An Introduction to R

Peter Haschke

on behalf of THE STAR LAB

Updated: Thursday 31st January, 2013

License

This document contains and incorporates material from the following sources:

Jonathan Olmsted's *the star lab's Introduction to R: A Short Course*,
available here: https://github.com/olmjo/R-Intro.

and

Brenton Kenkel's *An Introduction to R*,
available here: http://www.rochester.edu/College/gradstudents/bkenkel/data/rcourse.pdf

If you have any questions, comments, and or concerns relating to this document, please contact Peter Haschke (peter.haschke@rochester.edu).

Contents

The R Console

```
R version 2.15.2 (2012-10-26) -- "Trick or Treat"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

Chapter 1

The Course

This short course is designed to:

- serve as an introduction to the R language and it's uses
- teach you the basics of R's syntax
- provide an overview of how to implement some rudimentary statistical techniques and compute basic statistics
- showcase some of R's graphical capabilities
- have some fun in the THE STAR LAB

We will not cover all the things you will eventually need to know about programming in R. This course is merely meant to provide you with a basic understanding of how R works and how to get started. There are no prerequisites and I assume no prior programming knowledge. You should be able to use a mouse and a keyboard. If you feel underwhelmed, please be courteous to your colleagues. If you are overwhelmed, immediately let me know. With any luck, however, you will be able to throw away that old TI-81 at the end of this tutorial.

1.1 Housekeeping & Logistics

We will meet in THE STAR LAB from 7:30pm to 9:00pm, starting January 17, 2013. During each meeting we will go over the contents of one chapter of this tutorial. After each meeting/class, I will hand out a small problem set. Since this is not a graded course, it is entirely up to you to complete them. I nevertheless encourage you to work through them. I swear things will make more sense if you do. Also, the problems/puzzles may actually be fun. The answers and all course materials can be found by directing your favorite web-browser to this address: R-Course Webpage.

If you have any questions, please ask. More likely than not, I'll make mistakes, will be unclear, or just plain wrong. So let's clear problems/misunderstandings/confusions out of the way as they come up. So again, if something doesn't make sense or if you don't understand something interrupt and ask. If you encounter errors in my code, in the problem sets, & cetera, let me know. I wouldn't want anybody get frustrated and waste hours on an unsolvable problem.

1.2 Preliminaries

1.2.1 Why R?

Aside from the fact that you will be required to write your own programs in R for PSC 505 and perhaps even PSC 405, R has a number of virtues and advantages compared to other statistical software packages (e.g. Stata, etc ...).

1. It is open-source, it is free!
2. It is cross-platform (Windows, Mac, Linux).
3. It is what “real” scientists use.
4. It has a large active, helpful, and friendly user base.
5. It is updated regularly.
6. It has unrivaled graphical capabilities.
7. It is extremely flexible and can do or be made to do just about anything.
8. It is better than Stata. Period.[1]

1.2.2 What is the catch?

1. It is not a spreadsheet (e.g. Excel). So you do not “see” what’s going on.
2. There is no *real* GUI (i.e. no point-and-click interface).[2]
3. It is not the best tool for non-statistical programming (e.g. web scraping). Duh.
4. There is an initially somewhat steep learning curve (especially without any programming background).
5. It comes with ABSOLUTELY NO WARRANTY.

1.2.3 Installing R

R is installed on all computers in THE STAR LAB. If you don’t own your own computer or if you do not want to install it on your personal machine, you can ignore this section. Make THE STAR LAB your home. If you decide, however, that you want to use R at your other home. Here is how to get it up and running.

1. Navigate to the Comprehensive R Archive Network (CRAN) website at http://cran.r-project.org/.

[1] Fact!

[2] You can install GUIs and IDEs separately, if you want to be like that. You don’t want to be like that.

2. At the top of the page click the appropriate link for your operating system
   - **Windows**: Click on the link to the `base` subdirectory and then on Download R 2.15.2 for Windows. (This is the most recent version as of Dec. 2012.)
   - **Mac OS X**: Click the first link under the "Files" heading and download the file: R-2.15.2.pkg. (This is the most recent version as of Dec. 2012.)
   - **Linux**: Chose the directory of your Linux distribution and follow the instructions. Alternatively you can download the source code from the homepage and compile R yourself.
3. Unless you have chosen to compile R from source, run the executable you just downloaded.
4. You are done.[3]

R is updated regularly (roughly 3 times a year). These updates, contain improvements, bugfixes, and new features. Newly developed or updated packages also often are not backward compatible. As such you definitely want to make sure you keep R up-to-date. Updating can be tedious (e.g. on Windows R needs to be reinstalled). Make sure you keep track of all packages you download and backup your settings (e.g. your `Rprofile.site` file, et al.) to make the updating process as seamless and easy as possible.[4]

1.2.4 Text Editors

The R GUI is pretty sparse. When you start R you will only see the R-console which does include a few drop-down menus for some useful commands and actions. Beyond this the GUI is fairly limited when it comes to doing actual work, writing programs, and maintaining your code.

This is quite OK. After all, R is really just a command line interpreter and not a text editor or full-featured application. So R is simply designed to interpret your inputs.[5]. It does not care where those inputs come from, how you entered them, or if you saved them. To make a long story short, you will NEVER, EVER, EVER want to input commands/code directly into the R-console. Chances are that R will crash, the power goes out, or you close your R-session without saving, and all your precious code, and computations are gone - FOREVER.

To avoid endless frustration and to maintain good mental health, ALWAYS, ALWAYS, ALWAYS write all you code in a text or script editor. Good science requires repeatability and the communication of knowledge. Anything you manually type into the R-console or the command line will be lost forever after you close the terminal/console. You won't be able to replicate anything, or send your code and hard work to anybody, for help, debugging, or sharing of results.[6] You will not do science and just draw lines in the sand. So again R does not care which text editor you use. All R wants is interpretable input. There are a gazillion editor options that will do the trick and allow you to write your code and feed it to R. Below are some options and my two cents about them. I apologize for the emphasis on software for Windows.[7]

[3] On Windows, adding R to your `path` is advisable, especially if you do not want to use the R-console and instead spawn an R-session from the Windows Command Prompt, etc ...

[4] Your packages are located in the `/library` directory. For example `C:/Programs/R-2.1.5/library/`.

[5] You can do this at the command line, e.g. in the Windows Command Prompt, Apple's Terminal App, etc ...

[6] Imagine typing a paper straight into the command line to have LaTeX compile it to a `.pdf`. Once you close the terminal, you won't ever be able to edit or change anything.

[7] Did I mention that I hate everything Apple?

1. **Notepad**: It works. You can save your code. Nothing else.

   → Verdict: Do not use.

2. **R Editor**: The R Editor will automatically get installed with R. It is a marginal improvement over Notpad. It runs on all platforms. You can save your work. It allows you to send code directly to the R-console for interpretation (via keyboard shortcuts).

   → Verdict: Do not use.

3. **WinEdt**: This is a full-featured text editor. It comes with R integration via an R-package called RWinEdt. You can send code directly to R. It has syntax highlighting and a few other useful features. It is ugly. Think Windows 95. It costs money. It only runs on Windows. Duh. I will show you later on in this tutorial how to install and use R-packages.

   → Verdict: Do try. If it works for you, great.

4. **Notepadd++**: This is a really nice lightweight editor for Windows. It is free and open-source. It is highly customizable, theme-able, and good looking. Has auto-completion, function-completion, and function-folding capabilities. Multi-language support. It can be downloaded here: http://notepad-plus-plus.org/. Via a the NppToR plug-in you can also send code straight to R. It also allows PuTTy integration for passing to an R instance on a remote computer. NppToR can be downloaded here: http://sourceforge.net/projects/npptor/.

   → Verdict: I think this is a great choice if you are looking for a free, easy to use, and versatile editor for Windows.

5. **Emacs**: This is probably the editor of choice if you're some type of hardcore programmer, hacker, Linux-geek, or nerd. It is cross-platform. And does anything any text editor could ever be asked to do and more. There exist plug-ins making your R programming much easier (e.g. ESS: Emacs Speaks Statistics) but unless you have a ton of time on your hand or are crazy, you should stay away from Emacs. The learning curve is just as high as it is for R itself and we are talking about a text editor, here. I won't provide you with links.

   → If you are not using it already, stay away. You have better things to do.

6. **RStudio**: RStudio is not a text editor but a free and cross-platform IDE or software application that provides comprehensive facilities especially designed for R programmers.[8] It is extremely powerful and many of your colleagues here in the department swear by it. It can be downloaded here: http://www.rstudio.com/. You should definitely check out the "screencast" which lays out RStudio's capabilities: http://www.rstudio.com/ide/.

   → Verdict: If you don't have to do things your way or you like the look and feel of Stata, this is probably the best choice! We will not use an IDE for the purposes of this tutorial.

7. **Sublime Text 2**: This is by far the best looking text editor in this list. It is cross-platform but will cost you $50 for a license to disable rare pop-ups of the evaluation version. It is more versatile and has more features than say Notepad++. It has all the usual features of any good text editor, plus some nice stuff others do not. It is not specific to R. So you can write and compile your LaTeX documents right here. Then spawn an R-session right in the editor (via

[8] An alternative, not specific to R is Eclipse IDE found here: http://www.eclipse.org/

the SublimeREPL plug-in) while working on your Python or C++ code, etc. Think of it as the Emacs for dummies. You can get the editor here: http://www.sublimetext.com/. The SublimeREPL plug-in can be found here: https://github.com/wuub/SublimeREPL.

→ Verdict: Setting it up and customizing it to your needs will take some time (e.g.installing the package manager and the R and LaTeX plug-ins, etc). Once you have it running, it is the last editor you will ever use.

1.2.5 References & Help

You are not the first folks to learn R. Any problem you will encounter and any question you may have related to R over the next year or so will have been encountered or asked by somebody else.That's a good thing. It means that you will be able to get help. There exist many tutorials, papers, and books on how to use R and how to apply R to all sorts of problems. We will cover how to use and access R's internal help and reference features later in this tutorial. Below is a list of some reference materials I have found useful.

On the Web

1. Google it. In 99.9% of the cases you will find a solution to an issue or answers on how to do something with a quick Google search.

2. The official R manuals: Go to http://www.r-project.org/ and click on Manuals under the Documentation heading. You will find introductory materials and advanced tutorials on how to create your own packages.

3. *An Introduction to R* by Venables and Smith (2012) can be downloaded here: http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf.

4. *simpleR* by Verzani (2002) can be downloaded here: http://www.math.csi.cuny.edu/Statistics/R/simpleR/printable/simpleR.pdf.

5. The RGraph Gallery is a great source for help and code for making graphs: http://gallery.r-enthusiasts.com/.

Books

1. *The R Book* by Crawley (2007) should be available in The Star Lab.

2. *ggplot2: Elegant Graphics for Data Analysis* by Wickham (2009) is an excellent source for getting started with ggplot.

1.3 Almost there

Actually, one last thing before we get started. It is important to note that R is a compiled or interpreted language. This means that all commands and code you give R (either by sending code via an editor or by typing it directly into the console) will be executed immediately. Unlike in uncompiled languages (C, Fortran, etc, . . .) you are not required to build a complete program and instead R will intepret things for you. Throughout the tutorial you will see examples of R-input and R-output like so:

```
> print("Hello world!")
[1] "Hello world!"
```

When we reach these examples, you should input the R-code from the numbered lines into the R-console at your workstation. For the above example you should have typed: `print(''Hello world!'')` and then pressed enter. Whenever the output R spits back at you does not match what's in this tutorial, let me know and we shall hopefully figure things out.

Chapter 2

The Very Basics of the R Interpreter

OK, the computer is fired up. We have **R** installed. It is time to get started.

1. Start **R** by double-clicking on the **R** desktop icon.
2. Alternatively, open the terminal/console, and navigate to the directory in which **R** is installed: For example, go to `C:\Programs\R-2.15.2\bin` and type `R.exe` and hit enter.[1]

2.1 R as a Calculator

Depending on how you started **R**, you should see either the **R**-console or an **R** session should have spawned at the command line. Either way, you should see something like this:

```
R version 2.15.2 (2012-10-26) -- "Trick or Treat"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

[1] If **R** is on your system's `PATH` you can skip the navigation step.

Note the > at the bottom. Whenever you see this symbol, it means that R is not doing anything and just waiting for your input. It's called the prompt. Let's break the rule about text editors and type directly into the console.

2.1.1 Basic Arithmetic

```
> 1 + 2
[1] 3
```

When R is running it stores all variables, data, functions, and results in the active memory of your computer. In the example above, R knows 1 and 2, as well as the basic *operator* "+". It now creates an *object* in the active memory containing the results of the computation and implicitly executes a function to print the content of the results *object* to the screen. R let's you know that what it is displaying in the square brackets [ ]. In our case this *object* contains only one value and R tells you that it has printed the first and only value of our results.

You can scroll through previous commands you've entered by using the up (↑) and down (↓) keys on your keyboard. Often instead of giving you the prompt again, R will instead display a "+" (aka the continuation line). This means you entered an incomplete command and that R is waiting for more input. (Note R is not reporting the [ ], as it is not displaying any content from an object stored in memory.)

```
> 1 +
+
```

If you are not sure what's going on, and why R is asking for additional input, or what input it wants, you can just hit `Esc` or `Ctrl-C`. It will tell R to forget it and bring back the prompt. Of course we know that R wants another summand in this case. So let's give R what it wants.

```
> 1 +
+ 2
[1] 3
```

Same problem here. R is waiting for you to close the bracket.

```
> 7 / (1 + 3
+ )
[1] 1.75
```

R understands the following basic operators:

1. + and − for addition and subtraction
2. ∗ and / for multiplication and division
3. ∧ for exponents
4. %% is the modulo operator
5. %\% for integer division

R observes standard rules of operator precedence but you can use brakets if you don't remember those lessons your elementary school math teacher tried to hammer home.

This,

```
> 7 / (1 + 3)
[1] 1.75
```

is not the same as this:

```
> 7 / 1 + 3
[1] 10
```

2.1.2 Comments and Spacing

Comments

Another important operator is the comment operator #. Whenever R encounters this operator, it will ignore everything printed after it (in the current line). As can be inferred from its name, this is extremely useful to annotate your code.[2]

```
> 1 + 2 + 3 # Here R does some serious Maths.
[1] 6
```

[2] You can never use this functionality enough. Always annotate your code. Say you are writing a program to implement some non-standard estimator. The code works. Your results are nice and you submit them as a paper. You wait six months for the reviewers of your favorite journal to get back to you. When they do they mention that you should compute robust standard errors. You open up the code for your program but don't remember what you did and what any of it means. If you had annotated each line or section, you would probably be able to make sense of it all. Long story short. Annotate, annotate, annotate.

Be careful.

```
> 1 + 2 # + 20
[1] 3
```

Misplaced comments can break your code.

```
> 1 + # 2
+
```

Spacing

For the most part, R does not care about spacing. This:

```
> 1 + 2
[1] 3
```

produces the same result as this:

```
> 1    +          2
[1] 3
```

Spaces of course matter when you are dealing with character strings. This:

```
> print("Strings obey spacing.")
[1] "Strings obey spacing."
```

is clearly a different string than this:

```
> print(" Strings   obey       spacing  .   ")
[1] " Strings   obey       spacing  .   "
```

Semi-Colons

There is another special character, the semi-colon. The semi-colon is an important part of R's control structure. We have seen that R evaluates code line by line. A linebreak tells R that a statement is to be evaluated. Instead of a linebreak, you can use a semicolon to tell R where statements end.

```
> print("HELLO")
[1] "HELLO"

> print("HELLO") print("WORLD")
Error: unexpected symbol in "print("HELLO") print"

> print("HELLO"); print("WORLD")
[1] "HELLO"
[1] "WORLD"
```

2.1.3 Basic Functions

R comes with a slew of pre-installed functions. These functions are installed as part of the `base` package which is located in your `\library` directory. R treats all functions like *objects*. All functions have names and take arguments in parentheses: `function(...)`.

Let's consider the `print()` function. This function simply asks R to print *objects* to the screen. We can ask R to print known *objects*.

```
> print(1)
[1] 1
```

We can tell R to print any object, including a character string. Character strings are enclosed by quotation marks.

```
> print("We need more coffee!")
[1] "We need more coffee!"
```

Importantly, we can ask R to print named functions. Consider the function `exp()`. The function `exp()` computes the exponential function. Let's ask R to print it by using the function name as the argument for `print()`.

```
> print(exp)
function (x)  .Primitive("exp")
```

What R is telling you here is that `exp()` is a known function which takes one arguments `x`. (It also tells you what type or class of *object* `exp` is. The type here is .Primitive, which is a basic R function.) Let's try it.

```
> exp(x = 1)
[1] 2.718282
```

Let's ask R to print another function, say `log()` which computes logarithms. Here R reports that the function takes two arguments, separated by a comma. It needs and `x` and you can specify the `base`. If you do not specify the second argument, R will default to `base = exp(1)` (i.e. the natural logarithm).

```
> print(log)
function (x, base = exp(1))  .Primitive("log")
```

Ok ... let's try.

```
> log(x = 10, base = exp(1))
[1] 2.302585
```

Or, a log with base 10 . . .

```
> log(x = 10, base = 10)
[1] 1
```

R is pretty smart. You do not need to tell R all the specifics. Whenever you use a function, R knows what arguments can be supplied to the function. So you can simplify things, like this:

```
> log(10)
[1] 2.302585
```

R knows that the first argument you supply is the `x` and since you did not add anything else to the function, it returns the default logarithm (i.e. `base = exp(1)`)

You can do the same for the second argument. R knows that the second argument has to be for the `base`.

```
> log(10, 10)
[1] 1
```

Below is an excerpt of some of the basic mathematical functions R knows.

- `print()` – prints objects
- `log()` – computes logarithms
- `exp()` – computes the exponential function
- `sqrt()` – takes the square root
- `abs()` – returns the absolute value
- `sin()` – returns the sine
- `cos()` – returns the cosine
- `tan()` – returns the tangent
- `asin()` – returns the arc-sine
- `factorial()` – returns the factorial
- `sign()` – returns the sign (negative or positive)
- `round()` – rounds the input to the desired digit
- etc, etc, ...

2.1.4 Logical Operators

Among the most used features of R are logical operators. You will use these throughout your code and they are crucial for all sorts of data manipulation. When R evaluates statements containing logical operators it will return either `TRUE` or `FALSE`. Below is a list of most of them.

1.	<	less than
2.	<=	less than or equal to
3.	>	greater than
4.	>=	greater than or equal to
5.	==	equal
6.	!=	not equal
7.	&	and
8.	|	or

Let's try them out:

```
> 1 == 1
[1] TRUE

> 1 == 2
[1] FALSE

> 1 != 2
[1] TRUE

> 1 <= 2 & 1 <= 3
[1] TRUE

> 1 == 1 | 1 == 2
[1] TRUE

> 1 > 1 | 1 > 2 & 3 == 3
[1] FALSE

> 1 > 1 & 1 > 2 & 1 > 3
[1] FALSE
```

2.1.5 R's Help Function

At this point it should be obvious how to use R as a calculator. In this section, I will outline how you can ask R for help. The first thing to do if you have a question about one of R's functions is to ask R. This is important and you will use this functionality a lot.

Known Functions

Recall our example of computing a base 10 logarithm by calling `log()` with the argument `base = 10`? Assume we didn't know that `log()` had an argument called `base`, how could we have found out? We can simply pull up a help page for the `log()` function, like this:

```
> ?log
>
```

A help page will appear with the following sections (there may be others):

→ **Description**: purpose of the function

→ **Usage**: an example of a typical implementation

→ **Arguments**: a list of the arguments you can supply to the function and what each does

→ **Details**: more detailed information about the function and its arguments

→ **Value**: information about the likely output of a function (e.g. does the function return an integer, or a list, or a matrix, or something different)

→ **See Also**: a list of useful related functions

→ **References**: citations which can often be very useful

→ **Examples**: example code

Using the `?` command is generally a good start. And more likely than not, it will provide you with the answer you need. In many instances, however, the help is not helpful and it appears it was written in code. R programmers are not always good at explaining things in plain English. A good example is the help page for the straightforward function `sort()`. It is incomprehensible. In such cases, it may be useful to use the `example()` function, which displays the examples from the help page. These may or may not be helpful.

For some functions, especially basic operators, `?` may not work. In those cases you can use the `help()` function:

```
> help("+")
>
```

Unknown Functions

It maybe the case that you know what you want to do but you don't know how to do it in R. Say you want to estimate a logit but don't know how. Instead of using the `?`, you can use two `??`. This will initiate the help search page, and search the documentation for packages you have installed for that specific keyword. (This is the same as: `help.search("logit")`.)

```
> ??logit
>
```

An alternative is to use the `appropos()` function. It will return a list of all functions known to R containing the search term (e.g., "mean").

```
> apropos(mean)
 [1] ".colMeans" ".rowMeans"     "colMeans"  "kmeans"  "mean"
 [2] "rowMeans"  "weighted.mean" "mean.Date"
```

You can now use `?` to find out more.

```
> ?rowMeans
>
```

Comprehensive Help

If you cannot find your answer with the above methods, the best place to probe further is the overall R help feature via the `help.start()` function. Just type:

```
> help.start()
>
```

This opens a comprehensive helping of R documentation, manuals, and help for all installed packages in your web-browser. This maybe the best place to look but if not, it is time to exploit Google.com, or ask your friendly collegues, or The Star Lab fellow. Also, see Section 1.2.5 above.

2.2 Packages and Libraries

It may very well be that after all the searching for some functionality, you come up empty handed. For example, you desperately want to estimate an ordered probit. R does not know how to do that out of the box. You could program the log-likelihood yourself and estimate it by writing your own maximization routine. Or you could just install a package that has this feature built in.

2.2.1 Installing and Loading Packages

It turns out the ability to estimate ordered logistic or probit regression is included in the `MASS` package. To install this package you run the following command:

```
> install.packages("MASS")
```

You will be asked to pick a CRAN mirror from which to download (generally the closer the faster) and R will install the package to your library. R will still be clueless. To actually tell R to use the new package you have to tell R to load the package's library each time you start an R session, just like so:

```
> library("MASS")
>
```

R now knows all the functions that are canned in the `MASS` package. To see what functions are implemented in the `MASS` package, type:

```
> library(help = "MASS")
>
```

A list of functions will now be displayed and you can see that the function to estimate an ordered probit is polr(). You now can get help the normal way:

```
> ?polr
>
```

2.2.2 Maintaining your Library

Packages are frequently updated. Depending on the developer this could happen very often. To keep your packages updated enter this every once in a while:

```
> update.packages()
```

As mentioned above, R itself will be updated frequently. Unfortunately, this process is not well implemented and generally requires you to install a fresh copy of R and removing the old installation. Unless you specify and maintain a separate directory for where R can find your packages, the removal of the old R installation will generally also remove all your packages.

This is not a problem especially given the quality of CRAN's mirrors. Before deleting your old copy of R, simply check which packages you had previously installed (especially those you need in order for your code to run). To see the list of all installed packages, type:

```
> library()
```

Just save the names of the packages and after starting your new R installation install them all at once again like this:

```
> install.packages("MASS", "myPackageX", "myPackageY", "
   myPackageZ")
```

Chapter 3

The Building Blocks

Up to this point we have hopefully learned how to use R as a basic calculator and you know how to do some basic arithmetic, use basic functions, and access R's help functionality. To move beyond using R as a calculator this chapter will introduce the main building blocks of R – *objects* and their *modes*. The discussion that follows is not technically correct (in fact it is a gross mischaracterization) but it should help make sense of things and why things in R happen the way they do.

From here on out, you are no longer allowed type directly into the R Console!

3.1 Objects

R is an object oriented language. As I mentioned in passing above everything in R is an *object*. When R does anything, it creates and manipulates *objects*. R's *objects* come in different types and flavors. The most basic ones are:

→ **Vectors:** These are one-dimensional sequences of elements of the same *mode*. (More on *modes* later: see section 3.2.) For example, this could be vector of length 26 (i.e. one containing 26 elements) where each element is a letter in the alphabet.

→ **Matrices & Arrays:** These are two dimensional rectangular objects (matrices) and higher-dimensional rectangular objects (arrays). All elements of matrices or arrays have to be of the same mode.

→ **Lists:** Lists are like vectors but they do not have to contain elements of the same mode. The first element of a list could be a vector of the 26 letters of the alphabet. The second element could contain a vector of all the prime numbers below 1000. A third could be a 2 by 7 matrix.

→ **Data Frames:** Data frames are best understood as special matrices (technically they are a type of list). For most applications involving datasets you will use data frames. They are two dimensional containers with rows corresponding to 'observations' and columns corresponding to 'variables.'

→ **Factors:** Factors are vectors to classify categorical data. They behave differently than vectors containing numerical, integer, or character elements.

→ **Functions:** Functions are *objects* that take other *objects* as inputs and return some new *object*. We will deal with functions separately in a later chapter.

3.2 Modes

All objects have a certain *mode*. Some objects can only deal with one *mode* at a time, others can store elements of multiple *modes*. R distinguishes the following modes:

1. **integer**: integers (e.g. 1, 2 or -69)
2. **numeric**: real numbers (e.g 2.336, -0.35)
3. **complex**: complex or imaginary numbers
4. **character**: elements made up of text-strings (e.g. "text", "Hello World!", or "123")
5. **logical**: data containing logical constants (i.e. `TRUE` and `FALSE`)

3.3 Assignment and Reference

Knowing the types of objects R can work with is not terribly useful without knowing how to store these objects and without knowing how to recall or reference them when needed. You often will compute some statistic or manipulate some matrix. Instead of recalculating everything over and over again we can give things names and recall them later. Below we will cover how to create, assign to and refer to various *objects*. We shall use vectors as examples.

3.3.1 Playing with trivial Vectors

Recall our basic arithmetic examples from above. We implicitly relied on and then manipulated objects and R implicitly printed these objects to the screen.

```
> 1 + 2
[1] 3
```

Let's assign and recall names instead. We can do that by using the assignment operator "<-". Think of this as the M+ button on your calculator.

```
> Answer <- 1 + 2
>
```

We can use just about any name we like so long as it is not a number or does not start with a number (e.g. `3 <- 1 + 2` will not work, neither will `3Answer <- 1 + 2`). It is very useful to use descriptive names such as `NumberOfStudents <- 17` instead of `n <- 17`. Don't confuse yourself.

As you can see in the example above. R no longer gives you the answer to our problem. It just returns the prompt. Luckily you are familiar with R's `print()` function and you can recall or print the results to the screen.

```
> print(Answer)
[1] 3
```

If you give R the name of some object it knows you don't even have to use the `print()` function. Just type in the name and R will do it's thing.

```
> Answer
[1] 3
```

Whether you know it or not you have now already created an *object* of the vector type (of length 1). We can verify this with the `is()` function. When supplied with the name of an *object*, this function will tell you what type of *object* we have as well as its *mode*.

```
> is(Answer)
[1] "numeric" "vector"
```

Recall: 1, 2, 3, or 16 are internal objects. Try it!

```
> is(3)
[1] "numeric" "vector"
```

Named *objects* behave just like the ones R already knows. This is pretty useful:

```
> Answer * 2
[1] 6
```

Or . . .

```
> Answer2 <- Answer * sqrt(Answer)
> Answer2
[1] 5.196152
```

Keeping Track of Objects

To see what objects you have created (the ones R stored in active memory) you can use the `ls()` function.

```
> ls()
[1] "Answer"  "Answer2"
```

If you want to remove an *object* from memory use the `rm()` function. Be very careful. This will delete thing permanently. Don't delete things you need.

```
> rm(Answer2)
> ls()
[1] "Answer"
```

If you want to remove all *objects* from active memory this will do the trick:

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

3.3.2 *Real* Vectors

So far we have only created trivial vectors of length 1. Let's assign some longer ones. To do this you will use the `c()` function. The "c" stands for concatenate, and you can string a bunch of elements together, separated by commas.

```
> Vector1 <- c(1,2,3,4,5,6,7,8,9,10)
> Vector1
 [1]  1  2  3  4  5  6  7  8  9 10
```

How about a character vector?

```
> Vector2 <- c("a", "b", "c", "d")
> Vector2
[1] "a" "b" "c" "d"
```

Or ...

```
> Vector3 <- c("1", "2", "3", "4")
> Vector3
[1] "1" "2" "3" "4"
```

You can also string multiple vectors together with the `c()` function.

```
> Vector4 <- c(Vector2, Vector3, Vector2, Vector2, Vector2)
> Vector4
 [1] "a" "b" "c" "d" "1" "2" "3" "4" "a" "b" "c" "d" "a"
[14] "b" "c" "d" "a" "b" "c" "d"
```

Vector Operations

Most standard mathematical functions work with vectors.

```
> Vector1 + Vector1
 [1]  2  4  6  8 10 12 14 16 18 20
```

```
> Vector1 / Vector1
[1] 1 1 1 1 1 1 1 1 1 1
```

```
> log(Vector1)
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
[6] 1.7917595 1.9459101 2.0794415 2.1972246 2.3025851
```

Here we are nesting the `log()` function inside the `round()` function.

```
> round(log(Vector1))
[1] 0 1 1 1 2 2 2 2 2 2
```

The `round()` function takes an argument (`digit`) to specify how many decimals to display. It defaults to 0. Let's see a few more digits.

```
> round(log(Vector1), digit = 3)
 [1] 0.000 0.693 1.099 1.386 1.609 1.792 1.946 2.079 2.197
[10] 2.303
```

To do other useful things to vectors consider these functions:

Function	Description
`sum()`	sums of the elements of the vector
`prod()`	product of the elements of the vector
`min()`	minimum of the elements of the vector
`max()`	maximum of the elements of the vector
`mean()`	mean of the elements
`median()`	median of the elements
`range()`	the range of the vector
`sd()`	the standard deviation
`var()`	the variance (on `n-1`)
`cov()`	the covariance (takes two inputs `cov(x,y)`)
`cor()`	the correlation coefficient (takes two imputs cor(x,y))
`sort()`	sorts the vector (argument: `decreasing = FALSE`)
`length()`	returns the length of the vector
`summary()`	returns summary statistics
`which()`	returns the index after evaluating a logical statement
`unique()`	returns a vector of all the unique elements of the input

```
> sum(Vector1)
[1] 55

> prod(Vector1)
[1] 3628800

> median(Vector1)
[1] 5.5

> sd(Vector1)
[1] 3.02765

> sort(Vector1, decreasing = TRUE)
[1] 10  9  8  7  6  5  4  3  2  1

> length(Vector1)
[1] 10

> summary(Vector1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.25    5.50    5.50    7.75   10.00

>which(Vector1 >= 5) # note this returns the index not the
   elements (try it with Vector2)
[1]  5  6  7  8  9 10
```

Simplifying Vector Creation

Most of the time using the c() function will be tedious as you don't want to manually type all elements of a vector. Luckily the good folks responsible for R have thought of you.

You can use the colon to tell R to create an integer vector.

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

Or the seq() function, which is more general and has some neat features.

```
> seq(from = 0, to = 10) # you can drop the argment names
[1]  0  1  2  3  4  5  6  7  8  9 10

> seq(0, 10)
[1]  0  1  2  3  4  5  6  7  8  9 10

> seq(0, 10, by = 2) # the 'by' argument let's you set the
                    # increments
[1]  0  2  4  6  8 10

> seq(0, 10, length.out = 25) # the 'length.out' argument
                             # specifies the length of the
                             # vector and R figures out the
                             # increments itself

 [1]  0.0000000  0.4166667  0.8333333  1.2500000  1.6666667
 [6]  2.0833333  2.5000000  2.9166667  3.3333333  3.7500000
[11]  4.1666667  4.5833333  5.0000000  5.4166667  5.8333333
[16]  6.2500000  6.6666667  7.0833333  7.5000000  7.9166667
[21]  8.3333333  8.7500000  9.1666667  9.5833333 10.0000000
```

Another useful function is `rep()` which allows you to repeat things.

```
> rep(0, time = 10)
[1] 0 0 0 0 0 0 0 0 0 0

> rep("Hello", 3) # as always you can drop the argument name
[1] "Hello" "Hello" "Hello"

> rep(Vector1, 2) #  repeating Vector 1 twice
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9
[20] 10

> rep(Vector2, each = 2) # we can repeat each element as well

[1] "a" "a" "b" "b" "c" "c" "d" "d"
```

Indexing

Sometimes you do not want to print or manipulate an entire vector. This is where indexing comes in. Indexing vectors is done with [ ]. Check it out.

```
> Vector6 <- c("The", "Starlab", "Fellow", "is", "a Fool.")
> Vector6
[1] "The" "Starlab" "Fellow" "is" "a Fool"

> length(Vector6) # how long is Vector6
[1] 5

> Vector6[3] # with the bracket we reference the third
   element
[1] "Fellow"

> Vector6[2:4] # we can reference a sequence of elements
[1] "Starlab" "Fellow"  "is"

> Vector6[c(1,3,4)] #  or any elements we like
[1] "The" "Fellow" "is"

> Vector6[-2] #  all except the 2nd element
[1] "The" "Fellow" "is" "a Fool."

> Vector6[5] <- "great." # and we can change elements
> Vector6
[1] "The" "Starlab" "Fellow" "is" "great."
```

Logical operators come in handy when indexing:

```
> Vector7 <- c(1, 1, 2, 3, 4, 4.5, 6, 6, 10)
> Vector7
[1]  1.0  1.0  2.0  3.0  4.0  4.5  6.0  6.0 10.0

> Vector7[Vector7 == 1]
[1] 1 1

> Vector7[Vector7 >= 4]
[1]  4.0  4.5  6.0  6.0 10.0

> Vector7[Vector7 != sqrt(16) & Vector7 > 2]
[1]  3.0  4.5  6.0  6.0 10.0
```

More Functions

Consider the following three functions: na.omit(), subset(), and sample(). This will become very useful later when dealing with real data. Let's make a new vector called foo:

```
> foo <- c(2, 3, 4, 3, NA, NA, 6, 6, 10, 11, 2, NA, 4, 3)
> foo
[1]  2  3  4  3 NA NA  6  6 10 11  2 NA  4  3

> max(foo) # this won't work because many function can't deal with NAs
[1] NA

> summary(foo) # this works
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  2.000   3.000   4.000   4.909   6.000  11.000       3
```

This is where the na.omit() function comes in. This function returns the vector supressing the NAs and adds an attribute to it called na.action.

```
> na.omit(foo)
[1]  2  3  4  3  6  6 10 11  2  4  3
attr(,"na.action")
[1]  5  6 12
attr(,"class")
[1] "omit"
```

This is helpful because now we can compute all those functions that break when they encounter NAs. Instead of supplying the object foo we can supply the object returned by na.omit().

```
> max(na.omit(foo))
[1] 11
```

The summary() function is useful check whether NAs are present in your object. The is.na() function is more powerful. Combined with the subset() function we can remove the NAs manually. This requires you to write a logical statement. The first argument you need to supply is the object you want to subset. The second should be the logical statement R should evaluate.

```
>is.na(foo)
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
[9] FALSE FALSE FALSE  TRUE FALSE FALSE

> foo.noNA <- subset(foo, is.na(foo)==FALSE)
> foo.noNA
[1]  2  3  4  3  6  6 10 11  2  4  3
```

Of course the subset() function can be used for more than NA removal. Let's use it to find numbers divisible by 7.

```
> X <- 1:500 # creating a vector from 1 to 500
> Multiple7 <- subset(X, X%%7==0) # recall the modulo operator
> Multiple7
 [1]   7  14  21  28  35  42  49  56  63  70  77  84
[13]  91  98 105 112 119 126 133 140 147 154 161 168
[25] 175 182 189 196 203 210 217 224 231 238 245 252
[37] 259 266 273 280 287 294 301 308 315 322 329 336
[49] 343 350 357 364 371 378 385 392 399 406 413 420
[61] 427 434 441 448 455 462 469 476 483 490 497
```

The sample() function will also come in handy later. It takes the following arguments: size for the sample size, and replace = TRUE for whether you want to sample with or without replacement. Let's sample from our vector, Multiple7. Obviously, your output may/will look different than what I got here.

```
> sample(Multiple7, size = 10, replace = FALSE)
[1] 497 238 322  63  77 245 455 126 490 392
```

The print(), cat(), and paste() Functions

We already know that the print() function prints an object to the screen by explicitly creating an object in the computers active memory. The paste() function is a bit more useful as you can paste multiple objects together and print them to the screen (by creating an implicit object - a character vector). The cat() function does the same thing but it does not create an object in the computer's active memory.

```
> print(0.2)
 [1] 0.2

> X <- 0.2
> print(X)
 [1] 0.2

> paste(X, "is equal to", X)
 [1] "0.2 is equal to 0.2"

> cat(X, "is equal to", X) # notice the missing [1] below
0.2 is equal to 0.2 >
```

Chapter 4

Matrices

Most all statistical techniques for multi-variate data analysis require some matrix algebra. This chapter covers R's matrix algebra features. Before we get to matrices, I will show you how to save and load your objects and code.

4.1 Maintaining Code & Objects

In the previous chapter we began using a text editor to keep track of our code. Whenever you use R for data analysis you will want to save your code in order to replicate your analysis. Open Notepad and write something like the following:

```
############################
# DATE: 12/29/2012         #
# AUTHOR: Peter Haschke    #
# INFO: test script for R  #
############################

# delete all objects from active memory

rm(list = ls())

# Create two vectors

X<-c(1,2,3,4,5)
Y<-c(5,4,3,2,1)

# Do some maths

Z <- X - Y
MEAN.Z <- mean(Z)
MIN.Z <- min(Z)     # This finds the smallest element
MAX.Z <- max(Z)     # This finds the largest element

# This concludes my test script for R
```

4.1.1 Scripting

All code should be saved with the `.r` or `.R` extension. So click 'File' and then 'Save as...' and save the file onto your Z-Drive as `"Z:\test.R"`. Now open the R console and type the following and press enter:

```
> source("z:/test.R")
>
```

If all went well, R just executed your entire script. All the code you wrote has been evaluated and you can access all the objects you created. It's magic, I know.

```
> ls() # verify if all the objects we created are there
[1] "MAX.Z"  "MEAN.Z" "MIN.Z"  "X"      "Y"      "Z"

> Z
[1] -4 -2  0  2  4
```

Using the `source()` function like this can be tedious especially while you are working on a program or some code. In those cases it may be more efficient to use the console interactively, by copying or pasting from your editor (or by using an editor that can directly communicate with R). Writing properly source-able files, however, is really important for sharing code, and writing programs that can be replicated on any computer and by anyone. I expect all problem sets to be sent to me in such a self-contained format.

4.1.2 Saving and Loading Objects

Whenever you are writing programs that take a while to execute (e.g. you are inverting some crazy matrix, etc), saving your code as a source-able script only is not a good idea. Luckily, you can save and load output or results. The easiest way to do this is via the `save()` and `load()` functions. Objects are saved in the `.Rdata` format via the `file` argument.

```
> save(X, Y, Z, MEAN.Z, file = "z:/results.Rdata")
>
```

Let's see if it works. Of course it does.

```
> rm(list = ls()) # remove all objects in active memory
> ls() # check if they are really gone
character(0)

>  # Above: R telling you it found no named objects
> load("z:/results.Rdata")
> ls()
[1] "MEAN.Z" "X" "Y" "Z"
```

4.2 Creating Matrices

To create matrices we will use the matrix() function. The matrix() function takes the following arguments:

- data   an R object (this could be a vector)
- nrow   the desired number of rows
- ncol   the desired number of columns
- byrow   a logical statement to populate the matrix by either row or by column

Example 1:
A 3 by 3 matrix filled with 1s

```
> Matrix1 <- matrix(data = 1, nrow = 3, ncol = 3)
> Matrix1
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Example 2:
As always we can drop the argument names (as long as you remember that the first argument asks for the data, the second for the rows, and the third for the columns). Let's create a rectangular matrix (3 by 7) and fill it with NA's. NA is another useful special object existing in R. We have already seen TRUE and FALSE. NA is a kind of a special zero and most computations involving NA return NA (e.g. NA + 1 evaluates to NA, and NA == TRUE returns NA). Also, say hello to the dim() function.

```
> Matrix2 <- matrix(NA, 3, 7)
> Matrix2
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   NA   NA   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA   NA   NA
[3,]   NA   NA   NA   NA   NA   NA   NA

> dim(Matrix2) # this returns the dimensions of the matrix
[1] 3 7
```

Example 3:
Let's fill a matrix with a vector of values

```
> Vector8 <- 1:12
> Vector8
[1]  1  2  3  4  5  6  7  8  9 10 11 12

> Matrix3 <- matrix(data = Vector8, nrow = 4)
> Matrix3 # by default the matrix will be populated by column
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

> Matrix4 <- matrix(data = Vector8, nrow = 4, byrow = TRUE)
> Matrix4 # now we populated it by row
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```

Example 4:
You can also create matrices by pasting together vectors using the `rbind()` and `cbind()` functions.

```
> Vector9 <- 1:10
> Vector9
[1]  1  2  3  4  5  6  7  8  9 10

> Vector10 <- Vector9 ^ 2
> Vector10
 [1]   1   4   9  16  25  36  49  64  81 100

> Matrix5 <- rbind(Vector9, Vector10)
> Matrix5
         [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
Vector9     1    2    3    4    5    6    7    8    9    10
Vector10    1    4    9   16   25   36   49   64   81   100

> dim(Matrix5)
[1]  2 10
```

And `cbind()` . . .

```
> Matrix6 <- cbind(Vector9, Vector10, Vector9)
> Matrix6
      Vector9 Vector10 Vector9
 [1,]       1        1       1
 [2,]       2        4       2
 [3,]       3        9       3
 [4,]       4       16       4
 [5,]       5       25       5
 [6,]       6       36       6
 [7,]       7       49       7
 [8,]       8       64       8
 [9,]       9       81       9
[10,]      10      100      10


> dim(Matrix6)
[1]  10 3
```

As you can see, the `rbind()` and `cbind()` functions automatically label row or column names. You can use the `rownames()` and `colnames()` functions to manipulate these.

```
> colnames(Matrix6)
[1] "Vector9"  "Vector10" "Vector9"

> rownames(Matrix6) # the rownames do not exist
NULL

> colnames(Matrix6) <- c("A", "B", "C")
> rownames(Matrix6) <- c("a","b","c","d","e","f","g","h","i",
   "j")

> Matrix6
   A   B  C
a  1   1  1
b  2   4  2
c  3   9  3
d  4  16  4
e  5  25  5
f  6  36  6
g  7  49  7
h  8  64  8
i  9  81  9
j 10 100 10
```

The diag() function is useful for creating the identity matrix

```
> Matrix7 <- diag(5) # creates a 5 by 5 identity matrix
> Matrix7
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1

> Vector11 <- c(1, 2, 3, 4, 5)
> Matrix8 <- diag(Vector11) # Vector11 across the diagonal
> Matrix8
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5

>  diag(Matrix7) # extracts the diagonal from the matrix
[1] 1 1 1 1 1
```

4.2.1 Indexing Matrices

It should be kind of obvious how indexing works with matrices from looking at the output **R** generates and knowing that vectors are indexed via [ ]. Using [i,j] will retrieve the jth element of the ith row.

```
> Matrix9 <- matrix(1:9, 3)
> Matrix9
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> Matrix9[1,1] # extracts the first element of the first row
[1] 1

> Matrix9[2,3] # extracts the third element of the second row

[1] 8
```

You can also extract entire rows as vectors by leaving the column entry blank, and vice versa.

```
> Matrix9
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> Matrix9[ ,1] # extracts the first column
[1] 1 2 3

> Matrix9[2, ] # extracts the second row
[1] 2 5 8

> Matrix9[1:2, ] # extracts the first and second row
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8

> Matrix9[Matrix9[ ,2] > 4, ] # extracts all rows that in
   their second column contain values greater than four
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
```

4.3 Mathematical Operations

R can do matrix arithmetic. Below is a list of some basic operations we can do.

→	`+ - * /`	standard scalar or by element operations
→	`%*%`	matrix multiplication
→	`t()`	transpose
→	`solve()`	inverse
→	`det()`	determinant
→	`chol()`	cholesky decomposition
→	`eigen()`	eigenvalues and eigenvectors
→	`crossprod()`	cross product
→	`%x%`	kronecker product

4.3.1 Matrix Math-Examples

```
> X <- matrix(1:4, nrow = 2)
> X
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> Y <- diag(2)
> Y
     [,1] [,2]
[1,]    1    0
[2,]    0    1

> X * Y # by element multiplication
     [,1] [,2]
[1,]    1    0
[2,]    0    4

> X %*% Y # matrix multiplication
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> t(X) # transpose
     [,1] [,2]
[1,]    1    2
[2,]    3    4

> solve(X) # inverse
     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5

> sum(diag(X)) # trace
[1] 5

> det(X) # determinant
[1] 5-2
```

To compute eigenvalues and vectors you can use the eigen() function. Unlike most functions we have encountered so far, it does not return a vector or a matrix but a list.

```
> eigen(X)
$values
[1]  5.3722813 -0.3722813

$vectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736
```

As mentioned in before in Chapter 3.1, lists can be a collection of different types of *objects*. The eigen() function returns a list containing a vector and a matrix. You can extract named elements from a list with the $ symbol. Alternatively you can extract element with double [[ ]].

```
> Eigen.List <- eigen(X)
> names(Eigen.List) # prints the names of objects of the list
[1] "values"  "vectors"

> Eigen.List$vectors # returns the eigenvectors
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736

> Eigen.List$values # returns the eigenvalues
[1]  5.3722813 -0.3722813
```

4.3.2 Bonus Example

Last but not least, try this:

```
> Constant <- rep(1, times = 10)
> Variable <- c(1,2,3,1,2,3,5,6,7,8)
> X <- cbind(Constant, Variable)
> Y <- seq(1, 10)
> Beta.Hats <- solve(t(X)%*%X)%*%t(X)%*%Y
> colnames(Beta.Hats) <- "Estimate"
> Beta.Hats
         Estimate
Constant  1.34375
Variable  1.09375

> OMG!!!!
Error: unexpected '!' in "OMG!"
```

Chapter 5

Data Frames

Of the types of objects known to R, we have covered two in detail (vectors and matrices) and one in passing (lists). This chapter is devoted to the most flexible and arguably most used object type, data frames. Whenever you encounter datasets they will usually be stored in `data.frame` objects. In this chapter we will cover how to load and save datasets; how to manipulate, and edit them; and how to compute some data summaries.

5.1 Loading and Saving Datasets

Let's start with loading and saving datasets. The easiest way to load data is with the `data()` function. If entered without arguments, it will bring up a list of all datasets that come bundled with R.

```
> data()
>
```

Many R packages you may download also bundle their own datasets. To access all datasets known to R type:

```
> data(package = .packages(all.available = TRUE))
>
```

To load a package, simply add its name as the argument of the `data()`. Let's load a package that comes bundled with the ggplot2 package. To load it, we first need to load the ggplot2 library.

```
> rm(list = ls()) # remove all objects in active memory
> ls()
character(0)
> library(ggplot2)
> data(diamonds)
> ls()
[1] "diamonds"
```

We can verify that it is indeed a data frame with the `class()` function:

```
> class(diamonds)
[1] "data.frame"
```

As with lists, we can find names of components with the `names()` function:

```
> names(diamonds)
[1] "carat"   "cut"     "color"   "clarity" "depth"
[6] "table"   "price"   "x"       "y"       "z"
```

. . . and some other characteristics – `dim()`, `nrow()`, and `ncol()`:

```
> dim(diamonds) # the dimensions of the data frame
[1] 53940    10

> nrow(diamonds) # the number of observations
[1] 53940

> ncol(diamonds) # the number of variables
[1] 10
```

To save a copy of the dataset to your hard drive you can use the `save()` function we discussed before:

```
> save(diamonds, file = "z:/diamonds.Rdata")
>
```

5.1.1 Other Formats

In virtually all cases, you will not find the data you want or need bundled in R. Frequently, you will have created or downloaded data in some other program (e.g. Excel, or Stata, etc). Below we will go over loading and saving datasets with different formats and from various sources.

Comma Separated Values

The easiest way to import data is from files in the comma separated values `.csv` format.If you have created such a file, for example using Excel or a text editor, you can load it with the `read.csv()` function. A typical `.csv` file will look something like this

```
"Ozone","Solar.R","Wind","Temp","Month","Day"
41,190,7.4,67,5,1
36,118,8,72,5,2
12,149,12.6,74,5,3
18,313,11.5,62,5,4
NA,NA,14.3,56,5,5
28,NA,14.9,66,5,6
23,299,8.6,65,5,7
19,99,13.8,59,5,8
...
```

If you have a file in the `.csv` format you can load it into **R** via the `read.csv()` function. The `read.csv()` function takes the following arguments:

→ `file = " "` to specify the file name and location. this can be on your hard drive or a website

→ `sep = ","` to specify the character used as the delimiter (the default is a comma)

→ `header = TRUE` to specify whether your data has a variable names (the default is yes)

→ `quote = "\""` to specify what quotes look like in your dataset (the default is ")[1]

→ `dec = "."` to specify what decimals look like (the default is `.`)[2]

Let's load a dataset from my website. Since the dataset is formated nicely, we can just use the defaults (i.e. we don't have to specify all the above arguments).

```
> FE2013 <- read.csv(file = "http://www.peterhaschke.com/
    Teaching/R-Course/FE2013.csv")
>
```

[1] You may be confused by the `"\""` notation here. Whenever **R** encounters a delimiter of any kind (e.g. `{`, `(`, `[`, or `"`) It will wait for its counterpart to complete the operation. So if we had typed `quote = """`, **R** would be waiting for you to close the third quotation mark and produce an error. To tell **R** not treat the second `"` as a delimiter but a character, you will have to use **R**'s escape character, which happens to be \. We will come across the escape character again later on.

[2] This is useful when you are dealing with some European datasets that contain commas to instead of periods to identify decimals. Often they also use tabs, or semi-colons as delimiters. So pay attention to how your data is formatted.

You can also tell R to download the file for you so that you have a physical copy on your disk. You can then load it with the same function:

```
> download.file(url = "http://www.peterhaschke.com/Teaching/R-Course/FE2013.csv", destfile = "z:/FE2013.csv")
> FE2013 <- read.csv(file = "z:/FE2013.csv")
>
```

To save your dataset you can use the `save()` function we used before or the `write.csv()` function:

```
> save(FE2013, file = "z:/Dataset.Rdata")
> # or
> write.csv(FE2013, file = "z:/Dataset.csv")
>
```

Stata Data `.dta`

Importing datasets created in different formats is relatively straightforward. All you have to do is install the `foreign` package. This should be installed already in the The Star Lab but you will have to load it before being able to access its functions. The `foreign` package extends R's `read()` and `write()` functions. You now have access to `read.dta()` and `write.dta()` to read and write Stata files, and many others (e.g. `read.spss()`, etc).[3]

```
> library(foreign)
> Students <- read.dta("http://www.peterhaschke.com/Teaching/R-Course/Students.dta")
>
```

5.2 Manipulating Data Frames

Once you have your datasets loaded fun ensues. At this point we have three datasets in R's active memory, the diamonds dataset, the one on fuel economy and one on current Political Science Ph.D. students at the University of Rochester. Let's verify just to make sure:

```
> ls()
[1] "diamonds"  "FE2013"  "Students"
```

[3] Use the `help.start()` function to find out more about this package.

The first thing to note about data frames, is that it is usually not terribly helpful to print the object to the screen. Just for the heck of it, try it with the fuel economy data:

```
> FE2013
```

If you pressed enter, R will literally printed all the data to the screen. Unless you are dealing with tiny datasets containing only two or three variables, this is a waste of time and won't tell you anything. The best thing to do first is to use the names() and the dim() functions. This will tell you all the variable names of the data frame and give you some idea about the size of the dataset.

```
> names(FE2013)
 [1] "ModelYear"              "Manufacturer"
 [3] "Division"               "Model"
 [5] "Displacement"           "Cylinder"
 [7] "FEcity"                 "FEhighway"
 [9] "FEcombined"             "Guzzler"
[11] "AirAspiration1"         "AirAspiration2"
[13] "Gears"                  "LockupTorqueConverter"
[15] "DriveSystem1"           "DriveSystem2"
[17] "FuelType"               "FuelType2"
[19] "AnnualFuelCost"         "IntakeValvesPerCyl"
[21] "ExhaustValvesPerCyl"    "Class"
[23] "OilViscosity"           "StopStartSystem"
[25] "FErating"               "CityCO2"
[27] "HighwayCO2"             "CombinedCO2"

> dim(FE2013)
[1] 1082   28
```

And ...

```
> names(Students)
[1] "Name" "Year"

> dim(Students)
[1] 47  2
```

5.2.1 Extraction

Most of the basic extraction principles – namely the [ ] – we used for matrices also work for data frames. But you should remember that data frames are a special type of list and as such the $ will come in handy. For example, lets try to extract the variable called `"Gears"`. Unless you knew that `"Gears"` was the 13th variable you'd be trying around a bit. But either way works. Let's test it

```
> FE2013[,13]
   [1]  6  8  6  7  6  7  7  6  6  6  6  7  7  7  7  7  6  6  6  6  6  5  6  7  7
  [26]  7  7  6  7  7  7  7  7  7  5  5  6  6  6  6  6  6  6  6  6  6  6  6  7  6
  [51]  7  6  7  6  7  6  7  7  6  5  5  5  6  6  6  6  6  6  6  6  6  6  6  6  6
  [76]  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  6
 [101]  7  6  7  7  6  6  1  6  6  8  8  8  8  8  6  7  7  6  7  6  6  8  6  8  6
   ...

FE2013$Gears
   [1]  6  8  6  7  6  7  7  6  6  6  6  7  7  7  7  7  6  6  6  6  6  5  6  7  7
  [26]  7  7  6  7  7  7  7  7  7  5  5  6  6  6  6  6  6  6  6  6  6  6  6  7  6
  [51]  7  6  7  6  7  6  7  7  6  5  5  5  6  6  6  6  6  6  6  6  6  6  6  6  6
  [76]  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  6
 [101]  7  6  7  7  6  6  1  6  6  8  8  8  8  8  6  7  7  6  7  6  6  8  6  8  6
   ...
```

You can combine the $ notation and the [ ] notation since the $ extracts a vector and vectors are indexed via [ ].

```
> FE2013$Gears[1:3] # returns the first three elements of $
    Gears
 [1]  6  8  6
```

This of course also means that you can do anything to a data frame's variables that you can do to vectors.

```
> log(FE2013$Gears) + (FE2013$Gears + 100)
   [1] 107.7918 110.0794 107.7918 108.9459 107.7918
   [6] 108.9459 108.9459 107.7918 107.7918 107.7918
  [11] 107.7918 108.9459 108.9459 108.9459 108.9459
  [16] 108.9459 107.7918 107.7918 107.7918 107.7918
  [21] 107.7918 106.6094 107.7918 108.9459 108.9459
  [26] 108.9459 108.9459 107.7918 108.9459 108.9459
  [31] 108.9459 108.9459 108.9459 108.9459 106.6094
   ...
```

The with() Function

The problem with datasets being lists is that working with the $ notation is kind of tedious. Luckily there exists a function that makes dealing data frame names easier. Whenever you need to extract or index multiple variables and don't feel like typing dataset$variable.name each time, use the with() function.

```
> FE2013$Gears + FE2013$ModelYear / FE2013$Cylinder
 [1] 509.2500 511.2500 509.2500 342.5000 341.5000
 [6] 342.5000 132.8125 257.6250 257.6250 257.6250
[11] 257.6250 258.6250 258.6250 258.6250 258.6250
[16] 174.7500 207.3000 207.3000 207.3000 207.3000
[21] 509.2500 508.2500 509.2500 258.6250 258.6250
[26] 174.7500 510.2500 509.2500 342.5000 258.6250
  ...

> with(FE2013, Gears + ModelYear / Cylinder)
 [1] 509.2500 511.2500 509.2500 342.5000 341.5000
 [6] 342.5000 132.8125 257.6250 257.6250 257.6250
[11] 257.6250 258.6250 258.6250 258.6250 258.6250
[16] 174.7500 207.3000 207.3000 207.3000 207.3000
[21] 509.2500 508.2500 509.2500 258.6250 258.6250
[26] 174.7500 510.2500 509.2500 342.5000 258.6250
  ...
```

Warning NEVER use the attach() or detatch() to add a dataset to the search path of available R objects. If somebody tells you otherwise. They are wrong. These functions create all sorts of trouble.

5.2.2 Subsetting

You already know about the subset() function from our treatment of vectors. The subset() function works the same way for datasets. Let's look at the other data frame.

```
> Students
                         Name Year
1              Jeffrey Arnold    6
2      Sergio Ascencio Bonfil    2
3            Chitralekha Basu    3
4            Jonathan Bennett    1
5                  Peter Bils    1
6              Robert Carroll    4
7                   Hun Chung    1
8           Casey Crisman-Cox    2
9                  Trung Dang    3
10               Mason DeLang    2
11               David Gelman    1
```

```
12          Michael Gibilisco    2
13              Peter Haschke    6
14            YeonKyung Jeong    1
15               Doug Johnson    1
16               Gleason Judd    1
17              Kerim Kavakli    7
18          Brenton J. Kenkel    5
19                HyeSung Kim    5
20          Jonathan Klingler    6
21               Patrick Kuhn    7
22               Ben Laughlin    2
23              Youngchae Lee    7
24                Rabia Malik    3
25              Paulina Marek    6
26           Jeffrey Marshall    4
27           Justin Nicholson    1
28           Jonathan Olmsted    6
29                Lukas Pfaff    6
30         Barbara Piotrowska    1
31              Shawn Ramirez    8
32                Luke Reilly    2
33     Miguel R. Rueda Robayo    5
34         Kristin K. Rulison    5
35                Jeheung Ryu    1
36                Yoji Sekiya    8
37          Mattan Sharkansky    3
38              Bradley Smith    1
39             Jennifer Smith    2
40            William Spaniel    3
41              Jessica Stoll    7
42                  Ian Sulam    6
43            Susanna Supalla    4
44            Matthew Sweeten    1
45 Svanhildur Thorvaldsdottir    3
46        Ioannis Vassiliadis    2
47                    Jie Wen    1
```

Notice also that data frames are not printed the same as matrices (i.e. there are no [ ]).

Let's subset this dataset such that only first year students and myself are included:

```
> Rcourse <- subset(Students, Students$Year == 1 | Students$Name == "Peter Haschke")
> Rcourse
                 Name Year
4    Jonathan Bennett    1
5          Peter Bils    1
7           Hun Chung    1
11       David Gelman    1
13      Peter Haschke    6
14    YeonKyung Jeong    1
15       Doug Johnson    1
16       Gleason Judd    1
27   Justin Nicholson    1
30 Barbara Piotrowska    1
35        Jeheung Ryu    1
38      Bradley Smith    1
44    Matthew Sweeten    1
47            Jie Wen    1
```

5.2.3 Editing

R is not very good for editing datasets or data entry generally. This is not surprising since R is not a spreadsheet. If you want to use Excel to edit your datasets, feel free to do so. You know how to load and save from and to the `.csv` format which Excel can deal with. If you really feel so inclined – I do not advise this – you can use the `edit()` function. This will open up an interactive spreadsheet like environment for data entry and data manipulation. Again, just use Excel.

Let's do some manual data entry, anyway. We will use the $ operator to create a new variable in our `Rcourse` data frame.

```
> Rcourse$GreatFirstName <-  "No"
> Rcourse$GreatFirstName[c(2,5)] <- "Yes"
> Rcourse$New <- 1:length(Rcourse$Name)
> Rcourse$New2 <- rep(c("A", "B"), 7)
> Rcourse
                 Name Year GreatFirstName New New2
4    Jonathan Bennett    1              No   1    A
5          Peter Bils    1             Yes   2    B
7           Hun Chung    1              No   3    A
11       David Gelman    1              No   4    B
13      Peter Haschke    6             Yes   5    A
14    YeonKyung Jeong    1              No   6    B
15       Doug Johnson    1              No   7    A
```

```
16       Gleason Judd    1             No   8    B
27   Justin Nicholson    1             No   9    A
30 Barbara Piotrowska    1             No  10    B
35        Jeheung Ryu    1             No  11    A
38      Bradley Smith    1             No  12    B
44    Matthew Sweeten    1             No  13    A
47            Jie Wen    1             No  14    B
```

The transform() Function

To bulk edit or transform a number of variables at once, the transform() function can be used:

```
> Rcourse <- transform(Rcourse, Months = Year * 12
                              , New = New / 10 )
> Rcourse
                 Name Year GreatFirstName New New2 Months
4    Jonathan Bennett    1             No 0.1    A     12
5          Peter Bils    1            Yes 0.2    B     12
7           Hun Chung    1             No 0.3    A     12
11       David Gelman    1             No 0.4    B     12
13      Peter Haschke    6            Yes 0.5    A     72
14    YeonKyung Jeong    1             No 0.6    B     12
15       Doug Johnson    1             No 0.7    A     12
16       Gleason Judd    1             No 0.8    B     12
27   Justin Nicholson    1             No 0.9    A     12
30 Barbara Piotrowska    1             No 1.0    B     12
35        Jeheung Ryu    1             No 1.1    A     12
38      Bradley Smith    1             No 1.2    B     12
44    Matthew Sweeten    1             No 1.3    A     12
47            Jie Wen    1             No 1.4    B     12
```

5.3 More on *Objects*, *Modes* and other *Lies*

In Chapter 2 we talked rather loosely about *objects* and *modes*. I claimed that there exist a variety of different object types and that various objects can store elements of various modes.

For example. I claimed that there exists an object type called a vector. Moreover, I insisted that every vector can at most store elements of one mode. To determine what type of object and what mode we are dealing with the class(), mode(), and is() functions can be employed. Technically things are much more complicated and my treatment of *objects* and *modes* is quite forced.

Be that as it may. Our fancy data frame Rcourse can be understood as a special type of object (namely a special list) storing other objects (namely vectors and factors) of varying modes. Let's extract four of its components.

```
> Name <- Rcourse$Name
> Year <- Rcourse$Year
> New  <- Rcourse$New
> New2 <- Rcourse$New2
```

When we are using the class() function on the four vectors we have just created we will find the following.

```
> class(Name) # this tells us that the elements stored in
    this object are factors
[1] "factor"

> class(Year) # this tells us that the elements stored in
    this object are of mode integer
[1] "integer"

> class(New)  # this tells us that the elements stored in
    this object are of mode numeric
[1] "numeric"

> class(New2) # this tells us that the elements stored in
    this object are of mode character
[1] "character"
```

It is very important to realize that different types and modes affect the behavior of all R functions. For example factors are special vectors that contain an attribute called level. They are different from character vectors. To see this just print Name and New2:

```
> Name
 [1] Jonathan Bennett   Peter Bils
 [3] Hun Chung          David Gelman
 [5] Peter Haschke      YeonKyung Jeong
 [7] Doug Johnson       Gleason Judd
 [9] Justin Nicholson   Barbara Piotrowska
[11] Jeheung Ryu        Bradley Smith
47 Levels: Barbara Piotrowska ... Youngchae Lee

> New2
 [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A"
[12] "B" "A" "B"
```

In the above example we can see that printing each object produces different results. Printing a `factor` returns an abbreviated listing of its `levels`. To get the full list, type `levels(Name)`. You will see that this listing contains all the names of current graduate students even though none of them are part of the `Rcourse` data frame. In other words a `factor` is more than a vector of `characters` elements but and indicator vector. To beat a horse to death, type `summary(Name)` as well as `summary(New2)`. You can see that the summary of the `character` vector was not terribly useful.

Luckily, R is capable of changing mode and object types rather seamlessly. You will be bound to use many of the functions below:

`Function`	Description
`as.numeric()`	turns vectors, and matrices of other modes into a numeric ones
`as.character()`	turns vectors, and matrices of other modes into character ones
`as.integer()`	turns vectors, and matrices of other modes into integer ones[4]
`as.factor()`	will turn a vector, or matrices into factors
`as.matrix()`	will turn a vector, or data frame into a matrix[5]
`as.vector()`	will turn matrices into vectors
`as.data.frame()`	will turn vectors and matrices into data frames
`as.list()`	will turn vectors and matrices into lists

For our example above let's turn `New2` into a `factor` and try the summary command again. Instead of jibberish, the summary now produces a nice tabulation of frequencies.

```
> New2 <- as.factor(New2) # we are overwriting the old New2
> New2
 [1] A B A B A B A B A B A B A B
Levels: A B

> summary(New2)
A B
7 7
```

5.4 Data Summaries

Since you know how to extract and recall components of data frames, summaries can be computed with the functions you have used for vectors and matrices.

```
> mean(Rcourse$New) # the mean of the New variable for
   example
[1] 0.75

> summary(as.factor(Rcourse$GreatFirstName))
 No Yes
 12   2
```

You can also use the summary() function on the whole data frame.

```
> summary(Rcourse)   \\
        Names                Year      
 Barbara Piotrowska:1   Min.   :1.000  
 Bradley Smith     :1   1st Qu.:1.000  
 David Gelman      :1   Median :1.000  
 Doug Johnson      :1   Mean   :1.357  
 Gleason Judd      :1   3rd Qu.:1.000  
 Hun Chung         :1   Max.   :6.000  
 (Other)           :8                  

  GreatFirstName          New       
 Length:14          Min.   :0.100  
 Class :character   1st Qu.:0.425  
 Mode  :character   Median :0.750  
                    Mean   :0.750  
                    3rd Qu.:1.075  
                    Max.   :1.400  

     New2               Months     
 Length:14          Min.   :12.00  
 Class :character   1st Qu.:12.00  
 Mode  :character   Median :12.00  
                    Mean   :16.29  
                    3rd Qu.:12.00  
                    Max.   :72.00  
```

Another useful feature is the table() function. It allows you to create contingency tables. Use ?table to find out more on this.

```
> table(Students$Year)
 1  2  3  4  5  6  7  8 
13  8  6  3  4  7  4  2 

> table(Rcourse$Year, Rcourse$GreatFirstName)
    No Yes
  1 12   1
  6  0   1
```

A more complex table:

```
> table(FE2013$FErating, FE2013$Cylinder)
        3   4   5   6   8  10  12  16
  1     0   0   0   0   8   2   0   1
  2     0   0   0   0  53   1  15   0
  3     0   0   0   3  73   3   6   0
  4     0   6   0  86 106   0   1   0
  5     0  31   2 200  23   0   0   0
  6     0 121   8  59   0   0   0   0
  7     0 110   7   7   0   0   0   0
  8     0 119   0   4   0   0   0   0
  9     2  14   0   0   0   0   0   0
  10    0  11   0   0   0   0   0   0

> cor(FE2013$FErating, FE2013$Cylinder)
[1] -0.8114269
```

The str() function summarizes the structure of a dataset.

```
> str(Rcourse)
'data.frame':   14 obs. of  6 variables:
 $ Name          : Factor w/ 47 levels "Barbara Piotrows ...
 $ Year          : int  1 1 1 1 6 1 1 1 1 1 ...
 $ GreatFirstName: chr  "No" "Yes" "No" "No" ...
 $ New           : num  0.1 0.2 0.3 0.4 0.5 0.6 0.7 ...
 $ New2          : chr  "A" "B" "A" "B" ...
 $ Months        : num  12 12 12 12 72 12 12 12 12 12 ...
```

The describe() Function

For an even more detailed summary for our dataset, load the Hmisc package. You may have to install it first. As always check out: ?describe after loading the package.

```
> library(Hmisc)
> describe(Rcourse)
Rcourse

 6  Variables      14  Observations
---------------------------------------------------------
Name
      n missing  unique
     14       0      14

```

```
Barbara Piotrowska (1, 7%), Bradley Smith (1, 7%)
David Gelman (1, 7%), Doug Johnson (1, 7%)
Gleason Judd (1, 7%), Hun Chung (1, 7%)
Jeheung Ryu (1, 7%), Jie Wen (1, 7%)
Jonathan Bennett (1, 7%), Justin Nicholson (1, 7%)
Matthew Sweeten (1, 7%), Peter Bils (1, 7%)
Peter Haschke (1, 7%), YeonKyung Jeong (1, 7%)
---------------------------------------------------------
Year
      n missing  unique    Mean
     14       0       2   1.357

1 (13, 93%), 6 (1, 7%)
---------------------------------------------------------
GreatFirstName
      n missing  unique
     14       0       2

No (12, 86%), Yes (2, 14%)
---------------------------------------------------------
New
      n missing  unique    Mean     .05     .10     .25
     14       0      14    0.75   0.165   0.230   0.425
    .50     .75     .90     .95
  0.750   1.075   1.270   1.335

          0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2
Frequency   1   1   1   1   1   1   1   1   1 1   1   1
%           7   7   7   7   7   7   7   7   7 7   7   7
          1.3 1.4
Frequency   1   1
%           7   7
---------------------------------------------------------
New2
      n missing  unique
     14       0       2

A (7, 50%), B (7, 50%)
---------------------------------------------------------
Months
      n missing  unique    Mean
     14       0       2   16.29

12 (13, 93%), 72 (1, 7%)
---------------------------------------------------------
```

Chapter 6

Graphics

R is a fantastic tool when it comes to graphics. An entire short course could be devoted to making graphics and plotting in R, as there are many approaches and packages designed for this specific task. Unfortunately, we can only scratch the surface and the main purpose of this chapter is to showcase some of R's graphical capabilities and to point you in the right direction.

R comes with a slew of graphic capabilities and functions pre-installed as part of the `base` package. Many of these functions are well documented and I encourage you to learn about them and play around with various approaches by browsing the following website: http://gallery.r-enthusiasts.com/.

I believe that ultimately the choice of approach comes down to taste. If you are into a simple look, that may or may not be terrible consistent across different types of plots (e.g. histograms vs. pie charts) than the `base` graphics functions may work for you. At the end of the day any approach you may want to take requires some learning and practice. With this said, you might as well begin with a more flexible and consistent approach – `ggplot2`.[1] So let's use Hadley Wickham's `ggplot2` package. Extensive and detailed documentation and examples for his package can be found here: http://docs.ggplot2.org/current/.

6.1 ggplot2

For all examples we will use the fuel economy data found on my website. Let's load it up and also install and load the `ggplot2` package.

```
> FE2013 <- read.csv("http://peterhaschke.com/Teaching/R-Course/FE2013.csv")
> install.packages("ggplot2") # this may be installed in the starlab already
> library(ggplot2)
>
```

[1] The learning curves for plotting with the `base` functions vs. `ggplot2` or `lattice` are comparable. You should start learning the approach that you find most appealing, visually.

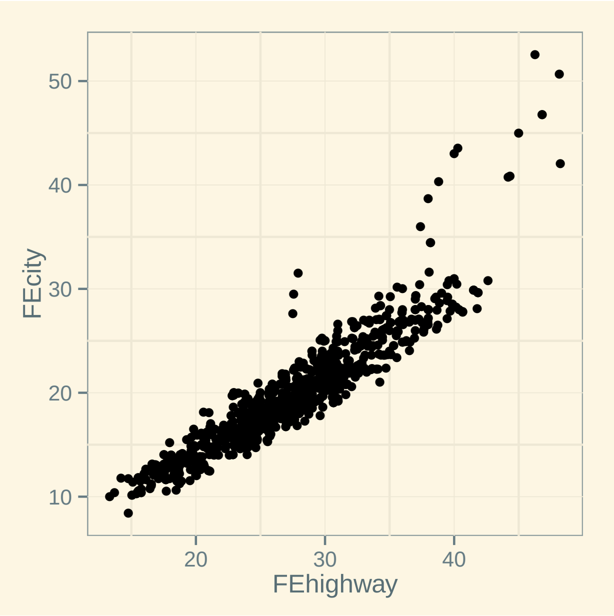
6.1.1 Scatterplots

Having installed and loaded ggplot2 we now have access to the packages ggplot() function. The ggplot() function does nothing other than create or initiate a ggplot object. It takes at the minimum one argument: data. All aspects of a plot are then subsequently added as layers with separate functions called geoms. The geom used to create scatterplots is geom_point(). The geom_point() function itself takes an argument called aes which assigns data to aesthetic properties. This sounds terrible complicated but really isn't. Let's give it a shot.

```
> plot1 <- ggplot(data = FE2013) # we are creating an empty ggplot object
> plot1 # if you print this plot to the screen, R's graphic device will open an empty plot
>
```

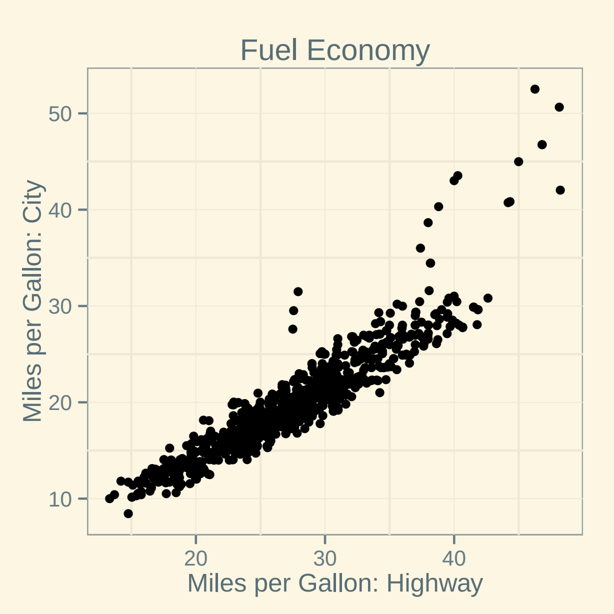
We now literally add a scatterplot layer to plot1 via the geom_point() function. The geom_point() function takes the aes() argument, which generates a mapping describing how variables in the data are mapped to visual properties of geoms. For geom_point() the arguments x and y must be supplied to aes. Let's compare highway and city fuel economies.

```
> plot1 <- plot1 + geom_point(aes(x = FEhighway, y = FEcity))
> plot1 # we now have added a layer and printing the object to the screen will open R's graphics device with a scatterplot.
>
```



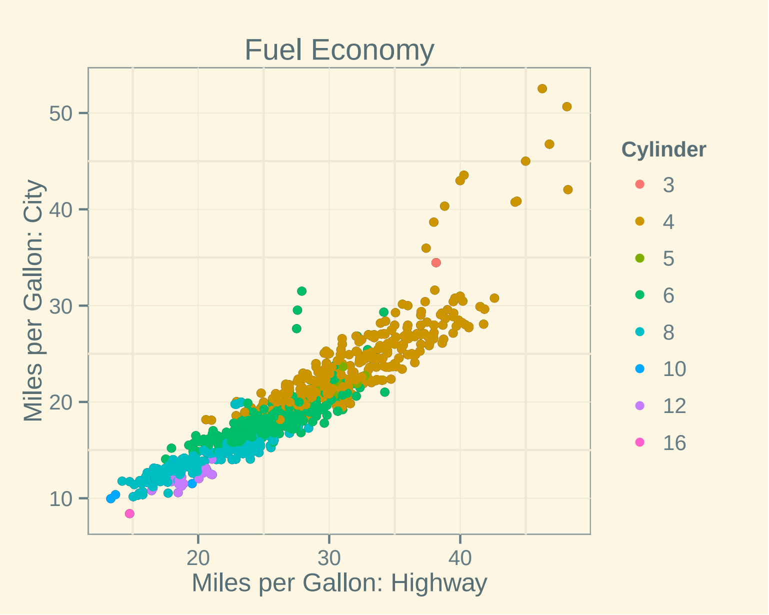
The plot is still pretty bare and poorly labeled. We can change all this by adding more layers. Let's add a layer called `labs()`. It takes the arguments `x`, `y`, and `title`.

```
> plot1 <- plot1 + labs(title = "Fuel Economy", x = "Miles per Gallon: Highway", y = "Miles per Gallon: City") # saving the labs layer to plot1
> plot1
>
```



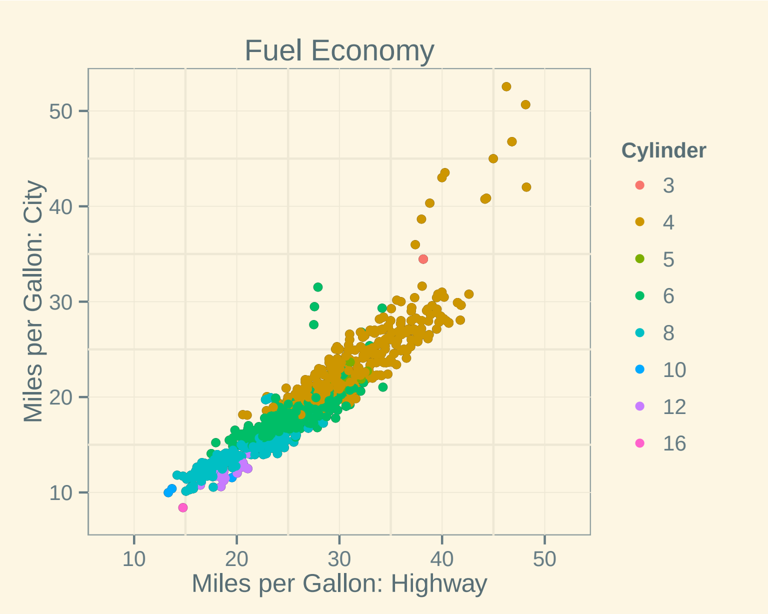
The great thing about `ggplot2` is that you can map additional data to aesthetics easily. Let's add a factor to the mapping via the `color` aesthetic. This will change the color of the points depending on a factor vector. In our dataset we have a variable called `Cylinder` that seems to be categorical but is currently an integer vector.

```
> class(FE2013$Cylinder)
[1] "integer"
> FE2013$Cylinder <- as.factor(FE2013$Cylinder) # let's change the Cylinder variable to a factor. note: we are overwriting the original variable
> plot1 <- plot1 + geom_point(aes(x = FEhighway, y = FEcity, color = Cylinder))
> plot1
```



Very nice. The plot looks pretty good. But just for good measure let's change the axis tick-marks. To change the scales layer we can use the `scale_x_continuous()` and `scale_y_continuous()` functions. Each takes an argument `breaks` which can be supplied with a vector of tick-mark locations. To change the limits of the axes we can use the `coord_cartesian()` function which take the argument `xlim` and `ylim`.

```
> plot1 <- plot1 + coord_cartesian(xlim = c(5,55), ylim = c(5,55)) # this changes the limits of the axes
> plot1 <- plot1 + scale_x_continuous(breaks = c(10, 20, 30, 40, 50)) # this changes the tick-marks on the x-axis
> plot1
>
```



6.1.2 Exporting Graphics

To export plots you have created in R to a format that you can use in your LaTeX documents you should use the pdf() function to save the plot as a .pdf.[2] The pdf() function will initiate the R graphics device and then save your plot to disk. There is a sequence that you need to follow in your code. First, you need to initiate the graphics device via the pdf() function. Secondly, you print the plot and all layers you have created to the device so that the pdf() function will save it. Lastly, you will have to close the graphics device with the dev.off() function. The pdf() function takes a variety of arguments. The most useful ones are: file to specify the location where your file will be saved, and width and height to set the size of the .pdf file. Let's save the plot we have just created.

```
> pdf(file = "Z:/Plot1.pdf", width=5, height=4) # Step 1
> plot1 # Step 2
> dev.off() # Step 3
null device
          1
>
```

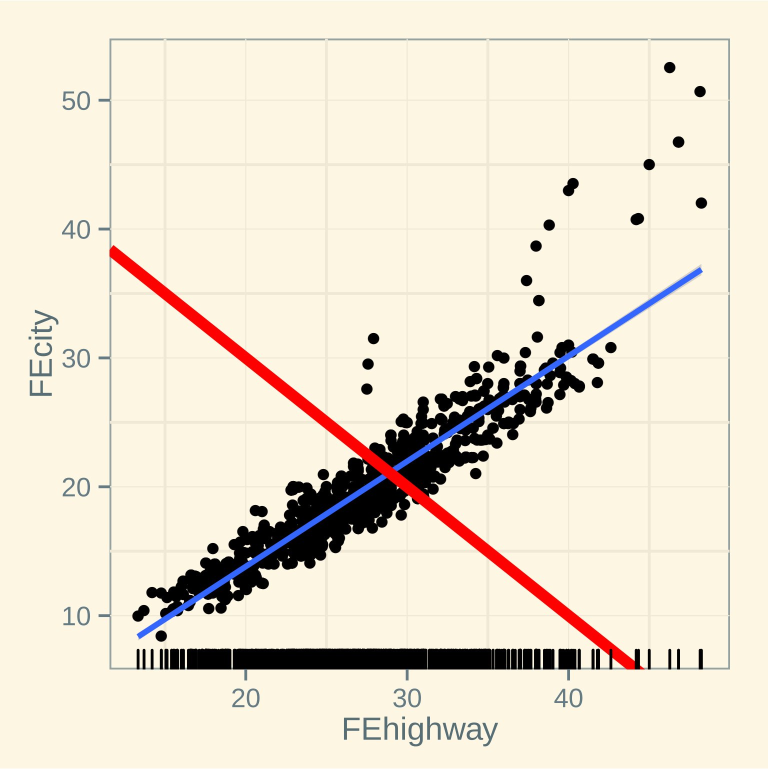
6.1.3 Adding more geoms

For exposition, the code for our figure so far has been unnecessarily complicated. Instead of creating a ggplot object and then successively saving additions to our plot, we can simplify things and add them all at once (i.e. we can write: Plot <- ggplot(...) + geom(...) + labs()). As long as each line we feed to R ends with a "+" R will know more input is coming. Let's go back to our basic plot and add geoms in one swoop. We will add a best fit line to the figure (via

[2] For exporting to other filetypes you can you the jpeg(), ps(), or png() functions.

the geom_smooth() function), an arbitrary line (via the geom_abline() function) and a rug plot (via the geom_rug() function). Note: that we are specifying the the aes() argument which maps the data to an aesthetic object, directly in the ggplot() function instead of separately in each geom.

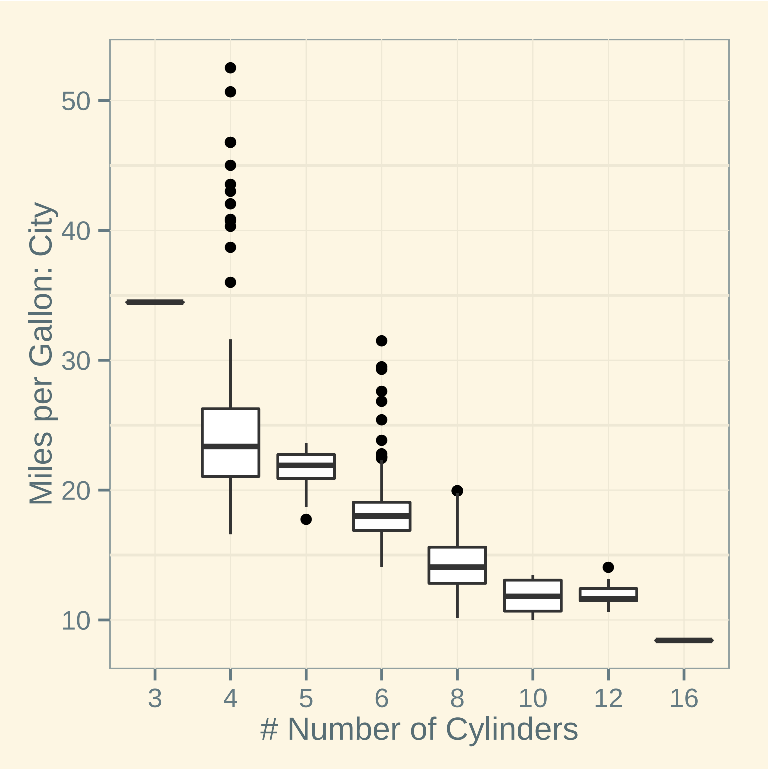
```
> plot2 <- ggplot(data = FE2013, aes(x = FEhighway, y =
   FEcity)) +
       geom_point() +
       geom_smooth(method = "lm", size = 1) +
       geom_abline(intercept = 50, slope = -1, color = "red",
          size = 2) +
       geom_rug(sides = "b")
> plot2
>
```



6.1.4 Boxplots: geom_boxplot()

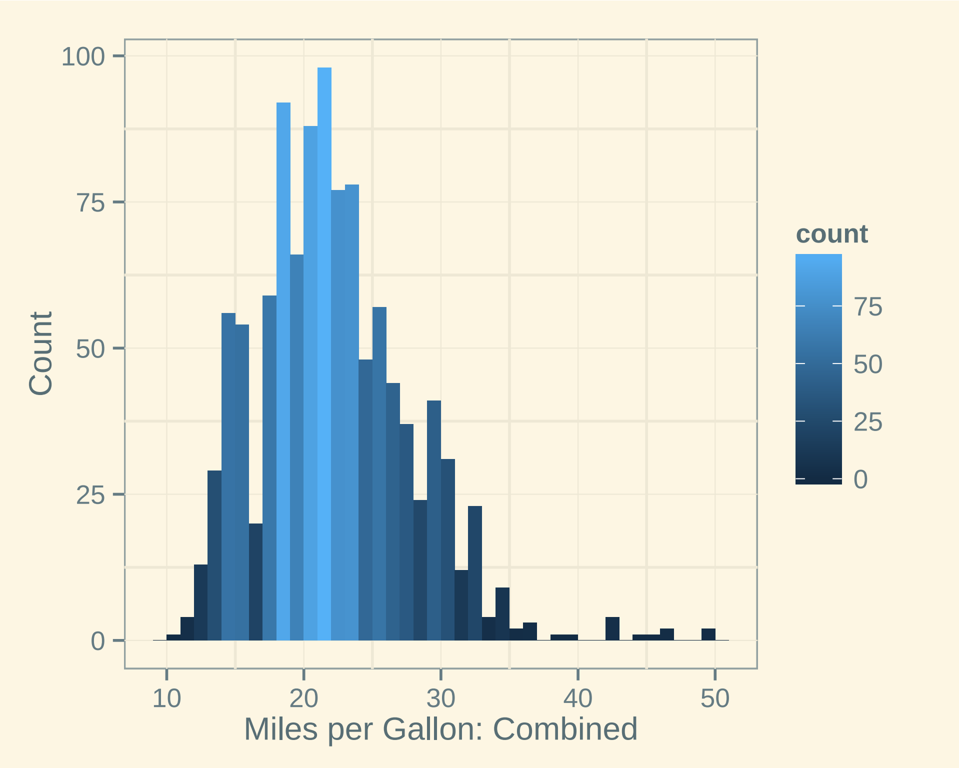
The ggplot2 package is quite powerful but as you can see it is also somewhat complicated to figure it all out. The remaining sections will provide some templates and examples of what ggplot2 can do for you.

```
> plot3 <- ggplot(data = FE2013, aes(x = Cylinder, y = FEcity
  )) +
      geom_boxplot() +
      labs(y = "Miles per Gallon: City", x = "# Number of
         Cylinders")
> plot3
```



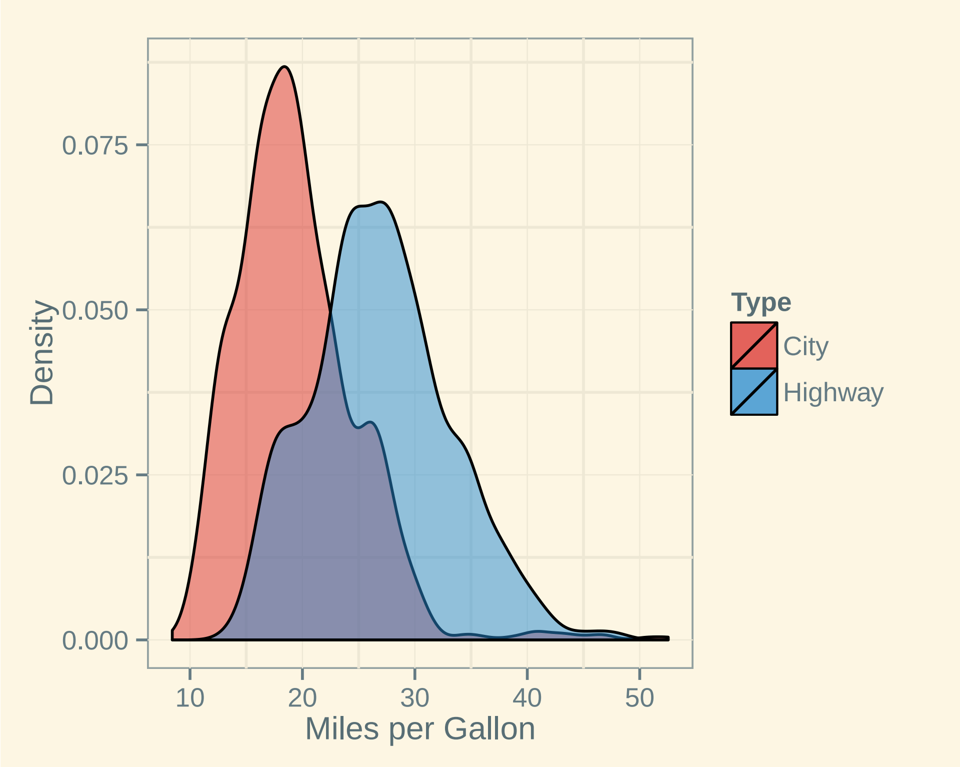
6.1.5 Histograms: geom_histogram()

```
> plot4 <- ggplot(data = FE2013, aes(x = FEcombined)) +
      geom_histogram(binwidth = 1, aes(fill = ..count..)) +
      labs(x = "Miles per Gallon: Combined", y = "Count")
> plot4
```



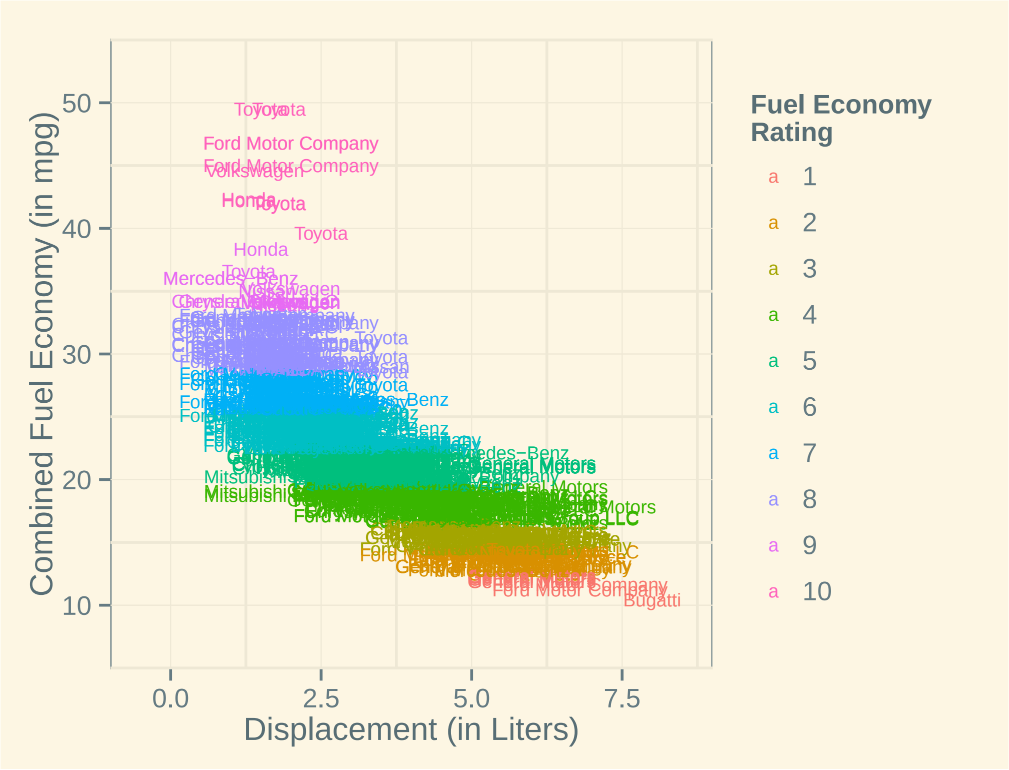
6.1.6 Density Plots: geom_density()

```
>plot5 <- ggplot(data = FE2013) +
      geom_density(aes(FEcity, fill = "City"), alpha = 0.5) +
      geom_density(aes(FEhighway, fill = "Highway"), alpha =
         0.5) +
      labs(x = "Miles per Gallon", y = "Density") +
      guides(fill = guide_legend(title = "Type"))
> plot5
```



6.1.7 Text Plots: `geom_text()`

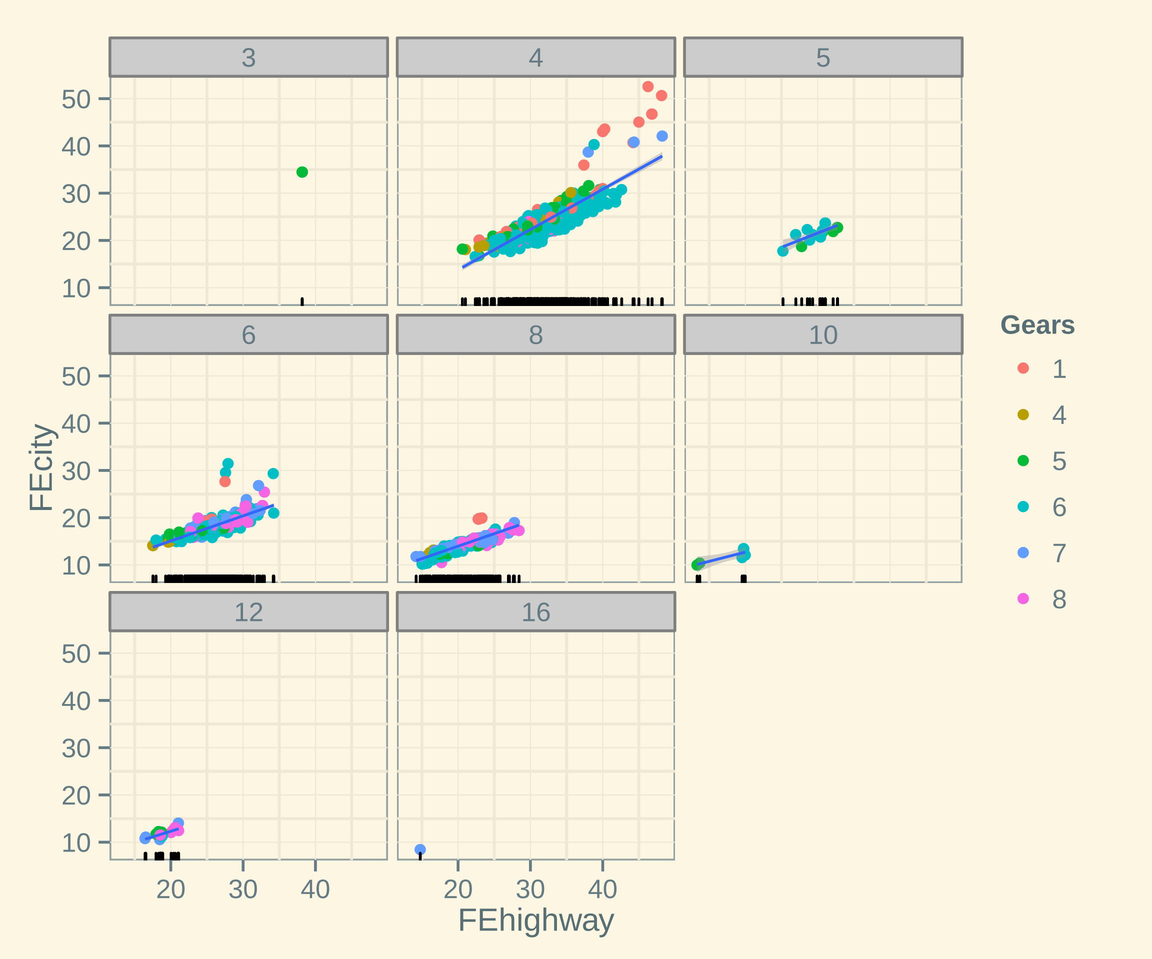
```
> plot6 <- ggplot(data = FE2013) +
      geom_text(aes(x = Displacement, y = FEcombined, label =
          Manufacturer, color = as.factor(FErating)), size =
         3) +
      labs(x = "Displacement (in Liters)", y = "Combined Fuel
          Economy (in mpg)") +
      guides(color = guide_legend(title = "Fuel Economy\
         nRating")) +
      coord_cartesian(xlim = c(-1,9), ylim = c(5,55))
> plot6
```



6.1.8 Faceting: facet_wrap() & facet_grid()

The facet_wrap() and facet_grid() functions allow for creating graphical contingency tables.

```
> FE2013$Gears <- as.factor(FE2013$Gears)
> FE2013$FErating <- as.factor(FE2013$FErating)
> plot7 <- ggplot(data = FE2013, aes(x = FEhighway, y =
   FEcity)) +
        geom_point(aes(color = Gears)) +
        geom_rug(sides = "b") +
        geom_smooth(method = "lm") +
        facet_wrap(~ Cylinder)
> plot7
```



6.1.9 Multiple Plots on One Page

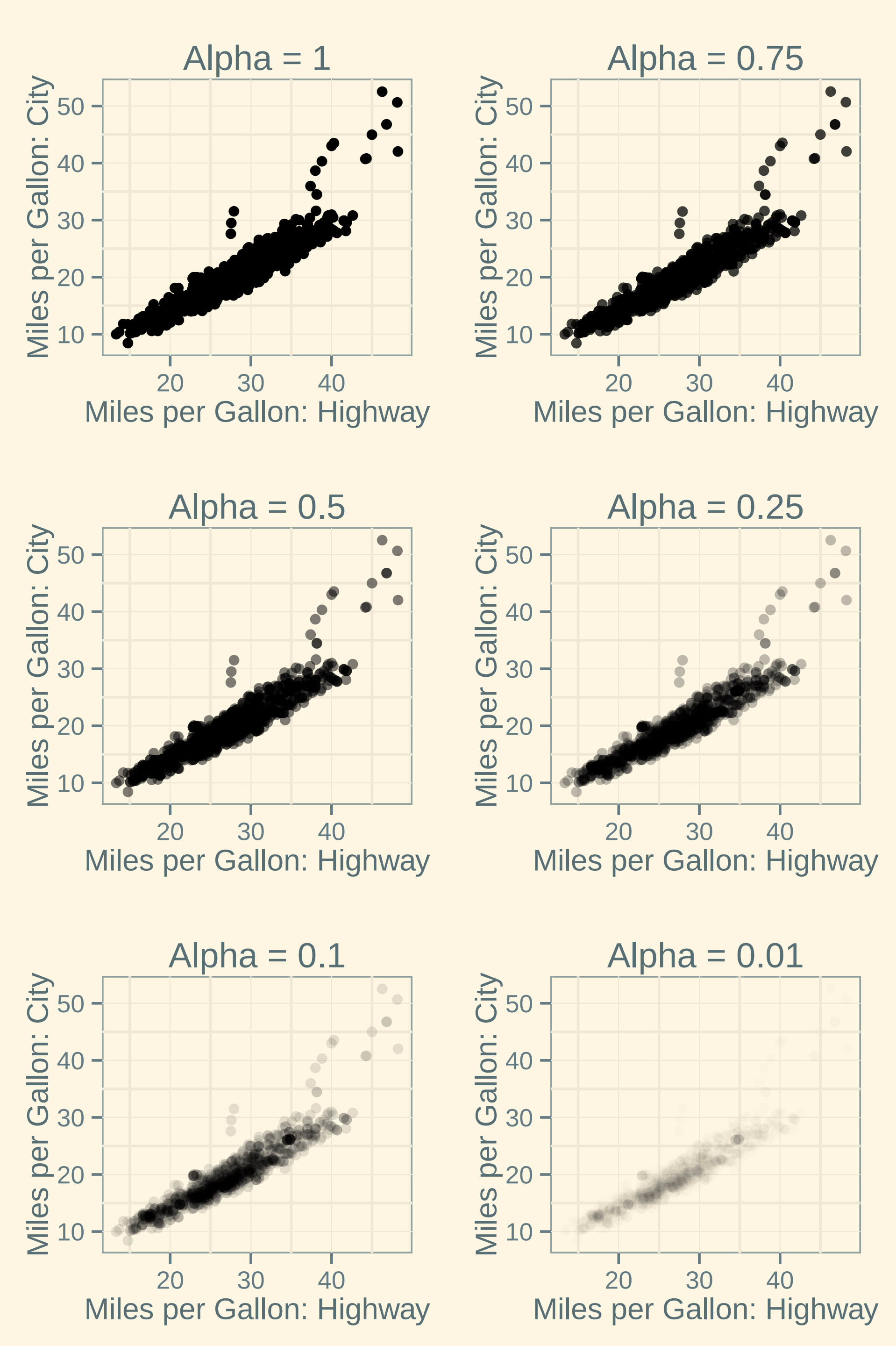
It is often the case that you will want to add multiple plots to a page. This can easily be done by initiating a device that is split up into a grid. To do this you will need to install a package called grid and then load it via the library() function.

Below, I created four separate scatter plots. They are all identical except that I am varying the transparency of the points with the alpha argument. This is a useful argument to highlight overplotting.

```
> A <- ggplot(data = FE2013) +
  geom_point(aes(x = FEhighway, y = FEcity), alpha = 1) +
  labs(title = "Alpha = 1", x = "Miles per Gallon: Highway"
     , y = "Miles per Gallon: City")

> B <- ggplot(data = FE2013) +
  geom_point(aes(x = FEhighway, y = FEcity), alpha = 0.5) +
  labs(title = "Alpha = 0.5", x = "Miles per Gallon:
     Highway", y = "Miles per Gallon: City")

> C <- ggplot(data = FE2013) +
  geom_point(aes(x = FEhighway, y = FEcity), alpha = 0.3) +
  labs(title = "Alpha = 0.3", x = "Miles per Gallon:
     Highway", y = "Miles per Gallon: City")

> D <- ggplot(data = FE2013) +
  geom_point(aes(x = FEhighway, y = FEcity), alpha = 0.1) +
  labs(title = "Alpha = 0.1", x = "Miles per Gallon:
     Highway", y = "Miles per Gallon: City")
```

To plot A, B, C, D, E, and F into a 3 by 2 grid, load the grid package and enter the following code:

```
> library(grid) # you may have to install the package first
> grid.newpage()
> pushViewport(viewport(layout = grid.layout(3, 2)))
> Layout <- function(x, y) viewport(layout.pos.row = x,
   layout.pos.col = y)
> print(A, vp = Layout(1, 1))
> print(B, vp = Layout(1, 2))
> print(C, vp = Layout(2, 1))
> print(D, vp = Layout(2, 2))
> print(E, vp = Layout(3, 1))
> print(F, vp = Layout(3, 2))
```


Alpha = 1
Alpha = 0.75
Alpha = 0.5
Alpha = 0.25
Alpha = 0.1
Alpha = 0.01
Miles per Gallon: City
Miles per Gallon: Highway
10
20
30
40
50
20
30
40

6.1.10 Recap: The Makings of a `ggplot`

A quick step by step summary of how to create a plot in `ggplot`:

1. Create a `ggplot` object with the `ggplot()` function. You will need to specify the data you will be using via the `data` argument.

   → `ggplot(data = MyDataset)`

2. Add a `geom` layer to the plot and specify the aesthetic mapping with `aes`

   → `geom_point(aes(x = Variable1, y = Variable2))`

   → `geom_boxplot(aes(x = Variable1))`

   → `geom_smooth(aes(x = Variable1, y = Variable2))`

3. You are done. Unless you want to mess with the defaults

4. Changing the default axis and plot labels:

   → `labs(title = "MyTitle", x = "X-Axis Label", y = "Y-Axis Label")`

5. Changing the range of the plot

   → `coord_cartesian(xlim = c(xmin,xmax), ylim = c (ymin,ymax))`

6.1.11 Common Aesthetics

All geoms require a set of aesthetic mappings (unless you already specified a default in the `ggplot()` function itself). The most common ones are x and y. For example to plot a scatterplot the `geom_point()` function needs to know which variable you want to map to the X-axis and which to the Y-axis. Below is a list of common additional ones that you can often specify within the `aes()` argument, or outside the `aes()` argument if you don't want things to be mapped to the legend. Always consult the documentation for a list of aesthetics each geom requires or understands: http://docs.ggplot2.org/current/.

`Aesthetic`	Description
`x`	for mapping a variable to the x-axis
`y`	for mapping a variable to the y-axis
`color`	for mapping a variable or constant to a color → `aes(color = Variable1)` or `aes(color = "red")`
`linetype`	for mapping a variable or constant to a linetype → `aes(linetype = Variable1)` or `aes(linetype = 2)`
`shape`	for mapping a variable or constant to the shape of points (squares, triangles, etc) → `aes(shape = Variable1)` or `aes(shape = 3)`
`size`	for mapping a variable or constant to the size or width of points or lines → `aes(size = Variable1)` or `aes(size = 5)`
`alpha`	for mapping a variable or constant to the transparency of lines or points, etc → `aes(alpha = Variable1)` or `aes(alpha = 0.5)`
`fill`	for mapping a variable or constant to the fill color of an area → `aes(fill = Variable1)` or `aes(fill = "green")`

Chapter 7

Programs

After an overview of R's capabilities and functions, this chapter will be devoted to some basic programming concepts.

7.1 Conditionals

One of the most basic programing concepts involves evaluating conditional statements and then performing some action based on the evaluation. Let's write a very simple conditional using the `if()` control-flow construct. The `if()` construct is simply a control statement that takes a conditional statement as its argument and then depending on the evaluation initiates some other function. Let's ask R if `1 + 1 = 2` and if so to print something to the screen with the `print()` function:

```
> if(1 + 1 == 2) print("Definitely!")
[1] "Definitely."
>
```

You can see that in this case the `if()` statement evaluated to `TRUE` and it activated the `print()` function. If the `if()` statement were `FALSE` nothing would happen. Try it.

```
> if(1 + 1 == 3) print("Definitely.")
>
```

If we wanted to R to also do something if the conditional is `FALSE` we would have to add the `else` statement.

```
> if(1 + 1 == 3) print("Definitely.") else print("Wrong.")
[1] "Wrong."
>
```

This can be pretty useful especially since we can ask R to perform much more complicated tasks. Before we try this I want to introduce you to some of R's random number generating functions.[1]

Function	Description
rnorm()	random variates for the normal distribution (arguments: n, mean, sd)
runif()	random variates for the uniform distribution (arguments: n, min, max)
rbinom()	random variates for the binomial distribution (arguments: n, size, prob)
rweibull()	random variates for the Weibull distribution (arguments: n, shape, scale)

Let's use these random number generators within an more complicated if() statement. To combine multiple tasks R should perform we have to use curly braces { }. Copy the following code and tell me what this program is doing.

```
 Test <- runif(n = 1, min = 0, max = 2)
 if(Test < 1) {
    X <- rnorm(n = 1000, mean = Test, sd = 1)
    ggplot() +
      geom_density(aes(X), fill = "red") +
      labs(title = paste("Mean = Test =", Test),
        x = "1000 Random Normal Deviates, sd = 1",
        y = "Density")
  } else {
     Y <- rweibull(n = 1000, shape = Test, scale = 1)
     ggplot() +
      geom_density(aes(Y), fill = "green") +
      labs(title = paste("Shape = Test =", Test),
        x = "1000 Random Weibull Deviates, scale = 1",
        y = "Density")
  }
>
```

7.1.1 The ifelse() Function

Instead of using the if() and else control statements, you can just use the ifelse() function. The function takes the arguments test, yes and no.

```
> Z <- 1:10
> Z
[1]  1  2  3  4  5  6  7  8  9 10
> Z.new <- ifelse(test = Z > 2 & Z < 8, yes = "Yes", no = "No")
> Z.new
[1] "No" "No" "Yes" "Yes" "Yes" "Yes" "Yes" "No" "No" "No"
```

[1] For a more complete list type ?Distributions.

This function comes in very handy to do data manipulation. Suppose we wanted to create a dummy variable in our Students dataset such that FirstYears = 1, else 0. As always we can drop the argument names and write something like this:

```
> library(foreign)
> Students <- read.dta("http://www.peterhaschke.com/Teaching/R-Course/Students.dta")
> names(Students)
[1] "Name" "Year"

> Students$FirstYears <- ifelse(Students$Year == 1, 1, 0)
> names(Students)
[1] "Name" "Year" "FirstYears"

> ifelse(Students$FirstYears == 1, print(as.character(Students$Name)), print("NA"))
 [1] "NA"                 "NA"
 [3] "NA"                 "Jonathan Bennett"
 [5] "Peter Bils"         "NA"
 [7] "Hun Chung"          "NA"
 [9] "NA"                 "NA"
[11] "David Gelman"       "NA"
[13] "NA"                 "YeonKyung Jeong"
[15] "Doug Johnson"       "Gleason Judd"
[17] "NA"                 "NA"
[19] "NA"                 "NA"
[21] "NA"                 "NA"
[23] "NA"                 "NA"
[25] "NA"                 "NA"
[27] "Justin Nicholson"   "NA"
[29] "NA"                 "Barbara Piotrowska"
[31] "NA"                 "NA"
[33] "NA"                 "NA"
[35] "Jeheung Ryu"        "NA"
[37] "NA"                 "Bradley Smith"
[39] "NA"                 "NA"
[41] "NA"                 "NA"
[43] "NA"                 "Matthew Sweeten"
[45] "NA"                 "NA"
[47] "Jie Wen"
```

7.1.2 Nested Control Flow Statements

If-statements can easily be nested. This can become quite ugly looking and you may easily lose track of your conditions.

```r
> r <- runif(n = 1, min = 0, max = 1)
> if(r < 0.2){
  cat("r is", r, "which is smaller than 0.2")
  } else {
    if(0.2 < r & r < 0.5 ) {
      cat("r is", r, "which is between 0.2 and 0.5")
    } else {
    if(r > 0.5 & r < 0.9) {
      cat("r is", r, "which is greater than 0.5")
    } else {
      cat("r is", r, "which is greater than 0.9")
    }
  }
}
```

Things tend to be a bit cleaner with the build in `ifelse()` function.[2]

```r
ifelse(r < 0.2,
  paste("r is", r, "which is smaller than 0.2"),
  ifelse(0.2 < r & r < 0.5,
    paste("r is", r, "which is between 0.2 and 0.5"),
    ifelse(r > 0.5 & r < 0.9,
      paste("r is", r, "which is greater than 0.5"),
      paste("r is", r, "which is greater than 0.9")
    )
  )
)
```

[2] Notice the use of the `paste()` function. We have to use `paste()` with the `ifelse()` function because `ifelse()` does not play nice with `cat()`. Ultimately, `paste()` does the same thing but it implicitly creates an object (i.e. a character vector) whereas `cat()` does not. Since you cannot assign the output of `cat()`, `ifelse()` which is a function manipulating and returning objects, breaks. The `if()` statement is not a function but a control flow operator and doesn't care if an object is created after the logical evaluation or not.

7.2 For Loops

Another control flow operator is the `for()` operator. For loops are the workhorses of R programming. They are extremely intuitive and straightforward to write and wrap your head around. Unfortunately, they also tend to be very inefficient. With modern computing power this may not be a problem for simple tasks. But you can bog your computer down with these if you are not careful.[3] The easiest way to understand how to use the `for()` operator is by example:

```
> for(i in 1:10) {print(i)}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

In the above example we asked R that for each *i* in the sequence from 1 and 10 we want that element printed to the screen. Or put differently, for each *i* in some object do the thing inside the curly braces.

Try this:

```
> X <- 1000000:1
> for(i in X) {print(i)}
>
```

To stop this silly loop hit the `Esc` button or press `Ctrl+C`. Now how about this:

Warning: This example may induce SEIZURES!

```
> X <- 10000:1
> for(i in X) {
    temp <- rnorm(n = 100, mean = 0, sd = 2)
    plot <- ggplot() + geom_density(aes(temp), fill = i) +
       labs(title = paste("i =", i))
    print(plot)
  }
>
```

[3] The often unintuitive alternative to for loops is vectorization via the `apply()` and `lapply()` functions. Feel free to google it.

7.2.1 Applications with Data

Let's actually use for loops to do something useful. We start by loading the Fuel Economy dataset and write a simple loop to start with.

```
> FE2013 <- read.csv("http://peterhaschke.com/Teaching/R-
   Course/FE2013.csv")
> levels(FE2013$Manufacturer) {
  [1] "Audi"             "Bentley"
  [3] "BMW"              "Bugatti"
  [5] "Chrysler"         "Ferrari"
  [7] "Ford"             "General Motors"
  [9] "Honda"            "Hyundai"
 [11] "Jaguar"           "Kia"
 [13] "Lamborghini"      "Land Rover"
 [15] "Lotus"            "Maserati"
 [17] "MAZDA"            "Mercedes-Benz"
 [19] "Mitsubishi"       "Nissan"
 [21] "Porsche"          "Rolls-Royce"
 [23] "Roush"            "Subaru"
 [25] "Suzuki"           "Toyota"
 [27] "Volkswagen"       "Volvo"

> for(i in levels(FE2013$Manufacturer)){{
    print(i)
  }
 [1] "Audi"
 [1] "Bentley"
 [1] "BMW"
 [1] "Bugatti"
 [1] "Chrysler"
 [1] "Ferrari"
 [1] "Ford"
 [1] "General Motors"
 [1] "Honda"
 [1] "Hyundai"
 [1] "Jaguar"
 [1] "Kia"
 [1] "Lamborghini"
 [1] "Land Rover"
 [1] "Lotus"
 [1] "Maserati"
 [1] "MAZDA"
 [1] "Mercedes-Benz"
 [1] "Mitsubishi"
 [1] "Nissan"
 [1] "Porsche"
 [1] "Rolls-Royce"
```

```
[1] "Roush"
[1] "Subaru"
[1] "Suzuki"
[1] "Toyota"
[1] "Volkswagen"
[1] "Volvo"
```

Nice. With the above loop we were able to print out each level of the variable manufacturer. Knowing this we can start adding some useful features to the loop.

```
> for(i in levels(FE2013$Manufacturer)){
    temp <- subset(FE2013, FE2013$Manufacturer==i)
    mean.mpg <- round(mean(temp$FEcombined))
    cat(mean.mpg, "mpg for", i,"\n")
  }
23 mpg for Audi
15 mpg for Bentley
23 mpg for BMW
10 mpg for Bugatti
22 mpg for Chrysler
14 mpg for Ferrari
22 mpg for Ford
20 mpg for General Motors
26 mpg for Honda
26 mpg for Hyundai
19 mpg for Jaguar
26 mpg for Kia
15 mpg for Lamborghini
16 mpg for Land Rover
21 mpg for Lotus
15 mpg for Maserati
26 mpg for MAZDA
20 mpg for Mercedes-Benz
24 mpg for Mitsubishi
22 mpg for Nissan
21 mpg for Porsche
14 mpg for Rolls-Royce
17 mpg for Roush
24 mpg for Subaru
25 mpg for Suzuki
23 mpg for Toyota
27 mpg for Volkswagen
21 mpg for Volvo
```

In the for loop above we did the following things for each i in `levels(FE2013$Manufacturer)` (i.e. for each car manufacturer):

1. For each manufacturer i, we subset our dataset such that it only contains observations for i. For each i we saved this subset of the dataset to the object `temp`.
2. After the subsetting, we compute the rounded mean of the combined fuel economy for the subset and store it in the object called `mean.mpg`.
3. After each loop we tell R to concatenate (`cat()`) the mean.mpg to the i^{th} manufacturer.

Suppose we actually wanted to save the output the loop generated instead of just printing it to the screen. The easiest way to do this is to create a matrix populated by `NA`'s which we can then populate it with the data the loop generates.

```
> Data <- matrix(NA, nrow = length(levels(FE2013$Manufacturer
  )), ncol = 4)
> rownames(Data) <- as.character(levels(FE2013$Manufacturer))
> colnames(Data) <- c("meanFE", "sdFE", "medianRating", "N")
> Data
               MeanFE   sdFE   medianRating      N
Audi               NA     NA             NA     NA
Bentley            NA     NA             NA     NA
BMW                NA     NA             NA     NA
Bugatti            NA     NA             NA     NA
Chrysler           NA     NA             NA     NA
Ferrari            NA     NA             NA     NA
Ford               NA     NA             NA     NA
General Motors     NA     NA             NA     NA
Honda              NA     NA             NA     NA
Hyundai            NA     NA             NA     NA
Jaguar             NA     NA             NA     NA
Kia                NA     NA             NA     NA
Lamborghini        NA     NA             NA     NA
Land Rover         NA     NA             NA     NA
Lotus              NA     NA             NA     NA
Maserati           NA     NA             NA     NA
MAZDA              NA     NA             NA     NA
Mercedes-Benz      NA     NA             NA     NA
Mitsubishi         NA     NA             NA     NA
Nissan             NA     NA             NA     NA
Porsche            NA     NA             NA     NA
Rolls-Royce        NA     NA             NA     NA
Subaru             NA     NA             NA     NA
Suzuki             NA     NA             NA     NA
Toyota             NA     NA             NA     NA
Volkswagen         NA     NA             NA     NA
Volvo              NA     NA             NA     NA
```

Now we can use a for loop to populate the matrix.

```
> for(i in levels(FE2013$Manufacturer)){
    temp <- subset(FE2013, FE2013$Manufacturer==i)
    Data[i,1] <- round(mean(temp$FEcombined), digits = 2)
    Data[i,2] <- round(sd(temp$FEcombined), digits = 2)
    Data[i,3] <- median(temp$FErating)
    Data[i,4] <- nrow(temp)
  }
> Data
               MeanFE    sdFE    medianRating    N
Audi            22.66    3.15             6.0   43
Bentley         14.85    1.77             2.0    9
BMW             23.37    4.84             6.0  139
Bugatti         10.44      NA             1.0    1
Chrysler        21.55    4.50             5.0   82
Ferrari         14.36    1.02             2.5    8
Ford            21.90    6.96             5.0   94
General Motors  19.78    5.25             4.0  177
Honda           25.83    6.17             6.0   31
Hyundai         25.98    4.39             6.0   34
Jaguar          18.65    1.23             4.0   13
Kia             25.90    3.06             7.0   34
Lamborghini     14.67    1.09             3.0    5
Land Rover      16.24    4.21             2.5    4
Lotus           21.46    1.03             5.0    4
Maserati        15.03    0.80             3.0    3
MAZDA           26.07    4.14             6.0   24
Mercedes-Benz   20.41    4.36             5.0   77
Mitsubishi      24.29    3.09             6.0   17
Nissan          21.80    5.47             5.0   57
Porsche         20.83    2.24             5.0   41
Rolls-Royce     14.34    0.74             2.0    6
Roush           17.42    1.07             4.0    2
Subaru          24.02    3.80             6.0   22
Suzuki          24.64    2.01             6.0   16
Toyota          23.15    7.62             5.0   77
Volkswagen      27.00    5.18             6.5   46
Volvo           21.27    1.95             5.0   16
```

7.2.2 Putting the Pieces Together

Although the code below may look complicated, most of it should be straightforward to interpret. Nothing you haven't seen before:

```
> library(ggplot2)
> FE2013$Gears <- as.factor(FE2013$Gears)
> MAKE<-as.character(levels(FE2013$Manufacturer))
> LIST <- as.list(rep(NA, length(MAKE)))
> names(LIST) <- MAKE

> for(i in levels(FE2013$Manufacturer)){
    temp <- subset(FE2013 , FE2013$Manufacturer==i)
    LIST[[i]] <- ggplot(data = temp, aes(x = FEcity, y =
       FEhighway)) +
    geom_point(aes(color = Gears)) +
    labs(title = paste("Manufacturer:",i), x = "Fuel Economy:
        City", y = "Fuel Economy: Highway ") +
    facet_wrap(~ Division) +
    if(nrow(temp) > 2 & nrow(temp) < 50) {
      geom_smooth(method = "lm")} else {
        if(nrow(temp) >= 50) {
          geom_smooth(method = "loess", span = 2 )}
    }
    pdf(file = paste("z:/", i, ".pdf", sep = ""), width=6,
       height=5)
    print(LIST[[i]])
    dev.off()
}
```

7.3 Other Loops

There are a few other types of loops and control flow operators. The `repeat` operator, simply repeats everything after it until you tell it to stop. It will loop until the lights go out. Like so:

```
> Number <- 1
> repeat{Number <- Number + 1; print(Number)}
>
```

To stop the looping simply hit Esc or Ctrl+C. Alternatively, you can tell R to stop loops via the break operator.

```
> repeat{Number <- runif(n = 1, min = 0, max = 1)
         print(Number)
         if(Number > 0.995) {break}
        }
>
```

Another useful control flow operator is while(). While loops are very similar to if() statements. Whereas the if() statement initiates some task if a condition is met, the while() operator will continue with some task as long as a condition is met. Note that if you set the condition to something that is always true, the while loop will not stop until the end of time.

```
> X <- 0
> while(X < 1000){
      X <- X + 2
      print(X)
      }
>
```

Another example:

```
> Y <- 0
> Count = 0
> while(Y < 13){ {
    S <- sample(size = 26, x = Students$FirstYears, replace =
        FALSE)
    Y <- sum(S)
    Count <- Count + 1
    cat("Sample:", S, "Number of First Years =", Y, "Trial:",
        Count, "\n")
 }
>
```

7.4 Functions

Writing functions in R is very easy. Whenever you want to run certain code over and over again, but don't want to source or paste things into the console repeatedly, you can and should create your own function with the `function()`. Functions take objects as arguments and return another object as output. Consider the following function:

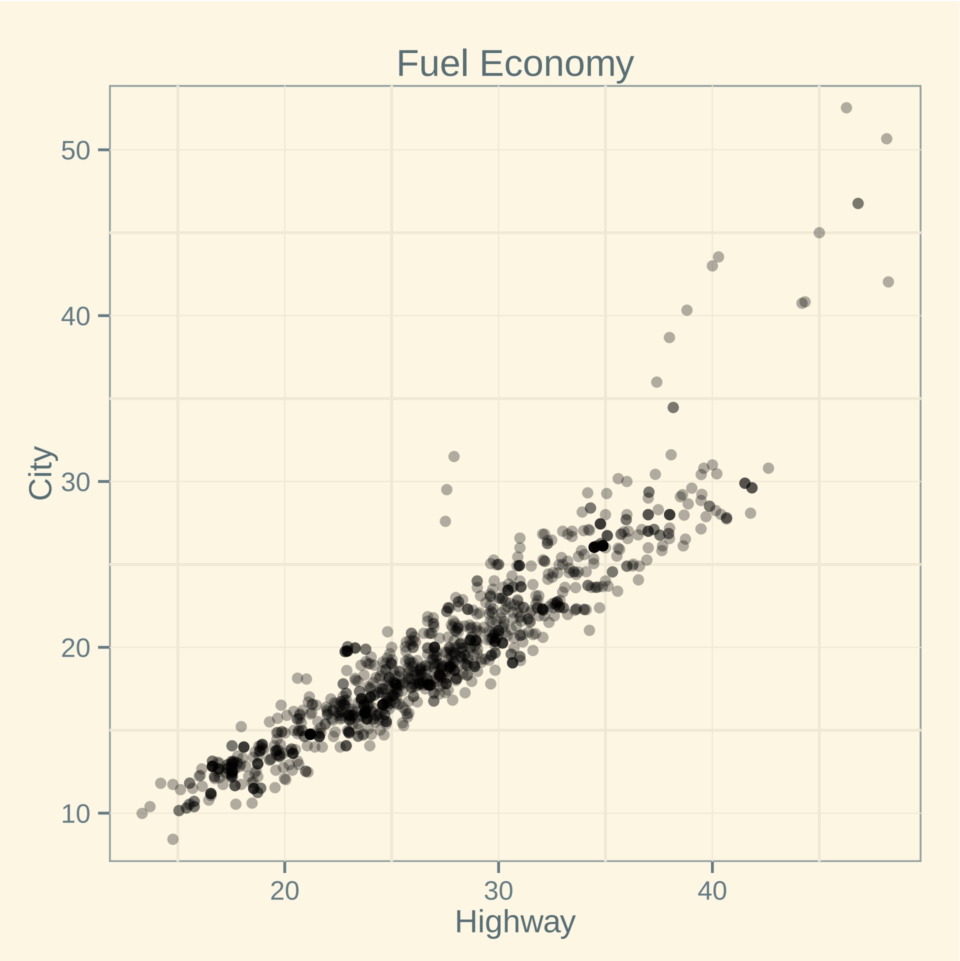
```
> MyFunction <- function(Object){
  Object + Object
  }

> MyFunction
function(Object){
 Object + Object
 }

> MyFunction(Object = 17)
[1] 34
```

You can wrap all sorts of functions, loops, and statements inside of a function to simplify repetitive tasks. You can specify as many parameters as you like. Let's make a function that makes a basic scatterplot. Note: the `require()` function will automatically load the `ggplot` package if needed.

```
> Scatterplot <- function(X, Y, Title, X.Axis, Y.Axis) {
    require(ggplot2)
    temp <- data.frame(X, Y)
    limits.X <- c(min(X) - 0.25*sd(X), max(X) + 0.25*sd(X))
    limits.Y <- c(min(Y) - 0.25*sd(Y), max(Y) + 0.25*sd(Y))
    ggplot(data = temp) +
      geom_point(aes(x = X, y = Y), alpha = 0.3) +
      labs(title = Title, x = X.Axis, y = Y.Axis) +
      coord_cartesian(xlim = limits.X , ylim = limits.Y)
  }
>
```

Let's take our function for a test drive. We can use the fuel economy data.

```
> Var1 <- FE2013$FEhighway
> Var2 <- FE2013$FEcity

> Scatterplot(X = Var1, Y = Var2, Title="Fuel Economy", X.
   Axis = "Highway", Y.Axis = "City")
>
```


Fuel Economy
City
50
40
30
20
10
20
30
40
Highway

Index of R Functions and Control Flow Operators

Functions are black and control flow operators blue.

Bibliography

Crawley, Michael J. 2007. *The R Book*. West Sussex, England: Wiley.

Venables, W. N. and D. M. Smith. 2012. *An Introduction to R – Notes on R: A Programming Environment for Data Analysis and Graphics*. R-Core Team.

Verzani, John. 2002. *simpleR – Using R for Introductory Statistics*.

Wickham, Hadley. 2009. *ggplot2: Elegant Graphics for Data Analysis*. New York, NY: Springer. **URL:** *http://had.co.nz/ggplot2/book*