

# *A “Level-Zero” Tutorial for Getting Started with R*

*April Galyardt*

*January 2, 2014*

A plethora of R tutorials are available. Why do we need another one? The issue is that most existing tutorials are written for people with some experience with programming or using command line interfaces. These tutorials can be difficult to work through if you’re trying to figure out file-types, working directories, different types of objects, and syntax for the first time all at once.

We’re going to work through all of these things one thing at a time, and try to keep our sense of humor. I really like the character Po from Kung Fu Panda. He may start at (the previously non-existent) level zero, but reaches mastery through his own path. That’s what we’re aiming for here.

Note, do not try to ‘read’ this tutorial. Instead, work through it. This tutorial is written for you to type each command into R as you get to it. If you skip a command, what comes next may not work, or may not make sense.

If you get an idea, try it out. Learn by doing, and don’t be afraid to play around with it.

## *Downloading and Installing R*

The main page for R is <http://cran.r-project.org/>, and it has versions of R available for Linux, Mac and Windows. You want to download the latest version of the “precompiled binary distribution”. These are the ones that will install just like any other application you might use on your computer, and they are the 3 links on the top of the page.

When you click on the link corresponding to your operating system, you may be asked to choose a CRAN mirror. “Mirror” means it’s an exact copy of the website. The different mirrors allow for faster downloads in different locations. Choose the mirror that’s closest to you. For those of us in Georgia, that’s Oak Ridge, TN. Once you download the software, you can install it as usual.

One important note here is that there are subtle differences in the GUI (graphical user interface) for R in different operating systems (OS). For example, the drop-down menu options for the Mac distribution are organized slightly differently than Windows drop-down menu options. However, the commands will not be. Any command you give R for an analysis will be exactly the same across any OS.



Figure 1: The R logo. If you don’t see this, you’re probably on the wrong website.

This tutorial will use screenshots from the Mac version, but other than appearance, everything should be similar in Windows.

### *Opening R for the First Time*

When you open R, the main window that opens is the R console, shown in Figure 2. A second window, for a script editor, may also open. We'll use the script editor later, but for now, just close the window. We're going to work just with the console and the command line.

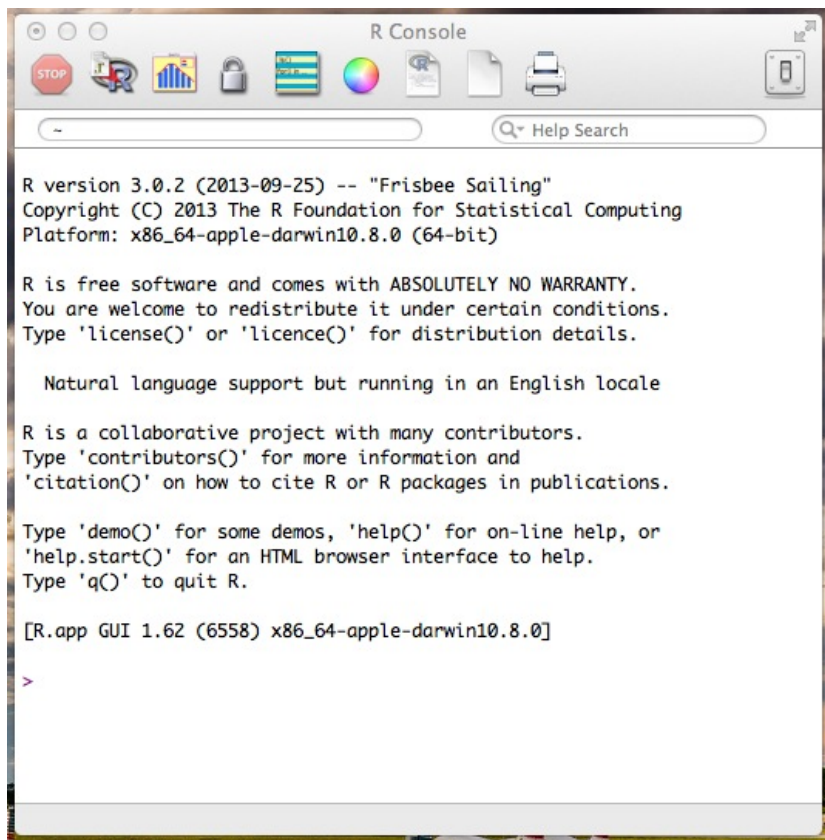


Figure 2: The R Console. The > at the bottom denotes the command line where you will type commands.

### *Using the Command Line*

We'll start by just using the command line like a ridiculously fancy calculator. To calculate  $5 * 3 + 1$ , at the command line, type

```
> 5*3+1
```

Similarly,  $\frac{6}{7} + 4 * 55$  is entered as

```
> 6/7 + 4*55
```

Notice that for arithmetic, spaces don’t matter. Try

```
> 6 / 7 + 4 * 55
```

Next are exponents, to calculate  $3^6$  enter

```
> 3^6
```

Now, sometimes we need to be more careful about the order of operations. Let’s say we want to calculate

$$5 + 3(2 - 10)^2$$

Most of the time spaces don’t matter in R, but there is one exception that we’ll talk about in the [Types of R objects](#) section.

$$5 + 3(2 - 10)^2 = 197$$

Here you need to enter

```
> 5 + 3*(2-10)^2
```

Try entering this without the \*. You should get an error:

```
> 5 + 3(2-10)^2
```

```
Error: attempt to apply non-function
```

When you use parenthesis with an arithmetic operation, like multiplication, R knows that you’re specifying the order of operations. But when you use parenthesis without an arithmetic operation, then R thinks you’re trying to use a function. In this example, R thinks you’re trying to use the `3(...)` function, but `3` isn’t a function, so it’s giving you an error.

We’ll talk about functions in detail, in the [Functions and “Help” Files](#) section.

**THE NIFTY UP-ARROW TRICK.** Let’s say that we want to calculate something more complicated, say

$$\left( \frac{1 + (2 - 1/3)^2}{5(1 - 3 * 6)} \right)^3$$

Until you get more practice, you want to be very careful about typing all of this out at one time. There’s bound to be a mis-placed parenthesis or some other typo. So you work through the order of operations one at a time, building up the command. Just start with the first step

```
> (2 - 1/3)^2
[1] 2.777778
```

Now, press the “Up arrow” key ↑. This will bring back up the last command that you entered on the command line. You can actually keep scrolling through all of your previous commands, but we just need the last command. Now we can add “1 +” and we’ll have the numerator.

```
> 1+(2 - 1/3)^2
[1] 3.777778
```

Press ↑ again, and this time, we’ll add parenthesis and the denominator.

```
> (1+(2 - 1/3)^2)/(5(1-3*6))
```

```
Error: attempt to apply non-function
```

Oops, I forgot that \* in the denominator! I can just press ↑ and fix the typo. This is one reason to build up complicated instructions one piece at a time. Fix the inevitable mistakes as you go.

```
> (1+(2 - 1/3)^2)/(5*(1-3*6))
```

```
[1] -0.04444444
```

Now, for the final step, I need to put parenthesis around the whole thing and cube all of it.

```
> ((1+(2 - 1/3)^2)/(5*(1-3*6)))^3
```

```
[1] -8.77915e-05
```

**Warning!** If you forget the outermost parenthesis, you’ll only cube the denominator, and you’ll get something else.  $(1 + (2 - 1/3)^2)/(5 * (1 - 3 * 6))^3$   
[1] -6.15148e-06

NOW TRY CALCULATING ON YOUR OWN:

$$\frac{3+1}{5^2} - 10 \left( \frac{3}{20} + 1 \right)^3$$

You should get -15.04875.

## Working Memory

So far we haven’t actually done anything with R that’s saved any of our work. After we entered the commands in the [Opening and exporting data in R](#) section, R executed them immediately, printed out the results and forgot about them.

What if we want to calculate something and then use it again later? For example, we might need to calculate the mean of a data set and then use it repeatedly in following calculations. We’ll talk about how to do this in just a little bit when we get to functions. For the mean time, let’s say we need to calculate, and then refer to;

$$\sqrt{4^2 + 3.1^2}$$

If we type

```
> sqrt(4^2 + 3.1^2)
```

```
[1] 5.060632
```

`sqrt()`  
is first function we’re meeting. It takes the square-root  $\sqrt{\quad}$

then we get the answer, but nothing gets stored in working memory, so we can’t use it later. In order to use this later, we need to give this quantity a name and save it as an *R object*. Let’s call this ‘a’:

```
> a = sqrt(4^2 + 3.1^2)
>
```

This time we don't get any other output from R. It's basically saying, "Ok, got it, what's next?" If we want to see what's stored as 'a', all we have to do is type the object name on the command line:

```
> a
[1] 5.060632
```

Now, we can refer to this quantity whenever we need it:

$$4^2 + 3.1^2$$

```
> a^2
[1] 25.61
```

$$\sqrt{4^2 + 3.1^2} + (9)(1.2)$$

```
> a + 9*1.2
[1] 15.86063
```

However, if I come back and store something else as 'a', then my previous 'a' is *gone*. Let's say I now enter:

$$a = \sqrt{5^2 + 9.4^2}$$

```
> a = sqrt(5^2 + 9.4^2)
> a
[1] 10.64707
```

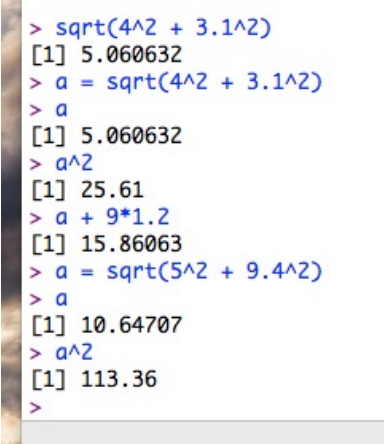
This doesn't change any of the calculations that R has previously done. But every calculation going forward will be with the new 'a'.

```
> a^2
[1] 113.36
```

**KEY POINT:** The stuff that R has printed out in the console is different from the stuff stored in working memory. What's displayed is a history of everything you've done so far. We need to use a command to get R to tell us everything that's stored in working memory.

The `ls()` command gives us a list of everything that's stored in working memory. Since it's a function, you have to type it with parenthesis, but we'll talk more about that in the [Functions and "Help" Files](#) section.

```
> ls()
[1] "a"
```



```
> sqrt(4^2 + 3.1^2)
[1] 5.060632
> a = sqrt(4^2 + 3.1^2)
> a
[1] 5.060632
> a^2
[1] 25.61
> a + 9*1.2
[1] 15.86063
> a = sqrt(5^2 + 9.4^2)
> a
[1] 10.64707
> a^2
[1] 113.36
>
```

Figure 3: What appears on the console is a record of what you've done so far. It doesn't tell you what's in the working memory.

Right now, the only object we've created is 'a', so it's the only thing in the list.

`ls()`  
Lists everything stored in working memory.

If we need to delete an object from working memory, we use the `rm()` command. Suppose that we want to delete 'a', then we put 'a' inside the parenthesis, `rm(a)`. Then, if we use `ls()` again, we can see that the working memory is empty.

```
> ls()
[1] "a"
> rm(a)
> ls()
character(0)
>
```

`rm()`  
Used to remove objects from working memory.

**NAMING R OBJECTS** You can name an R object pretty much anything you want to as long as it starts with a letter. `a0` is a good name, but `0a` won't work. Note that R is case-sensitive, so `x` is a different object than `X`.

```
> a0 = 5
> a0
[1] 5
> 0a
Error: unexpected symbol in "0a"
```

You want to get in the habit of calling your R objects something that makes sense, and you can remember what's what. `x` is pretty common for generic predictor variables, and `y` is common for generic response variables. But if you have a variable that's measuring the diameter of trees, then call it `diam` or `d.trees` or something where you can remember what it is.

If you're working with data and you imported it and called it, for example `x`, and you create a new variable and call it `x` too, then your data is gone from working memory. To prevent this, if you want to save something new under a variable name, just type that name on the command line. If something is already saved under that name, then R will print it out.

```
> a0
[1] 5
```

If there's nothing saved there, then R will print out an error, and you're free to save whatever you want under that variable name.

```
> a1
Error: object 'a1' not found
```

```
> a1 = (3+1.4)^2
> a1
[1] 19.36
```

### *Types of R objects*

Everything in R is saved as an 'object' in working memory. But there are lots of different types of objects, and they all behave a little differently. If you want to find out what kind of object something is, you can use the function `class()`.

```
> class(a)
Error: object 'a' not found
>
```

`class()`  
Tells you what kind of R object you've got.

Oh, right. We deleted 'a'. We'll just re-create it. [Hint: Use the ↑.]

```
> a = sqrt(5^2 + 9.4^2)
> class(a)
[1] "numeric"
>
```

### *numeric*

The most basic type of object is just a number. By default R uses double-precision calculations. That just means that calculations will be accurate to as many significant digits as you'll pretty much ever need.

So our 'a' above is just a single number, but sometimes we want to do the same operation to a whole list of numbers. Think of a linear function

$$y = 3 + 2x$$

Let's say that we want to compute  $y$  for every  $x$  from 0 to 10. We could do that this way;

```
> 3+2*0
[1] 3
> 3+2*1
[1] 5
> 3+2*2
[1] 7
```

and etcetera etcetera etcetera. But even with only 11 numbers to calculate that would get painfully tedious. So instead we make a numeric *vector* of  $x$ 's, and do the 11 calculations all at the same time.

The most basic way to make a vector is to use the 'concatenate' function `c()`.

`c()`  
The concatenate function puts things together into a vector.

```
> x = c(0,1,2,3,4,5,6,7,8,9,10)
> x
[1] 0 1 2 3 4 5 6 7 8 9 10
> class(x)
[1] "numeric"
```

Like 'a', our vector 'x' is numeric, but 'x' is an ordered list of 10 numbers. And that is effectively what a vector is, it's an ordered list of numbers. Of course, if you ask a physicist what a vector is, you'll get a much longer more complicated definition. But for our purposes, it's just a list of numbers.

Now, if I want to calculate

$$y = 3 + 2x$$

all I have to do is this

```
> 3+2x
Error: unexpected symbol in "3+2x"
```

Darn it, I forgot that \* again. Notice that the error message is different than the one we got before, but it's really the same mistake.

```
> 3+2*x
[1] 3 5 7 9 11 13 15 17 19 21 23
```

Now suppose that you want to calculate y for x all the way from 0 to 100 instead of for just 0 to 10. I don't want to have to type out 100 numbers! Fortunately, there are 2 shortcuts. The first shortcut is 0:100

a:b  
Makes a vector of numbers from a to b.

```
> 0:100
[1] 0 1 2 3 4 5 6 7 8 9 10 11 12
[14] 13 14 15 16 17 18 19 20 21 22 23 24 25
[27] 26 27 28 29 30 31 32 33 34 35 36 37 38
[40] 39 40 41 42 43 44 45 46 47 48 49 50 51
[53] 52 53 54 55 56 57 58 59 60 61 62 63 64
[66] 65 66 67 68 69 70 71 72 73 74 75 76 77
[79] 78 79 80 81 82 83 84 85 86 87 88 89 90
[92] 91 92 93 94 95 96 97 98 99 100
```

Here we want to notice 2 things: First those numbers in the brackets are the index for my vector. The first object in my list [1] is a zero. The fourteenth object, [14] is a 13, and so on.

The second thing we need to notice is that I haven't saved my list of numbers as anything, so they're just printed out onto the screen. If I want to save them, I have to store them under some name:

```
> x1 = 0:100
>
```



This way, if I want to calculate  $y = 3 + 2x$  for all 101 numbers, I only need to type 2 little commands, and I'm done.

```
> x1 = 0:100
> y = 3+2*x1
>
```

We'll talk more about making graphics later, but just to make sure it worked, and we calculated x1 and y correctly, we can plot them.

```
> plot(x1, y)
```

The little colon sequence has another trick or two up it's sleeve. Try these commands and see what you get.

```
> -5:5
> 1:10
> 10:1
```

There's another shortcut for creating vectors, seq(). Let's say I need a list of all the even numbers between 50 and 100, then I'll enter

```
> seq(50, 100, by=2)
```

Or If I need 400 numbers from 100 to 200

```
> seq(100,200, length=400)
```

Try the next two commands but before you enter them, try to guess what you'll get:

```
> seq{0,10, by=0.5}
> seq(21, 7, length=31)
```

Were your guesses right?

One last useful trick is that you can combine these commands in any combination, for example

```
> c(0:5, 10:16, seq(50, 65, by=3), 32:38)
[1]  0  1  2  3  4  5 10 11 12 13 14 15 16 50 53 56 59 62
[19] 65 32 33 34 35 36 37 38
```

### *character*

A character object is just that, characters. For example,

```
> type = 'red'
> type
[1] "red"
> class(type)
[1] "character"
```

plot()

Basic plotting function for R. We will talk about it in detail in section [Plotting Data](#)

seq(a,b)

Makes a sequence of numbers from a to b. It has some useful options/tweaks.

The first thing to notice when creating these objects is that they type of parenthesis doesn't really matter. And the second thing, is that you can make a vector of characters too.

```
> type = c('red', 'black', 'white')
> type
[1] "red" "black" "white"
> class(type)
[1] "character"
```

This kind of object might seem less imminently useful than the numeric objects that we use for calculations, but they often work together. For example, they are one way that we can control the color in plots:

```
> x = seq(1,100, length=26)
> y = 3 -0.5*x
> type = rep(c('red', 'black'), 13)
> type
[1] "red" "black" "red" "black" "red" "black"
[7] "red" "black" "red" "black" "red" "black"
[13] "red" "black" "red" "black" "red" "black"
[19] "red" "black" "red" "black" "red" "black"
[25] "red" "black"
> plot(x,y)
> plot(x,y, col=type)
```

`rep(x,n)`  
Makes a vector where 'x' is repeated 'n' times.

Hmm... I'm not terribly happy with that plot, the points aren't visible enough, and it's a little too hard to see the colors.

```
> plot(x,y, col=type, pch=19)
```

There are lots and lots of options for tweaking plots in R. `col` changes the color of the points, and the `pch` option will change the shape of points plotted. More in section [Plotting Data](#).

Let's see what kind of object we get if we string together a character and a number:

```
> tmp = c('one', 1)
> tmp
[1] "one" "1"
> class(tmp)
[1] "character"
```

A numeric vector can't store characters, but a character vector will treat a number as a character.

One other thing that we need to think about, is that the order in which we compose functions matters a great deal. The two commands below will produce very different results. Before you enter them, try to predict what you will get.

```
> rep(c('red','black','white'), 5)
> c(rep('red',5), rep('black', 5), rep('white', 5))
```

Were your predictions correct?

### *factor*

Factors are another way to store data from categorical variables. Some R functions need categorical variables as inputs. So the first thing we'll do is turn a character into a factor.

```
> tmp = rep(c('red','black','white'), 5)
> class(tmp)
[1] "character"
> type = factor(tmp)
> class(type)
[1] "factor"
> type
 [1] red   black white red   black white red   black white
[10] red   black white red   black white
Levels: black red white
> levels(type)
[1] "black" "red"   "white"
```

In earlier versions of R factors took up less memory than character objects, but it doesn't make as much difference now. As you get more practice, this is something to remember, if you can take up less working memory or you can use fewer operations, then you'll get results faster.

`factor()`  
creates a factor.

`levels()`  
Gives you the levels of a factor.

There's another way we can create exactly the same object

```
> rep(1:3, 5)
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
> type.2 = factor(rep(1:3, 5), labels=c("red", "black", "white"))
> type.2
 [1] red   black white red   black white red   black white
[10] red   black white red   black white
Levels: red black white
```

Notice that the default plot for a factor object is a bar chart, which should be your default choice for displaying a categorical variable.

```
> plot(type.2)
> plot(type.2, col = c('red','black','white'))
```

### *data frame*

Now most of the time when we do a scientific study, we have multiple variables on each unit of observation. If we're doing a study in the behavioral sciences, then we'll have any number of variables for each participant in our study including predictor variables such as race, gender, and experimental condition as well as one or more

response variables. We'd like to store all of these variables together in a single R object. That's what a data frame does.

Rather than create one, let's use one of the built-in R data sets. There are many many data sets available through R, some as built-in data sets, others are included in R packages. Type `iris` on the command line, what happens? Let's explain,

```
> class(iris)
[1] "data.frame"
> dim(iris)
[1] 150  5
```

This is telling us that `iris` is a data frame with 150 rows and 5 columns. When you just typed `iris` on the command line, you asked R to display the whole object. So R complied and printed out all  $5(150) = 750$  numbers and filled up the console window a couple of times over. That's rarely useful, so we want to get a few other tools for seeing what's in a data frame.

```
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"
```

This is where it becomes obvious that we really are talking about real flowers. Four variables are well named measurements of physical attributes of the flowers, and the fifth variable is the species of iris. This is useful, but it's also useful to look at the first couple rows of data, which is where the next command comes in.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4          0.2  setosa
2          4.9         3.0          1.4          0.2  setosa
3          4.7         3.2          1.3          0.2  setosa
4          4.6         3.1          1.5          0.2  setosa
5          5.0         3.6          1.4          0.2  setosa
6          5.4         3.9          1.7          0.4  setosa
```

It's also useful to be able to get the basic summary statistics on each variable in the data frame. The `summary` function is a generic function that can be applied to a lot of different types of R objects. When it is applied to a data frame, it gives us the basic statistics on each variable, depending on whether the variable is quantitative or categorical. Note, that it's a very very basic summary, it doesn't even give us standard deviations.

```
> summary(iris)
```

`dim()`  
Gives you the dimensions of a data frame, matrix or array in R:  
(rows, columns)

`names()`  
lists the names of the variables in a data frame.



`head()`  
prints out the first 6 rows of a data frame.

`summary()`  
general use function. When applied to a data frame, displays basic summary statistics for each variable.

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100	setosa :50
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300	versicolor:50
Median :5.800	Median :3.000	Median :4.350	Median :1.300	virginica :50
Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199	
3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800	
Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500	

Now, let's say that I want to examine the distribution of `Sepal.Length`; how do I get the variable I'm interested in all by itself? If I just type in `Sepal.Length`, I get an error.

```
> Sepal.Length
Error: object 'Sepal.Length' not found
```

The variable is contained within the data frame, so we can call it as part of the data

```
> iris$Sepal.Length
[1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8
[14] 4.3 5.8 5.7 5.4 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0
.....
[144] 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

`data.frame$variable`  
calls a variable within a data frame object.

If we "attach" a data frame to the workspace, then we can call the variables directly.

```
> attach(iris)
> Sepal.Length
[1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8
.....
```

`attach()`  
attaches a data frame, so that you can call variables directly.

If you're only working with one data set, then attaching a data frame makes a lot of sense. However, if you're working with multiple data sets, then you probably don't want to attach all of the data sets at the same time. For example, if you're doing homework, and you've moved on to problem 2, you don't want to leave everything from problem 1 hanging around.

```
> detach(iris)
> Sepal.Length
Error: object 'Sepal.Length' not found
```

`detach()`  
detaches a data frame, removes the variables from the workspace.

To really use data frames well, we need one more tool in our toolbox, and that's *indexing*. There's another way that we can get the variable `Sepal.Length` by itself.

```
> iris[,1]
```

[,]  
square brackets are used for indexing.

Sepal.Length is the first variable in the iris data frame, it's the first column, which is identified by the index "[,1]". If I wanted to pull off the first row, I'd type

```
> iris[1,]
```

If I want to pull out the sepal width of the 10th flower, that's the second column of the 10th row, so I type

```
> iris[10,2]
```

Or, I could get this same number a couple of different ways, for example

```
> iris$Sepal.Width[10]
> iris[10,]$Sepal.Width
```

Indexing is an incredibly flexible tool; we can pull off a subset of the data with just the variables we're interested in. Let's suppose that we wanted to ignore the petal measurements and focus on the sepal measurements and the species. Then we want variables 1, 2, and 5, though we may want to double check the order of the variable names just in case.

```
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length"
[4] "Petal.Width"  "Species"
> sepal = iris[,c(1,2,5)]
> head(sepal)
  Sepal.Length Sepal.Width Species
1          5.1          3.5  setosa
2          4.9          3.0  setosa
3          4.7          3.2  setosa
4          4.6          3.1  setosa
5          5.0          3.6  setosa
6          5.4          3.9  setosa
```

Remember what that `c()` command does? If I want to get several rows or several columns at the same time, we need to call all of them together, so we need to use the concatenate function. How would you pull out the data for the 4th, 20th and 100th flowers? Try it. You should get:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
4	4.6	3.1	1.5	0.2	setosa
20	5.1	3.8	1.5	0.3	setosa
100	5.7	2.8	4.1	1.3	versicolor

Let's say I want just the subset of data for flowers with sepal width less than 3. Let's start by creating an index vector

```
> small = (iris$Sepal.Width <3)
```

Get R to print out `small`, it should be a vector of TRUE and FALSE. It's TRUE if the sepal width is less than 3, and FALSE if it's not. Now we can create our data subset.

```
> iris.sm.sw = iris[small,]
```

```
> dim(iris.sm.sw)
```

```
[1] 57 5
```

```
> summary(iris.sm.sw)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
Min. :4.400	Min. :2.00	Min. :1.300	Min. :0.200	setosa : 2
1st Qu.:5.600	1st Qu.:2.50	1st Qu.:4.000	1st Qu.:1.200	versicolor:34
Median :6.000	Median :2.70	Median :4.500	Median :1.400	virginica :21
Mean :5.953	Mean :2.64	Mean :4.509	Mean :1.449	
3rd Qu.:6.300	3rd Qu.:2.80	3rd Qu.:5.100	3rd Qu.:1.800	
Max. :7.700	Max. :2.90	Max. :6.900	Max. :2.400	

So the new `iris.sm.sw` only includes the 57 rows where the sepal width was less than 3. We can also see that the summary statistics have changed for the other variables as well, which we would expect. Now, we need to note that our new dataset has the same variable names as the original iris data, if we attach both of them, we'll get the one we attached second, and it can be really, really hard to remember which one that was. A good rule of thumb would be to not attach multiple data sets with the same variable names. We should probably double check and make sure we've got everything detached. The `detach` command will detach anything and everything that we've attached.

```
> detach()
```

One other useful trick is that I don't have to actually create the vector `small`, I can just combine it into the command to create `iris.sm.sw`:

```
> iris.sm.sw.2 = iris[iris$Sepal.Width <3,]
```

Use the commands we've learned so far to convince yourself that `iris.sm.sw` and `iris.sm.sw.2` are the same. Check the dimensions, the summary statistics, the first few rows....

The *versicolor* species seems to have the skinniest sepals, why don't we take a closer look at those flowers? Alright, so we want to pull out the rows where the Species is *versicolor*

```
> versi = (iris$Species=versicolor)
```

```
Error: object 'versicolor' not found
```

Alright, let's debug this. R is looking for an object named 'versicolor', which doesn't exist. Ok, 'versicolor' is a level of a factor object, and as such it's really a string or character object. Since we want R to match one character object to another, we need to put it in quotes.

```
> versi = (iris$Species='versicolor')
> versi
[1] "versicolor"
```

Hmm... we didn't get an error, but we didn't get what we wanted either. We need a TRUE/FALSE vector that's true for the versicolor flowers. The problem is the = signs. First, we set iris\$Species to be 'versicolor', then we set versi to be the same thing. Let's check the iris data frame and make sure we didn't screw it up too bad.

```
> summary(iris)
...
   Species
Length:150
Class :character
Mode  :character
```

Yup, we overwrote the Species variable. So first we need to fix it, then we need to actually get what we want. Since we overwrote the data in R's working memory, the easiest way to fix this is just reload the original data back into the working memory.

```
> data(iris)
> summary(iris)
...
   Species
setosa    :50
versicolor:50
virginica :50
```

Ok, we've got the data back, but we still need to figure out how to get the data for just the versicolor flowers. If we want to select just the elements of a vector that are equal to a particular value, we need to use ==:

```
> versi = (iris$Species=='versicolor')
```

There's a bit of an urban legend that in the earliest days of computers, 'debugging' really did mean physically going and looking for bugs in the computer.

<http://en.wikipedia.org/wiki/Debugging>

This is a key lesson for data analysis in any software:

Never. Never. Never do anything that might overwrite your original data file. Keep your analysis separate from your data.

*[Organizing and Saving Work in R](#)*



```

> versi
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[15] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[29] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[43] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
[57] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[71] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[85] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[99] TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[113] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[127] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[141] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> versicol = iris[versi, ]
> summary(versicol)
  Sepal.Length   Sepal.Width   Petal.Length   Petal.Width   Species
Min.   :4.900   Min.    :2.000   Min.    :3.00   Min.    :1.000   setosa    : 0
1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.00   1st Qu.:1.200   versicolor:50
Median :5.900   Median :2.800   Median :4.35   Median :1.300   virginica : 0
Mean   :5.936   Mean    :2.770   Mean    :4.26   Mean    :1.326
3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.60   3rd Qu.:1.500
Max.   :7.000   Max.    :3.400   Max.    :5.10   Max.    :1.800

```

Yay! We got it. And by looking at the summary of the Species column we know we got it right. Data frames are very common. You really want to get comfortable with them, and comfortable with pulling out the pieces that you need.

You also want to get comfortable with "debugging". This is the biggest part of programming. Try something, check it, adjust, check it again, adjust again, and so on and so forth until it works. Of course, check & adjust is also a life-skill ;).

### *Functions and "Help" Files*

There are a few other types of R-objects that we haven't talked about (e.g., matrices, arrays, lists), but the most important one remaining is functions. By now, you've met quite a few functions, but it's worth it to spend some time talking about functions in general.

By now you've probably noticed that we used parenthesis ( ) for functions, and brackets [ ] for indexes. We'll talk about how to use braces { }, later.

( ) for functions  
[ ] for indices  
{ } for control statements

You have to use the parentheses to call the function. Note the difference between the results

```

> ls
> ls()

```

The second calls the function so you get the list of everything saved in the workspace. The first prints out the function. That is mostly annoying, but it can be useful if you need to figure out exactly what the function's doing, or if you need to look at an example of how to write a function.

We just finished working with data frames, so let's start by taking a closer look at `data.frame()`. First we'll look at the "help" file.

```
> ?data.frame
```

Why is "help" in quotes? Because the "help file" is just the documentation page. It contains the technical details, but the documentation requires a little practice to understand.

```
help(command.name)
OR ?command.name
Opens the documentation for the
command.
```

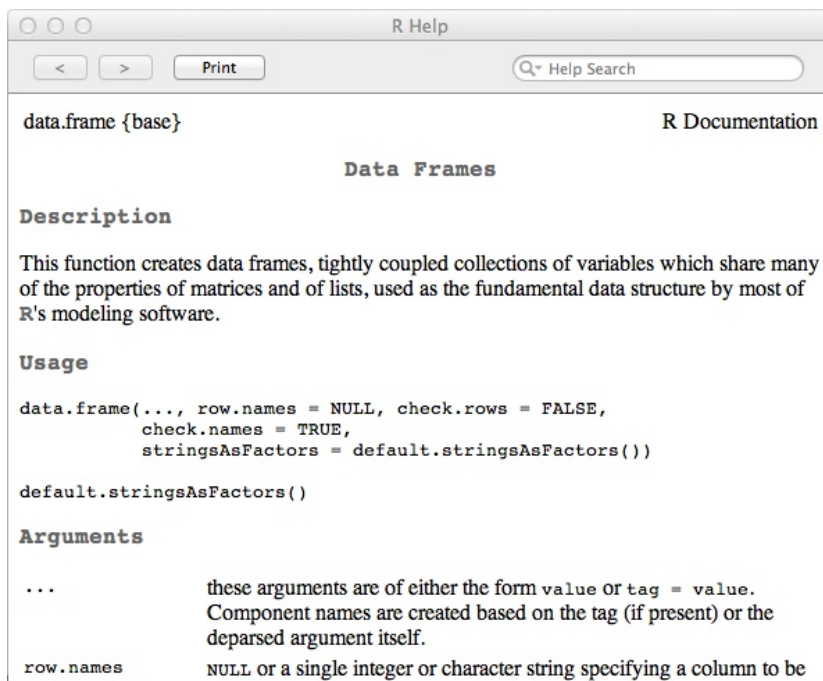


Figure 4: The R documentation for `data.frame()`.

You should notice that the page has several headings. The Description gives a basic description of what the function does. The one for `data.frame` is pretty typical; the first phrase tells you what you need to know and a lot of the rest is full of jargon. Then there's the Usage section, where you get an outline of what to type in to use the function, the inputs. The Arguments section usually gives some more detail on the inputs the function takes. What's in the Details section varies from function to function. In some cases, it contains mathematical details on how the function does its calculations. In the `data.frame` case, it contains a little more information about how the function works depending on whether or not some of the optional inputs are included, and how data frames work in general. The

Value section tells you what the outputs of the function are. It's often a list that looks like the arguments list. In the `data.frame` documentation, the output is a data frame, so it describes that data frame a little bit more. Then there's the Note and References sections. The See Also section usually has links to related functions. If you can't find what you're looking for on the first documentation page you look at, sometimes you can find it on one of those related pages. Then at the very very bottom, we find the Examples. When you're trying to get a function to work, look at the Usage, then the Arguments, then the Examples.

For `data.frame()`, if we look at the Usage, it's not clear what that first argument `"..."` is supposed to be. So we look at the Arguments section, "these arguments are of either the form `value` or `tag=value`." That probably doesn't make any sense right now. So now we take a look at the Examples to see if we can figure it out. We'll start with the first segment:

```
L3 <- LETTERS[1:3]
fac <- sample(L3, 10, replace = TRUE)
(d <- data.frame(x = 1, y = 1:10, fac = fac))
## The "same" with automatic column names:
data.frame(1, 1:10, sample(L3, 10, replace = TRUE))

is.data.frame(d)
```

Now, take it line by line. What does the first line do? Type it in.

```
> L3
[1] "A" "B" "C"
> class(L3)
[1] "character"
```

Ok, it creates a character vector with letters 1:3 of the alphabet. Let's double check what that `LETTERS` object is, just to be sure.

```
> LETTERS
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Yes, `LETTERS` is in fact a vector of the alphabet, so `LETTERS[1:3]` takes the first 3 letters.

Alright, let's move on to the second line. Type it in. What do you get?

```
> fac
[1] "A" "A" "C" "C" "C" "B" "A" "A" "A" "A"
```

# is the R "comment character"  
Good code has comments in it that explain what it does. The # lets us include these comments. Everything on the same line that comes after that character is ignored by R.  
See [Commenting your work](#).

<-  
`x<-5` is the same as `x=5`.  
In older versions of R, `<-` worked better, so you'll see it more in older examples.

You almost certainly got something different than I did. The `sample()` command takes a random sample, so your random sample is most likely different than my random sample. In this case `fac` is a random sample of size 10 from the `L3` vector.

Now let's try line 3.

```
> d <- data.frame(x = 1, y = 1:10, fac = fac)
> d
  x  y fac
1  1  1  A
2  1  2  A
3  1  3  C
4  1  4  C
5  1  5  C
6  1  6  B
7  1  7  A
8  1  8  A
9  1  9  A
10 1 10  A
> class(d)
[1] "data.frame"
> is.data.frame(d)
[1] TRUE
```

This created my data frame with 3 variables, `x`, `y` and `fac`. So how is line 5 different?

```
> data.frame(1, 1:10, sample(L3, 10, replace = TRUE))
  X1 X1.10 sample.L3..10..replace...TRUE.
1   1     1                             B
2   1     2                             C
3   1     3                             A
```

It put the same stuff in the data frame, but since we didn't give it variable names, it made some up. The first one's ok. The second one's not terrible, the third one is horrific. Alright, scroll back up to the Arguments section in the help file. Now that `"..."` makes sense. That's where we put the variables that we want to include in the data frame. If we just put the value in, then it makes up a name for the variable based on whatever we put in. On the other hand, if we put in `tag=value`, then `"tag"` is what the data frame will name our variable.

However, in R, like most programming languages, there's almost never just one way to do something. The following 2 lines will create an object just like `d`.

```
> b = data.frame(1, 1:10, sample(L3, 10, replace = TRUE))
```

`sample(x,n,...)`

Takes a sample of size `n` from the vector `x`, also has some optional arguments you'll want to pay attention to.

commas!

The commas in between function arguments are essential. If you forget one, you'll get an error.

Error: unexpected symbol in "..."

```
> names(b) = c('x', 'y', 'fac')
> b
  x y fac
1 1 1  A
2 1 2  A
3 1 3  A
4 1 4  A
...
```

`names()`  
This is the same function that we used to get the names of a data frame. We can also use it to set the names of an object.

Now, what about the rest of those arguments? They're *optional* arguments. They give the function more flexibility. For example if our data is on the 50 states, then we might want to use the state names as the row names. The `row.names` option gives us the flexibility to do this if we want to, but it's optional so we don't have to.

## Plotting Data

As we move on to plotting, we'll get more practice using functions. We'll go through two basic types of plots, (*histogram*, and *scatterplot*) learn how to alter plots in *Adding Things to Plots*, and practice reading the help files as we go.

Before we get started, let's clean up the workspace. First, detach anything that might be hanging around.

```
> detach()
```

Then we'll clear the workspace using the dropdown menu.

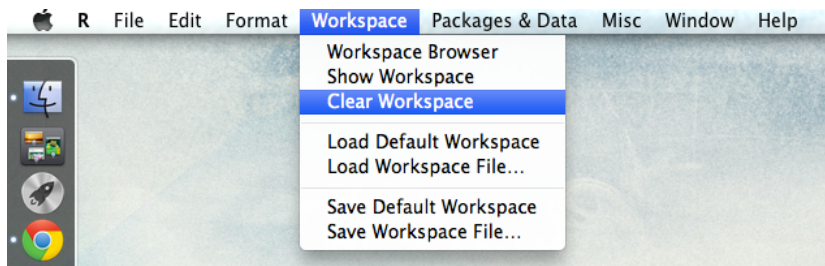


Figure 5: This will delete everything from R's working memory. So you will need to say "yes I really mean it" in a confirmation window.

You can check that it worked by using `ls()`

```
> ls()
character(0)
```

## histogram

Let's pull that iris data back up. We'll just start by making a basic plot of `Sepal.Length`, then tweaking it.

```
> attach(iris)
> names(iris)
> hist(Sepal.Length)
```

Now, let's open the help file and see if we can use the options to make it better, `?hist`. Scroll down to the Arguments list.

The first argument is of course the data, `x`. Note that `hist` expects the data to be a numeric vector. So it's worth remembering that if we give it a matrix or a data frame, we may not get what we're after.

The second argument is `breaks`. This controls "the breakpoints between histogram cells", otherwise known as bin-width. This is usually the most important adjustment we can make to a histogram. And you can see that R has a number of different ways we can specify the breaks. Let's start with "a single number giving the number of cells for the histogram". Of course, we also get the warning, "the number is a suggestion only; the breakpoints will be set to pretty values". But let's try a couple of different things, and see what we get.

```
> hist(Sepal.Length, breaks=10)
> hist(Sepal.Length, breaks=15)
> hist(Sepal.Length, breaks=20)
> hist(Sepal.Length, breaks=5)
```

Sometimes I'm not really a fan of these "pretty" breaks, it doesn't give you fine enough control. Let's say we really, really want 12 or 15 bins, rather than 8 or 20. If we specify the breaks as "a vector giving the breakpoints between histogram cells" then it will actually do what we want. First let's check the range of the variable.

```
> range(Sepal.Length)
[1] 4.3 7.9
```

So if our breaks go from 4 to 8, we should be fine. If we want 12 breaks, then we want a vector from 4 to 8 of length 12+1. Do you remember how to do this? ([numeric](#))

```
> br = seq(4, 8, length=13)
> hist(Sepal.Length, breaks=br)
```

Create a histogram with 15 bins. It should look like Figure 7.

Back to the list of arguments: Try changing `freq` to `FALSE`, see what happens. We also see that the option `probability` does the same thing as `freq`. These options make the histogram harder to interpret. What it's labeling as "density", isn't the usual definition of density for a continuous variable. These aren't options we want to use under normal circumstances. We really want to stick with counts/frequency unless we have a spectacular reason for doing otherwise.

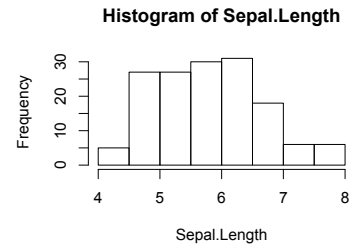


Figure 6: The initial plot should look something like this.

`range()`  
gives the minimum and maximum of the data.

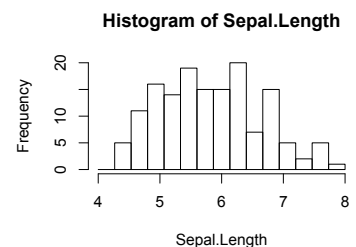


Figure 7: Histogram of sepal length with 16 bins.

The next two options give you some really fine-grained control over whether the bins are  $[a,b)$  or  $(a,b]$ . When your variable is highly discrete, these can be useful, but in the case of the continuous `Sepal.Length`, it really doesn't make any difference.

The next two arguments `density` and `angle` will make some shading lines on the bins. We can play with them a little bit, and see what they do.

```
> hist(Sepal.Length, breaks=br, density=1)
> hist(Sepal.Length, breaks=br, density=5)
> hist(Sepal.Length, breaks=br, density=10)
> hist(Sepal.Length, breaks=br, density=10, angle=45)
> hist(Sepal.Length, breaks=br, density=10, angle=90) # particularly terrible
> hist(Sepal.Length, breaks=br, density=10, angle=120)
```

I'm not really a fan of the density arguments, I think `col` and `border` produce prettier results.

```
> hist(Sepal.Length, breaks=br, col='wheat')
> hist(Sepal.Length, breaks=br, col='tomato')
> hist(Sepal.Length, breaks=br, col='steelblue') #one of my favorite R colors
> hist(Sepal.Length, breaks=br, col='steelblue', border='wheat')
> hist(Sepal.Length, breaks=br, col='steelblue', border='tomato')
> hist(Sepal.Length, breaks=br, col='steelblue', border='slateblue')
> hist(Sepal.Length, breaks=br, col='wheat', border='steelblue')
```

By now, we've got something that looks pretty good. We just need to fix those labels. We don't want to use the label `Sepal.Length`. So we need to use the `main` and `xlab` options:

```
> hist(Sepal.Length, breaks=br, col='steelblue',
+ xlab='Sepal Length', main="Histogram of Sepal Length")
> hist(Sepal.Length, breaks=br, col='steelblue',
+ xlab='Sepal Length', main="")
```

Ok. That gives us something we can use. Figure 8.

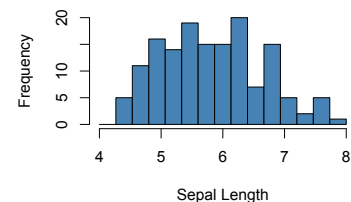


Figure 8: `hist(Sepal.Length, breaks=br, col='steelblue', xlab='Sepal Length', main='')`

### *scatterplot*

Now we're ready to work on plotting 2 variables. For this, we use the generic `plot()` function. So let's start by looking at the help file, `?plot`.

The description reads "Generic function for plotting R objects". What does that mean? It means that this same function command will plot all kinds of different R objects. If we fit a regression model, and name it `reg.model`, then `plot(reg.model)` will produce some summary and diagnostic plots for the model.

Right now, we just want to focus on creating a simple scatterplot, so let's click on `plot.default`

The description of `plot.default` reads "Draw a scatterplot with decorations..." so we know this is what we're looking for. Now, we look at the usage, and see:

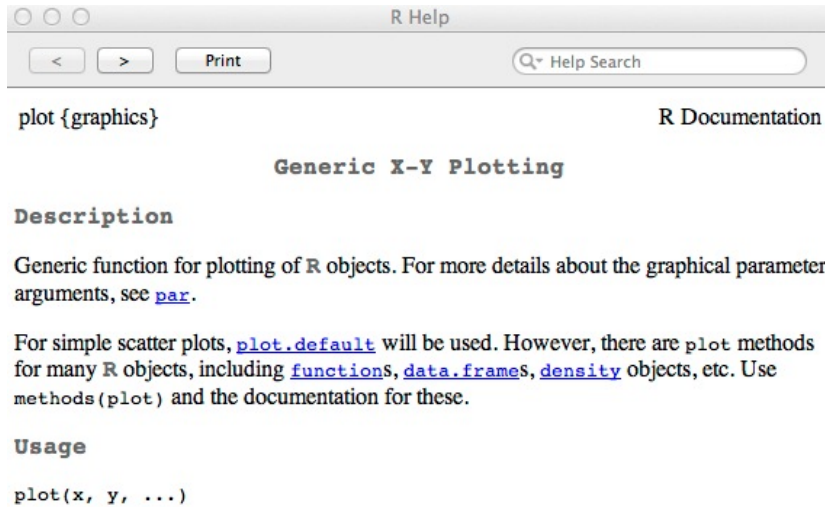


Figure 9: Help page for the plot function.

```
plot(x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Let's just take this one argument at a time. The first argument is `x`, so that's where we'll start.

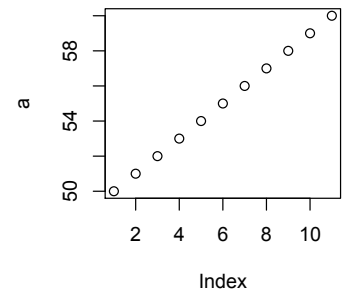
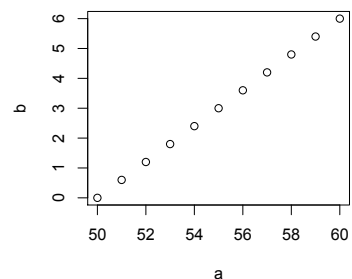
```
> a = 50:60
> a
[1] 50 51 52 53 54 55 56 57 58 59 60
> plot(x=a)
> plot(a)
```

Notice first that these two plot commands do exactly the same thing. R assumes that whatever you put in the function as the first argument is `x`. However, since `a` is a vector, R has plotted the index on the x-axis and the number on the y-axis, so the first point is (1,50) and the last is (11, 60).

So that's what plot does if you only give it one numeric variable, so what if we give it two? Let's create a `b` to go with our `a`.

```
> b = seq(0,6, length=length(a))
> b
[1] 0.0 0.6 1.2 1.8 2.4 3.0 3.6 4.2 4.8 5.4 6.0
> plot(x=a, y=b)
> plot(a,b)
```

Again, notice that the last two commands do exactly the same thing. Now, let's look at the next argument type. If we look at the Usage,

Figure 10: `plot(a)`Figure 11: `plot(a,b)` Note the difference in the x-axis from figure 10



we see `type="p"`, so "p" for points is the default for type. If we don't specify something to override this, then we get points, as in figures 10 and 11. So let's try the other types available,

```
> plot(a,b, type='l')
> plot(a,b, type='b')
> plot(a,b, type='c')
> plot(a,b, type='o')
> plot(a,b, type='s')
> plot(a,b, type='h')
```

Just remember we don't have to use all the arguments or use them in any specific order.

```
> plot(iris$Sepal.Length, iris$Petal.Length, xlab='Sepal Length',
ylab="Petal Length", xlim=c(0,8), ylim=c(0,8))
```

And more options for adjusting graphs are available with `?par`. Take a look at the documentation, and see if you can figure out what the `mfrow` argument does.

```
> par(mfrow=c(1,2))
> plot(iris$Sepal.Length, iris$Petal.Length, xlab='Sepal
Length', ylab="Petal Length", pch=19, col='blue')
> plot(iris$Sepal.Width, iris$Petal.Width, xlab='Sepal
Width', ylab="Petal Width", pch=19, col='red')
```

`par()`  
Graphical parameters can either be set with the `par` command, or added as arguments to plots.

### *Adding Things to Plots*

Close the last quartz device, so we can start with a clean slate. By now, you've certainly noticed that when you make one plot, the last plot is overwritten and gone. So this leaves us with two questions: 1) Can I get two plots at the same time? and 2) How do I add something to a plot?

We've already seen one way to get multiple plots at the same time with the `par(mfrow=...)` command. There's another useful way to do this, with `dev.new()`.

```
> hist(iris$Petal.Length, xlab='Petal Length', main=NULL,
col='tomato')
> dev.new()
> plot(iris$Petal.Length, iris$Petal.Width, xlab="Petal
Length", ylab="Petal Width", pch=18)
```

`dev.new()`  
Opens a new plotting device.

## Organizing and Saving Work in R

Data are sacred. You don't want anything or anyone to alter the original data in any way. Data are typically time consuming and/or expensive to collect<sup>1</sup>, particularly experimental data.

In R, we have the ability to treat data as "read only". The data are stored in their own original data file, perhaps a .csv, or a .rdata file. We read it into R's working memory, do the analysis, we save the commands to do the analysis, save graphs and other necessary output, and never have to alter the data file.

This has a couple of advantages. First, when you work with data interactively, as in SPSS or Excel, a simple easy to make mistake can modify the data, and this means that you are never sure where the data came from or how it has been modified. This has caused real and serious problems with published research; a discussion of a recent example can be found in the Colbert Report archives<sup>2</sup>.

The second advantage is that by preserving the original data and saving every single command that you used to do the analysis, you are making your work **replicable**. The generated output is completely disposable. At any point you can go through, delete the output, and rerun the analysis from the beginning to double check it. You can then share a small number of files with other researchers who are interested in replicating your findings, or post the files on your website with the paper. Increasing the ease of replicability increases the quality of the science.

<sup>1</sup> <http://nicercode.github.io/blog/2013-04-05-projects/>

<sup>2</sup> <http://www.colbertnation.com/the-colbert-report-videos/425749/april-23-2013/austerity-s-spreadsheet-error---thomas-herndon>

## Controlling the Working Directory

R can save a couple of different file types, but before we talk about them, we have to talk about what a working directory is and how to control it.

```
> getwd()
```

This will print out the full path for R's current working directory. It tells you where in the computer's file structure you currently are. Check out the visualization in Figure 12. If the current working directory is John's work folder, then we'll get

```
> getwd()
[1] "/users/john/work"
```

When we tell R to save work, it will automatically save it to the current working directory. In addition, when we talk about opening data later (*Opening and exporting data in R*), R will look for the data in the current working directory. So any time we're getting stuff into or out of R, we need to pay attention to the working directory.

```
getwd()
Gives you R's current working directory.
```

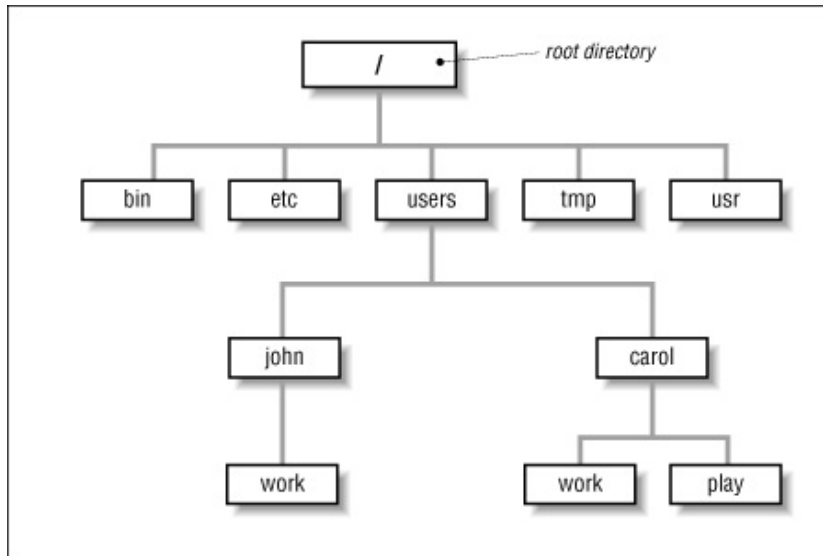


Figure 12: Visualization of a directory tree, or file structure in a computer.

Note that it has to be checked every time you open R. You don't get to just set the working directory once and be done with it. But this is a good thing. If you set the working directory for one project, finish it and start working on another project, you don't want to keep using the folder for the first project.

There are two ways to change the working directory. The first is to use the command `setwd`, for example,

```
> setwd('/users/carol/play')
```

`setwd()` Sets the working directory.

However, if you're not used to typing out path names, this can be a little annoying, since you have to be very careful to not make any typos at all. Also, path names should not contain any spaces. Good practice is to name folders with dashes or underscores instead of spaces. So "Learning\_R" is a much better folder name than "Learning R". If your current folder names have spaces, they won't play nice with a command line environment, like we're using in R.

There's also a drop-down menu option for managing the working directory. For Macs it's under "Misc" as shown in Figure 13.

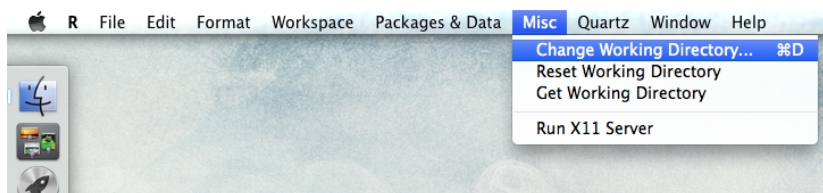


Figure 13: Menu option for changing the working directory.

*\*.history files*

We already know that you can scroll through your earlier commands on the command line by pressing  $\uparrow$ . You can also look at and save the entire history of commands that you've entered.

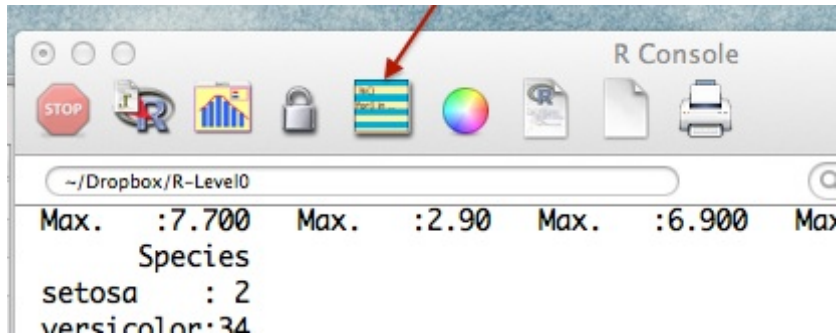


Figure 14: In the Mac version of R, this icon will let you view/hide the entire command history, and give you an option to save it.

For Mac users, take a look at Figure 14. For non-Mac users, try the command

```
> savehistory(file='file_name.history')
```

Note that R automatically saves the history as a hidden file in the working directory where you open R. But until you get more practice with command lines, you may not want to go looking for hidden files. This gives you a way to save a copy of the history where you can find it.

Now, the problem with saving only the history is that it saves absolutely every command you've entered, and only the commands. So all the errors, all the typos, everything gets saved, with no record of which commands actually worked, and no comments to say what these commands are supposed to do.

*\*.R Scripts*

Script files solve all the problems with using history files, and have a few other useful features. When you open R, two windows usually open, the console and a new blank script document. So far, we've used only the console. Now we're ready to write a script.

1. Either open a new R session, or clear the workspace.
2. Use `ls()` to make double sure the workspace is clear.
3. Create a folder "LearningR" and change the working directory to be that folder.

4. Use `getwd()` to make sure that the working directory is what you want it to be.
5. If you don't already have a new script editor or 'document' window open, then open one up.

We're going to create a short script that generates a sample of data from a standard normal distribution, plots a histogram of the sample, and shows the sample mean and population mean on the plot. We'll also add a legend.

Type this into the script editor (or copy and paste):

```
#### Draws and plots a sample from the standard normal distribution
n = 50 # sample size
x = rnorm(n, 0, 1) #draws the sample
x.bar = mean(x) #sample mean
br = seq(min(x), max(x), length=13) #sets breaks for the histogram
hist(x, col='wheat', breaks=br) # makes the histogram
abline(v=0, lty=3, col='red', lwd=2) # adds population line
abline(v=x.bar, lty=2, col='blue', lwd=2) # adds sample line
legend('topright', lty=c(3,2), col=c('red', 'blue'), lwd=2,
legend=c('pop mean', 'sample mean'), cex=0.8) #adds the legend
```

Note that we've got a line at the top that says what the script does, and a label on each line of the code. We talk more about why this is important in the [Commenting your work](#) section.

Once you've got it all typed in, save the file as `ExampleScript.R` in your `LearningR` folder. Now it's time to try to run the code and de-bug it if necessary (and it's usually necessary).

If the file is located in the folder that's the current working directory, then the command to run the file is:

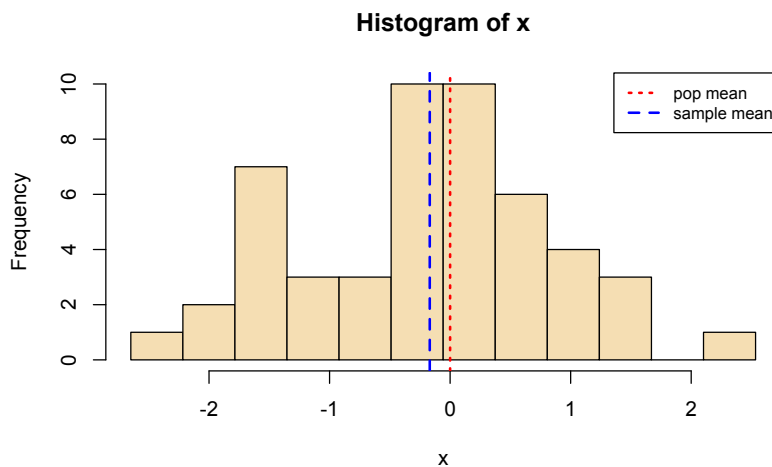
```
> source('ExampleScript.R')
```

`source()` Runs an R script.

If the location of the file and the current working directory don't match, then you'll get this error:

```
> source('ExampleScript.R')
Error in file(filename, "r", encoding = encoding) :
  cannot open the connection
In addition: Warning message:
In file(filename, "r", encoding = encoding) :
  cannot open file 'ExampleScript.R': No such file or directory
```

If everything works perfectly, you should get something similar to this plot



If you didn't get that plot, try to figure out why not. The most common culprits are missing or misplaced commas and parenthesis. "unexpected symbol" usually, but not always, means it's a comma problem. "unused argument" or "missing argument" usually, but not always, mean that there's a parenthesis problem. "object 'X' not found" probably means there's a typo; for example, you named your data x and tried to take the mean of X. R is case-sensitive.

SCRIPTS ARE THE LIFE-BLOOD OF R. If you are going to do anything that's more than a few lines long, you should create a script file. You don't have to write a script file like we did in the example, where you write a complete script file and then run it all at once. Rather, as you're getting started write the commands out and comment them in a script file, then you can cut and paste onto the command line. Using the command line and the script editor together will let you build up your code line by line and keep a good record of what the code is supposed to do and which commands work. The script and the original data file should be all that you ever need to share with other researchers to make your work replicable.

### *\*.Rdata files*

You can save all the objects and functions that you have created in a .RData file, by using the `save()` or the `save.image()` functions.

```
> save.image(file='file_name.Rdata')
> save(object1, object2, object3,..., file='file_name.Rdata')
```

Note that both of these functions will save the file into your current working directory. So if you can't find the file, use `getwd()` to see where to look for it.

`save.image()`  
Saves everything in the current workspace.

`save()`  
Saves the listed objects.

So right now, try

```
> ls()
[1] "br"      "n"       "x"       "x.bar"
> save.image(file='Example.Rdata')
```

Now close R. To re-open R and load the file, just double click on the `Example.Rdata` file. Use `ls()` to make sure that you've preserved all of the R objects.

```
> load("/path_name/LearningR/Example.Rdata")
> ls()
[1] "br"      "n"       "x"       "x.bar"
> getwd()
[1] "/path_name/LearningR"
>
```

Note that when you open R in this way, your working directory is set automatically as the folder that contains the file. This little trick also works when you double click on `*.R` script files.

### *File Management Habits*

Each project needs to be in its own folder. For small projects, like homework, you can put everything in one folder. Just name the folder "ERSH8150-HW1 and the data, graphics, and R files can all go in there together.

However, bigger projects grow and evolve over time. Laying out the file structure well in the beginning can save some serious headache later on. A project will often start life as a couple of notes, then you write some code to test out the ideas, then you might start drafting a manuscript to record your observations in, by the end you've got a couple of iterations of code, and a ton of figures, and a few different drafts of a paper. If it's not well organized, things can get lost and mixed up. There isn't one way to lay out a project, but a good basic structure might look something like this:

```
project/
|----- data/
|----- doc/
|----- graphics/
|----- output/
```

Put your data in the `/data` directory and treat it as *read only*. Depending on the project, you might have csv files, a database, or even additional subdirectories in here. If you have 2 or more data files, say from a pilot study and the main study. You should have a `/data` subdirectory.

Write-ups, whether formal papers, or informal notes, or something in between should go in the `/doc` subdirectory. LaTeX tends to be popular with statisticians, and has the benefit that it’s very easy to update the graphics automatically. If you’re using Word, you’ll need to remember to paste in the most recent versions of your figures.

Of course, the `/graphics` directory contains the graphics. A good idea is to make sure this directory contains only generated files. Meaning, that you can delete the contents and regenerate them from your code and data at any time.

Other output should go in the `/output` folder; for example, simulation output or processed data. It may be a while before you’re working on projects that are big enough to need this directory.

Initially, you can put your R code in the main project directory. As the project gets bigger, you may want to pull out functions into their own script that you can source. When you get to this point, it’s probably best to create another directory `/R_functions` to hold these functions.

The main thing to remember is data is precious, the generated output is disposable. You can always re-run the analysis, recreating the data is pretty much impossible.

### *Commenting your work*

We saw our first example of commented code in the [Functions and “Help” Files](#) section, now it’s time to think about this more deeply. If you’re working on a project, and you have to leave it for a little while, perhaps as little as a couple of days to a couple of weeks, when you sit down to work again, your work may look like gibberish:

```
load('forest.rdata')
attach(ufcgf)
plot(Dbh, Height)
lin.model = lm(Height~Dbh)
abline(lin.model)
summary(lin.model)
dev.new()
plot(Dbh, Height)
abline(lin.model)
sm.model = loess(Height~Dbh)
points(Dbh, sm.model$fitted, pch=19, col='red')
```

Example of badly commented code.  
The worst comments are no comments.

On the other hand, if you want to be able to easily pick up where you left off, then commenting the code makes all the difference in the world.

```
load('forest.rdata')
```

Example of decently commented code.



```

attach(ufcgf)
#####
## 1991 forest inventory measures from the Upper Flat Creek
## stand of the University of Idaho Experimental Forest
#####
## Plot - Plot number
## Tree - ID for tree within plot
## Species - stored as a factor
## Dbh - Diameter in mm
## Height - Height of tree, in decimeters
#####
## We want to examine the relationship between Height and Dbh
#####
plot(Dbh, Height) # makes the scatterplot
lin.model = lm(Height~Dbh) # fits the regression line
# Height = a + b*Dbh
abline(lin.model) # adds the fit line to the scatterplot
summary(lin.model) #summary of the regression model
#####
dev.new()
plot(Dbh, Height) # makes a new scatterplot
abline(lin.model) # adds the line from the linear regression
sm.model = loess(Height~Dbh) # fits a smooth function
# Height = g(Dbh)
points(Dbh, sm.model$fitted, pch=19, col='red')
# adds the smooth function to the plot

```

When it comes to commenting code, there are many different conventions. Some people label blocks of code, some people label each line, some people only label the unusual lines. Some will put comments before a set of lines, some will put a comment after or below each line.

As a place to start, you should add a couple of comments at the beginning explaining the goal of the analysis, and what's in the data. Then comment all of the lines that you can't just look at and know what they do. The lines you can't just "read" should be labeled.

Adequate comments also increase the replicability of your work. When you share your work, others can more easily read it and know what you did. If you ever plan to share your work, you should be writing comments as you write the code. And this is doubly true if you have collaborators that will need to be able to understand your analysis.

## Opening and exporting data in R

Data comes in many different varieties, and now it's time to learn how to get it into R so we can work with it.

### *\*.csv and \*.txt files*

This is one of the most common formats for small to medium data sets. The data are stored as plain text, separated by commas or tabs. These files are fairly popular because every statistical software can open this type of file easily. You can even open them with a text editor like Notepad or TextEdit. However, storing data as text is not very efficient in terms of computer memory, so it really is only used for relatively small data.

Download the dataset from here <http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv>, and save it in your LearningR folder. Its a small data set on advertising expenditures and sales.

First, open it up in a text editor to look at it. A csv will usually look something like this

```
"", "TV", "Radio", "Newspaper", "Sales"
"1", 230.1, 37.8, 69.2, 22.1
"2", 44.5, 39.3, 45.1, 10.4
"3", 17.2, 45.9, 69.3, 9.3
"4", 151.5, 41.3, 58.5, 18.5
"5", 180.8, 10.8, 58.4, 12.9
"6", 8.7, 48.9, 75.7, 7.2
"7", 57.5, 32.8, 23.5, 11.8
"8", 120.2, 19.6, 11.6, 13.2
"9", 8.6, 2.1, 1.4, 4.8
"10", 199.8, 2.6, 21.2, 10.6
```

You can see that the first row contains column labels, and the first column contains row labels. The columns are separated (delimited) by commas, and the rows are separated by line breaks. In order to read the data into R correctly, we need to pay attention to these details.

To open this data in R, you check your working directory and then use `read.table()`

```
> getwd()
[1] "/path_name/LearningR"
> ?read.table # to look at the help file
starting httpd help server ... done
> ads = read.table('Advertising.csv', header=TRUE, sep=',')
> head(ads)
  X    TV Radio Newspaper Sales
1 1 230.1  37.8      69.2  22.1
2 2  44.5  39.3      45.1  10.4
```

You can locate data on just about anything at <http://www.Data.gov>, but formatting is highly variable.

`read.table()`  
Imports data in text format into R.

```

3 3 17.2 45.9      69.3  9.3
4 4 151.5 41.3     58.5 18.5
5 5 180.8 10.8     58.4 12.9
6 6  8.7 48.9     75.0  7.2

```

The `header=TRUE` option tells `read.table` that there is a row at the top with column names, while `sep=','` indicates that the data is separated by commas. However, we notice that our new data frame `ads` isn't quite right. We have an extra column in the beginning. It looks like the row names were read in as an extra variable. To fix this, we add the option `row.names=1` to indicate that the first column contains row names.

```

> ads = read.table('Advertising.csv', header=TRUE, sep=',', row.names=1)
> head(ads)
      TV Radio Newspaper Sales
1 230.1  37.8      69.2  22.1
2  44.5  39.3      45.1  10.4
3  17.2  45.9      69.3   9.3
4 151.5  41.3      58.5  18.5
5 180.8  10.8      58.4  12.9
6   8.7  48.9      75.0   7.2
>

```

Note that if you don't store the data in working memory as an R object, in this case `ads`, then R will just print out the data frame onto the screen. For comparison, see what happens when you enter just:

```
> read.table('Advertising.csv', header=TRUE, sep=',', row.names=1)
```

This will read in any plain text file, but for csv files, we get an extra shortcut. The important thing is not which command you use, but rather that the options you give the function match how the data is recorded in the file.

`read.csv()`  
Imports data in csv format into R.

```

> ads2 = read.csv('Advertising.csv', row.names=1)
> head(ads2)

```

```

      TV Radio Newspaper Sales
1 230.1  37.8      69.2  22.1
2  44.5  39.3      45.1  10.4
3  17.2  45.9      69.3   9.3
4 151.5  41.3      58.5  18.5
5 180.8  10.8      58.4  12.9
6   8.7  48.9      75.0   7.2

```

TO IMPORT AN EXCEL FILE INTO R, first open the file in excel and export it to a csv. Then open the csv file in R.

TO EXPORT A CSV FROM R, use the `write.csv()` function. We'll take the `iris` dataset that we were working with earlier and write it to a csv in the `LearningR` folder. So first check your working directory.

`write.csv()`  
Writes a data frame to a csv file.

```
> getwd()
[1] "/path_name/LearningR"
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1          3.5          1.4          0.2  setosa
2          4.9          3.0          1.4          0.2  setosa
3          4.7          3.2          1.3          0.2  setosa
4          4.6          3.1          1.5          0.2  setosa
5          5.0          3.6          1.4          0.2  setosa
6          5.4          3.9          1.7          0.4  setosa
> write.csv(iris, file='Test_iris.csv')
>
```

Look in your working directory and open up `Test_iris.csv` in a plain text editor. Make sure everything worked.

### *\*.sav files*

Importing and exporting data from SPSS is a bit different. SPSS uses it's own proprietary format and so we have to work a little bit harder to get data in and out of R.

First we need to install and load the `foreign` package, see the [Packages](#) section.

```
> library(foreign)
> ?read.spss
> imported.data = read.spss('filename.sav', to.data.frame = TRUE)
```

There are some real idiosyncrasies here, and honestly I would avoid importing data this way if at all possible. In some ways, it might be easier to get SPSS to export the data to a csv, and then import the csv into R.

### *Packages*

One of the best things about R is that it is open source and easily extended. These days, it's pretty common for statistics researchers to put together and share an R package to go with the new methods they develop. So if there's any method you want to use, there is almost certainly an R package or two or three that will do it.

Here, we'll install and load a package that will fit a Gaussian mixture model, often used for clustering, `mclust`. There is a command

line option for installing packages, but this is one thing it's much easier to do with the drop down menus. Find the menu option for the package installer, as in Figure 15. That should open up the window for the installer, shown in Figure 16.

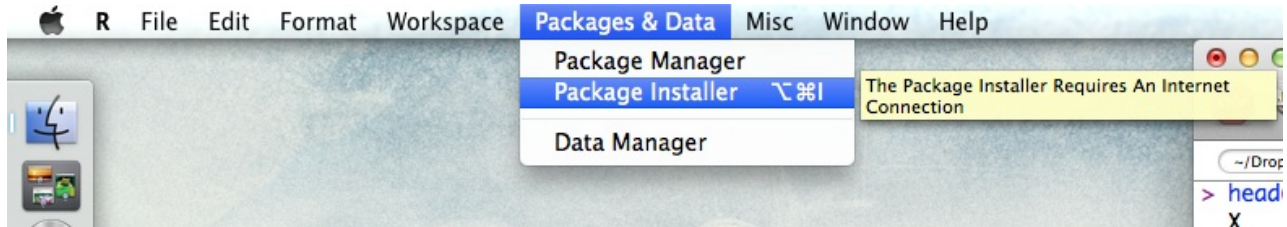


Figure 15: Location for the package installer menu option on Macs.

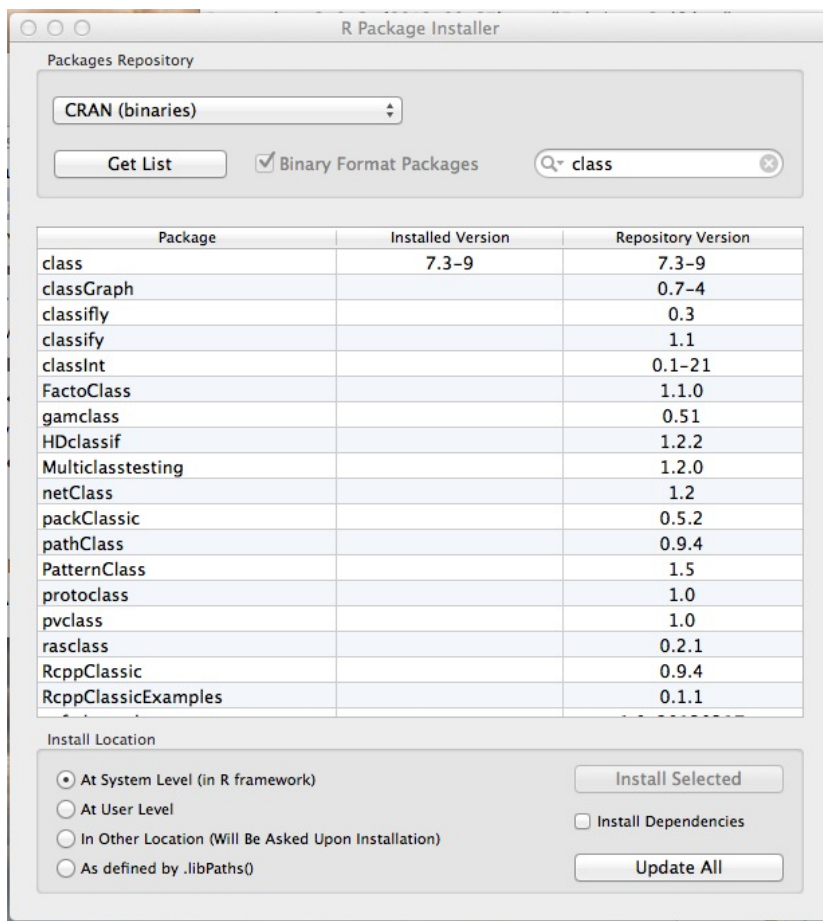


Figure 16: The package installer window.

First, click the Get List button to bring up the list of available packages. You can try scrolling through, but you will find that the list is very long and the names don't really tell you much on their own. Use the search bar to find the package that you're looking for, in this case mclust. Highlight the package you want to install. Before

you click `Install Selected`, I recommend selecting 2 options. First, go ahead and install the packages at the system level where you installed R. Second, click `install dependencies`. Some packages make use of other packages, so you usually want to install these "dependencies" so everything will work correctly. When the package is installed, the version number for the installed version will appear, if it matches the repository version number, then you're up to date.

Now to check out what's in the package and be able to use the new functions, we need to load the package. One of the main functions in the package is `Mclust`, note that if we try to look at the help file for it before we load the package, we get an error.

```
> ?Mclust
No documentation for 'Mclust' in specified packages and libraries:
you could try '??Mclust'
> library(mclust)
Package 'mclust' version 4.2
>
> ?Mclust
starting httpd help server ... done
```

You can look at the documentation for the whole package, including the list of all the functions in the package in a couple of ways. First, scroll all the way down to the bottom of the `Mclust` help file, and click on the index link. Second, use the package manager.

### *Congratulations*

Lastly, just remember that google is your friend. R is open source, so any question you have has probably already been asked and answered on some forum somewhere.

