different types*

Some notation

- When you see the $>$ at the beginning of a line, that means I am showing you the code *and* the console output from that code on the next line(s). Do not copy the $>$ sign when you follow the examples. When I omit the $>$ sign, you can copy the line(s) directly.
- Assignment: We will not use the equal sign (=) to assign content to a new vector/ variable etc.; somewhat eclectically *R* uses `<-` to assign something on the *right* to something the *left*. We reserve the equal sign for input to functions and logical comparisons (e.g. double equal).[1]
- Concatenation/ Combining: `c(1, 2, 3)` creates a vector that combines the element 1, 2, and 3. This is probably the most often used *R* function after the assignment function.

[1] Click here to learn why.

Vectors of atomic objects

```
> x <- c(1.1, 2.2, 3.3)
> is.numeric(x)
[1] TRUE
> x <- c(1L, 2L, 3L)
> is.integer(x)
[1] TRUE
> x <- c(1+0i, 2+4i, 3+6i)
> is.complex(x)
[1] TRUE
> x <- c(TRUE, FALSE, TRUE)
> is.logical(x)
[1] TRUE
> x <- c("I", "like", "R")
> is.character(x)
[1] TRUE
```

What data type is this vector? `x <- c("R", 2, FALSE)`

Operations with vectors (I)

You can call most functions on a vector, such as functions for the type of vector, like `is.numeric()` or most statistical functions.

```
x <- 1:10
mean(x)
sd(x)
var(x)
length(x)
sum(x)
```

We can and often will work with subsets of vectors:

```
> x[1:5]
[1] 1 2 3 4 5
> mean(x[1:5])
[1] 3
```

Operations with vectors (II)

We can do all sorts of simple math with vectors. Note that *R* by default does vector operations element-wise, for vector algebra we have to use a different notation (advanced use).

```
> a <- 1:10
> b <- 11:20
> a + b
 [1] 12 14 16 18 20 22 24 26 28 30
```

Recycling: if some vectors are too short, many operations make them equal length by repeating the shorter vector(s)

```
> a <- c(1,1,1,1)
> b <- c(2,4)
> a * b
[1] 2 4 2 4
```

Matrices (I)

Matrices are collections of vectors. They have dimensions $r \times k$ where r is the number of rows and k is the number of columns. The input vectors need to be of equal length and equal type.

```
> x1 <- 1:3; x2 <- 4:6; x3 <- 7:9
> x <- cbind(x1,x2,x3)
> x
     x1 x2 x3
[1,]  1  4  7
[2,]  2  5  8
[3,]  3  6  9
> dim(x) # returns r and k
[1] 3 3
> length(x) # returns r times k
[1] 9
```

Matrices (II)

Matrices are built up column-wise. We can take a long vector and break it up into rows and columns.

```
> mat <- matrix(1:8, nrow = 2, ncol = 4)
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

Or as in the example before, we combine existing vectors column-wise `cbind()` or row-wise `rbind()`:

```
> x1 <- 1:3; x2 <- 4:6
> x <- cbind(x1,x2); dim(x)
[1] 3 2
> x <- rbind(x1,x2); dim(x)
[1] 2 3
```

Matrices (III)

Simple math operators also perform all matrix operations element-wise. For matrix algebra we need to use special functions, such as %*% for matrix multiplication or `t()` for the transpose.

```
> m <- matrix(1:9, 3, 3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m*m
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

Factors (I)

Factors store categorical data that may be ordered or unordered. Like “yes” and “no”, or “disagree”, “neutral” and “agree”, or “BMW”, “Mercedes”, and “Volkswagen”.

```
> 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
```

Factors (II)

Often factors have an intrinsic order (for example a Likert scale). The `levels` option makes sure the factors are not ordered on first come first serve basis, but how you want. Some statistical functions require the use of `ordered()` instead of `factor()`.

```
> x <- factor(c("agree", "agree", "neutral", "disagree"),
+             levels = c("disagree", "neutral", "agree"))
> x
[1] agree    agree    neutral  disagree
Levels: disagree neutral agree
> unclass(x) # shows how it's really stored
[1] 3 3 2 1
attr(,"levels")
[1] "disagree" "neutral"  "agree"
```

Data frames

Data frames are the most important data type for statistical analysis. They can hold all atomic types provided they are in vectors of equal length. Think of an excel sheet/ table that records different characteristics for different units of observations.

```
> x <- data.frame(id = 1:5, male = c(T, T, F, F, F),
+                 age = c(29, 45, 23, 62, 59))
> x
  id  male age
1  1  TRUE  29
2  2  TRUE  45
3  3 FALSE  23
4  4 FALSE  62
5  5 FALSE  59
```

Lists

Unlike data frames (which are special lists), a list can hold any type of vector consisting of different atomic elements, no matter what length.

```
mylist <- list(beers = c("Pils", "Lager",
+                         "Pale Ale", "Dark Ale"),
+              cars = c("BMW", "Mercedes", "Volkswagen"))
> mylist
$beers
[1] "Pils"     "Lager"    "Pale Ale" "Dark Ale"
$cars
[1] "BMW"        "Mercedes"   "Volkswagen"
```

While lists are extremely useful, we will try to avoid lists in this course where possible.

Naming objects

All *R* data objects can be assigned names with the `names()`, `colnames()` or `rownames()` functions. Typically we will not name every element of a vector or name each row and column of a matrix. For **data frames**, however, **column names** are really important. They correspond to the **variable name**. For example:

```
> x <- data.frame(id = 1:5, male = c(T, T, F, F, F),
+                 age = c(29, 45, 23, 62, 59))
> names(x) #print names
[1] "id"   "male" "age"
> # let's rename the first two
> names(x) <- c("personID", "mgender", "age")
> names(x)
[1] "personID" "mgender"   "age"
```

Missing and other special values

R has a few special symbols. Most importantly: missing values. Missing values can be of any atomic type: character, number, and so on. However, *R* also has designated signs for "not a number", "positive infinity", and "negative infinity". For example:

```
> x <- 1 / 0
> x
[1] Inf
> log(0)
[1] -Inf
> x <- c(1, NA, 3, 4)
> is.na(x)
[1] FALSE  TRUE FALSE FALSE
> x <- 0 / 0
> x
[1] NaN
```

What is a function?

Just like in math, e.g. $y = f(x)$, an *R* function receives one or multiple inputs, then does something with these inputs and returns something. For example, if we look at the help file (`?mean`) for the function `mean()`, it tell us what this function returns (duh) the arithmetic mean and what it expects as an input.

Generally the (somewhat cryptic) documentation provides:

- the name of the function and the package where it is located
- a short description of what it does
- a short description of the syntax
- a list of the required and optional arguments
- what the function returns and the data type returned
- references, other links and some example usage

Documentation for `mean()`

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An `R` object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

Your first *R* function

In this course we will mostly use built-in functions, but programming *R* functions is incredibly easy. Let's write a function that computes the mean of a vector (assuming there are no missing values). All we need to do is this:

```
# Define the function
mymean <- function(x) {
  mean <- sum(x)/length(x)
  return(mean)
}
# Create a new vector and evaluate
x <- c(1, 2, 3)
mymean(x)
```

It will give the same results as `mean()` and now shows up in your workspace on the top right.

Logical Operators (I)

Operator	Description
`<`	Test for less than
`<=`	Test for less than or equal to
`>`	Test for greater than
`>=`	Test for greater than or equal to
`==`	Test for equality
`!=`	Test for if not equality
`!x`	Boolean negation, for vectors
`x | y`	Boolean x OR y, for vectors
`x & y`	Boolean x AND y, for vectors
`x || y`	Boolean x OR y, for scalars
`x && y`	Boolean x AND y, for scalars
`isTRUE(x)`	Boolean test if X is TRUE, for scalars

Logical Operators (II): numeric input

```
> x <- c(1,2,3)
> y <- c(3,4,5)
> x < y # element-by-element comparison
[1] TRUE TRUE TRUE
> x <= y
[1] TRUE TRUE TRUE
> x > y
[1] FALSE FALSE FALSE
> x == y
[1] FALSE FALSE FALSE
> x != y
[1] TRUE TRUE TRUE
> mean(x) == mean(y) # also works with results
[1] FALSE
```

Logical Operators (III): boolean input

```
> x <- c(TRUE, FALSE, TRUE)
> y <- c(TRUE, TRUE, TRUE)
> !x # negation, set theory: complement
[1] FALSE  TRUE FALSE
> x | y # x or y is true, set theory: union
[1] TRUE TRUE TRUE
>  x & y # x and y is true, set theory: intersection
[1]  TRUE FALSE  TRUE
> isTRUE(y) # aha, what's going on here?
[1] FALSE
> x[2] && y[2] # scalar
[1] FALSE
> x[2]==T || y[2]==F # scalar, can stack conditions
[1] FALSE
```

Before we load data I: setting the working dir

When you open *R* and *RStudio* it will work in a default directory. To see this directory, type

```
> getwd()
[1] "/home/richard"
```

To specify a new directory, type (for example)

```
setwd("/home/richard/Desktop")
```

Forwards slashes are familiar to UNIX/MAC users. On Windows *you also have use forward slashes* or *escape the backwards slashes* to set a path like `"C:\\Users\\Name\\"`. I recommend you make a new folder on your desktop called "summerschool" and place all materials there. Then set this as your working directory.

Before we load data II: clearing objects and saving work

You may want to start each *R* script with a clean slate. To empty the memory and remove all objects, type

```
> rm(list = ls())
```

To save your current *R* script (`*.R`), just click on "save" in the top left of *RStudio*. To save one or multiple objects (e.g. data frames) from your current workspace as an `*.Rdata` file, type

```
> x <- c(1, 2, 3)
> save(x, file = "X.RData")
```

You can save more than one object by separating them with commas, e.g. `save(x, y, file =...)`. To load them again, use

```
> load(file = "X.RData")
```

Let's start a new script and load a data set

We'll now work through our first example with a real data set and save our work in the end. You can open a new *R* script by clicking on the plus button or CTRL+SHIFT+N. Then copy and paste this code while filling in your working directory:

```
# In class example, day 1

# Clear workspace
rm(list=ls())

# Set working dir, your path here
setwd("/home/richard/Desktop/summerschool/")
```

Loading data from CSV files

Typically we do not want to create a data frame by hand but load data from a comma-separated text file or other formats. In this course we will mostly use *CSV* or *Rdata* files, but *R* can read lots of formats.

```
> url <- "http://www.stern.nyu.edu/~wgreene/Text/"
> df <- read.csv(paste0(url,"Edition7/TableF4-3.csv"))
> df <- df[,1:5]
> head(df,5)
       BOX MPRATING BUDGET STARPOWR SEQUEL
1 19167085        4   28.0    19.83      0
2 63106589        2  150.0    32.69      1
3  5401605        4   37.4    15.69      0
4 67528882        3  200.0    23.62      1
5 26223128        2  150.0    19.02      0
```

The `str()` and `summary()` functions

When we load a data set, the first thing we typically want to know is what data is inside and how it has been recorded. Here, the `str()` and `summary()` functions are extremely useful.

```
> str(df[,1:3])
'data.frame':   62 obs. of  3 variables:
 $ BOX     : int  19167085 63106589 5401605 67528882 ...
 $ MPRATING: int  4 2 4 3 2 3 3 3 3 4 ...
 $ BUDGET  : num  28 150 37.4 200 150 37 130 80 40 ...
> summary(df[,1:3])
      BOX               MPRATING         BUDGET
 Min.   :  511920   Min.   :1.000   Min.   :  5.00
 1st Qu.: 6956492   1st Qu.:2.000   1st Qu.: 30.50
 Median :16930926   Median :3.000   Median : 37.40
 Mean   :20720651   Mean   :2.968   Mean   : 53.29
 3rd Qu.:26696144   3rd Qu.:4.000   3rd Qu.: 60.00
 Max.   :70950500   Max.   :4.000   Max.   :200.00
```

Subsetting data frames

Much like vectors, we can subset a data frame by specifying which rows and columns we would like to work with. We always need to specify two dimensions or leave one blank. Matrix subsetting works just the same way. Since the variables in data frames have names, we can refer to them directly:

```
> df[1:5,"BOX"]
[1] 19167085 63106589  5401605 67528882 26223128
> df$BOX[1:5]
[1] 19167085 63106589  5401605 67528882 26223128
> df[1:5,1]
[1] 19167085 63106589  5401605 67528882 26223128
```

In each case, we ask *R* to return the first five rows of the variable "BOX" as a numeric vector.

Generating and replacing variables

The box office returns are measured in dollars. Suppose we would like to change the scale to millions of dollars instead. We would need to either replace "BOX" with its rescaled counterpart or create a new variable.

```
> df$BOXM <- df$BOX / 10^6
> df[,"BOXM"] <- df[,"BOX"] / 10^6
> summary(df[,"BOXM"])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.5119  6.9560 16.9300 20.7200 26.7000 70.9500
```

Both lines do the same with a different syntax. Note how you *always need to specify the data frame* you are using even on the right hand side. Otherwise *R* will search for a vector named "BOX" in the workspace and not in the data frame!

Recoding factor variables

The variable "MPRATING" is an integer in the raw data, but in fact signifies the MPAA rating of the movie. The codes are 1=G, 2=PG, 3=PG13, and 4=R. We need to create a new factor.

```
> df$MPAA <- factor(df$MPRATING, levels = c(1,2,3,4),
+                   labels = c("G", "PG", "PG13", "R"))
> summary(df$MPAA)
   G   PG PG13    R
   2   15   28   17
```

For tables, we sometimes want to split up other variables.

```
> df$BOXcat <- cut(df$BOXM, breaks=c(0,10,20,30,Inf))
> summary(df$BOXcat)
  (0,10]  (10,20]  (20,30] (30,Inf]
      19       17       14       12
```

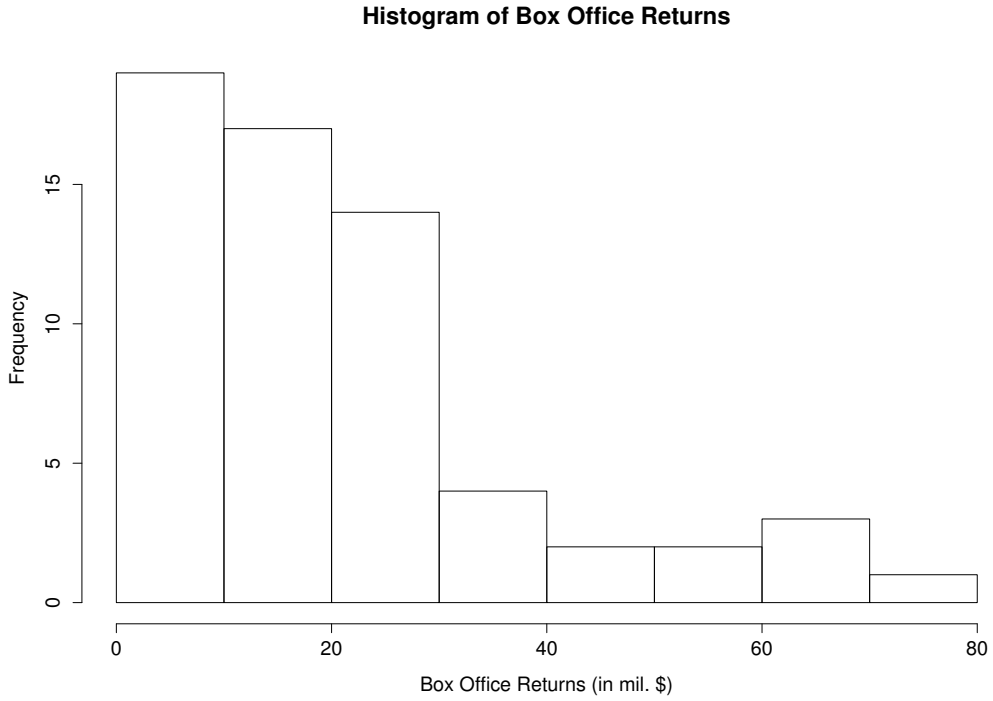
Simple tables

R has many tabulating capabilities. For now, I am only introducing three basic types: 1. one-way frequency tables, 2. two-way frequency tables and 3. tables of proportions

```
> table(df$MPAA)
   G   PG PG13    R
   2   15   28   17
> table(df$BOXcat,df$MPAA)
          G PG PG13 R
  (0,10]  0  3    8 8
  (10,20] 2  4    8 3
  (20,30] 0  4    6 4
  (30,Inf] 0  4    6 2
> prop.table(table(df$BOXcat))
   (0,10]    (10,20]    (20,30]   (30,Inf]
0.3064516 0.2741935 0.2258065 0.1935484
```

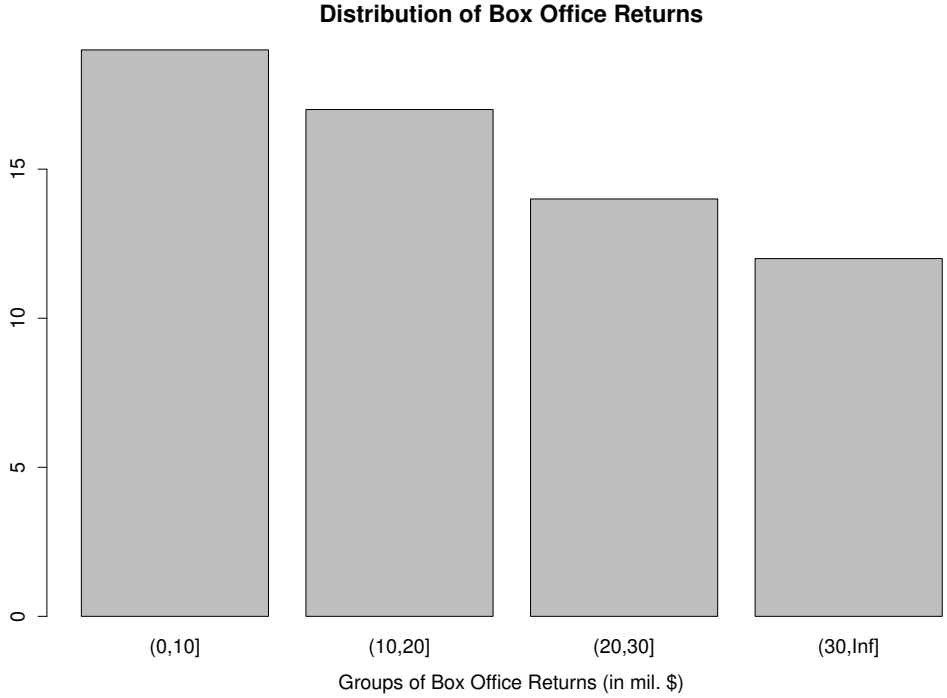
Graphing distributions: Histograms

```
hist(df$BOXM, main="Histogram of Box Office Returns",
     xlab="Box Office Returns (in mil. $)")
```



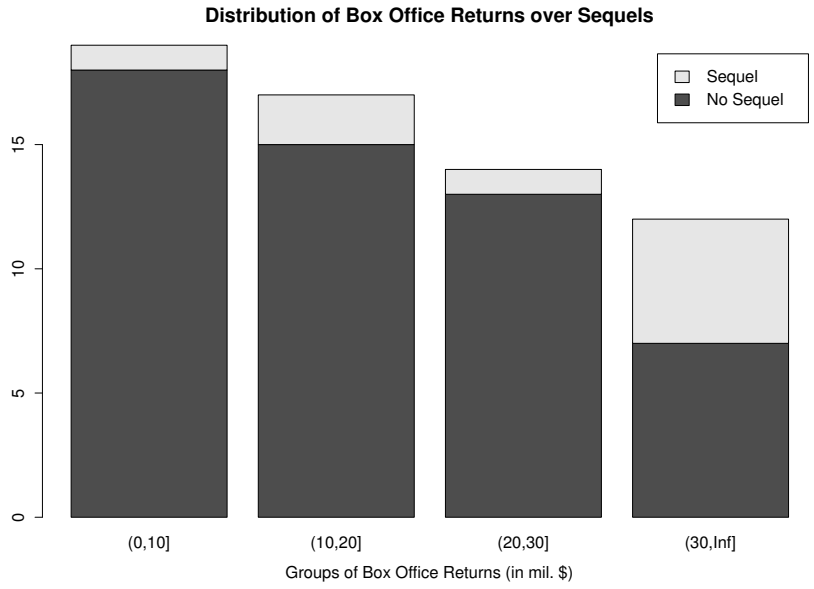
Graphing distributions: Bar plots (I)

```
freq <- table(df$BOXcat)
barplot(freq, main="Distribution of Box Office Returns",
        xlab="Groups of Box Office Returns (in mil. $)")
```



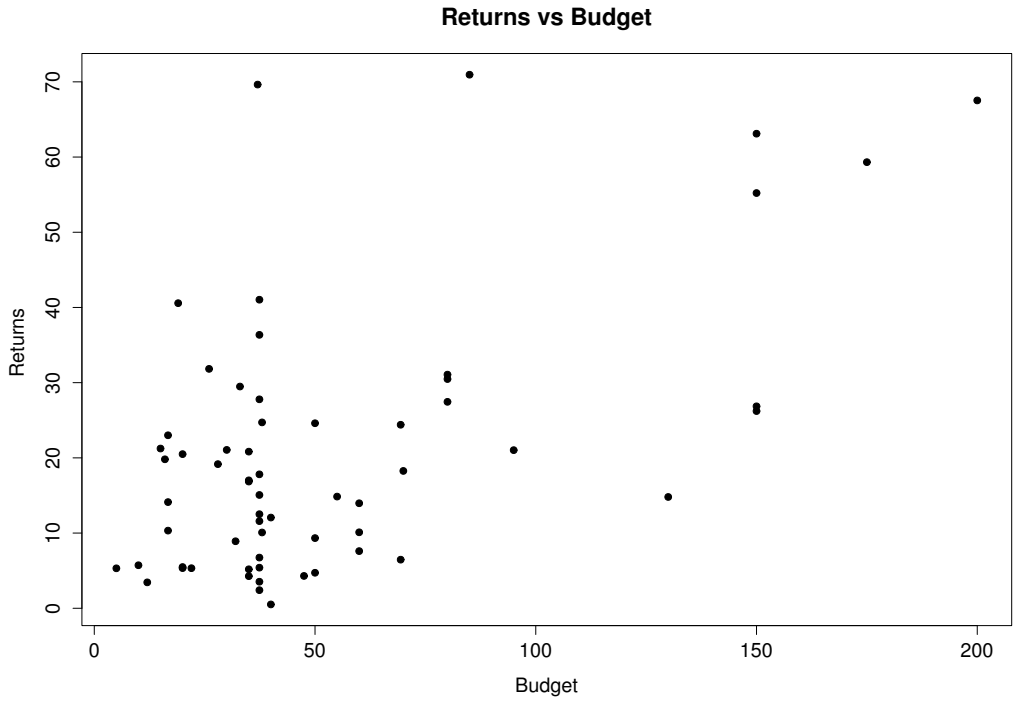
Graphing distributions: Bar plots (II)

```
df$SEQUEL <- factor(df$SEQUEL, levels = c(0,1),
            labels = c("No Sequel","Sequel"))
freq <- table(df$SEQUEL,df$BOXcat)
barplot(freq, main="Box Office Returns over Sequels",
        xlab="Groups of Box Office Returns (in mil. $)",
        legend = rownames(freq))
```



Graphing distributions: Two-way graphs

```
plot(df$BUDGET, df$BOXM, main="Returns vs Budget",
     xlab="Budget", ylab="Returns ", pch=19)
```



Saving the example

Let's save the code we have written so far by clicking on "save" in *RStudio*. You may call the file "Example1.R".

You may also want to save the data frame we have created, so that next time you don't have to run the entire script to continue working with the movie data in another example. Just add this to the bottom of your file

```
>  save(df, file = "MovieData.RData")
```

run and save the script again. Next time we can open the data with `load(file = "MovieData.RData")` and the data frame `df` will show up in your workspace again.

Graphing capabilities

- *R* has a great deal of graphing capabilities
- Most of the more advanced (and prettier) graphs come from user-written packages like `lattice` or `ggplot2`
- For simplicity, we only use the graphing capabilities in base *R*, but feel free to check out the add-ons on your own time.
- `http://www.cookbook-r.com/Graphs/` has a lot of examples for `ggplot2`
- `http://www.statmethods.net/advgraphs/trellis.html` has some examples for the Trellis graphics system in the `lattice` package
- Both packages have a dedicated book demonstrating their capabilities

Help me!

- OMG! This is so much to take in, am I expected to know all this by tomorrow? Short answer: No.
- Long answer: We will build up your *R* knowledge each day with new applications, the learning curve in the beginning is steep but I am there to help. Later it becomes easier.
- *R* can also help you, for any function just type: `help(function)` or `?function`
- *R* can show you examples, for most functions you can use: `example(function)`
- Otherwise, Google it! Many people are learning *R* and there are countless FAQs and tutorials out there.
- An important website for slightly more complicated questions is `http://stackoverflow.com/questions/tagged/r`, it's for many languages but has 31,911 Q&As on *R*!

Homework: Part I

Write a function that returns the sample variance of a numeric vector. Recall that the sample variance is defined as the sum of squared deviations divided by $(N-1)$, that is

$$\widehat{\text{var}}(X) \equiv S_X^2 = \frac{1}{N-1}\sum_i^N (X_i - \bar{X})^2$$

Use following shell, you may use `mean()`, `sum()` and `length()`. Test it by comparing your result with *R*'s built-in `var()` function.

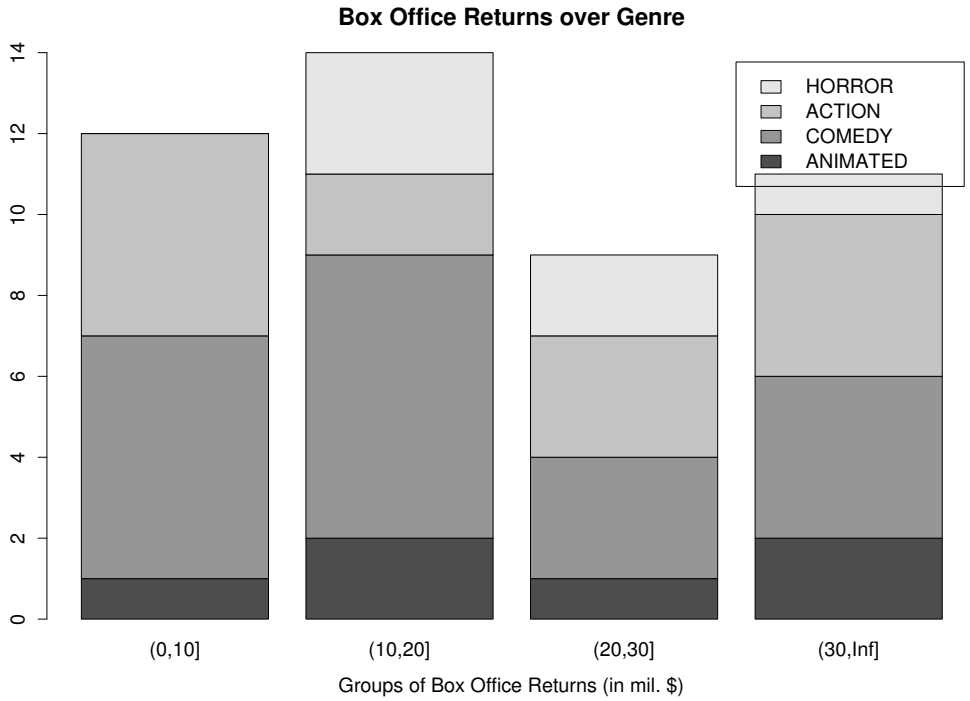
```
myvar <- function(x) {
  # your computations
  # assign the final result to var: var <- something
  return(var)
}
```

Homework: Part II (a bit harder)

- Load the full data MovieRatings data set from the web as a `*.CSV` file, keep all variables and not only the first five
- Create a new factor that has hold the genre of the movie, code 1 for animated, 2 for comedy, 3 for action and 4 for horror flick.
- *Hint*: you need to create a count variable first that goes from 1 to 4 for each genre, like `df$genre[df$ANIMATED == 1] <- 1`, and then recode it as a factor.
- Copy the code that creates the box office returns in millions and the code that cuts the box office returns into four categories (0 to < 10, 10 to < 20, 20 to < 30, and 30 to ∞).
- Make a bar graph plotting the frequency of box office returns (the four categories) over the different genres. Name the x-axis, title the graph and make a legend for the genres.

Homework: Part II (the graph)

Your graph should look like this:



Optional homework

- Go to the *R* section of Codeschool at `http://tryr.codeschool.com`
- Complete the entire TryR course
- You can do this from any computer without *R* being installed
- The interactive tutorial goes through most of the concepts we covered today plus a little extra (sometimes a little less)
- If you already did that last weekend, then go to `http://www.rstudio.com/ide/docs/` and read the section "Using RStudio" to get familiar with our favorite R IDE

Additional resources (more advanced)

- A great many *R* books:
  - The Art of R Programming (Matloff, 2011)
  - R Cookbook (Teetor, 2011)
  - R in a Nutshell (Adler, 2012)
- Online tutorials and fully fledged courses:
  - QuickR: `http://www.statmethods.net`
  - Roger Peng's *Computing for Data Analysis*, YouTube Playlist and Coursera MOOC
- Econometrics with R (still lacking more econometrics books, but lots of statistics books):
  - Applied Econometrics with R (Kleiber and Zeileis, 2008)
  - Applied Linear Regression (Weisberg, 2005) + An R Companion to Applied Regression (Fox, 2011)
  - *Use R!* Springer series on very particular fields/ techniques

Appendix: A few statistical functions in *R*

- `mean(x)` computes the mean
- `sd(x)` computes the sample standard deviation
- `var(x)` computes the sample variance
- `median(x)` computes the media of a vector
- `quantile(x,probs=...)` computes the supplied quantiles of a vector
- `summary(x)` summarizes the input object (for a vector, mean, min, max, etc.)
- `cor(x,y)` computes the correlation of two vectors
- `cov(x,y)` computes the covariance of two vectors

Appendix: A few mathematical functions in *R*

- Self explaining: `+`, `-`, `*`, `/`
- Exponentiation: `x^p`, e.g. `x^2` or `x^10`
- `abs(x)` takes the absolute value of a vector
- `sqrt(x)` takes the square root of a vector
- `log(x)` takes the natural logarithm of a vector
- `exp(x)` exponentiates the vector
- `min(x)` returns the minimum of a vector
- `max(x)` returns the maximum of a vector
- `sum(x)` returns the sum of a vector
- `prod(x)` returns the product of a vector
- `round(x, digits)` rounds the vector to specified # digits
- `trunc(x)` truncates the vector to an integer
- `cumsum(x)` returns the running sum of a vector

Appendix: Control structures

Two major control statements (`for` and `if ... else if ... else`), but also support for C-style control structures (`while` and `repeat`, not covered here).

A simple for-loop:

```
for (i in 1:100) {
  print(i)
}
```

A simple if-else statement:

```
x <- F
if (x == T) {
  print("yes")
} else {
  print("no")
}
```

Appendix: Alternatives to explicit loops

A popular alternative is to use the `apply()` family of functions, here's an abbreviation of what `??apply` brings up:

```
apply   Apply Functions Over Array Margins
by      Apply a Function to a Data Frame Split by Factors
eapply  Apply a Function Over Values in an Environment
lapply  Apply a Function over a List or Vector
mapply  Apply a Function to Multiple List or Vector Arguments
rapply  Recursively Apply a Function to a List
tapply  Apply a Function Over a Ragged Array
```

This is an *R* idiosyncrasy and is recommended in some books (as `apply()` is sometimes faster). We focus on loops for simplicity.

Moreover, loops could be avoided with a recursive function (a function that calls itself) but that should be reserved for problems that actually require recursion.