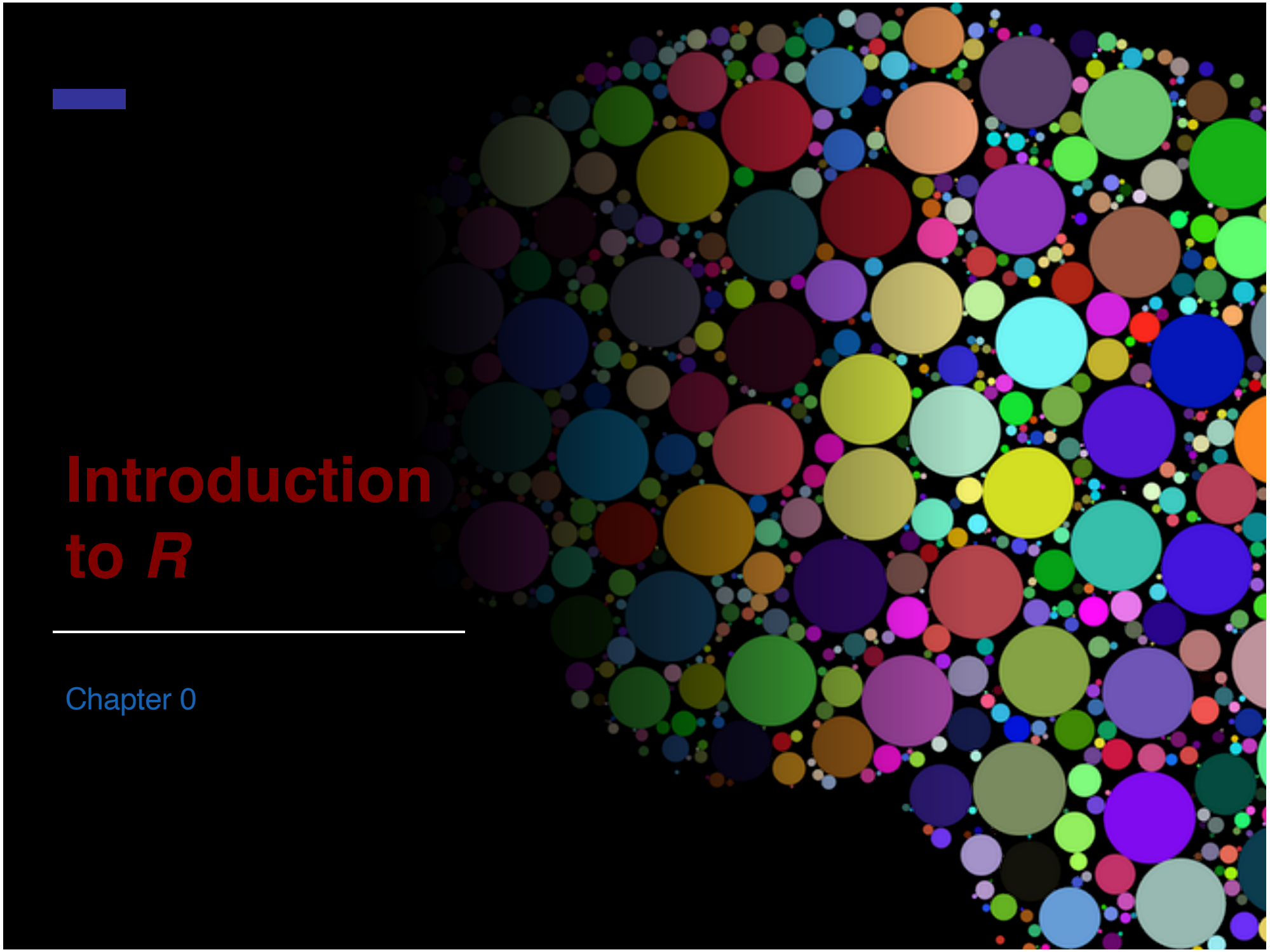


# Introduction to *R*

Chapter 0

# First Note:

> Helpful websites:

- Quick-R: www.statmethods.net
- R documentation: www.rdocumentation.org
- TryR: tryr.codeschool.com
- Swirl: www.swirlstats.com
- Stack Overflow: www.stackoverflow.com

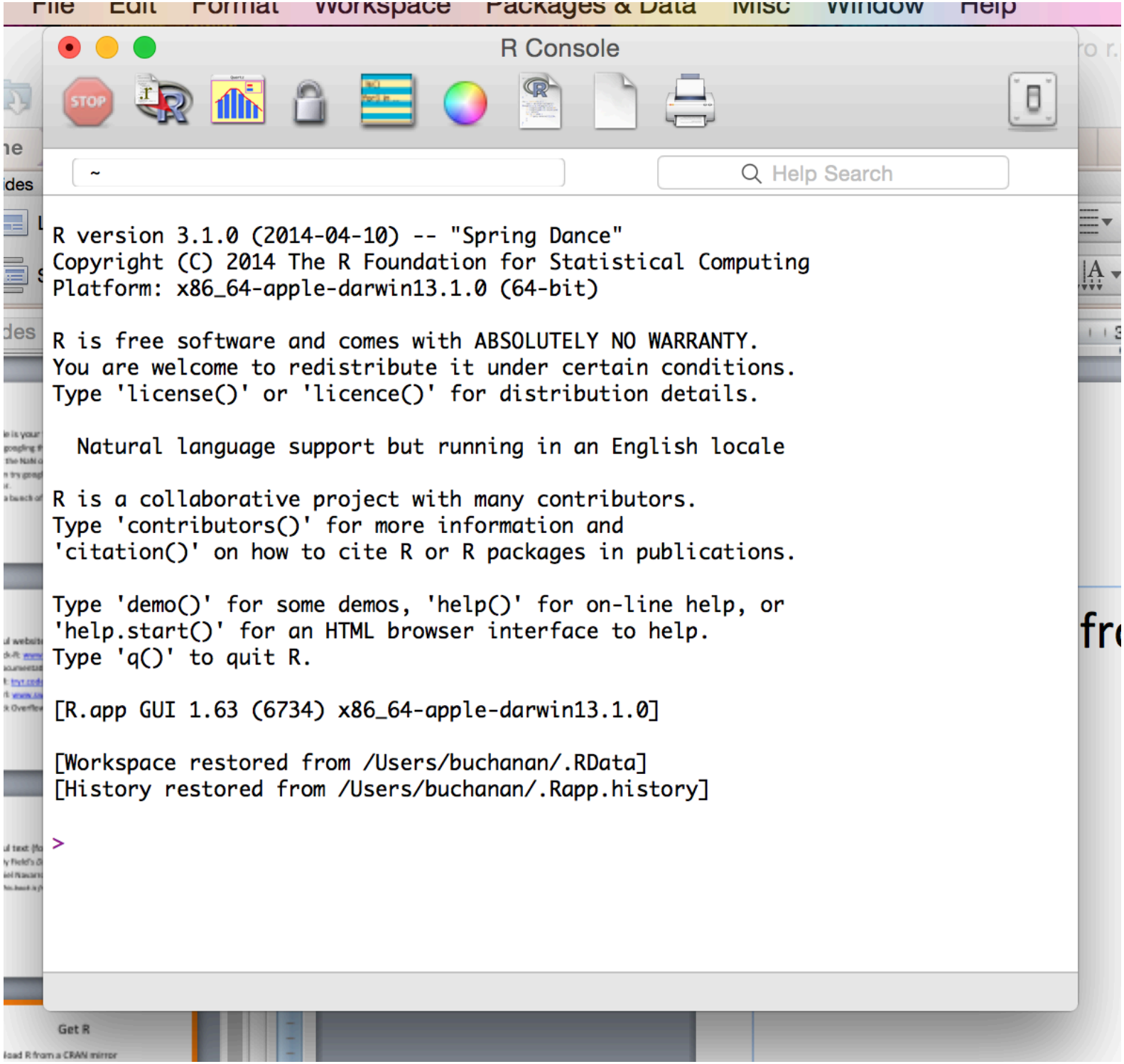
# Get R

> Download R from a CRAN mirror

- http://cran.r-project.org/

> Note:

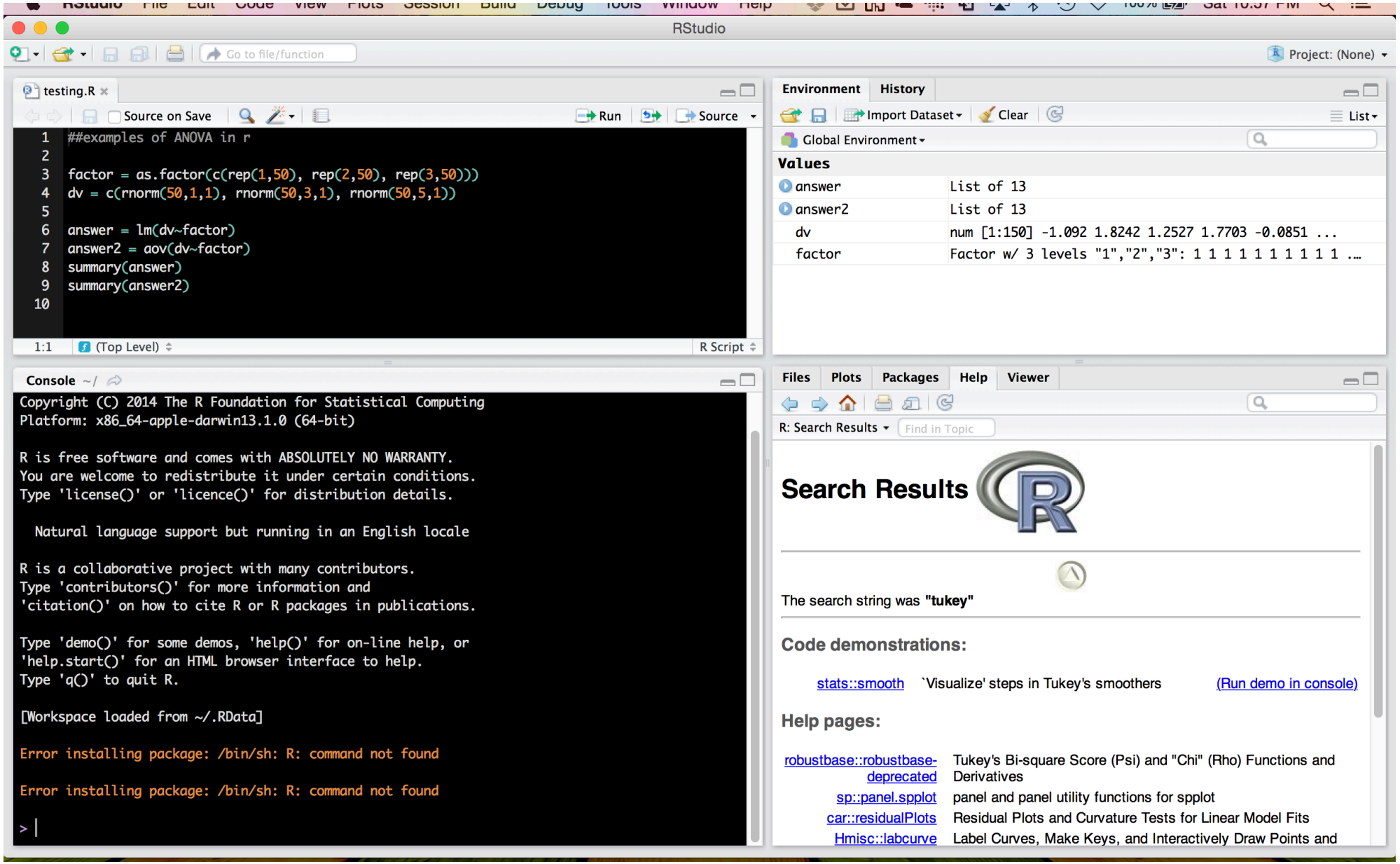
- Mac users will also need XQuartz for plotting


File Edit Format Workspace Packages & Data Misc Window Help
R Console
STOP
~
Help Search
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.1.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[R.app GUI 1.63 (6734) x86_64-apple-darwin13.1.0]
[Workspace restored from /Users/buchanan/.RData]
[History restored from /Users/buchanan/.Rapp.history]
>
Get R

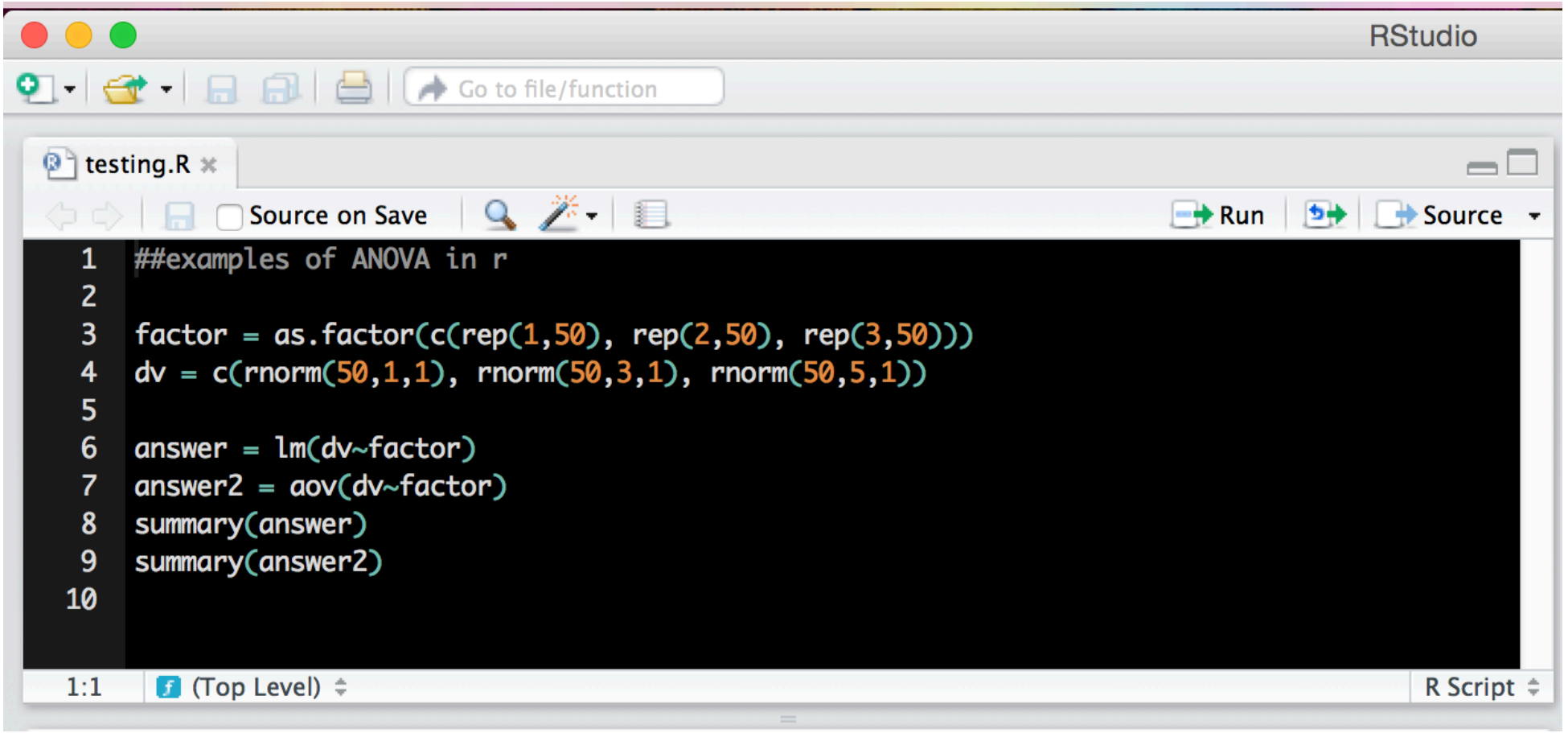
# Get RStudio

> For the love of sanity RStudio:

- http://www.rstudio.org/

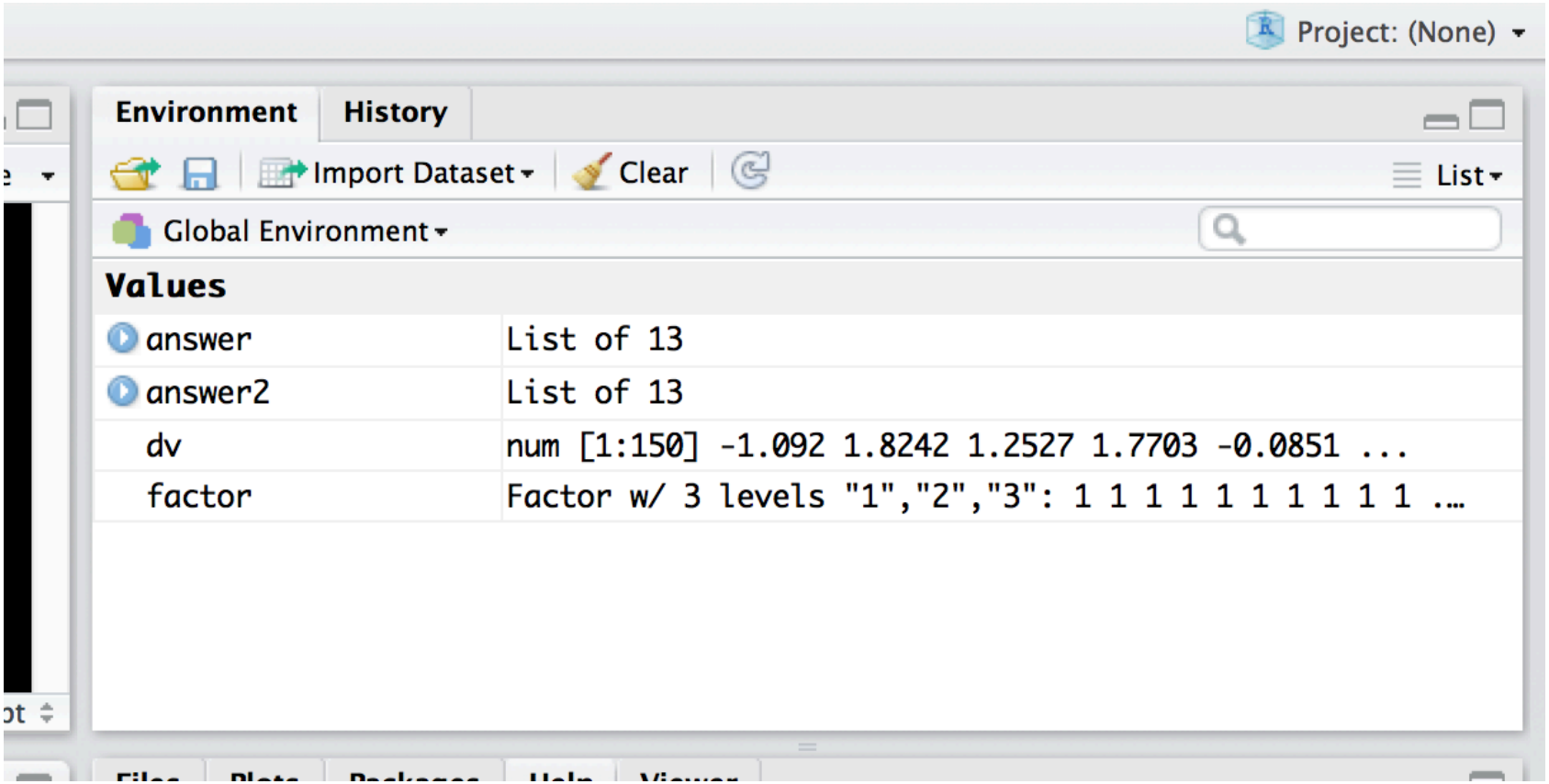

RStudio
testing.R
Source on Save
Run
Source
1 ##examples of ANOVA in r
2
3 factor = as.factor(c(rep(1,50), rep(2,50), rep(3,50)))
4 dv = c(rnorm(50,1,1), rnorm(50,3,1), rnorm(50,5,1))
5
6 answer = lm(dv~factor)
7 answer2 = aov(dv~factor)
8 summary(answer)
9 summary(answer2)
10
1:1 (Top Level) R Script
Console ~/
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.1.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/.RData]
Error installing package: /bin/sh: R: command not found
Error installing package: /bin/sh: R: command not found
>
Environment History
Import Dataset Clear List
Global Environment
Values
answer List of 13
answer2 List of 13
dv num [1:150] -1.092 1.8242 1.2527 1.7703 -0.0851 ...
factor Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 ...
Files Plots Packages Help Viewer
R: Search Results
Find in Topic
Search Results
The search string was "tukey"
Code demonstrations:
stats::smooth `Visualize' steps in Tukey's smoothers (Run demo in console)
Help pages:
robustbase::robustbase-deprecated Tukey's Bi-square Score (Psi) and "Chi" (Rho) Functions and Derivatives
sp::panel.spplot panel and panel utility functions for spplot
car::residualPlots Residual Plots and Curvature Tests for Linear Model Fits
Hmisc::labcurve Label Curves, Make Keys, and Interactively Draw Points and

# Top left window



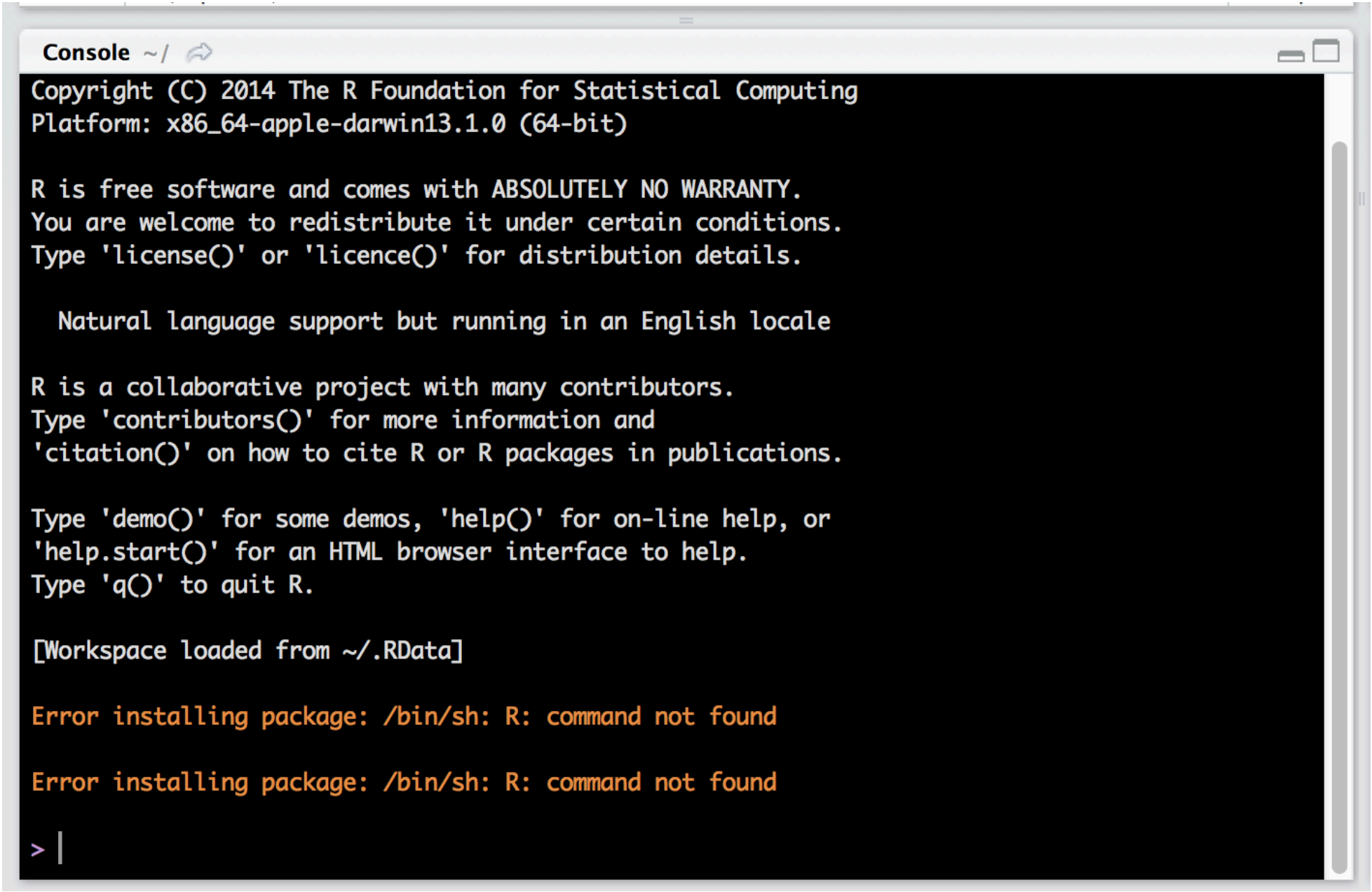

Scripting window – these files end with .R extension.

# Top right window

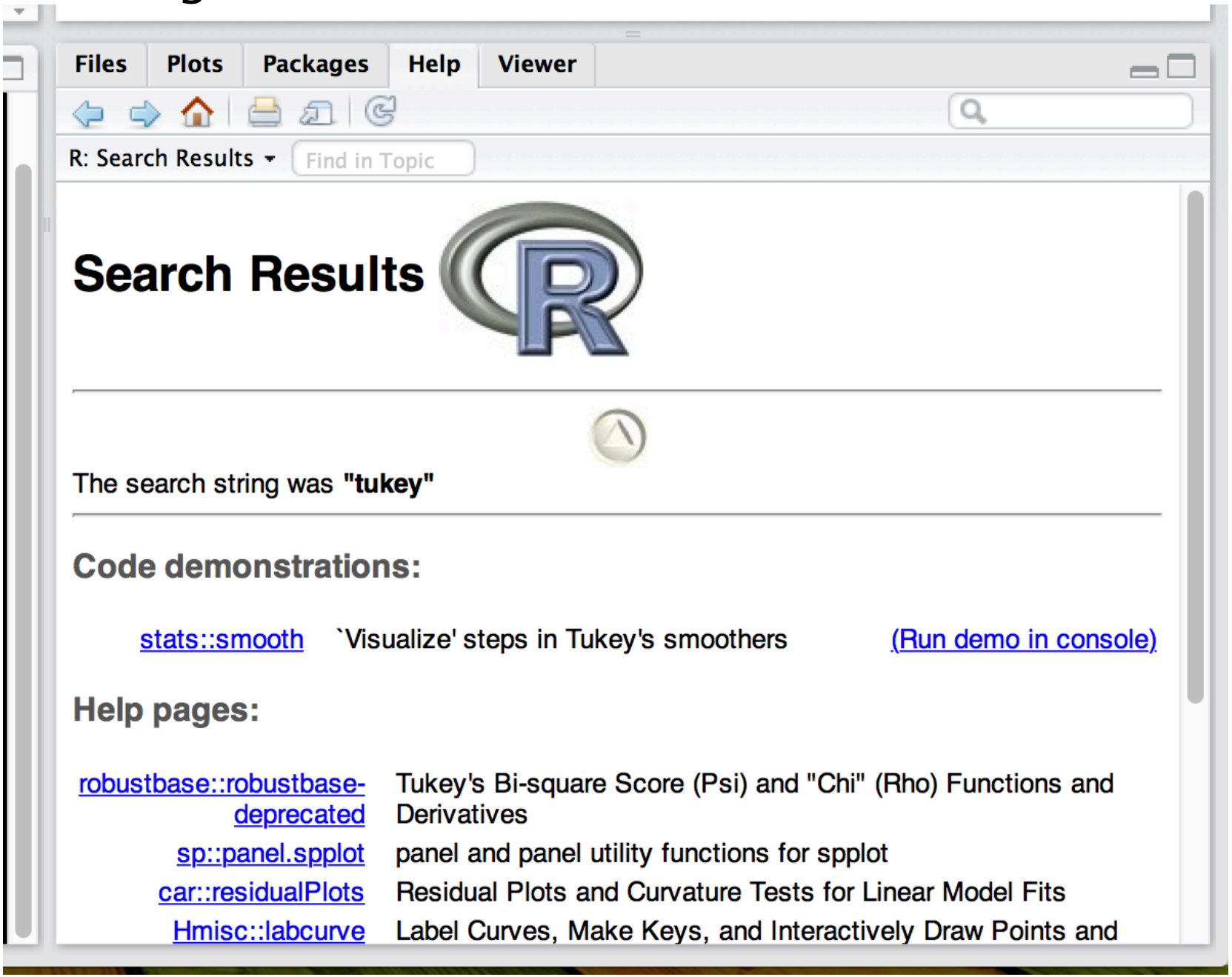



Environment window – you can save the environment (va
.Rdata files.

# Bottom left window



# Bottom right window



# Commands

> *Commands* are the code that you tell *R* to do for you.
  - They can be very simple.
  - They can be very complex.

> A note: *R* only does what you tell it. So when you encounter a mistake, it may be a typo or it may be an error in what you coded.

# Commands

> Commands can be typed in two places in RStudio.

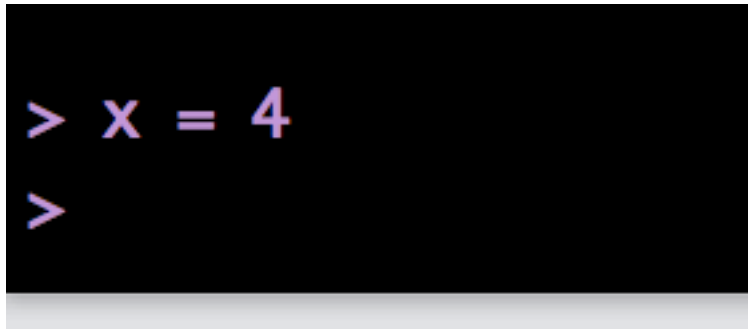
- At the top in an open R document.
- Directly in the R console.

# Commands

> Try something simple in the console:

> X = 4

- What are the > signs?
- Those indicate what has been run and a > with the cursor indicates it's ready for the next command.

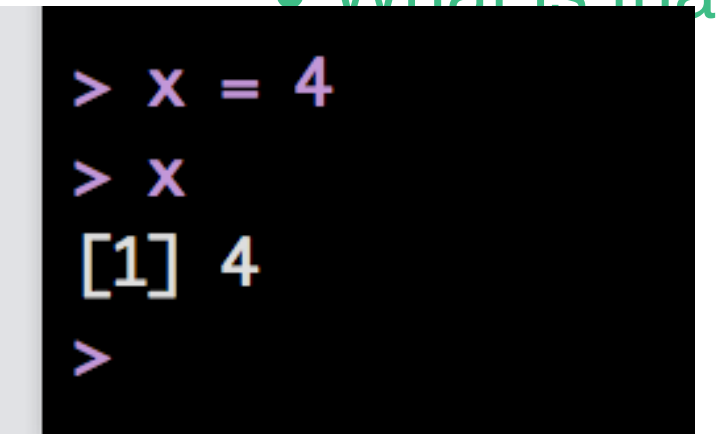


# Commands

> Now type X in the console (cApItAliZing matters!).

> Now it shows you the value(s) of X.

- It's only one number: 4.
- What is that [1] thing?! (give me a minute).



# Commands

> Note:

- = and <- are equivalent.
- In Swirl use <- to get credit.

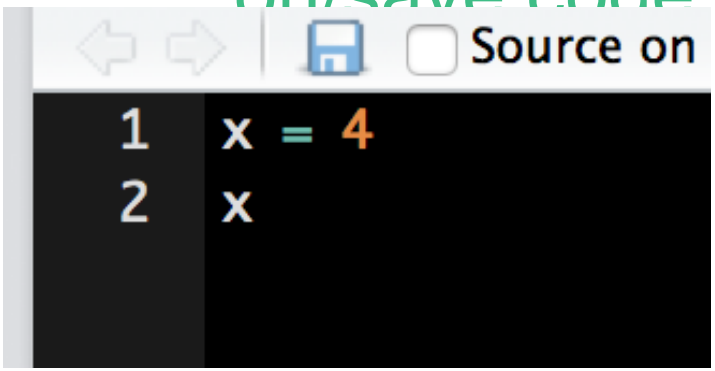
> An important thought here:

- The question in R is never “can R do this?” it’s “how can R do this?”
- (Un)fortunately, the answer is often: about 10 different ways.

# Commands

> Now type the same thing into the R script window.

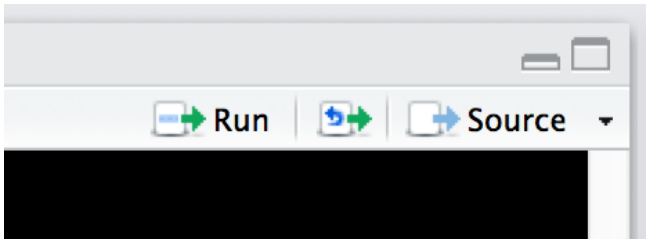
- You should notice that nothing happened in the console.
- The script window allows you to work on/save code without running it.



# Commands

> You can run the code by highlighting it and clicking on run or by hitting:

- Windows: ctrl + r
- Mac: command (apple) + return



# Commands

> The script window is great for saving projects so you can remember and recreate your steps when you need to later.
  - No need to save multiple files! ;)

# Commands

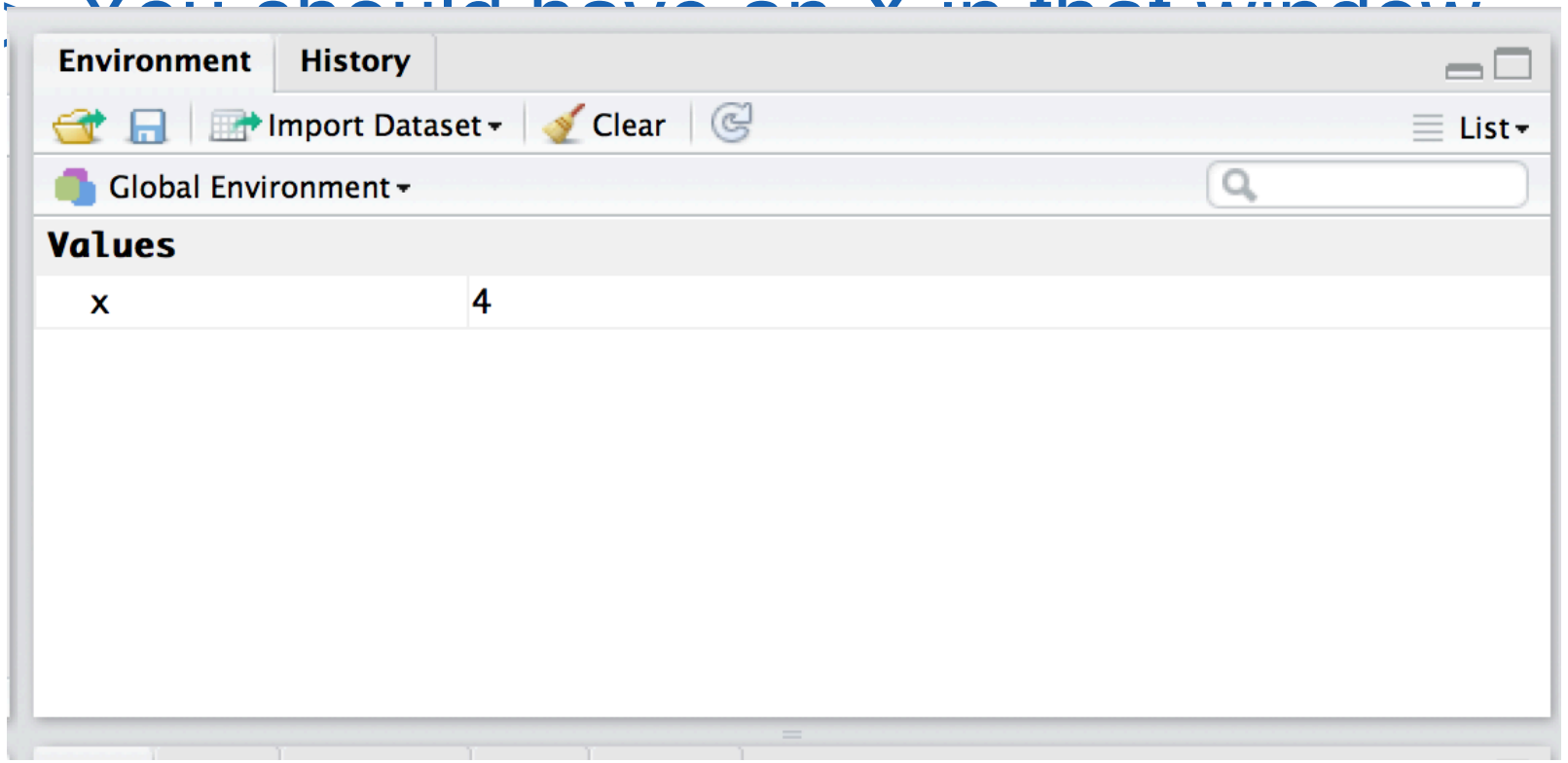
> Two console shortcuts:

- Hit the up arrow – you can scroll through the last commands that were run.
- Hit the tab key – you'll get a list of variable names and options to select from.

# Commands

> Anytime you create/use a variable, it will appear in the environment window.

> You should have an X in that window



# Commands

> The environment window also allows you to see all the types of the variables you have open.

- I find this very handy for *lists and dataframes*, which can be difficult to remember what all is stored in them.

# Object Types

> Vectors

> Lists

> Matrices

> Data Frames

# Object Types

> Within these objects, values can be:

- Character
- Numeric/Integer/Complex
- Logical
- NaN (versus NA)

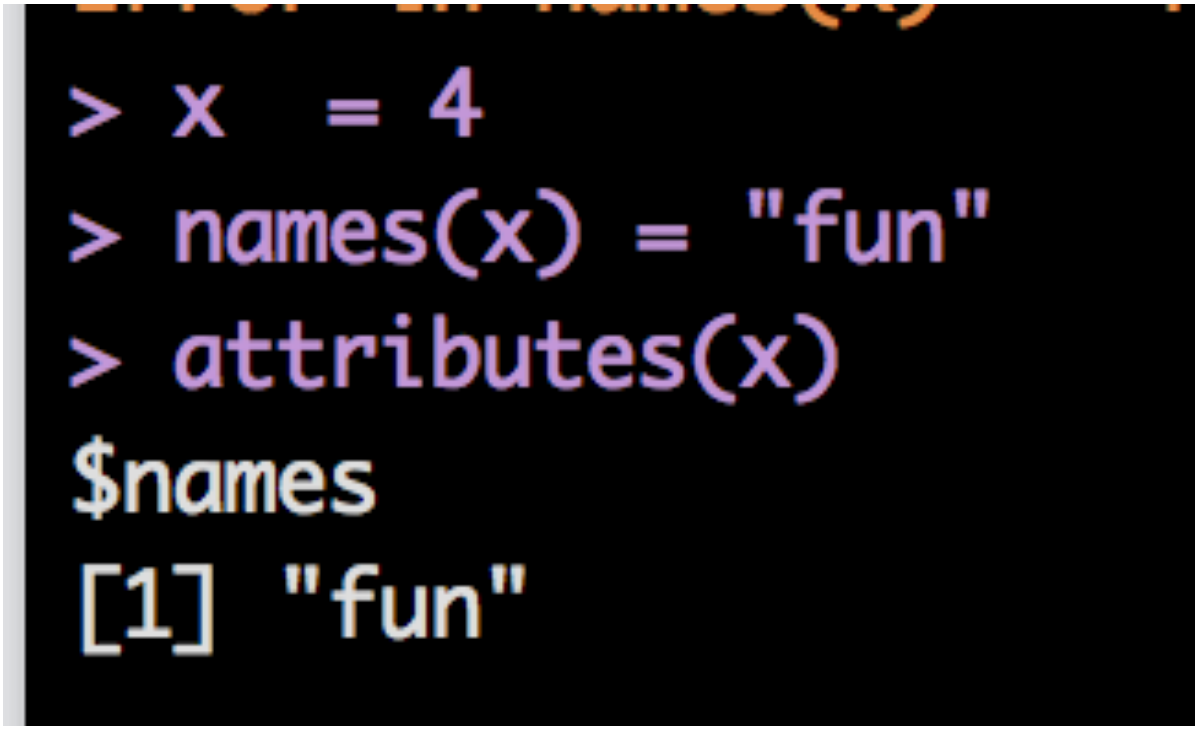
# Object Types

- And furthermore, within objects you can have *attributes*
  - The most important that you'll use are their *names*
  - Example:
    - After you run a regression, you can get a variable that has the residuals and coefficients stored. They are stored with specific names, so you can *call* them.

# Object Types

> Example

- attributes(OBJECT NAME)



# Object Types

```
> data(airquality)
> attributes(airquality)
$names
[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"

$class
[1] "data.frame"

$row.names
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22
 [23]  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44
 [45]  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66
 [67]  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88
 [89]  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110
[111] 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
[133] 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153

> |
```

# Object Types

> Another way to see what's going on with an object:

- ls(OBJECT NAME)
  - > LS gives you a list of the names in an object.
- str(OBJECT NAME)
  - > STR gives you the structure of an object, which is helpful to know what things are stored in that object.
  - > It's the same thing you can see in the environment window.

# Object Types

```
> 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 ...
> ls(airquality)
[1] "Day"     "Month"   "Ozone"   "Solar.R" "Temp"    "Wind"
```

**Data**

| | |
|---|---|
| airquality | 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 ... | |

**Values**

# Object Types

> Vectors

- You can think about a vector as one row or column of data
- All the objects must be the same class/type (number, logical, etc.).
  - > If you try to mix and match, it will *coerce* them into the same type or make them NA if not.

# Object Types

> Let's make a vector.

- You've already done that! Go you!
- X = 4, makes a vector.

> X

- That vector has one value (hence the [1] when you print it out) and is a class type numeric.

# Object Types

> Let's make some more vectors! Learn four new things:

- A = 1:20
- B = seq(1, 20, 2)
- C = c("cheese", "is", "great")
- D = rep(1,30)

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

- Notice you got a sequence of 20 numbers with int
- The [1] doesn't mean 1 row of data, it means here
- Try a = 1:200.
- Now you see that the [] corresponds to the item pl

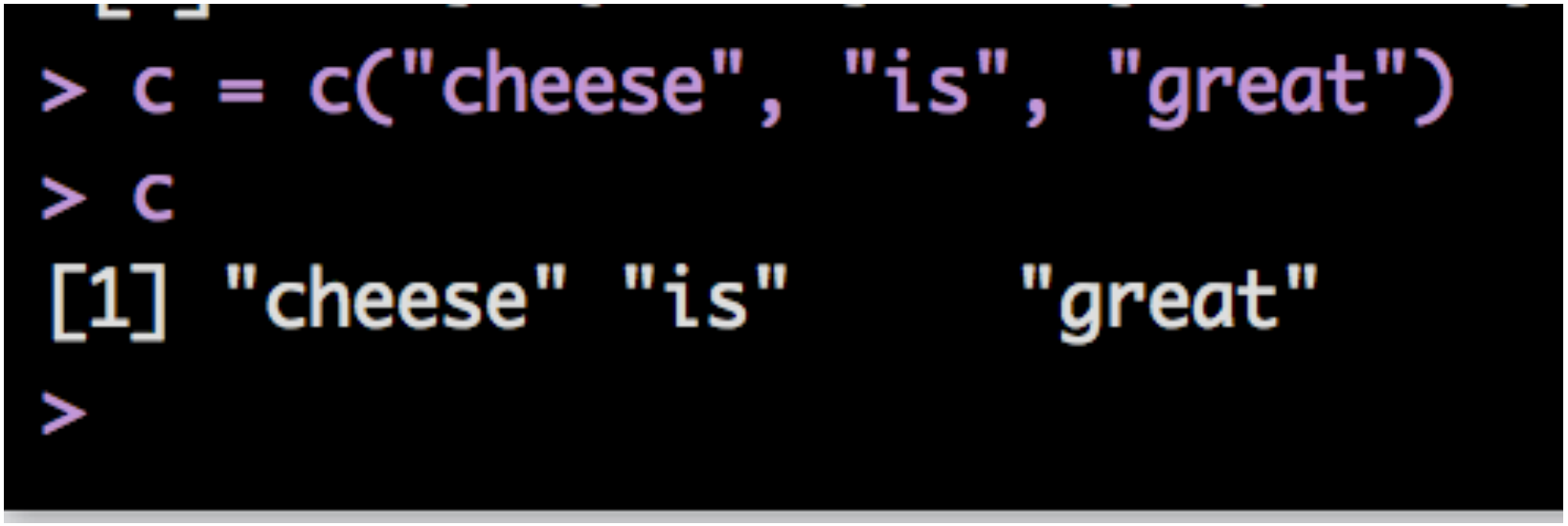
```
> a
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22
 [23]  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44
 [45]  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66
 [67]  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88
 [89]  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110
[111] 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
[133] 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154
[155] 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176
[177] 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198
[199] 199 200
>
```

```
> a[4]
[1] 4
>
```

```
> b = seq(1, 20, 2)
> b
 [1]  1  3  5  7  9 11 13 15 17 19
```

The sequence function works in the following way:

- Seq(start, end, by)
- Start at the number 1
- End at the number 20
- Increment by 2




The c function = concatenate or combine.
This function lets you combine things into one vector.

```
> d = rep(1,30)
> d
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
> 
```

The rep function lets you repeat an object over and ove
rep(repeat this, so many times)
Try rep(1:4, 10)

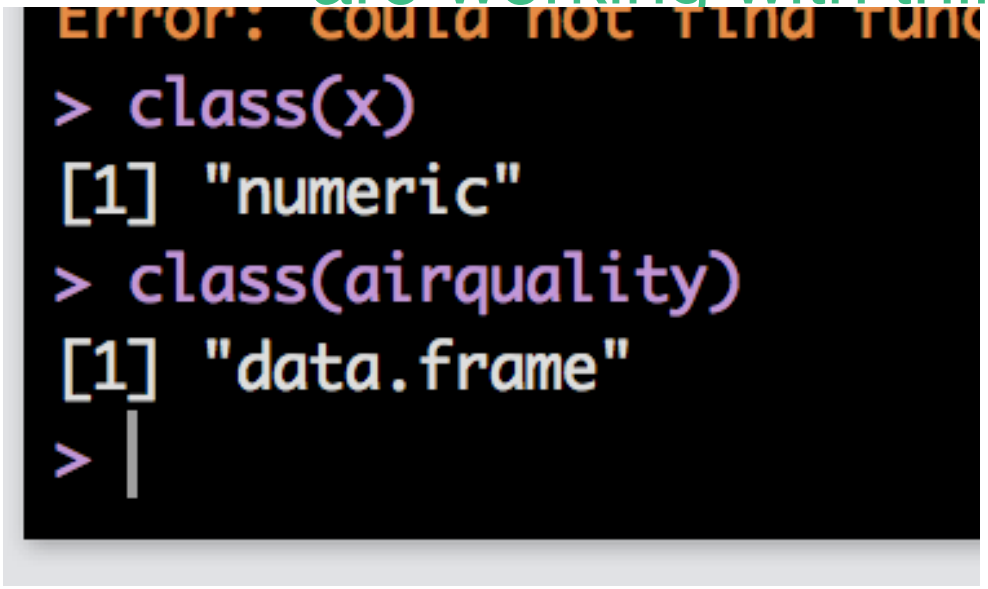
```
> rep(1:4, 10)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
```

You can combine all of these functions to get the exact sequences you need.

# Object Types

> Some things to try with vectors:

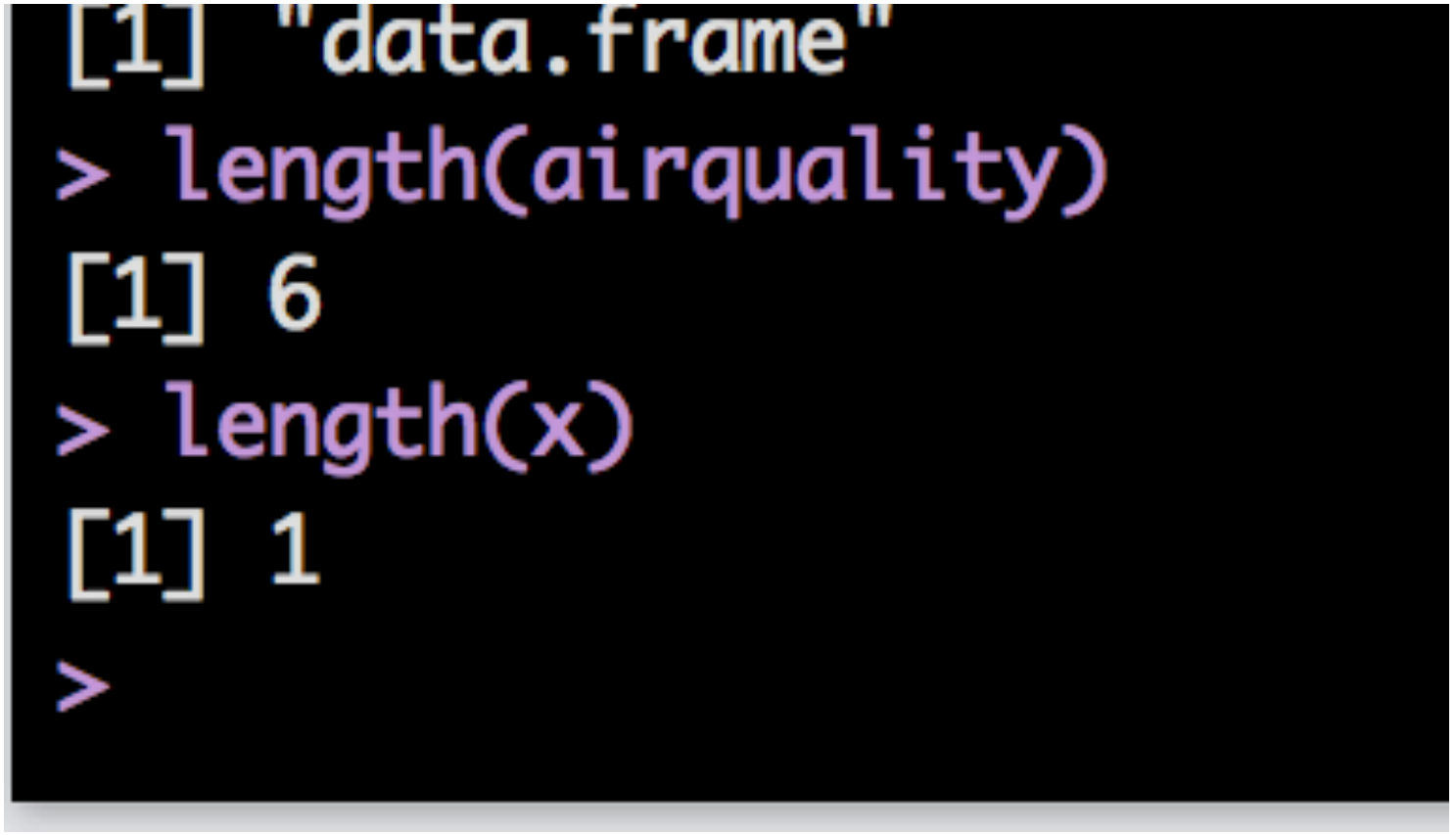
- Class(OBJECT NAME) – tells you the type of data stored or the type of object if you are working with things bigger than



# Object Types

> Some things to try with vectors:

- Length(OBJECT NAME): tells you the number of things in that object.
- That can be tricky though – the length of a vector is the number of objects, but the length of a list/data frame is the number of things in a list, not the number of items of each sub variable.


[1] "data.frame"
> length(airquality)
[1] 6
> length(x)
[1] 1
>

# Object Types

> Factors

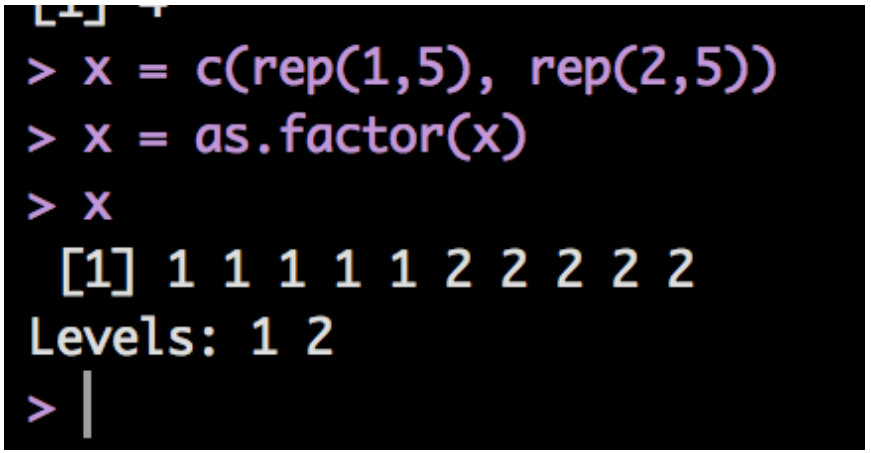
> Use the as.factor(OBJECT NAME) command.

- Let's make a vector:

  >X = c(rep(1,5), rep(2,5))

  >X = as.factor(X)

- You can also use text.
- More on the factor() command later.



## Object Types

```
> x = as.factor(c(rep("a",5), rep("b",5)))
> x
 [1] a a a a a b b b b b
Levels: a b
```

| | |
|---|---|
| x | Factor w/ 2 levels "a","b": 1 1 1 1 1 2 2 2 2 2 |

**Functions**

# Object Types

## > Matrices

- Matrices are vectors with dimensions (like a 2X3).
- You can create a matrix with the following:

>matrix(things to put into the matrix, rows,

```
> matrix(1:10, 2, 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
>
```

Notice it filled in the numbers going down.

# Object Types

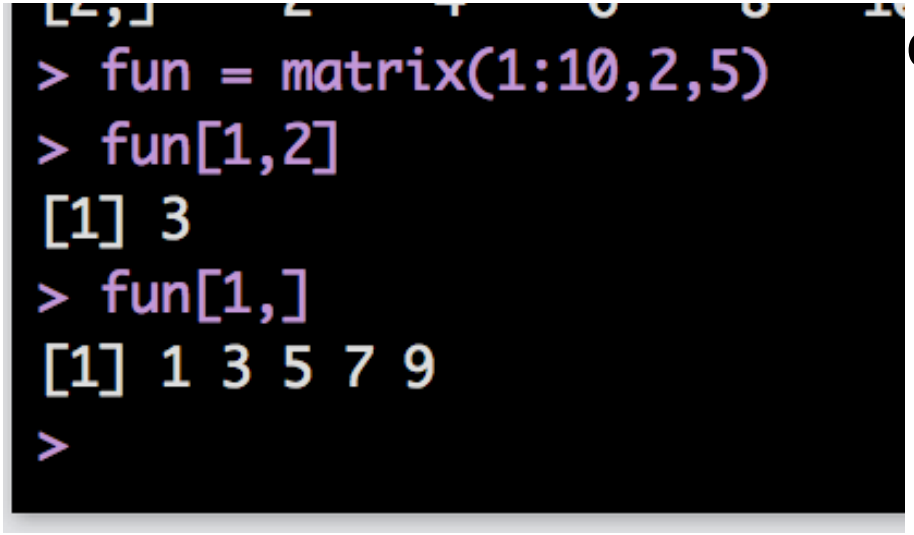
> Now what are the [,1] and [1,]?

- Matrices are called by listing [row, column]
- If you all one row or all one column:

  >fun[1, ]

  >Leave the indicator blank, and it will assume

em.



# Object Types

> What if I want to combine things?

- You can't use c(row, row) – c is for vectors because there's only one row.
- You can use:

  >cbind(column, column)

  >rbind(row, row)

- Let's combine two vectors:

# Object Types

> X = c(1:5) (note x = 1:5 is the same thing)

> Y = c(6:10

> cbind(x,y)

> rbind(x,y)

```
> x = 1:5
> y = 6:10
> cbind(x,y)
     x  y
[1,] 1  6
[2,] 2  7
[3,] 3  8
[4,] 4  9
[5,] 5 10
```

```
> rbind(x,y)
  [,1] [,2] [,3] [,4] [,5]
x    1    2    3    4    5
y    6    7    8    9   10
```

# Object Types

- Be careful when the objects are not the same length:

```
> x = 1:6
> cbind(x,y)
     x  y
[1,] 1  6
[2,] 2  7
[3,] 3  8
[4,] 4  9
[5,] 5 10
[6,] 6  6
Warning message:
In cbind(x, y) :
  number of rows of result is not a multiple of vector length (arg 2)
>
```

# Object Types

> Matrices are like vectors, everything must be of the same class/type (all numeric, all character, etc.)

# Object Types

> Data Frames are your friend

- We will use these the most often.
- They are special matrices, where everything must be the same length.
- BUT the columns can be different classes.
  - > So you can have characters, logicals, and numbers all together.

# Object Types

> Before talking about how to call values in a data frame, let's talk about lists.

> Lists are *vectors* that allow you to have different classes/types.

- Lists are handy because they allow you to have different lengths of groups, unlike data frames.
- You can create them with the list() function.

```
> x = list(1, "c", 3, "b")
> x
[[1]]
[1] 1

[[2]]
[1] "c"

[[3]]
[1] 3

[[4]]
[1] "b"
```

```
> b = list(1:4, "a", c("b", "d"))
> b
[[1]]
[1] 1 2 3 4

[[2]]
[1] "a"

[[3]]
[1] "b" "d"
```

What's with the [[]]?

- [[]] indicates which number of the list the smaller vector is, while [] indicate which item in that smaller vector.
- Try getting d out of the b list:

```
[1] "b"
> b[[3]][2]
[1] "d"
```

# Object Types

> Lists can also have names:

- names(b) = c("fun", "times", "had")

```
> names(b) = c("fun","times","had")
> b
$fun
[1] 1 2 3 4

$times
[1] "a"

$had
[1] "b" "d"

>
```

Now try to get d ou

```
> b$had[2]
[1] "d"
>
```

# Object Types

> Why even talk about lists when most of our data is going to be data frames?

- When you save data from a function (like anova or regression), you will often get all the information from that function as a list.
- It *really* helps to know how the heck to get the piece of information you need – especially when it's not provided automatically with the summary function.

# A quick example:

```
> regression = lm(Temp~Solar.R, data=airquality)
> ls(regression)
 [1] "assign"        "call"          "coefficients"  "df.residual"   "effects"
 [6] "fitted.values" "model"         "na.action"     "qr"            "rank"
[11] "residuals"     "terms"         "xlevels"
> 
```

```
> summary(regression)

Call:
lm(formula = Temp ~ Solar.R, data = airquality)

Residuals:
     Min       1Q   Median       3Q      Max
-22.3787  -4.9572   0.8932   5.9111  18.4013

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 72.863012   1.693951  43.014  < 2e-16 ***
Solar.R      0.028255   0.008205   3.444 0.000752 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 8.898 on 144 degrees of freedom
  (7 observations deleted due to missingness)
Multiple R-squared:  0.07609,   Adjusted R-squared:  0.06967
F-statistic: 11.86 on 1 and 144 DF,  p-value: 0.0007518
```

# So what if you wanted the residuals?

```
> regression$residuals
          1           2           3           4           7           8           9
-11.23139214  -4.19705852  -3.07295216 -19.70671207 -16.31114720 -16.66022048 -12.39984979
         10          12          13          14          15          16          17
 -9.34441067 -11.09619796 -15.05685550 -12.60478136 -16.69956294 -18.30005938 -15.53718427
         18          19          20          21          22          23          24
-18.06687318 -13.96100377 -12.10621563 -14.08904882  -8.90449451 -12.56937760 -14.46243805
         25          26          28          29          30          31          32
-17.72781757 -22.37874429  -6.23032199   1.01682058  -0.16379505  -4.74605453  -2.94383697
```

# Object Types

> Now what is up with the $?

- Lists and data frames are dimensional, so you can use the following to find something:
  - > Lists: [[number]][number]
  - > DF: [row, column]

# Object Types

> That's tedious.

> If they have names, you can use the $ to indicate the column (or sub-vector) name.

> Therefore:

- airquality$Temp and airquality[,4] are the same thing.

# Object Types

> REMEMBER:

- Just because you know that the airquality dataset is open and Temp is a variable, doesn’t mean that R cares.
- You must specify what object to look at for the Temp piece.

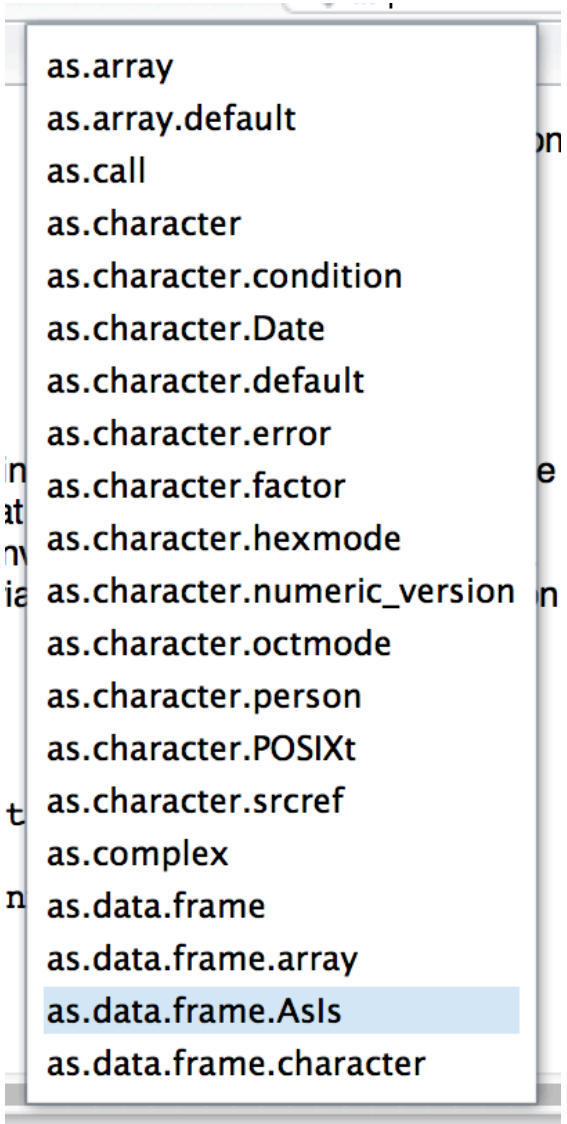
# Object Types

- You can also do the following:
  - with(OBJECT NAME, THING YOU WANT TO DO)
    - With applies the data frame/list name to all the variables you are using. It basically says “hey the stuff they want you to use is HERE”.
  - attach(OBJECT NAME)
    - Attach does the same thing, but it’s more permanent. It leaves that object open the whole time.
    - This version can get tricky if you have multiple

# Object Types

> Let's say you want to change from one type to another:

- as.data.frame
- as.numeric
- as.factor
- as.logical
- Etc.



# Object Types

> Let's make a character vector:

- Y = c("a", "b", "c")
- Try changing that to other types of data:

>X = as.numeric(y)

```
Error: object 'a' not found
> y = c("a", "b", "c")
> x = as.numeric(y)
Warning message:
NAs introduced by coercion
> x
[1] NA NA NA
>
```

# Object Types

> Summary: it's good to understand types of objects, because it allows you to understand:

- How to get to the information you need
- Why you might be getting NaN or NA values
- Etc.

# Subsetting

> *Subsetting* is pulling out the rows/columns that you need given some criteria.

- We already talked about how to select one row/column with [1,] or [,1] and the $ operator.
- What about cases you want to select based on scores, missing data, etc.?

# Subsetting

## > Examples

- Let's use the airquality dataset.
- Pick out the first two rows of the whole dataset.
- airquality[1:2,]
- Remember t
  these by set
  name.

```
> airquality[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
>
```

# Subsetting

> You can also use the logical operator to determine how to pick something.

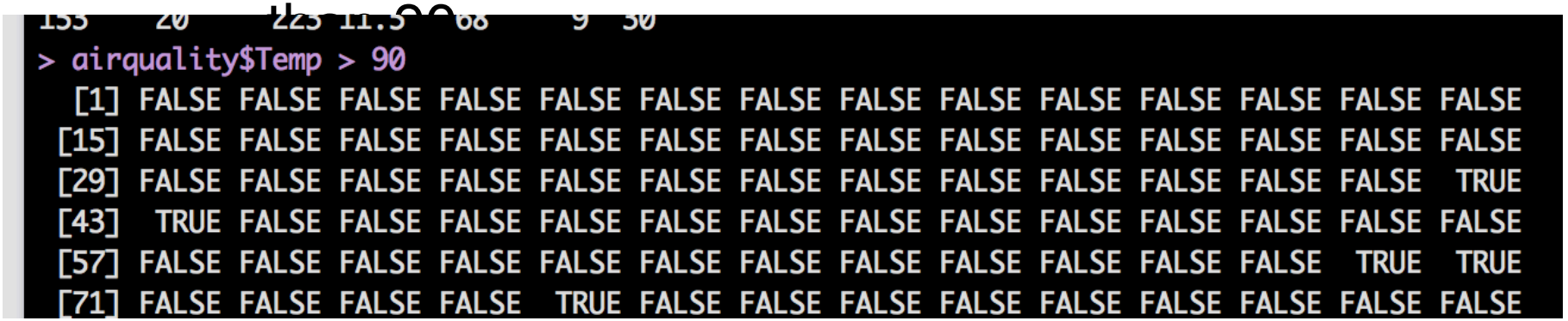
- Logical
- airquality[airquality$Temp> 90,]

```
> airquality[airquality$Temp>90,]
    Ozone Solar.R Wind Temp Month Day
42     NA     259 10.9   93     6  11
43     NA     250  9.2   92     6  12
69     97     267  6.3   92     7   8
70     97     272  5.7   92     7   9
75     NA     291 14.9   91     7  14
102    NA     222  8.6   92     8  10
120    76     203  9.7   97     8  28
```

# Subsetting

> How does the logical operator work?

- It analyzes each row/column for the appropriate logical question
  - > So, in this example, we asked it to analyze the Temp column.
  - > Each row was tested to see if it was greater



# Subsetting

> Therefore, all the columns with TRUE were selected.

- True = 1
- False = 0

> You can use this feature to your advantage when selecting for missing cases.

# Subsetting

> You can use multiple arguments to get the exact data you want.

- airquality[airquality$Temp>90 & airquality$Ozone<90,]

```
127    91    189  4.6   93     9   4
> airquality[airquality$Temp>90 & airquality$Ozone<90,]
     Ozone Solar.R Wind Temp Month Day
NA      NA      NA   NA   NA    NA  NA
NA.1    NA      NA   NA   NA    NA  NA
NA.2    NA      NA   NA   NA    NA  NA
NA.3    NA      NA   NA   NA    NA  NA
120     76     203  9.7   97     8  28
122     84     237  6.3   96     8  30
123     85     188  6.3   94     8  31
125     78     197  5.1   92     9   2
126     73     183  2.8   93     9   3
>
```

You'll still get NAs though, we'll get to how to deal with them.

# Subsetting

> Let's say you wanted everything BUT the first column

- airquality[,-1]

> Let's say you wanted everything BUT the last column

- airquality[,-ncol(airquality)]

```
153    20     223 11.5   68     9
> airquality[,-ncol(airquality)]
    Ozone Solar.R Wind Temp Month
1      41     190  7.4   67     5
2      36     118  8.0   72     5
3      12     149 12.6   74     5
```

```
126     73     183  2.8   93     9
> airquality[,-1]
    Solar.R Wind Temp Month Day
1       190  7.4   67     5   1
2       118  8.0   72     5   2
3       149 12.6   74     5   3
```

# Subsetting

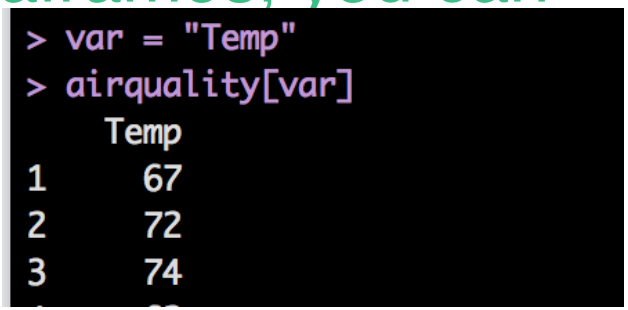
> Something important:

- When subsetting lists or dataframes, you can do this:

  > var = “Temp”

  > airquality[var]

  > Airquality$Temp




- But not:

  > airquality$var

- When you use the $, an exact match is expected, but the [] will fill in with the variable value.

# Subsetting

> Another subsetting function is subset()

> Subset requires

- (data set name, logical function)
- Optional includes what columns you want to select

> subset(airquality, Temp > 80, select = c(Ozone, Temp))

# Missing Values

> Missing values are considered NAs

- NaN is not a number, NA is a missing value.
- However, NaN sometimes is the error when you get when you try to do something with a NA.

> You sometimes can get away with having them in the dataset, often not.

# Missing Values

> Many functions have options that allow you to omit them to run the function.

> Not all of them, and they often do the missing omission differently.

# Missing Values

> There are several NA functions:

- But they often don't do what you think.
- complete.cases(DATA SET)

```
> fun = complete.cases(airquality)
> fun
  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
 [15]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
 [29]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
 [43] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
 [57] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
 [71]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
 [85]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
 [99]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
[113]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[127]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[141]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
>
```

# Missing Values

> The missing values functions often return logical vectors with:
  - TRUE = yes missing data
  - FALSE = no missing data

> Complete cases tells you if the entire row is complete.
  - TRUE = complete, FALSE = incomplete

# Missing Values

> Another function that is faster than complete cases is na.omit()

> na.omit(airquality)

# Missing Values

> is.na() works in much the same way, but works in individual values, rather than the entire row like complete cases.

> Usually, you can use this function for individual columns or rows.

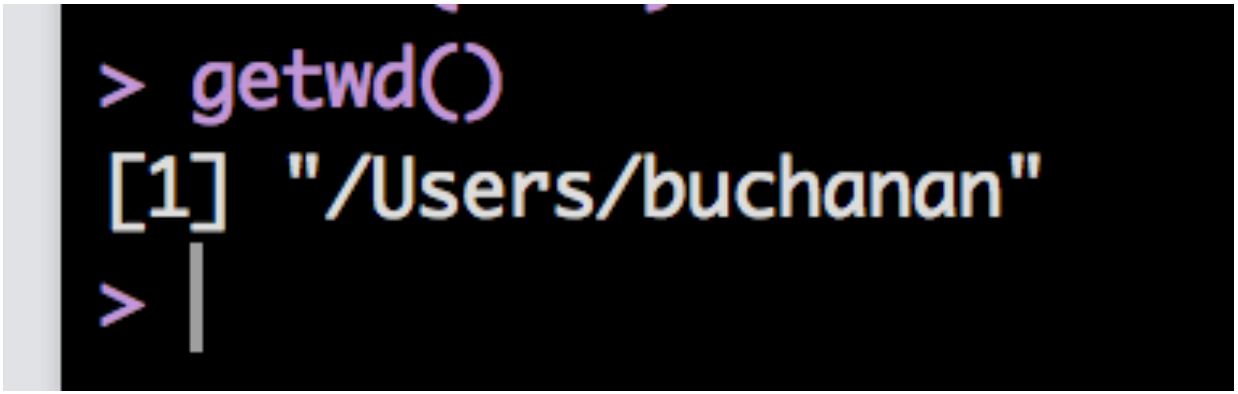
- Is.na can help with checking for missing values in data screening.

# Working Directory

> The *working directory* is where you are currently saving data in R.

> What is the current working directory?

- Type in getwd()
- You'll see the path for your directory



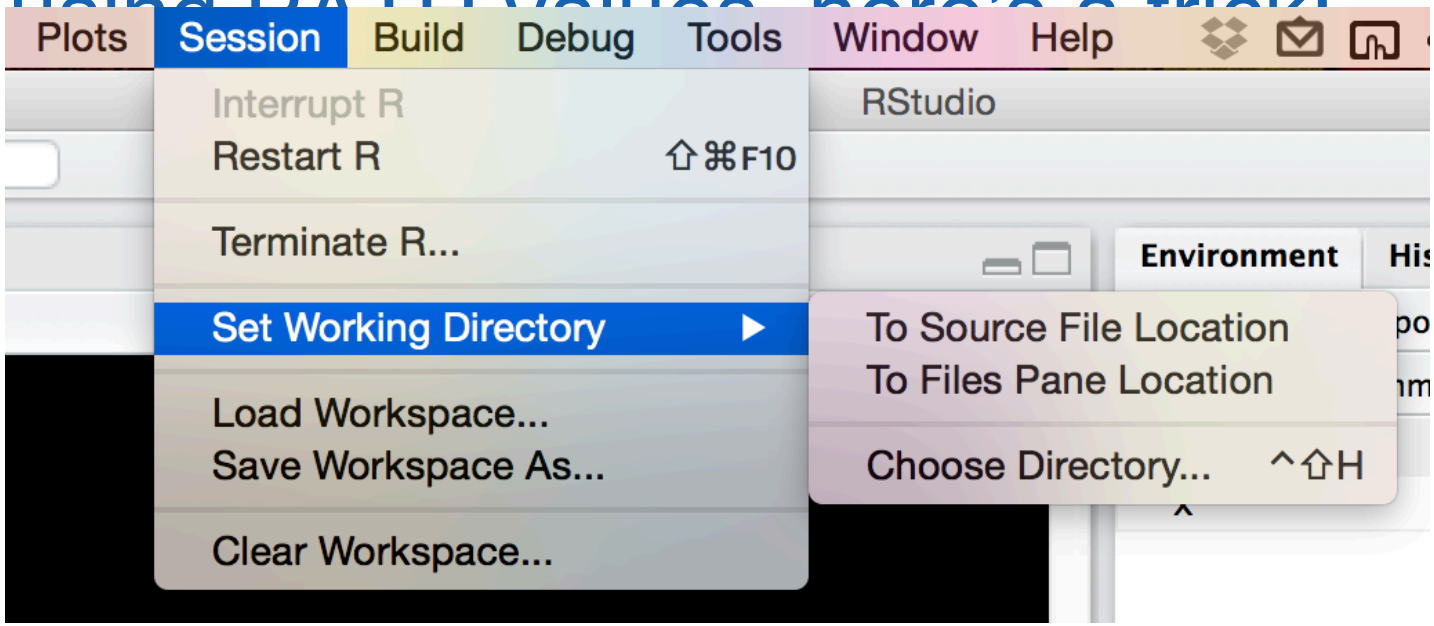

Note: I'm using a M

# Working Directory

> How to set the working directory:

- setwd("PATH")

> If you aren't really familiar or good with using PATH values, here's a trick:

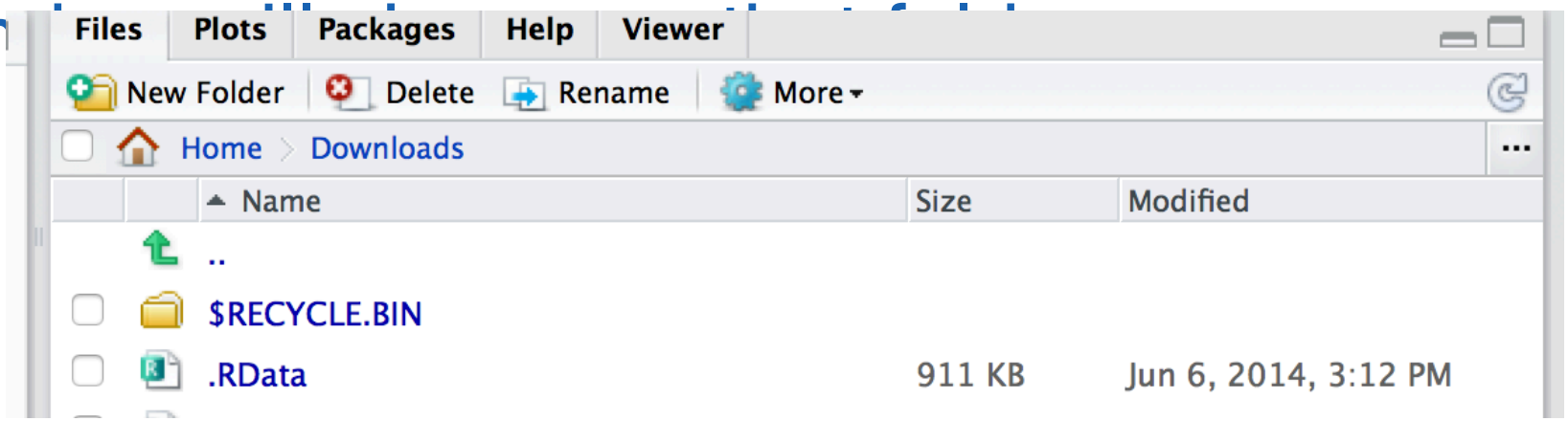


# Working Directory

> Now you can pick the folder you are interested in saving your files to.

```
> setwd("~/Downloads")
```

> Once you do that, the bottom right win



# Working Directory

> Why is all this important?

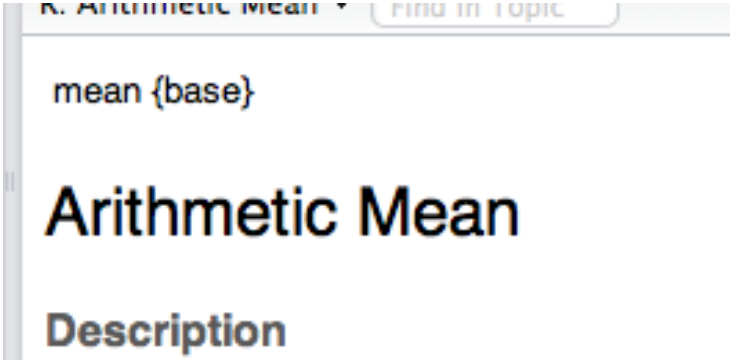
- You can use getwd and setwd in saved R scripts to point the analyses to specific files.
- Basically, you can set it to import a file from a specific spot and use that over and over, rather than importing the file each time you open R.

# Packages

> Packages are add-ons to R that allow you to do different types of analyses, rather than code them yourself.

> R comes with many pre-programming functions – lovingly called *base R.*

At the top of the help window, you can tell which package a function is included in.



# Packages

> Packages are checked/monitored by the CRAN people.

- That means there's some oversight to them.
- Many other types of functions can be downloaded from GitHub.

  > Use at your own risk.

# Packages

> Note: each time R updates, the packages sometimes come with it, sometimes they don't.

- If you are looking for a specific package, and it doesn't want to install the normal way (next couple slides), but you know it exists → google it and get the TAR files.
- You can install them from the TAR files.

# Packages

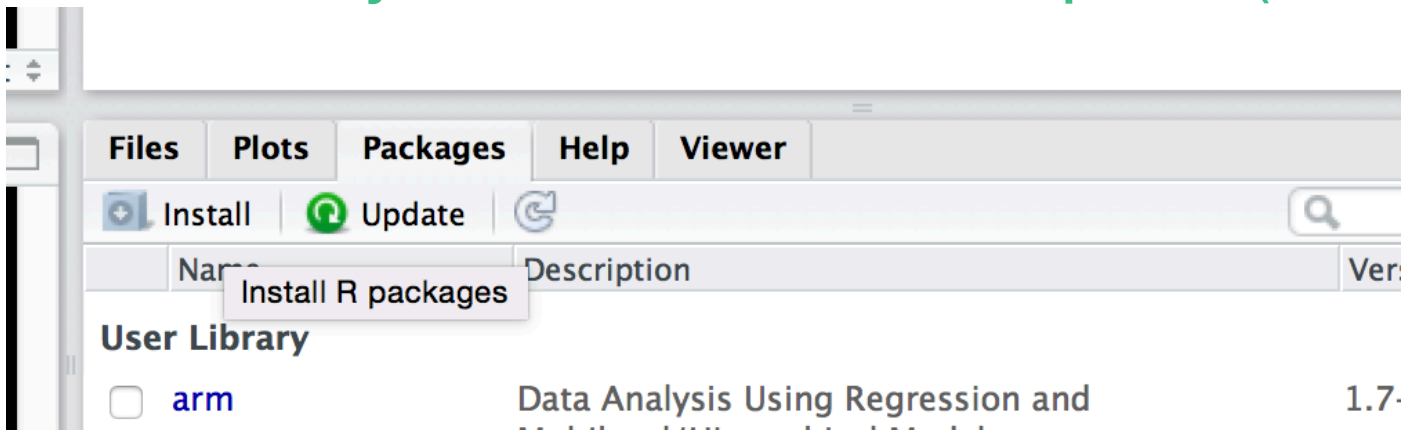
> How to install:

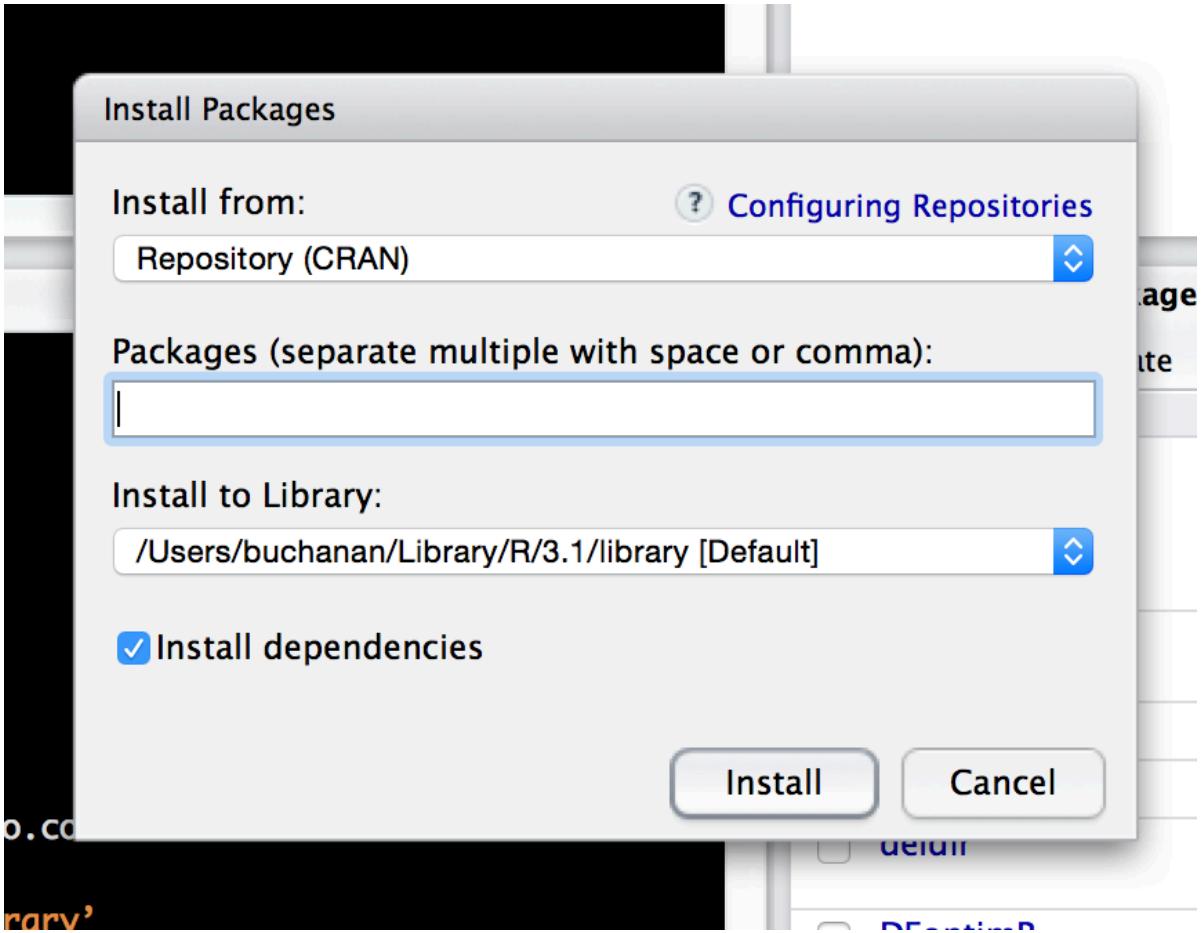
- Console: install.packages("NAME OF PACKAGE")
- Let's try it!
  - > install.packages("car")
  - > Note: you have to be connected to the internet for packages to install.

# Packages

> How to install:

- Through RStudio
- Click on packages, click on install.
- Note: you can see here in this window what all you have installed, and if you click on them, you will load that help file (or click






Start typing the name of the package – a drop down will ap

# Packages

> Now it's installed! Awesome!

> That doesn't mean that it *loads* every time.

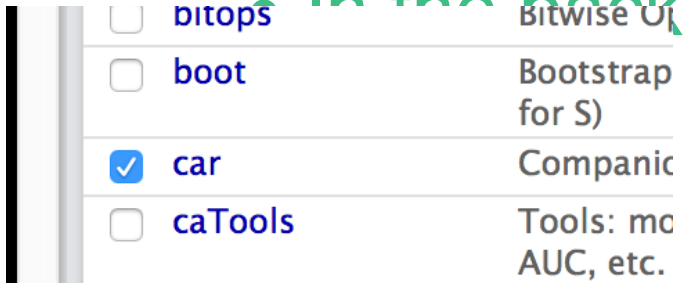
- Annoying!
- But this saves computing power by not loading unless you need it.
- You *will* run something without turning on the right package. It's cool – all the cool kids do it.

# Packages

> Packages are also called *libraries*.

> You can load them two ways:

- In the console: library(car) (look no “” this time).
- In the packages window by clicking on the




```
> library("car", lib.loc="/Library/Frameworks/R.framework/Versions/3.1/Resources/library")
>
```

# Packages

> I suggest adding the code to your script to load the packages you need to save yourself the headache of trying to remember which ones were important.

# Working with Files

> Data files (like the airquality dataset) come with base R.

- You don’t technically have to load them, but you can get them to appear in environment window by:
- data(SET NAME)

# Working with Files

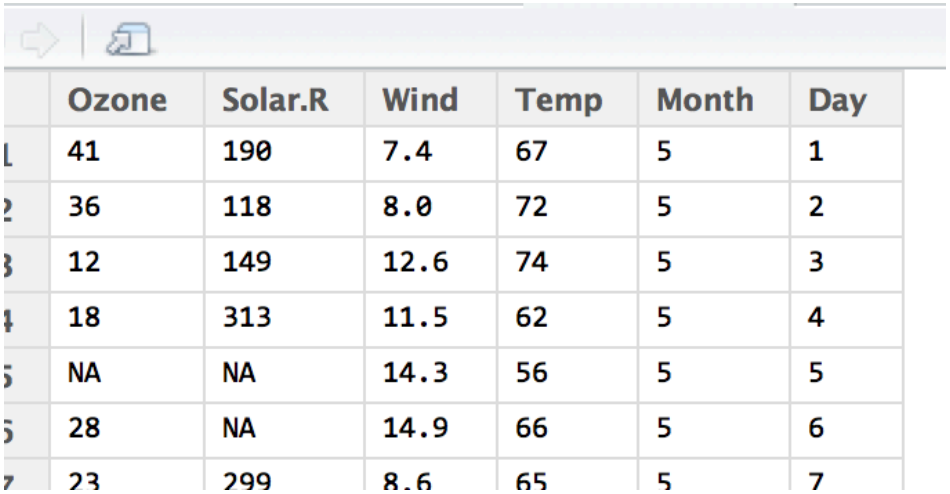
> If you want to see what's available, type data()

> Use the help(DATA SET NAME) or ?DATA SET NAME to see what is included/is part of the data set.

# Working with Files

> RStudio can give you somewhat of a visual.

- Type View(airquality) to get a visual (note V is capital)
- Or click on it in environment window



| | 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 |
| 7 | 23 | 299 | 8.6 | 65 | 5 | 7 |

# Working with Files

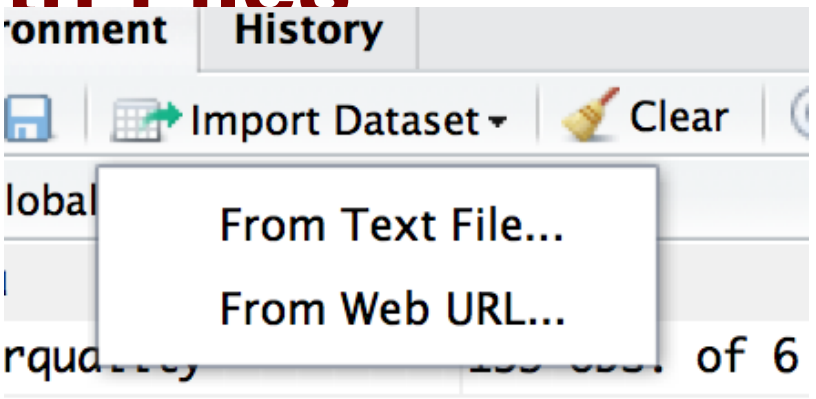
> You can import all types of files, including SPSS files.

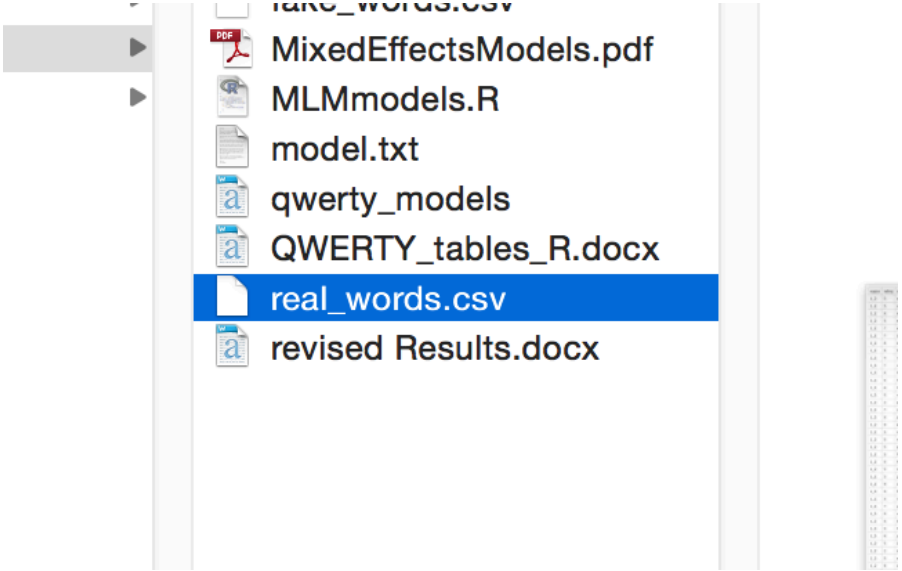
- I find .csv easiest but that's me.
- You can do .txt with any separator (comma, space, tab)

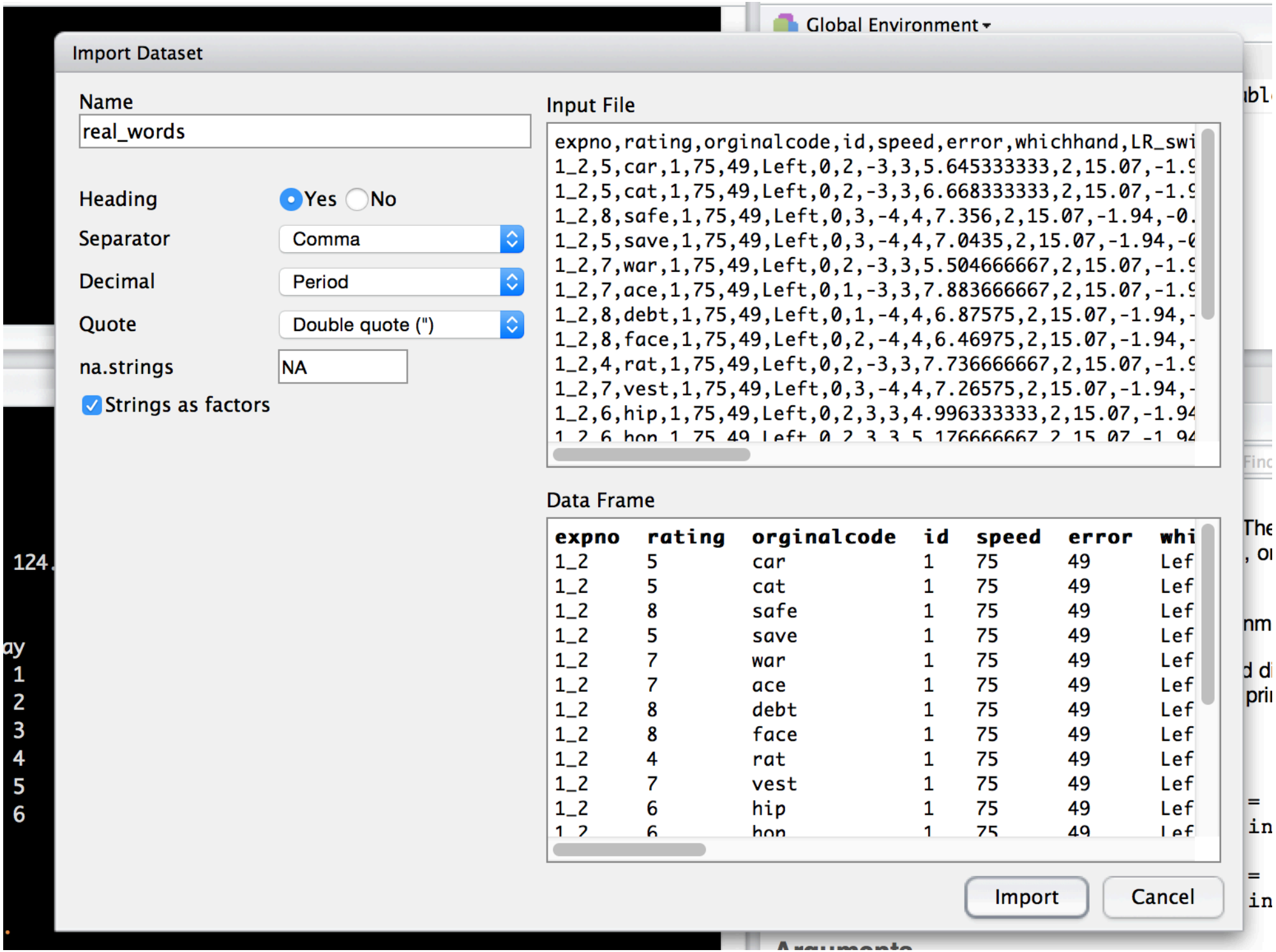
# Working with Files

## > Import from Rstudio

- Pick your file and click open




Global Environment
Import Dataset
Name
real_words
Heading
Yes
No
Separator
Comma
Decimal
Period
Quote
Double quote (")
na.strings
NA
Strings as factors
Input File
expno,rating,orginalcode,id,speed,error,whichhand,LR_swi
1_2,5,car,1,75,49,Left,0,2,-3,3,5.645333333,2,15.07,-1.9
1_2,5,cat,1,75,49,Left,0,2,-3,3,6.668333333,2,15.07,-1.9
1_2,8,safe,1,75,49,Left,0,3,-4,4,7.356,2,15.07,-1.94,-0.
1_2,5,save,1,75,49,Left,0,3,-4,4,7.0435,2,15.07,-1.94,-0
1_2,7,war,1,75,49,Left,0,2,-3,3,5.504666667,2,15.07,-1.9
1_2,7,ace,1,75,49,Left,0,1,-3,3,7.883666667,2,15.07,-1.9
1_2,8,debt,1,75,49,Left,0,1,-4,4,6.87575,2,15.07,-1.94,-
1_2,8,face,1,75,49,Left,0,2,-4,4,6.46975,2,15.07,-1.94,-
1_2,4,rat,1,75,49,Left,0,2,-3,3,7.736666667,2,15.07,-1.9
1_2,7,vest,1,75,49,Left,0,3,-4,4,7.26575,2,15.07,-1.94,-
1_2,6,hip,1,75,49,Left,0,2,3,3,4.996333333,2,15.07,-1.94
Data Frame
expno rating orginalcode id speed error whi
1_2 5 car 1 75 49 Lef
1_2 5 cat 1 75 49 Lef
1_2 8 safe 1 75 49 Lef
1_2 5 save 1 75 49 Lef
1_2 7 war 1 75 49 Lef
1_2 7 ace 1 75 49 Lef
1_2 8 debt 1 75 49 Lef
1_2 8 face 1 75 49 Lef
1_2 4 rat 1 75 49 Lef
1_2 7 vest 1 75 49 Lef
1_2 6 hip 1 75 49 Lef
Import
Cancel

# Working with Files

> This process is the same as:

- real_words <- read.csv(”FILE NAME")
- The read.csv function – which has a lot more settings, but this process make it easy to start working with files.

# Working with Files

- You can also use the read.table function – which reads more than just csv files, allows you more flexibility in how you import the files.

# Working with Files

> Importing SPSS files.

- You need the haven package.

> read_spss(“nameoffile.sav”)

> read_sav(“nameoffile.sav”)

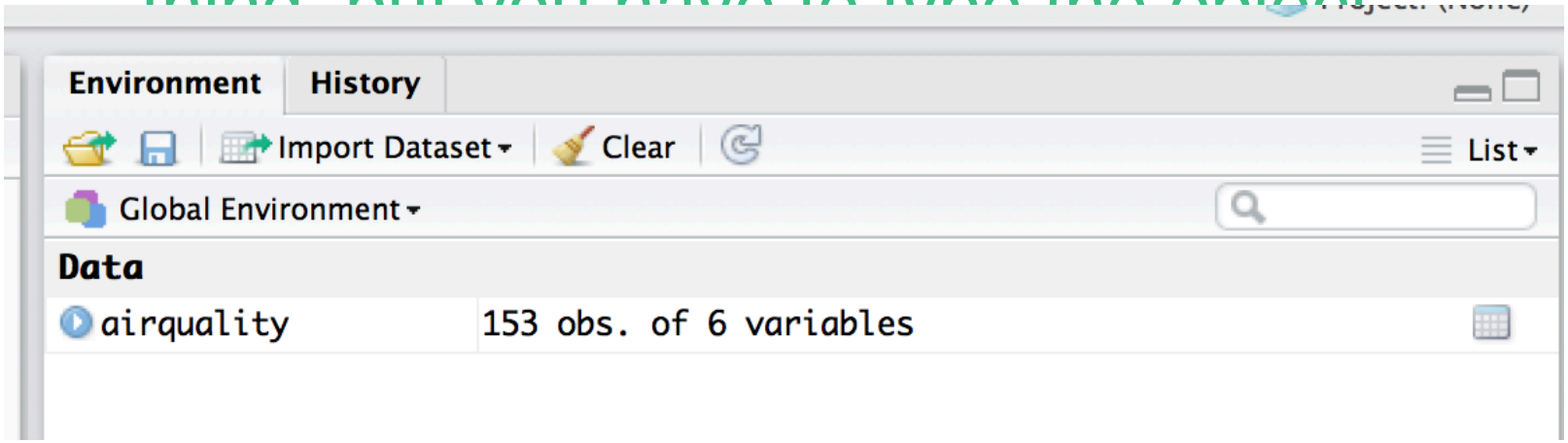
# Working with Files

> All of these options will import your data set as a data frame.

# Working with Files

## > Clear the workspace

- You don't have to do this, but it helps if you want to start over. Click on the broom in the environment window.
- rm() and remove() functions do the same thing, but you have to type the object



# Functions

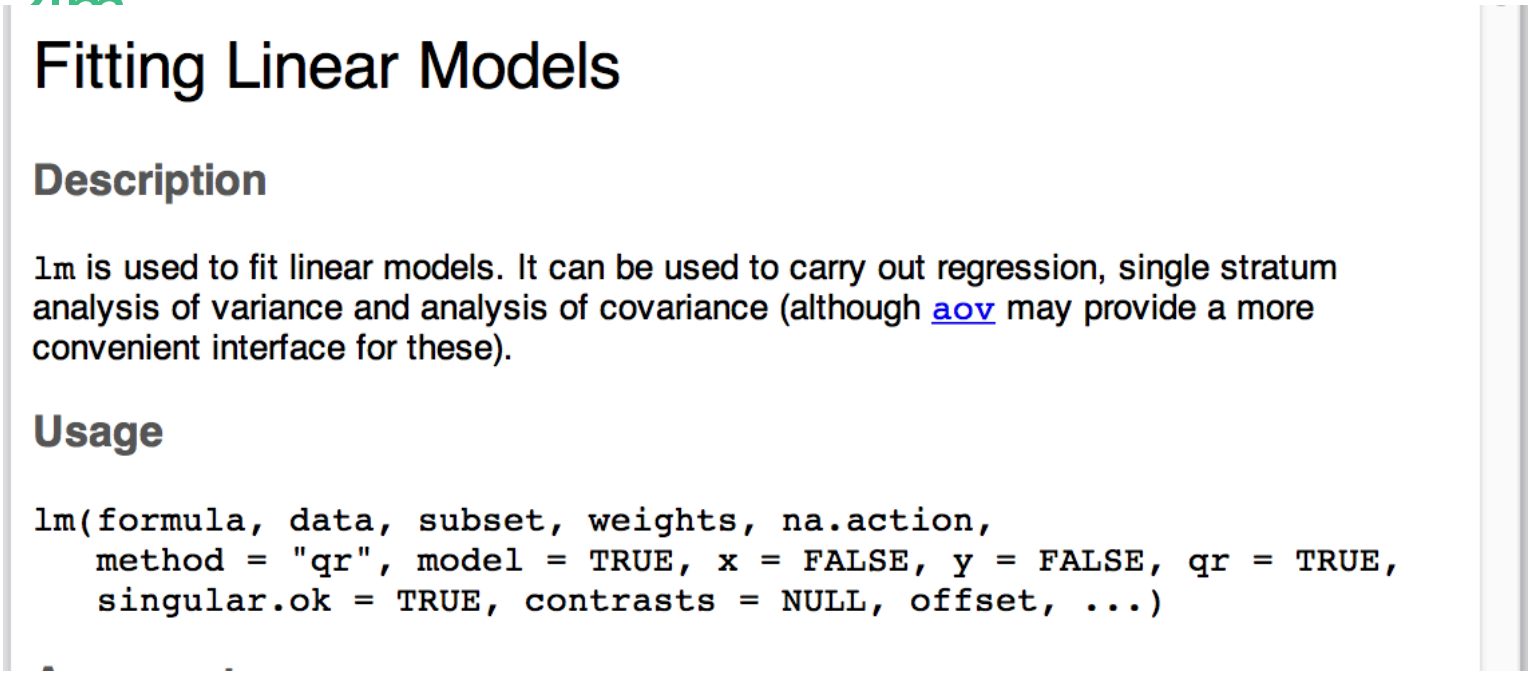
> Functions are pre-written code to help you run analyses (so you don't have to do the math yourself!).

- So there are functions for the mean, variance, z-tests, ANOVA, regression, etc.

# Functions

> How to get help on a function (or anything really)

- ?function/name/thing
- Try ?lm



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, ...)
```

# Functions

> More help on functions:

- help(function) – same as ?function
- args(function) – tells you all the arguments that the function takes

```
> 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
```

# Functions

> What do you mean *arguments?*

> Functions have a couple of parts

- The name of the function – like lm, mean, var
- The arguments – all the pieces inside the () that are required for the function to run.

# Functions

> Get help on functions:

- example(function)
- Gives you an example of the function in action.

# Functions

> Let's write a very simple function to exponents.

> You do have to save them, set them equal to something.

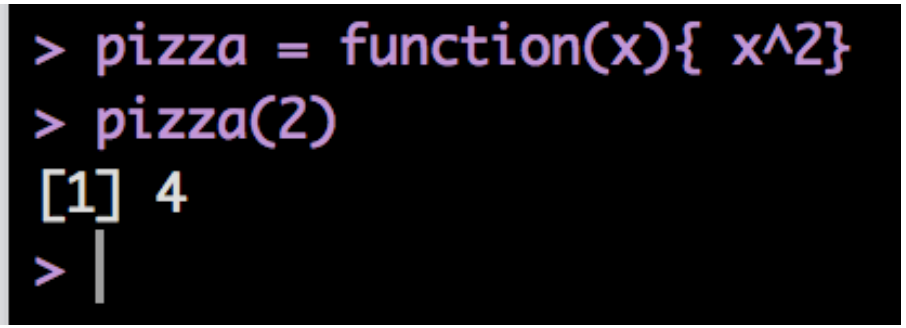
- pizza = function(x) { x^2 }
- You can make more complex function, adding more to the (x) part like (x,y,z).
- The variables can be named anything, they just have to match in () and within the {}.

# Functions

- pizza = function(x) { x^2 }
  - This part is called the formal argument – that's where you define the function.

> pizza(2)

  - The actual argument – that's where you call the function and use it.



# Functions

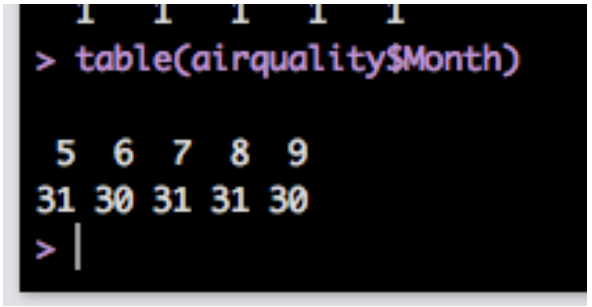
> Example functions:

- table()
- summary()
- cov()
- cor()
- mean()
- var()
- sd()

# Table Function

> The table function gives you a frequency table of the values in a vector/column.

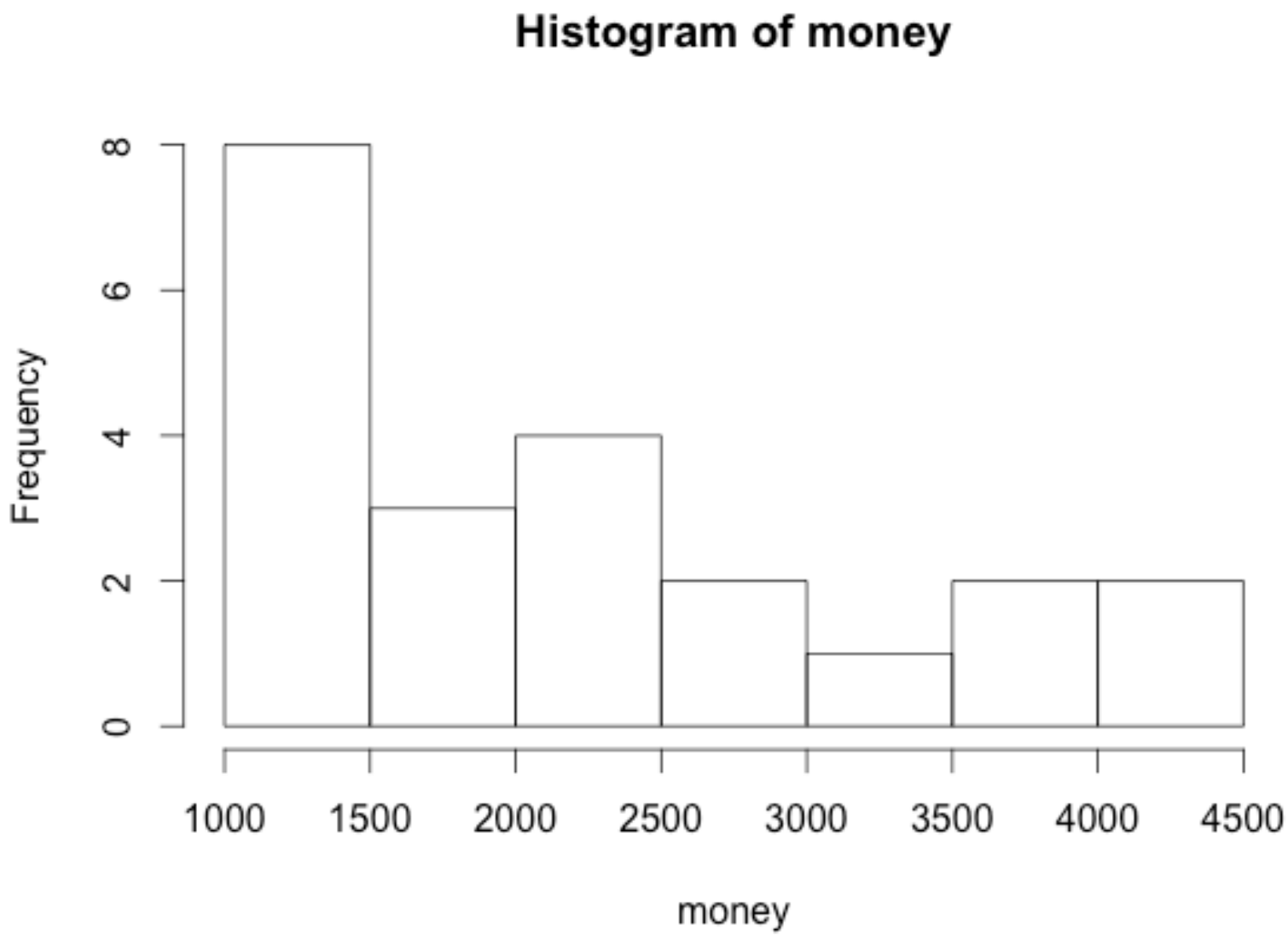
> table(OBJECT NAME)



# Histogram Function (hist)

> The hist() function will give you a histogram of your vector/column (or all the columns if they are the right type).

> hist(data/column)


Histogram of money
Frequency
0
2
4
6
8
1000
1500
2000
2500
3000
3500
4000
4500
money

# Summary Function

> The summary function has several uses:

- On a vector/data frame, it will give you basic statistics on that information
- On a statistical analysis, it will give you the summary output.

```
> summary(airquality)
     Ozone           Solar.R          Wind             Temp           Month            Day
 Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00   Min.   :5.000   Min.   : 1.0
 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00   1st Qu.:6.000   1st Qu.: 8.0
 Median : 31.50   Median :205.0   Median : 9.700   Median :79.00   Median :7.000   Median :16.0
 Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88   Mean   :6.993   Mean   :15.8
 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00   3rd Qu.:8.000   3rd Qu.:23.0
 Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00   Max.   :9.000   Max.   :31.0
 NA's   :37       NA's   :7
>
```

# Summary Function

```
> regression = lm(Temp ~ Month, data=airquality)
> summary(regression)

Call:
lm(formula = Temp ~ Month, data = airquality)

Residuals:
     Min       1Q   Median       3Q      Max
-20.5263  -6.2752   0.9121   6.2865  17.9121

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  58.2112     3.5191  16.541  < 2e-16 ***
Month         2.8128     0.4933   5.703 6.03e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 8.614 on 151 degrees of freedom
Multiple R-squared:  0.1772,    Adjusted R-squared:  0.1717
F-statistic: 32.52 on 1 and 151 DF,  p-value: 6.026e-08
```

# Descriptives

## > Basic Descriptives

- cov() – covariance table
- cor() – correlation table
- mean() – average
- var() – variance
- sd() – standard deviation

# Descriptives

> Try taking the average of airquality$Ozone

```
> mean(airquality$Ozone)
[1] NA
```

> mean(airquality$Ozone)

- Darn!
- Stupid NAs!

> We've talked about how to deal with NAs globally, but here's how they are handled in functions (generally)

# Descriptives

> Try this line instead:

- mean(airquality$Ozone, na.rm=TRUE)
- Na.rm = remove NAs ??
- The default is FALSE (lame).

> So you can subset the data or use that argument to tell it to ignore NAs.

```
> mean(airquality$Ozone, na.rm=TRUE)
[1] 42.12931
```

**when they are actually helpful that is.

# Descriptives

> Help / args are your friend**.

- The var() function has na.rm
- Cov() and cor() do not.

**Usage**

```
var(x, y = NULL, na.rm = FALSE, use)

cov(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))

cor(x, y = NULL, use = "everything",
    method = c("pearson", "kendall", "spearman"))
```

# Descriptives

> Try:

- cor(airquality, use =

```
> cor(airquality, use="pairwise.complete.obs")
              Ozone     Solar.R        Wind       Temp       Month         Day
Ozone    1.00000000  0.34834169 -0.60154653  0.6983603  0.164519314 -0.013225647
Solar.R  0.34834169  1.00000000 -0.05679167  0.2758403 -0.075300764 -0.150274979
Wind    -0.60154653 -0.05679167  1.00000000 -0.4579879 -0.178292579  0.027180903
Temp     0.69836034  0.27584027 -0.45798788  1.0000000  0.420947252 -0.130593175
Month    0.16451931 -0.07530076 -0.17829258  0.4209473  1.000000000 -0.007961763
Day     -0.01322565 -0.15027498  0.02718090 -0.1305932 -0.007961763  1.000000000
> |
```

# Rescoring Functions

> scale() will mean center or z-score your column.

```
> meancenter = scale(airquality$Ozone)
> mean(meancenter, na.rm=TRUE)
[1] -2.818084e-17
> sd(meancenter, na.rm=TRUE)
[1] 1
```

- scale(VARIABLE)
  - > Z-scored
- scale(VARIABLE, scale=FALSE)
  - > Mean centered

```
> meancenter = scale(airquality$Ozone, scale=FALSE)
> mean(meancenter, na.rm=TRUE)
[1] -9.830199e-16
> sd(meancenter, na.rm=TRUE)
[1] 32.98788
>
```