What is R?

R is an open source platform for statistical computing.

It offers

- a general purpose *interpreted* programming language
- a vast library of subroutines, both built-in and contributed
- and a support community

all with an emphasis on data manipulation and statistical inference.

R is maintained by a team of developers around the world.

Updated versions of the R core are released twice a year, usually with minor improvements, enhancements, and bug fixes.

- Not much has changed in the last several years.

User contributed add-on libraries are updated regularly, with dozens of additions every month.

- For many these are what make R attractive.

What is R most like?

S/S-PLUS: a commercial software that pre-dates R.

- R began as an open source version of S, contributed to by many of the original creators of S.
- R has essentially killed S, although its current owners (TIBCO) continue to get mileage out of supporting both softwares.

The modern language that R is most similar to is MATLAB.

Differences:

- R is free and open.
- R is object oriented; the MATLAB language is more streamlined.
- MATLAB is faster out of the box.
- The MATLAB stats “toolbox” is weak.
- Other, more engineering-oriented MATLAB toolboxes, e.g, for Signal Processing, are superior.
- MATLAB’s documentation is more professional.
- R’s contributed add-ons are of generally higher quality, and are easier to navigate.

R has a richer feature set, and is more highly customizable, than other statistical software suites. E.g.,

- SAS: great for Big Data
- STATA: popular with economists
- Minitab: what?
- Excel: perfect for making pie charts.

It is designed for tinkering, prototyping, simulating, high performance computing, and sharing.

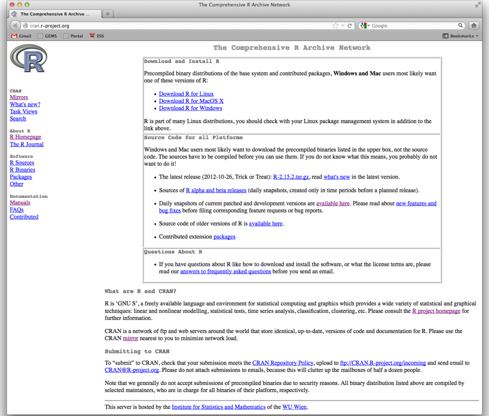
- R is a one-stop shop.

R **is not**

- as flexible as Python, but it is easier to learn
- a compiled language, but subroutines can be
- a spreadsheet, but it can interface with Excel
- a database, although it can interface with them
- a scripting language like Perl, although you can get by in a pinch
- for application building, although it could be part of a back-end or serve as a front-end

Obtaining R

http://cran.r-project.org



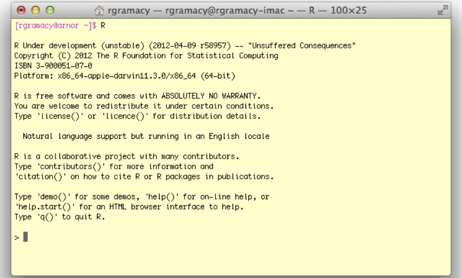
- Binaries for Windows and Mac OSX are available for download directly from CRAN.
- Full source code is available for compilation on any machine.
- Many linux distributions (e.g., Ubuntu) provide binaries.

There are commercial services that will sell you custom/optimized binaries

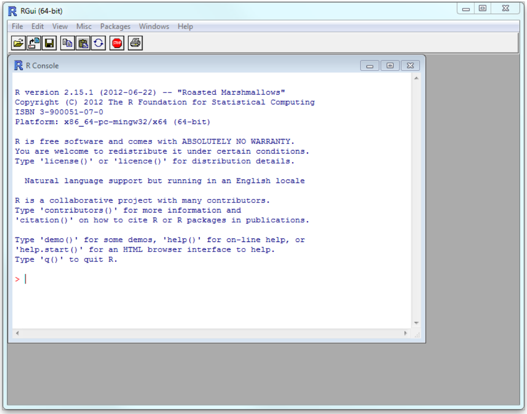
- e.g., Revolution Analytics

Interfaces

- Terminal (DOS/Unix/Linux/OSX); full-featured

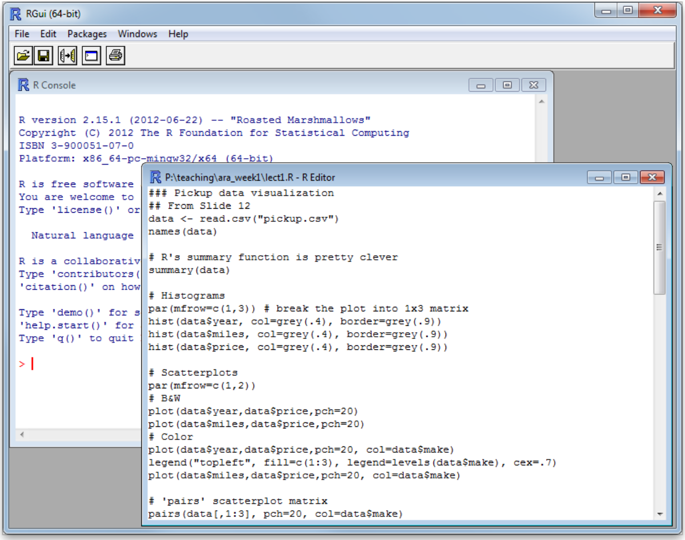


- Windows GUI

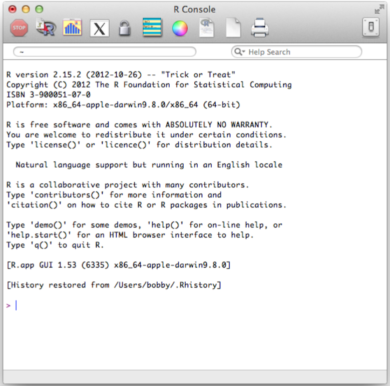


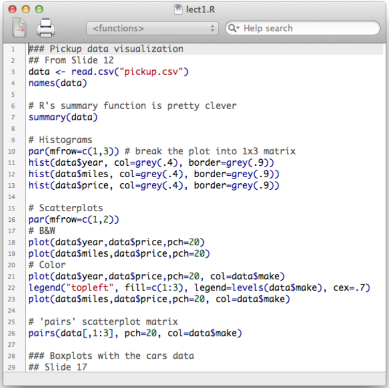
... not more features than the terminal version, although the built-in editor, file browser, and menus can at times be helpful.

It is not a fancy editor, and there is a window within a window.



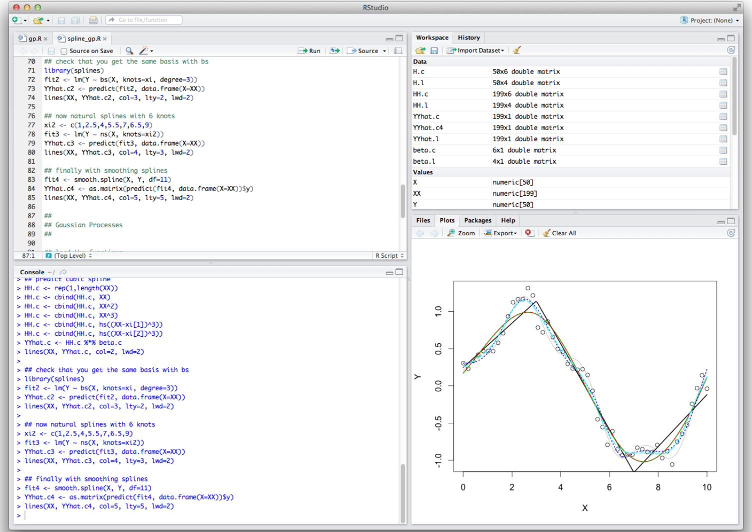
- OSX GUI





... on par with the Windows one, but without windows within windows, and the editor does syntax highlighting.

- Rstudio (http://www.Rstudio.com)



The R console

The console is the main way of interacting with R.

It allows you to type commands into R and see how the system responds.

It may seem rudimentary but it is state of the art. Although point-and-click has replaced command lines throughout much of computing (think DOS), the console remains the best way to analyze data.

It is not just an efficient way to interact with a computing platform. It makes it easy to keep a record of everything you do so you can recreate it later if needed.

R is waiting for your input ...

```
R version 2.15.2 (2012-04-09 r58957) -- "Trick or Treat"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin11.3.0/x86_64 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

... at the command prompt “>”.

Basic Operations

When you enter an expression into the R console and press Enter, R will evaluate it and display the results (if any).

At its most basic, R can be your calculator.

```
> 1 + 2 + 3
[1] 6
> 1 + 2*3
[1] 7
> (1 + 2)*3
[1] 9
```

The output in each case is a [1]-element vector.

Vectors are first-class citizens in R. They are formed by

- concatenation, e.g.,

```
> c(0,1,1,2,3,5,8)
[1] 0 1 1 2 3 5 8
```

- or by sequences, e.g.,

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

The brackets show the index of the first element of each row.

When you perform an operation on two vectors, R will match their elements pairwise and return a vector.

```
> c(1,2,3,4) + c(10,20,30,40)
[1] 11 22 33 44
> c(1,2,3,4) * c(10,20,30,40)
[1]  10  40  90 160
> (1:4) - c(1,1,1,1)
[1] 0 1 2 3
```

If the vectors aren't the same size, R will repeat the smaller sequence multiple times.

```
> c(1,2,3,4) + 1
[1] 2 3 4 5
> 1/(1:5)
[1] 1.0000000 0.5000000 0.3333333
[4] 0.2500000 0.2000000
> c(1,2,3,4) + c(10,100)
[1]  11 102  13 104
> c(1:5) + c(10, 100)
[1]  11 102  13 104  15
Warning message:
In c(1:5) + c(10, 100) :
  longer object length is not a multiple of shorter
  object length
```

You can also enter expressions with characters:

```
> "Hello world."
[1] "Hello world."
```

This is called a character vector of length 1.

Here is one of length 2:

```
> c("Hello world", "Hello R interpreter")
[1] "Hello world"
[2] "Hello R interpreter"
```

Comments

You can add comments to your R code. Anything after `#` is ignored:

```
> ## ... at the beginning of a line
> 1 + 2 + # ... in the middle
+   + 3
[1] 6
```

Editors may format the comments differently depending on their multiplicity; R doesn't care.

- Judicious commenting is an integral part of programming.

Functions

In R, the operations that do all the work are called *functions*.

- R is said to be a hybrid procedural and functional programming language.

Most are of the form

```
f(arg1, arg2, ...)
```

where `f` is the name of the function, and `arg1`, `arg2`, ... are the arguments,

- some of which may have default values.

A few example functions:

```
> exp(1)
[1] 2.718282
> cos(3.141593)
[1] -1
> cos(seq(-pi, pi, 1))
[1] -1.0000000 -0.5403023  0.4161468
[4]  0.9899925  0.6536436 -0.2836622
[7] -0.9601703
> log(1)
[1] 0
> log(exp(1))
[1] 1
```

When functions take more than one argument you can specify them by name

```
> log(x=64, base=4)
[1] 3
> log(base=4, x=64)
[1] 3
```

Or, if you give the arguments in the default order, you can omit the names.

```
> log(64, 4)
[1] 3
> log(4, 64)
[1] 0.3333333
```

Not all functions are of the form `f(...)`.

Some are operators. For example, for addition we use the "`+`" operator.

```
> 17+2
[1] 19
> 2^10
[1] 1024
> 3 == 4
[1] FALSE
```

Variables

R lets you assign values to variables and refer to them by name.

- Once assigned, the R interpreter will substitute that value in-place of the variable name when it evaluates an expression.

```
> x <- 1
> y <- 2
> z <- c(x,y)
> z
[1] 1 2
```

The substitution is done at the time that the value is assigned, not later when it is evaluated in an expression.

```
> y <- 4
> z
[1] 1 2
```

- R is provides no visual output when assigning, but

  ```
  > print(y <- 4)
  [1] 4
  ```

- You can use `=` and `->` but I don't recommend it.

  ```
  > x = 2
  > print(c(x,y) -> z)
  [1] 2 4
  ```

Referring to members of vectors:

```
> b <- (1:12)^2
> b
 [1]   1   4   9  16  25  36  49  64  81
[10] 100 121 144
> b[7]
[1] 49
> b[1:6]
[1]  1  4  9 16 25 36
> b[c(1,11,6)]
[1]   1 121  36
> b[b %% 3 == 0]
[1]   9  36  81 144
```

Puzzled about a compound expression?

- Break it into its constituent parts.

```
> b %% 3
 [1] 1 1 0 1 1 0 1 1 0 1 1 0
> print(b30 <- b %% 3 == 0)
 [1] FALSE FALSE  TRUE FALSE FALSE  TRUE
 [7] FALSE FALSE  TRUE FALSE FALSE  TRUE
> b[b30]
[1]   9  36  81 144
```

Notice how indexing with logicals differs from integers.

Careful with `=` and `==`.

```
> one <- 1
> two <- 2
> one = two
> one
[1] 2
> one <- 1
> one == two
[1] FALSE
```

Functions

A function in R is just another object that is assigned to a symbol.

You can make your own functions in R, assign them a name, and then call them like the built-in functions.

```
> f <- function(x,y) { c(x+1, y+1) }
> f(1, 2)
[1] 2 3
> f
function(x,y) { c(x+1, y+1) }
```

Loops and control

R has a several ways of repeating code, or branching execution upon condition.

E.g.,

```
> fib <- rep(NA, 12)
> fib[1:2] <- 0:1
> for(i in 3:length(fib)) {
+   fib[i] <- fib[i-1] + fib[i-2]
+ }
> fib
 [1]  0  1  1  2  3  5  8 13 21 34 55 89
```

Data Structures

You can construct more complicated data structures than just vectors.

An array is a multidimensional vector.

- Arrays and vectors are stored (internally) in the same way, but an array may be displayed and accessed differently.
- It is basically a vector that has an additional dimension attribute.

```
> a <- array(c(1,2,3,4,5,6,7,
+              8,9,10,11,12), dim=c(3,4))
> a
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Particular cells can be reference via 2-d coordinates:

- first row, then column

  ```
  > a[2,3]
  [1] 8
  ```

A vector lacks that extra structure.

```
> as.vector(a)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

And a matrix is just a two-dimensional array.

```
> m <- matrix(data=c(1,2,3,4,5,6,7,
+                8,9,10,11,12), nrow=3, ncol=4)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Arrays can have more than two dimensions.

```
w <- array(1:12, dim=c(2,3,2))
```

```
> w
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

```
> w[1,3,2]
[1] 11
> w[1,3,]
[1]  5 11
> w[,,2]
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Arrays/vectors can be subset by other (integer) arrays/vectors.

- we just saw a couple of examples

```
> a[1:2,]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
> a[c(1,3),]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    3    6    9   12
```

List objects

Vectors, arrays, and matrices are data structures based on a single underlying (e.g., numeric or character) type.

The most generic data structure for collecting mixed-type data is a list.

Entries in a list can be entered and referenced by name and/or by location, i.e., by number.

Here is an example of a list with two named components.

```
> e <- list(thing="hat", size=8.25)
> e
$thing
[1] "hat"

$size
[1] 8.25
```

You can access an item in a list multiple ways.

```
> e$thing
[1] "hat"
> e[[1]]
[1] "hat"
```

A list can even contain other lists.

```
> g <- list("lists within lists", e)
> g
[[1]]
[1] "lists within lists"

[[2]]
[[2]]$thing
[1] "hat"

[[2]]$size
[1] 8.25
```

Data frames

A data frame is a list that contains multiple named vectors that are the same length.

- It is a lot like a spreadsheet or a database table
- It is stored like a matrix, but like a list it allows columns to differ in type
- They are particularly good at representing experimental data.

Here is an example list containing win/loss results for baseball teams in the NL East in 2008:'

```
> teams <- c("PHI", "NYM", "FLA", "ATL", "WSN")
> w <- c(92, 89, 94, 72, 59)
> l <- 162 - w
> nleast <- data.frame(teams, w, l)
> nleast
  teams  w   l
1   PHI 92  70
2   NYM 89  73
3   FLA 94  68
4   ATL 72  90
5   WSN 59 103
```

You can refer to the components of a data frame by name or by column number.

```
>  nleast$w
[1] 92 89 94 72 59
> nleast[,2]
[1] 92 89 94 72 59
```

You can use logical expressions to pick out particular rows:

```
> nleast[nleast$teams == "FLA",]
  teams  w  l
3   FLA 94 68
```

Objects and classes

R is an object-oriented language, and so every object has a class.

```
> class(teams)
[1] "character"
> class(w)
[1] "numeric"
> class(nleast)
[1] "data.frame"
> class(class)
[1] "function"
```

Some functions are associated with a specific class; these are called methods.

When methods for different classes share the same name they are called generic functions.

Generic functions serve two purposes.

1. They make it easy to guess the right function for a unfamiliar class.
2. They make it possible to use the same code for objects of different types.

For example, `+` is a generic function for adding objects.

- You can add numbers.

```
> 17 + 6
[1] 23
```

- And it probably does something sensible with other objects, e.g., those of the date class.

```
> d <- as.Date("2009-08-08")
> class(d)
[1] "Date"
> d + 7
[1] "2009-08-15"
```

`print()` is another good example.

Charts, graphics and summaries

R has many ways to inspect/visualize data. Consider the `cars` data provided in the base R library.

```
> cars
   speed dist
1      4    2
2      4   10
3      7    4
...
> dim(cars)
[1] 50  2
> names(cars)
[1] "speed" "dist"
```

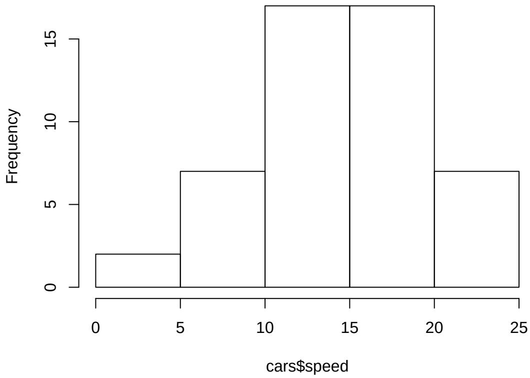
Each of 50 observations records the speed of the car and the distance required to stop.

The generic `summary()` function is a useful first exploratory data analysis (EDA) tool.

```
> summary(cars)
     speed           dist       
 Min.   : 4.0   Min.   :  2.00  
 1st Qu.:12.0   1st Qu.: 26.00  
 Median :15.0   Median : 36.00  
 Mean   :15.4   Mean   : 42.98  
 3rd Qu.:19.0   3rd Qu.: 56.00  
 Max.   :25.0   Max.   :120.00  
```

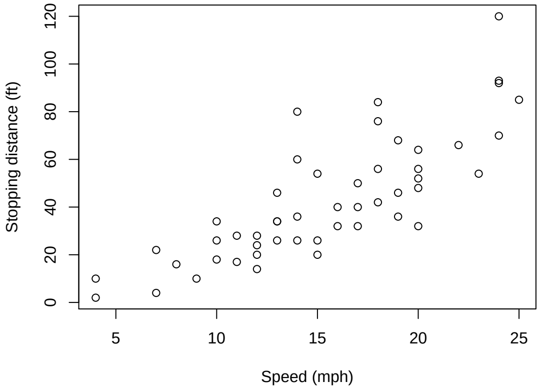
For a visual summary, try a histogram.

```
> hist(cars$speed, main="")
```



Scatterplots are useful.

```
> plot(cars, xlab="Speed (mph)",
+      ylab="Stopping distance (ft)")
```



Statistical modeling

There would appear to be a linear relationship between speed and stopping distance. Lets check.

```
> cars.lm <- lm(dist~speed, data=cars)
```

This envokes an OLS fit to

$$\mathrm{dist}_i = \beta_0 + \beta_1 \mathrm{speed}_i + \varepsilon_i, \quad \varepsilon_i \overset{\mathrm{iid}}{\sim} \mathcal{N}(0, \sigma^2)$$

for $i = 1, \ldots, n$.

- `dist~speed` is a formula encoding that model.

`print()` provides a brief summary of the fitted model:

```
> cars.lm

Call:
lm(formula = dist ~ speed, data = cars)

Coefficients:
(Intercept)        speed
    -17.579        3.932
```

- Those coefficients are $\hat{\beta}_0$ and $\hat{\beta}_1$.
- see `print` and `print.lm`

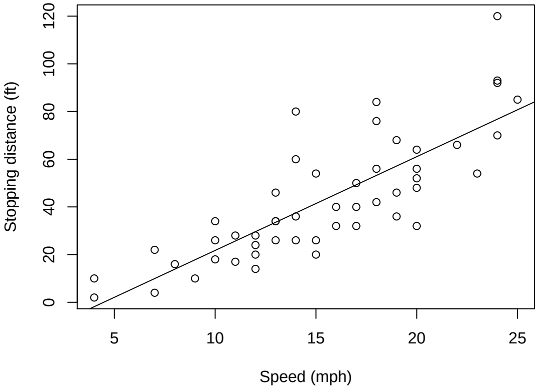
`summary()` provides more info.

```
> summary(cars.lm)

Call:
lm(formula = dist ~ speed, data = cars)

Residuals:
    Min      1Q  Median      3Q     Max
-29.069  -9.525  -2.272   9.215  43.201

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -17.5791     6.7584  -2.601   0.0123 *
speed         3.9324     0.4155   9.464 1.49e-12 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 15.38 on 48 degrees of freedom
Multiple R-squared: 0.6511, Adjusted R-squared: 0.6438
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

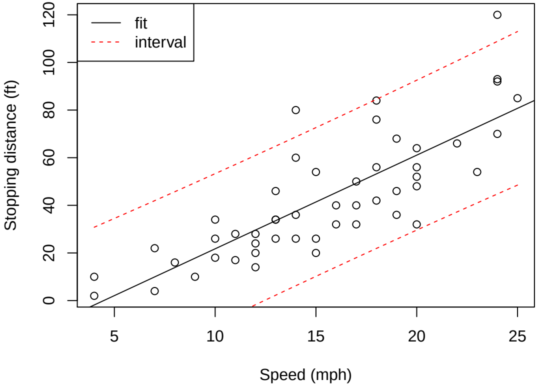
- see `summary` and `summary.lm`

Adding the line of best fit is easy, using another generic function.

```
> abline(cars.lm)
```



Finally, it helps to understand the full scope of uncertainties.



(code in supplement)

Also try `plot(cars.lm)`.

Getting Help

R includes a help system to help you get information about the core language and installed packages.

For help on fitting linear models, type:

```
> help(lm)
```

or equivalently

```
> ? lm
```

How the help interface looks depends on your GUI.

R: Fitting Linear Models

127.0.0.1:42870/help/library/stats/html/lm.html

lm {stats} R Documentation

Fitting Linear Models

Description

`lm` is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Usage

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

Arguments

`formula`	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.
`data`	an optional data frame, list or environment (or object coercible by `as.data.frame` to a data frame) containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which `lm` is called.
`subset`	an optional vector specifying a subset of observations to be used in the fitting process.
`weights`	an optional vector of weights to be used in the fitting process. Should be `NULL` or a numeric vector. If non-NULL, weighted least squares is used with weights `weights` (that is, minimizing `sum(w*e^2)`); otherwise ordinary least squares is used. See also 'Details',
`na.action`	a function which indicates what should happen when the data contain `NA`s. The default is set by the `na.action` setting of `options`, and is `na.fail` if that is unset. The 'factory-fresh' default is `na.omit`. Another possible value is `NULL`, no action. Value `na.exclude` can be useful.
`method`	the method to be used; for fitting, currently only `method = "qr"` is supported; `method = "model.frame"` returns the model frame (the same as with `model = TRUE`, see below).

That R's help is cryptic is a fair criticism.

- But there is some method to the madness.

I start with the examples, which are at the bottom.

- You can cut-and-paste, or
- use `example(lm)`.

The "usage" section is where I look next, to see what the arguments are and check the defaults.

I read the "details" section last or not at all.

You can search the help system for a topic, which is handy if you don't know (or have forgotten) the relevant function name.

```
> help.search("regression")
```

or equivalently

```
> ?? regression
```

The search is based on *keywords* hidden in the documentation system and sadly isn't very helpful.

- You're better off Googling.

R **resources**

Google really is the best place to start.

- Often, you end up being directed to R-sponsored pages and discussion groups.

But if you're looking for something more structured or more polished, you can try:

- `http://cran.r-project.org/manuals.html`
- `http://cran.r-project.org/other-docs.html`

which are linked from the CRAN page.

R **packages**

Packages are the life blood of R.

A package is a related set of functions, help files, and data files that have all been bundled together.

R offers an enormous array of packages

- displaying graphics, statistical tests,
- machine learning, signal processing
- analyzing microarray data, modeling credit risk

Some are included in R; others are contributed by the public and are available online via package repositories.

To use a package you need to *load* it into your current session.

The packages loaded by default are:

```
> (.packages())
[1] "stats"     "graphics"  "grDevices"
[4] "utils"     "datasets"  "methods"
[7] "base"
```

The full set that come with R are:

```
> (.packages(all.available=TRUE))
 [1] "base"       "boot"       "class"
 [4] "cluster"    "codetools"  "compiler"
 [7] "datasets"   "foreign"    "graphics"
[10] "grDevices"  "grid"       "KernSmooth"
[13] "lattice"    "MASS"       "Matrix"
[16] "methods"    "mgcv"       "nlme"
[19] "nnet"       "parallel"   "rpart"
[22] "spatial"    "splines"    "stats"
[25] "stats4"     "survival"   "tcltk"
[28] "tools"      "utils"
```

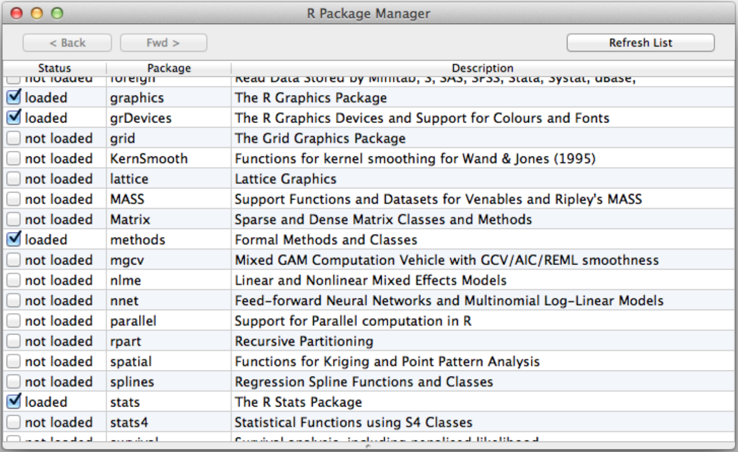
Also try `library()`.

Packages are loaded with the `library()` function, supplying the name of the desired package as an argument.

```
> library(rpart)
```

You can use the GUI too.

- Each works a little differently.
- A drawback is that you have to remember to re-do the load in each new session; whereas the `library()` call can be scripted.


R Package Manager
< Back
Fwd >
Refresh List
Status
Package
Description
loaded graphics The R Graphics Package
loaded grDevices The R Graphics Devices and Support for Colours and Fonts
not loaded grid The Grid Graphics Package
not loaded KernSmooth Functions for kernel smoothing for Wand & Jones (1995)
not loaded lattice Lattice Graphics
not loaded MASS Support Functions and Datasets for Venables and Ripley's MASS
not loaded Matrix Sparse and Dense Matrix Classes and Methods
loaded methods Formal Methods and Classes
not loaded mgcv Mixed GAM Computation Vehicle with GCV/AIC/REML smoothness
not loaded nlme Linear and Nonlinear Mixed Effects Models
not loaded nnet Feed-forward Neural Networks and Multinomial Log-Linear Models
not loaded parallel Support for Parallel computation in R
not loaded rpart Recursive Partitioning
not loaded spatial Functions for Kriging and Point Pattern Analysis
not loaded splines Regression Spline Functions and Classes
loaded stats The R Stats Package
not loaded stats4 Statistical Functions using S4 Classes

Package Repositories

You will find thousands of R packages online. The two biggest sources are:

- CRAN (Comprehensive R Archive Network)
  http://cran.r-project.org
- Bioconductor, primary for genomic analysis
  http://www.bioconductor.org

R-Forge (http://r-forge.r-project.org) is another interesting place to look for R packages

- but it is more of a collaborative/works in progress site.

Packages can be installed in several ways; CRAN packages are the easiest.

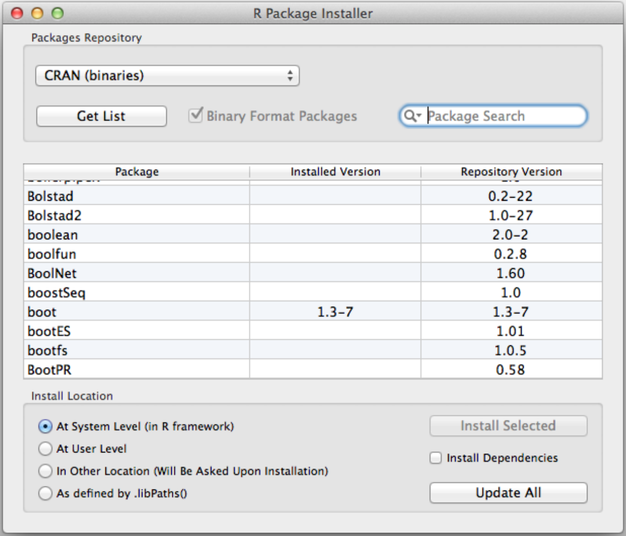
```
> install.packages("tgp", dependencies=TRUE)
...
```

Finding packages for your task can be challenging.

- There *is* an app for that.
- Use Google.

The GUIs are also an option.

- You only need to install a package once per machine.


R Package Installer
Packages Repository
CRAN (binaries)
Get List
Binary Format Packages
Package Search
Package
Installed Version
Repository Version
Bolstad
0.2-22
Bolstad2
1.0-27
boolean
2.0-2
boolfun
0.2.8
BoolNet
1.60
boostSeq
1.0
boot
1.3-7
1.3-7
bootES
1.01
bootfs
1.0.5
BootPR
0.58
Install Location
At System Level (in R framework)
At User Level
In Other Location (Will Be Asked Upon Installation)
As defined by .libPaths()
Install Selected
Install Dependencies
Update All

“Homework”

- Write a function returning

$$f(x) = 1 - x + 3x^2 - x^3, \quad x \in [-0.5, 2.5].$$

- Plot the function over that range and note the critical points. Check them against the truth (via calculus).
- Use an R library function to find those critical points numerically, and check them against the plot/truth.
- How many iterations did it take to find each critical point?