Lecture 2: Basic Computation and Stats

Kelly

And how will all of this help me?

You might be asking yourself: "I very rarely care to add vectors together or to create variables or to do any of the things that **R** seems to be able to do. Why might I care?"

To which we respond: you will be surprised. You plot things all the time, right? (If not, you should!) You often have questions like "Yes, I know that 5 seaplanes leave Lake Union normally, but yesterday there were 9. That seems like a lot… is it really a lot? Or might that just happen by chance?" Or perhaps "I talked to 10 people at the Council meeting, and 3 were in favor of the proposal. Is that more than I expected, given the polling numbers?" **R** is your tool for these kinds of things. And for plotting… perhaps you want to plot the number of planes that take off each day of the month from Lake Union, and then you want to do that for each month of the year. Easy in **R**, hard in Excel.

And all kinds of other fun stuff you haven't even thought about yet.

So, to review:

R is a language, rather than a particular product. As a result, it can be confusing when people talk about how they work in with **R**. You can write in any language using a plain text editor (such as Notepad++, Atom, BBedit, Textwrangler, or just the plain old notepad that came by default on your computer). But the text editor couldn't *run* the code… for that, you would need a program that speaks and executes the **R** language. And wouldn't it be easier if you could write code, run that code, and visualize the code more easily? I thought so. That's why people use **R**-specific software to do it. That software comes in a few flavors:

- Command-line **R** can be run on a terminal (command-prompt) with plain text. For super-nerds.
- For the less-nerdy, the GUI (Graphical User Interface… i.e., where you can use a mouse and click on things) version of **R** is available on CRAN (CRAN stands for the Comprehensive R Archive Network, which is also where most contributed packages live). This is a big improvement for most people, but has in part been superseded by…
- RStudio, which is an *Integrated Development Environment* (IDE), the idea of which is to keep everything all in one place: you can write code, run code, see plots, see what files are where, see what objects you've got loaded into an environment, etc… all on one screen. That's why we're using RStudio.

But the point is that **R** is the language, and RStudio (or whatever you might otherwise choose… there are other options) is the environment in which you're using **R**.

Some Vocabulary of Note:

- *Rmarkdown* is a handy way of keeping code and plots and calculations and text and notes all in one place. The document you are reading now is written in Rmarkdown, and you'll write your problem sets, etc, the same way. The details don't matter, but just know that Rstudio is the way that you easily create Rmarkdown documents. You can find the basics of Rmarkdown here or in the cheatsheets provided in the files associated with our class. (Rmarkdown is itself a kind of mini-language for formatting text. Note that the tab character, and new lines, are important in *Rmarkdown* for formatting, in a way that they are generally ignored in the *R* language, so be careful.)

- The *console* is where the action takes place in RStudio. If you run code from the *source* window (which is your writing/editing space), you see the code being carried out in the *console*.

- Just so you know, you can create freestanding bits of code (*scripts*) that you can call from the command line, when you're more advanced. You run them like programs (which they are), without having to edit anything in Rstudio, etc.

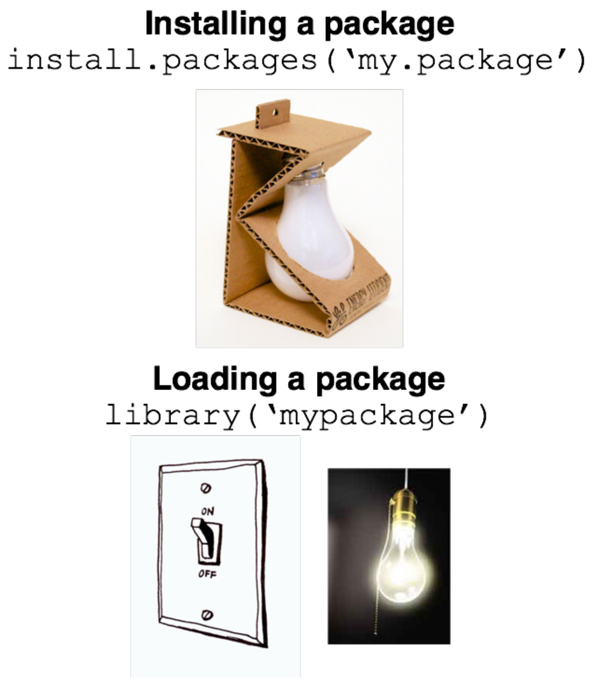


Figure 1: An R package is like a lightbulb. First you need to order it with install.packages(). Then, every time you want to use it, you need to turn it on with library()

Packages : Extensions for R

You already installed and loaded a package in **R** last week. But let's take a minute to see what that's all about. The *Pirate's Guide* says:

> When you download and install R for the first time, you are installing the Base R software. Base R will contain most of the functions you'll use on a daily basis like `mean()` and `hist()`. However, only functions written by the original authors of the R language will appear here. If you want to access data and code written by other people, you'll need to install it as a *package*. An R package is simply a bunch of data, from functions, to help menus, to vignettes (examples), stored in one neat package.
>
> A package is like a light bulb. In order to use it, you first need to order it to your house (i.e.; your computer) by *installing* it. Once you've installed a package, you never need to install it again. However, every time you want to actually use the package, you need to turn it on by *loading* it. Here's how to do it.

Installing a new package

> Installing a package simply means downloading the package code onto your personal computer. There are two main ways to install new packages. The first, and most common, method is to download them from the Comprehensive R Archive Network (CRAN). CRAN is the central repository for R packages. To install a new R package from CRAN, you can simply run the code `install.packages("name")`, where "name" is the name of the package. For example, to download the `yarrr` package, which contains several data sets and functions we will use in this book, you should run the following:

```r
# Install the yarrr package from CRAN
#   You only need to install a package once!
install.packages("yarrr")
```

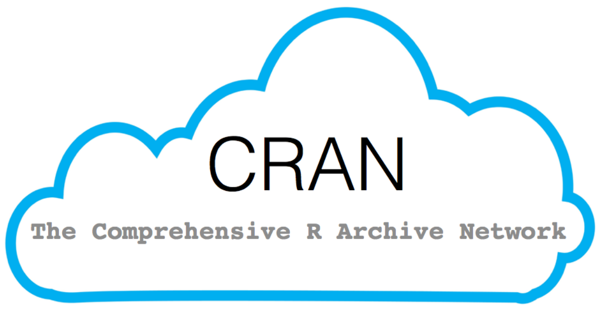
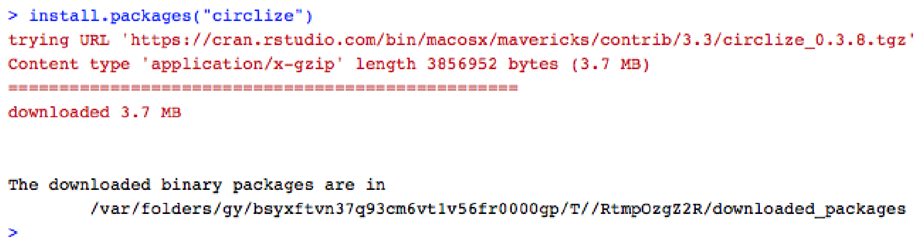


Figure 2: CRAN (Comprehensive R Archive Network) is the main source of R packages



```
> install.packages("circlize")
trying URL 'https://cran.rstudio.com/bin/macosx/mavericks/contrib/3.3/circlize_0.3.8.tgz'
Content type 'application/x-gzip' length 3856952 bytes (3.7 MB)
==================================================
downloaded 3.7 MB


The downloaded binary packages are in
	/var/folders/gy/bsyxftvn37q93cm6vt1v56fr0000gp/T//RtmpOzgZ2R/downloaded_packages
>
```

Figure 3: When you install a new package, you'll see some random text like this you the download progress. You don't need to memorize this.

When you run `install.packages("name")` R will download the package from CRAN. If everything works, you should see some information about where the package is being downloaded from, in addition to a progress bar.

Like ordering a light bulb, once you've installed a package on your computer you never need to install it again (unless you want to try to install a new version of the package). However, every time you want to use it, you need to turn it on by loading it.

Loading a package

Once you've installed a package, it's on your computer. However, just because it's on your computer doesn't mean R is ready to use it. If you want to use something, like a function or dataset, from a package you *always* need to *load* the package in your R session first. Just like a light bulb, you need to turn it on to use it!

To load a package, you use the `library()` function. For example, now that we've installed the `yarrr` package, we can load it with `library("yarrr")`:

```r
# Load the yarrr package so I can use it!
#   You have to load a package in every new R session!
library("yarrr")
```

USING *R*

OK, back to us for a minute, away from the *Pirate's Guide*…

Some basic math

R is a calculator, among other things. It does what you'd expect. [Note here that **R** has created a different syntax for these basic operations… the normal `function(argument)` syntax works, but so does `3 + 2`, because well… that just seems like it should work, no? And so the good people of **R** made it so.]

```
3 + 2
```

```
## [1] 5
```

```
#or equivalently:
sum(c(3,2))
```

```
## [1] 5
```

```
3*2
```

```
## [1] 6
```

```
sqrt(4)
```

```
## [1] 2
```

```
4^2
```

```
## [1] 16
```

```
#scientific notation, if that's your thing
10e-3
```

```
## [1] 0.01
```

…but you don't really need another calculator. Your phone can do these things. But can your phone do this? :

Handy Functions

summary()

This gives – you guessed it – a summary of an object. If you use it to summarize a numeric vector, you get something like this:

```
summary(AirPassengers)  #a vector that is one of many datasets included in RStudio by default. Monthly
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   104.0   180.0   265.5   280.3   360.5   622.0
```

If you use **summary()** to summarize a data frame, you get

```
summary(CO2) #a dataframe of Carbon Dioxide Uptake in Grass Plants; contains some numeric data and some
```

```
##      Plant             Type         Treatment       conc
##  Qn1    : 7   Quebec     :42   nonchilled:42   Min.   :  95
##  Qn2    : 7   Mississippi:42   chilled   :42   1st Qu.: 175
##  Qn3    : 7                                    Median : 350
##  Qc1    : 7                                    Mean   : 435
##  Qc3    : 7                                    3rd Qu.: 675
##  Qc2    : 7                                    Max.   :1000
##  (Other):42
##      uptake
##  Min.   : 7.70
##  1st Qu.:17.90
##  Median :28.30
##  Mean   :27.21
##  3rd Qu.:37.12
##  Max.   :45.50
##
```

This is a quick way of seeing what's going on with your dataset.

dim() and length()

dim() gives the dimensions of your object; for example the number of rows and columns in a dataset.

length gives the length of a vector.

Note in RStudio, the "environment" tab usually gives you this information, so you don't need to call the functions.

```
#using the same example datasets as above:
dim(CO2) #the dataframe has 82 rows and 5 columns
```

```
## [1] 84  5
```

```
dim(AirPassengers) #this doesn't work on vectors...
```

```
## NULL
```

```
length(AirPassengers) #this is the equivalent for vectors.
```

```
## [1] 144
```

Subsetting and Indexing

Two ways of taking a slice of your data are 1) referring to the slice you want by its row or column number, and 2) referring to the slice by its column name. There are other ways of subsetting, but these are most common, and we'll start with them.

Numerical Indexing

When you want to take a certain set of your data, but not the whole dataset, we call that *subsetting*. More generally, the idea that you can call particular elements by name or number is called *indexing*. For example, let's say you have a vector of 10 numbers (below).

```
x <- c(1,5,7,2,8,10,3,8,5,4)
x
```

```
##  [1]  1  5  7  2  8 10  3  8  5  4
```

Let's now say you wanted to grab the 7th number in the vector, and multiply it by 100. We can do that using the notation `[]` where inside the square brackets is the *index* of the element you want to select. So if we want to select the 7th element in the vector x, we do that as follows:

```
x[7]
```

```
## [1] 3
```

And we can manipulate it as follows:

```
x[7] * 100
```

```
## [1] 300
```

Note that this doesn't change the stored version of the vector; if we want to do that, we have to store it explicitly.

```
x
```

```
##  [1]  1  5  7  2  8 10  3  8  5  4
```

```
x[7] <- x[7]*100  #we just want to change the 7th element, and we do this explicitly

x
```

```
##  [1]   1   5   7   2   8  10 300   8   5   4
```

If we want to see elements 3 through 10, we do it in a similar way; and we can even change the order in which the elements are shown to us.

```
x[3:10] #shows elements 3 through 10
```

```
## [1]   7   2   8  10 300   8   5   4
```

```
x[c(4,7,2)]  #shows the 4nd, 7th, and 2nd elements in a vector
```

```
## [1]   2 300   5
```

Here we've indexed by number, but you can do it by element *name* as well, shown in the worked example below.

Dollar Sign

And in many contexts, the dollar sign is the most useful way of subsetting. When you want to take a single column of a dataframe, for example, and use it as a vector, this is the way to do it.

Let's say we want to just pull the number of treasure chests each pirate has, out of the `pirates` dataset.

```
pirates$tchests  #this is just that column of data.
```

```
##   [1]   0  11  10   0   6  19   1  13  37  69   1   5   6  12  70   3   3
##  [18] 107  75  30  11  13  11   3  18  10   2  15   1  44  54   6   3  15
##  [35]   2  14   3   1  39  18  58   1  10   0  17  14  56  26   9   1  13
##  [52]   0   6 102   1  17  36  35  25  20  38   3   0  14  20  26  34  18
##  [69]  16  31  38  17  59  31  17   4   9  64   2  41  54   2   0   5 122
##  [86]   1  73   1  25   7   0  13  75   7  21  18  66  10  34  72  10  24
## [103]  22   6  15  15  23  37  29  63   2  59   8   2   9   9   6   0  67
## [120]   2  24   9  22  21   0  33   0  10   3  36  14  19  48  27  29   4
## [137]  20   3   3  69  92  14  35  74  36   7  28  24   5   4  19  43   5
## [154]  13  18  21  30   4   6  14  60   9  27  28  28   1  72  13  13   8
## [171]  26  13  12  55 110  13  14   2  60  25   4  34   2  19  11  15  10
## [188]  44  58   1  23   6  20   5  22  46  23   4  15   8  12   1  10  49
## [205]  25   1   2   7   2 122  11   4   7  12  28  54  37  11  26  52   9
## [222]  37  28  13  25 123   2  12  21   9   9  48  18   5   7   4  17  63
## [239]  15 125   2  18  59  23  21 101  13  43  16  21   1  10   8  47   6
## [256]   9  32  19  10  17   0  15   0   1   8  14  32   9  14  10  12  16
## [273]  28  20  60  32  32   0  54  33  30   5  35   4  18  47   7  30  30
## [290]   1  40   2  32   9  78   2  51  15   9   0  37   1  10  44   3  10
## [307]  18  53  10  15  33  11   5 135  18   8  12  66  12   2  20  12  71
## [324]  18  54   1   4   9  15   2   3  22   2   6  11  22  27 139   3   6
## [341]  12   7  24   7  13   6   5   9  13  10   1  45  11   5   5  15  15
## [358]  27  41  32   5   1  23  46   5  23  49  46  16   6  19  31  84   6
## [375]  15  53  45   6   4  12  15   4  21  22  21  25   7   6  55  14  47
## [392]   8  32  10  23 109  14  14  47  19  15  35   2  12   6   3  24  24
## [409]  10   0  14  52   3   1   4  15  25  22   0  10   3   4  60   7  40
## [426]   3   5  18  28   7  23  25  12  10   9  37  32   4  10   9  12   6
## [443]  26   2   2   4  36   5   3  16   9  12   2  16  33   5  13  27  13
## [460]   3 147  19  53   7   5  34   1   5  36   1  80   8  35  67  10   4
## [477]   5  12   5  12  10  13  19  16   3   3   4  13  32   5  20  14 134
##  [494]  11  33  17   2   3  28 129  13  72  26  11  74  21   3  39  54  53
##  [511]   8   1  51   5  29  14   5  21  11  38  18   8   1  27  18  13   7
##  [528]  10  36  46  16  52   6  39   3  16  17   4   3  21  19  33   1  36
##  [545]  10  27  64   0  33   8   6  37   8  17  18  33  10   1  57  39  14
##  [562]   3   4  39   7  22  14  36   8  12   5  56   3  77   5  30  19  17
##  [579]   4   8  15  22   0  28   0   9   8  21  15  19  27  46   5  17  34
##  [596]  13  12  50  28  30   2  42   1   1   3   5  21  22  45  16  47  11
##  [613] 126  13   6  50   5   3  23  10  23   1  28  35  18   8  17   8  11
##  [630]  36  44   1   4  13   4  36  12 135  66  35  17   1   1  19   1   5
##  [647]  22  58  35  17 101   1  11  30  17  94  14  91   1  18   6  43  21
##  [664]  51   2   3  45  26   4   6   7   1 138 101  57   7   8   6  26  16
##  [681]  86  11  13  44  16   3  33  37   9  10  11  41  12  62  22  37  64
##  [698]  29  68   5  46  20  80  23  14  58   0   1   9   0  13  28  26   3
##  [715]   1   1  31 139   5  19  26   8  20  30   2  58  88   2   9  23  17
##  [732]   2  14  19  31  30  12   3  25   1   7   3  41   3   3  19  14  90
##  [749]   4  13  11  59   3  30  42  22  28   4  33 106   5  26  24   7   5
##  [766]  47  20  20  15   7  18   6  14  20  15   4  14 118  17   4   2  11
##  [783]   5   3  31   1   4   9  60  52  23  76  27  24  29  41  20  20  15
##  [800]  33 115  13  51   1  28   6   6  86  60  22  20  30  30  19  50  76
##  [817]   5  10  22  11  21   6   3   9  35  42   8  12  57  50   1   7   2
##  [834]   8  28  40  13  13  35  26   5   8   3  51  12   5   3  48   5  19
##  [851]   6   2  15   6   4   6  15  16  80   4  65  14  50   2  26  82  44
##  [868]   4  10  19   6   3  71   6  23  11  22   0  16   6  19  34  10   3
##  [885]  37  33  21  39   5 102  11  21   7  83   1   2  26  21   4   4   3
##  [902]  74   3   2  19  18  14  10  41   8   5  29   5   8  24   7  15  29
##  [919]  41  31  20   8  11  30  21  18   2  12  18   8   1   5  13  10   4
##  [936]  52  14   7  22  24  38   7  84  14  15  28  21  14   8  61  23  27
##  [953]  14  49   6   4   3  26   1  14  10  12   7  23  21   9   3   6   1
##  [970]  17   3   7   4  65   4  26  11   1  41  20  64  51  24  26   0  24
##  [987]  30  19  26  37   6   6  15  12  18  11  48  95   6  36
```

```
#And a single column of data is, by definition, a vector.

#we could then summarize this or plot it, or whatever...
summary(pirates$tchests)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    6.00   15.00   22.69   30.00  147.00
```

#A Worked Example

Here's why subsetting is useful:

Take the sample dataset (already installed) called *WorldPhones* (the number of telephones over time, by geography).

```
WorldPhones
```

```
##      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
## 1951  45939  21574 2876   1815    1646     89      555
## 1956  60423  29990 4708   2568    2366   1411      733
## 1957  64721  32510 5230   2695    2526   1546      773
## 1958  68484  35218 6662   2845    2691   1663      836
## 1959  71799  37598 6856   3000    2868   1769      911
## 1960  76036  40341 8220   3145    3054   1905     1008
## 1961  79831  43173 9053   3338    3224   2005     1076
```

Let's select and plot the growth of phone service in the Americas (overall). First, we select the columns we care about:

```
(Amers <- WorldPhones[,c("N.Amer", "S.Amer", "Mid.Amer")])
```

```
##      N.Amer S.Amer Mid.Amer
## 1951  45939   1815      555
## 1956  60423   2568      733
## 1957  64721   2695      773
## 1958  68484   2845      836
## 1959  71799   3000      911
## 1960  76036   3145     1008
## 1961  79831   3338     1076
```

Note a few things here:

- First, you have to tell R that you want all rows for the selected columns. In general, we index dataframes by [rows, columns]. To select all rows for only some columns, you don't put a value in the rows spot, so it looks like this: [,columns].
- Second, we selected columns by column name. We could have put in `WorldPhones[,c(1,4,7)]`, indexing by number, and gotten the same result.
- Third, we stored the subset dataframe in a variable called *Amers*. And we printed the results to the screen at the same time by using a little trick: the whole expression was in parentheses.
- Forth, we created a vector `c("N.Amer", "S.Amer", "Mid.Amer")` to tell **R** we wanted to pull multiple columns from the dataset. Most times you are giving **R** a set of things, you are going to use this notation. Here, it's a vector of character strings… the names of the columns.

Now we want to sum across rows. Fortunately, **R** has a built-in function to do that:

```
(totals<-rowSums(Amers))
```

```
##  1951  1956  1957  1958  1959  1960  1961
## 48309 63724 68189 72165 75710 80189 84245
```

And finally, we want to plot this across years. But right now the years aren't stored as numbers… they are just the row names of the dataframe. We can check this, and it's true:

```
is.numeric(row.names(Amers))
```

```
## [1] FALSE
```

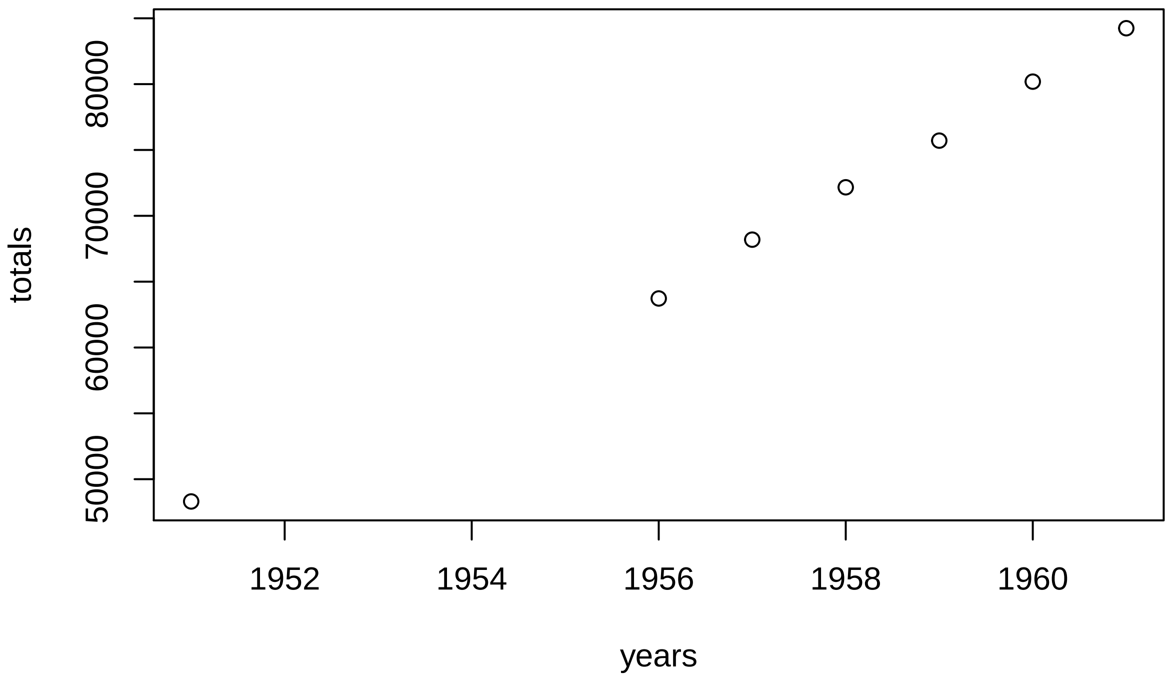
So we are going to *coerce* the row names into numeric values, as follows:

```
(years <- as.numeric(row.names(Amers)))
```

```
## [1] 1951 1956 1957 1958 1959 1960 1961
```

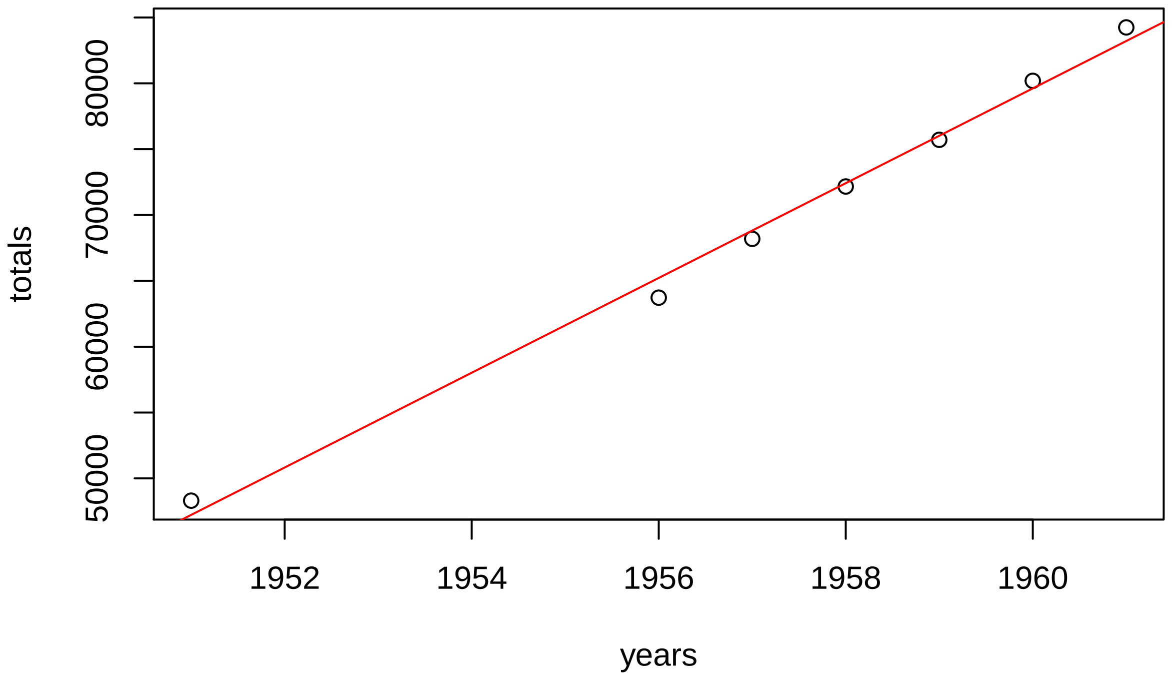
And now we can make our plot:

```r
plot(totals~years)
```



We can add a best-fit trendline if we are feeling saucy.

```r
plot(totals~years)
  abline(lm(totals~years), col="red")  #we'll do more of this soon;  lm is for linear model
```



Reading-in and Writing-out data

Let's go back to the *Pirate's Guide*:

read.table()

If you have a .txt file that you want to read into R, use the `read.table()` function.

Argument	Description
`file`	The document's file path relative to the working directory unless specified otherwise. For example `file = "mydata.txt"` looks for the text file directly in the working directory, while `file = "data/mydata.txt"` will look for the file in an existing folder called `data` inside the working directory.If the file is outside of your working directory, you can also specify a full file path (`file = "/Users/CaptainJack/Desktop/OctoberStudy/mydata.txt"`)
`header`	A logical value indicating whether the data has a header row – that is, whether the first row of the data represents the column names.
`sep`	A string indicating how the columns are separated. For comma separated files, use `sep = ","`, for tab–delimited files, use `sep = "\t"`
`stringsAsFactors`	A logical value indicating whether or not to convert strings to factors. I **always** set this to FALSE because I *hate*, *hate*, *hate* how R uses factors

The three critical arguments to `read.table()` are `file`, `sep`, `header` and `stringsAsFactors`. The `file` argument is a character value telling R where to find the file. If the file is in a folder in your working directory, just specify the path within your working directory (e.g.; `file = data/newdata.txt`. The `sep` argument tells R how the columns are separated in the file (again, for a comma–separated file, use `sep = ","`}, for a tab–delimited file, use `sep = "\t"`. The `header` argument is a logical value (TRUE or FALSE) telling R whether or not the first row in the data is the name of the data columns. Finally, the `stringsAsFactors` argument is a logical value indicating whether or not to convert strings to factors (I *always* set this to FALSE!)

Let's test this function out by reading in a text file titled `mydata.txt`. Since the text file is located a folder called `data` in my working directory, I'll use the file path `file = "data/mydata.txt"` and since the file is tab–delimited, I'll use the argument `sep = "\t"`:

```
# Read a tab-delimited text file called mydata.txt
#  from the data folder in my working directory into
#  R and store as a new object called mydata

mydata <- read.table(file = 'mydata.txt',          # file is in my working directory
                     sep = '\t',                   # file is tab--delimited
                     header = TRUE,                # the first row of the data is a header row
                     stringsAsFactors = FALSE)     # do NOT convert strings (i.e., "words" or text) to f
```

Reading files directly from a web URL

A really neat feature of the `read.table()` function is that you can use it to load text files directly from the web (assuming you are online). To do this, just set the file path to the document's web URL (beginning with `http://`). For example, I have a text file stored at `http://goo.gl/jTNf6P`. You can import and save this tab–delimited text file as a new object called `fromweb` as follows:

```
# Read a text file from the web
fromweb <- read.table(file = 'http://goo.gl/jTNf6P',
                      sep = '\t',
                      header = TRUE)

# Print the result
fromweb
```

```
##              message randomdata
## 1 Congratulations!         1
## 2              you         2
## 3             just         3
## 4       downloaded         4
## 5             this         5
## 6            table         6
## 7             from         7
## 8              the         8
## 9             web!         9
```

I think this is pretty darn cool. This means you can save your main data files on Dropbox or a web-server, and always have access to it from any computer by accessing it from its web URL.

Debugging

When you run `read.table()`, you might receive an error like this:

Error in file(file, "rt") : cannot open the connection

In addition: Warning message:

In file(file, "rt") : cannot open file 'asdf': No such file or directory

If you receive this error, it's likely because you either spelled the file name incorrectly, or did not specify the correct directory location in the `file` argument.

Excel, SPSS, and other data files

A common question I hear is "How can I import an SPSS/Excel/... file into R?". The first answer to this question I always give is "You shouldn't". **S**hitty **P**iece of **S**hitty **S**hit files can contain information like variable descriptions that R doesn't know what to do with, and Excel files often contain something, like missing rows or cells with text instead of numbers, that can completely confuse R.

Rather than trying to import SPSS or Excel files directly in R, I always recommend first exporting/saving the original SPSS or Excel files as text `.txt.` files – both SPSS and Excel have options to do this. Then, once you have exported the data to a `.txt` file, you can read it into R using `read.table()`.

Warning: If you try to export an Excel file to a text file, it is a good idea to clean the file as much as you can first by, for example, deleting unnecessary columns, making sure that all numeric columns have numeric data, making sure the column names are simple (ie., single words without spaces or special characters). If there is anything 'unclean' in the file, then R may still have problems reading it, even after you export it to a text file.

If you absolutely *have* to read a non-text file into R, check out the package called `foreign` (`install.packages("foreign")`). This package has functions for importing Stata, SAS and SPSS files directly into R. To read Excel files, try the package `xlsx` (`install.packages("xlsx")`). But again, in my experience it's *always* better to convert such files to simple text files first, and then read them into R with `read.table()`.

There are many functions other than `read.table()` for importing data. For example, the functions `read.csv` and `read.delim` are specific for importing comma-separated and tab-separated text files. In practice, these functions do the same thing as `read.table`, but they don't require you to specify a `sep` argument. Personally, I always use `read.table()` because it always works and I don't like trying to remember unnecessary functions.

Writing Data Out

And writing data to files is pretty much the same process. If you want to write out a table, for example, you could do it like this:

```
write.table(ChickWeight,                        #write a table consisting of the dataframe "ChickWeight
            file = "NewLocalFileCreated.txt",   #call the file this, and put it in the working director
            quote = FALSE,                      #do not put quotes around the values in the dataframe
            sep = "\t",                         #use tabs to separate values, so it's tab-delimited
            row.names = FALSE,                  #do not include the row-names in the written file
            colnames = TRUE,                    #do include the column names
            na = "NA")                          #for missing or otherwise invalid characters, use "NA"
```

Otherwise, if you want to save your R objects directly (rather than writing files as tables, etc), follow the *Pirate's Guide*:

.RData files

The best way to store objects from R is with `.RData files`. `.RData` files are specific to R and can store as many objects as you'd like within a single file. Think about that. If you are conducting an analysis with 10 different dataframes and 5 hypothesis tests, you can save **all** of those objects in a single file called `ExperimentResults.RData`.

save()

To save selected objects into one `.RData` file, use the `save()` function. When you run the `save()` function with specific objects as arguments, (like `save(a, b, c, file = "myobjects.RData")` all of those objects will be saved in a single file called `myobjects.RData`

For example, let's create a few objects corresponding to a study.

```
# Create some objects that we'll save later
study1.df <- data.frame(id = 1:5,
                        sex = c("m", "m", "f", "f", "m"),
                        score = c(51, 20, 67, 52, 42))

score.by.sex <- aggregate(score ~ sex,
                          FUN = mean,
                          data = study1.df)

study1.htest <- t.test(score ~ sex,
                       data = study1.df)
```

Now that we've done all of this work, we want to save all three objects in an a file called `study1.RData` in the data folder of my current working directory. To do this, you can run the following

```
# Save two objects as a new .RData file
#   in the data folder of my current working directory
save(study1.df, score.by.sex, study1.htest,
     file = "data/study1.RData")
```

Once you do this, you should see the `study1.RData` file in the data folder of your working directory. This file now contains all of your objects that you can easily access later using the `load()` function (we'll go over this in a second. . .).

save.image()

If you have many objects that you want to save, then using `save` can be tedious as you'd have to type the name of every object. To save *all* the objects in your workspace as a .RData file, use the `save.image()` function. For example, to save my workspace in the `data` folder located in my working directory, I'd run the following:

```r
# Save my workspace to complete_image.RData in th,e
#  data folder of my working directory
save.image(file = "data/projectimage.RData")
```

Now, the `projectimage.RData` file contains *all* objects in your current workspace.

load()

To load an `.RData` file, that is, to import all of the objects contained in the `.RData` file into your current workspace, use the `load()` function. For example, to load the three specific objects that I saved earlier (`study1.df`, `score.by.sex`, and `study1.htest`) in `study1.RData`, I'd run the following:

```r
# Load objects in study1.RData into my workspace
load(file = "data/study1.RData")
```

To load all of the objects in the workspace that I just saved to the data folder in my working directory in `projectimage.RData`, I'd run the following:

```r
# Load objects in projectimage.RData into my workspace
load(file = "data/projectimage.RData")
```

I hope you realize how awesome loading .RData files is. With R, you can store all of your objects, from dataframes to hypothesis tests, in a single `.RData` file. And then load them into any R session at any time using `load()`.