Introduction to R

Ryan Womack*

May 24, 2011

*Data and Economics Librarian, Rutgers University, New Brunswick, NJ, rwomack@rci.rutgers.edu

Abstract

This is a guide to getting started in R. First, basic file commands and R conventions are reviewed. Second, elementary statistical operations are illustrated. Third, the use of graphics packages are demonstrated. Fourthly, data creation, import, and manipulation are discussed. Finally, the use of editors, GUIs, and other R functions are briefly presented.

This material was developed for the "Introduction to R" workshop, presented with Harrison Dekker, at the IASSIST Annual Conference, May 31, 2011, held in Vancouver, Canada.

1 Preliminaries

This document presents illustrative examples of R code. A file (Intropduction_to_R.R) containing the R commands necessary to reproduce the results is available on my personal home page. This may be useful for executing commands via cut-and-paste, or for review. This workshop presents a compressed version of material used in my three-part workshop series at Rutgers. If you are interested in reviewing the longer version, see my R Libguide.

There are many useful references to R, but this document in particular relies on the author's study of three books, *R for SAS and SPSS Users* (Mue09), *Data Manipulation with R* (Spe08), and *Introductory Statistics with R* (Dal08). In addition, the books *ggplot2: Elegant Graphics for Data Analysis* (Wic09) and *Lattice: Multivariate Data Analysis with R* (Sar08) provide comprehensive descriptions of their respective graphics packages, written by the authors of the packages themselves.

If you do not already have R, obtaining and installing it is easy. Since R is open source software, licensed under the GPL, you can use it freely for just about anything except creating closed source software. Information about R is available at the R project site, and the software itself is downloadable from CRAN, the Comprehensive R Archive Network, comprised of synchronized mirror sites around the world. Also, the freedom of open source means that you can install R in as many locations as you like: all of your public workstations, your web servers, your home machines, your netbook,USB drives, your friends' machines, ... you get the idea.

You can download Windows, Linux, or Mac versions. The Windows version of the `base` package is a self-contained executable containing all necessary files to get your R installation running. The `contrib` package contains additional modules, or *packages*, as they are known in R parlance. It is usually easier to download and install packages individually as you need them, as explained below.

In Linux, it is possible to install R from source, but it will usually be more convenient to wait for the latest version to be packaged for release in a major Linux distribution, such as Ubuntu or Fedora, and to download and install it using that distribution's tools. This simplifies the resolution of dependencies and staying current with updates. Rpms and .deb files are also available from CRAN, but may not always be in sync with the latest R version.

The Mac version is also available at CRAN as a downloadable package, although the author has little experience with it.

2 Getting Around in R

Once you have installed R, you are ready to run it: in Windows by clicking the R icon, or in Linux by simply typing R at the terminal prompt. Now you are presented with the most challenging part of your R experience, the empty command line. What to do?

You can operate R entirely from the command line, entering text in interactive mode. R is a full statistical programming environment, and some of its greatest power comes from writing longer programs that automate complex processes and can be run on demand. There are also optional GUIs that run on top of R, providing a feel closer to software such as SPSS, but navigating the command line is the fundamental way of interacting with the software. For now, let's try typing some commands.

There are a few basic commands that will help you navigate your workspace. First, let's find out where we are.

Type

```
> getwd()
```

```
[1] "/home/ryan/Desktop/2011"
```

This command will show you the default path for your R files.

Now type `getwd`, this time without the parentheses.

```
> getwd
```

```
function ()
.Internal(getwd())
<environment: namespace:base>
```

What you see now is the actual definition of the function in R. This is a nifty feature that gives you a clue to one of the primary characteristics of R. It is simple yet powerful at the same time. Typing any function name without its arguments will return the function itself. This becomes more interesting as you access functions created by other contributors in their packages, and can see exactly how their tools work. And you can use this functionality to easily modify existing functions and create your own. Any arguments to a function are enclosed within parentheses. With nothing inside the parentheses (), R will use the default values and settings for the function.

Note that R is case-sensitive so

```
> Getwd()
```

will not work.

If you type `getwd(` without closing the parentheses, you'll notice that the > symbol changes to a `+` and nothing else happens. Commands can be entered over multiple lines. Since the original command was not completed, the `+` indicates that R is waiting for you to type something to complete a command. This often happens if the final parenthesis is missed. Just type `)` to complete the command, and you'll be on your way.

We can perform mathematical operations in R with the usual operators.

```
> 2 + 2
```

```
[1] 4
```

We can also create and modify our own functions. For a simple and silly example of this, see the following:

```
> funkyadd <- function(x, y) {
+     x + y + 1
+ }
> funkyadd(2, 2)
```

```
[1] 5
```

The curly brackets enclose the actual functional expression. Once defined, the function can be reused just as though it were part of the base R system.

You can change the working directory by typing

```
> setwd("pathname")
```

Within R, lots of Unix conventions are used, so paths are specified with a single forward-slash separator, even on Windows systems. So `setwd("C:/Documents and Settings/username/My Documents")` would be used to point to the My Documents directory in Windows.

You can list the objects in your workspace with `ls()`, and remove them with `rm("objectname")`. Notice that we don't have much in our workspace yet, but we will after we have created some objects.

We have mentioned packages in passing already. Packages are add-ons or extensions to R's functionality. The base R installation will let you perform most basic statistical and graphing operations, but the real power of R lies in the over 3000 packages distributed on CRAN (the number of packages is expanding virtually exponentially). These packages provide ready-to-use implementations of all kinds of statistical methods, representing the latest research and specialized techniques. Because of R's open source nature, anyone can create a package, and once checked for basic quality standards, the package can be distributed worldwide on CRAN. This rapid adaptability is one reason for R's success in the research community.

In order to use a package, you must install it. Let's do this for some packages that we will need later.

```
> install.packages("Hmisc")
> install.packages("foreign")
> install.packages("ISwR")
> install.packages("gdata")
> install.packages("Rcmdr")
> install.packages("lattice")
> install.packages("ggplot2")
> update.packages()
```

R will ask us which mirror we want to use. Choose your favorite country (if you want speed, choose Canada!). The `dependencies=TRUE` option will check for other packages that your package needs and install those too. You probably want to do this, unless you are really fine-tuning your system or are a control freak! But for today we will leave it out just to save some setup time. The final call to update the packages is how you would maintain your system on a regular basis. This is much less time-consuming than the inital installation. Empty parentheses will update all installed packages.

R will automatically locate packages that have been officially accepted into CRAN without trouble at all. Also, since R is flexible and powerful, it is also relatively easy to create your own packages with your own custom functions and data included, which you might distribute locally. For local packages, you'd have to specify an explicit path to where R could find the package.

You can type `library()` to see all of the available packages that have been installed on your system. The `search()` command will show what has been actively loaded in your current environment.

To load a specific package, use the library command again, but with an argument this time. We can now see that it is loaded with `search()`. The order of packages listed by the search command is the order in which the packages will be searched when a command is entered.

```
> library()
> library("ISwR")
> search()
```

```
 [1] ".GlobalEnv"        "diamonds"          "cystfibr"
 [4] "package:foreign"   "diamonds"          "package:ggplot2"
 [7] "package:proto"     "package:grid"      "package:reshape"
[10] "package:plyr"      "package:lattice"   "cystfibr"
[13] "package:ISwR"      "package:stats"     "package:graphics"
[16] "package:grDevices" "package:utils"     "package:datasets"
[19] "package:methods"   "Autoloads"         "package:base"
```

To get help in R about a particular function or object, use the question mark. This will access a complete functional reference or description of an object. Try `?library` as an example. To launch the interactive help system, allowing you to search help, browse functions, and read manuals, type `help.start()`.

Finally, even though we haven't yet created any data or output worth saving, we can learn how to save and load our workspace. A simple `save("objectname", file="yourfilename")` command will save a single item from a workspace (for example, a matrix that you created). To save the entire workspace, use the `save.image` command. One of the very useful features of R is the ability to save not only data and output files, but to save all functions and intermediate objects created in the course of a session as part of the workspace. A complete workspace of this kind is usually saved with the extension `.RData`. The `load` command loads a previously saved object.

```
> save.image("mydata.RData")
> load("mydata.RData")
```

3 Statistical Functions

The data files used in this section are part of the "ISwR" package that accompanies *Introductory Statistics with R* (Dal08). This is the package we just loaded in the example above.

```
> library("ISwR")
> `?`(ISwR)

No documentation for 'ISwR' in specified packages and libraries:
you could try '??ISwR'

> data()
> library(help = ISwR)
```

Notice that ISwR doesn't have any associated help file, but we can see that there are many datasets included in it by executing the `data()` command or `library(help=packagename)`.

R can easily produce summary statistics. To illustrate this, let's load a particular dataset. We do this with the `data` command. We will use the `cystfibr` dataset from `ISwR`, which contains data on lung function in cystic fibrosis patients from a medical study. In a well-documented package, like `ISwR`, we can find this out by typing `?cystfibr`. This is another example of R's ability to package data and supporting data and documentation in a single, easy-to-distribute file.

Next we run the `summary` command to get a quick overview of the dataset.

```
> data(cystfibr)
> `?`(cystfibr)
> summary(cystfibr)

      age             sex            height          weight          bmp
 Min.   : 7.00   Min.   :0.00   Min.   :109.0   Min.   :12.9   Min.   :64.00
 1st Qu.:11.00   1st Qu.:0.00   1st Qu.:139.0   1st Qu.:25.1   1st Qu.:68.00
 Median :14.00   Median :0.00   Median :156.0   Median :37.2   Median :71.00
 Mean   :14.48   Mean   :0.44   Mean   :152.8   Mean   :38.4   Mean   :78.28
 3rd Qu.:17.00   3rd Qu.:1.00   3rd Qu.:174.0   3rd Qu.:51.1   3rd Qu.:90.00
 Max.   :23.00   Max.   :1.00   Max.   :180.0   Max.   :73.8   Max.   :97.00
      fev1             rv              frc             tlc           pemax
 Min.   :18.00   Min.   :158.0   Min.   :104.0   Min.   : 81   Min.   : 65.0
 1st Qu.:26.00   1st Qu.:188.0   1st Qu.:127.0   1st Qu.:101   1st Qu.: 85.0
 Median :33.00   Median :225.0   Median :139.0   Median :113   Median : 95.0
 Mean   :34.72   Mean   :255.2   Mean   :155.4   Mean   :114   Mean   :109.1
 3rd Qu.:44.00   3rd Qu.:305.0   3rd Qu.:183.0   3rd Qu.:128   3rd Qu.:130.0
 Max.   :57.00   Max.   :449.0   Max.   :268.0   Max.   :147   Max.   :195.0
```

It is easy to run basic descriptive statistics in R. For example, to get the mean of a variable like age in this dataset, just type `mean(cystfibr$age)`, using the `$` notation to refer to `age` in the `cystfibr` object. Note that if we type `mean(age)`, we get an error. Since R can handle multiple datasets and objects in its workspace at once, it does not make assumptions about what you might want. But to make things easier, we can use the `attach` function as follows to specify the default dataset. Now our shorthand works, and will continue to work until we `detach` the dataset.

```
> mean(cystfibr$age)

[1] 14.48

> attach(cystfibr)

The following object(s) are masked from 'cystfibr (position 4)':

    age, bmp, fev1, frc, height, pemax, rv, sex, tlc, weight
The following object(s) are masked from 'cystfibr (position 13)':

    age, bmp, fev1, frc, height, pemax, rv, sex, tlc, weight
The following object(s) are masked from 'package:ISwR':

    tlc

> mean(age)

[1] 14.48
```

There are also techniques for breaking down results by group. The `by` function is one easy way to do this. It takes three arguments: the dataset, the variable to group by, and the function to apply. So, the following will give us two separate summaries (0 is male, 1 is female):

```
> by(cystfibr, cystfibr["sex"], summary)

sex: 0
      age             sex        height          weight           bmp
 Min.   : 7.00   Min.   :0   Min.   :109.0   Min.   :13.10   Min.   :64.00
 1st Qu.: 9.75   1st Qu.:0   1st Qu.:134.0   1st Qu.:22.40   1st Qu.:68.25
 Median :15.50   Median :0   Median :165.5   Median :43.65   Median :78.50
 Mean   :15.21   Mean   :0   Mean   :155.9   Mean   :41.36   Mean   :79.71
 3rd Qu.:19.75   3rd Qu.:0   3rd Qu.:174.8   3rd Qu.:53.73   3rd Qu.:92.00
 Max.   :23.00   Max.   :0   Max.   :180.0   Max.   :73.80   Max.   :97.00
      fev1             rv             frc             tlc
 Min.   :22.00   Min.   :171.0   Min.   :104.0   Min.   : 95.0
 1st Qu.:33.25   1st Qu.:184.8   1st Qu.:127.8   1st Qu.:101.5
 Median :38.50   Median :215.0   Median :135.0   Median :106.0
 Mean   :39.86   Mean   :234.9   Mean   :148.4   Mean   :113.6
 3rd Qu.:48.00   3rd Qu.:249.8   3rd Qu.:153.2   3rd Qu.:125.5
 Max.   :57.00   Max.   :441.0   Max.   :268.0   Max.   :147.0
     pemax
 Min.   : 70.0
 1st Qu.: 95.0
 Median :100.0
 Mean   :117.5
 3rd Qu.:152.5
 Max.   :195.0
------------------------------------------------------------
```

```
sex: 1
      age              sex        height          weight          bmp
 Min.   : 7.00   Min.   :1   Min.   :112.0   Min.   :12.90   Min.   :65.00
 1st Qu.:11.50   1st Qu.:1   1st Qu.:144.5   1st Qu.:29.55   1st Qu.:67.50
 Median :14.00   Median :1   Median :153.0   Median :34.80   Median :70.00
 Mean   :13.55   Mean   :1   Mean   :148.8   Mean   :34.64   Mean   :76.45
 3rd Qu.:16.50   3rd Qu.:1   3rd Qu.:157.0   3rd Qu.:39.65   3rd Qu.:89.50
 Max.   :19.00   Max.   :1   Max.   :176.0   Max.   :60.10   Max.   :93.00
      fev1             rv           frc             tlc             pemax
 Min.   :18.00   Min.   :158   Min.   :118.0   Min.   : 81.0   Min.   : 65.00
 1st Qu.:22.00   1st Qu.:210   1st Qu.:126.5   1st Qu.:102.0   1st Qu.: 85.00
 Median :28.00   Median :253   Median :146.0   Median :120.0   Median : 90.00
 Mean   :28.18   Mean   :281   Mean   :164.3   Mean   :114.5   Mean   : 98.45
 3rd Qu.:30.00   3rd Qu.:337   3rd Qu.:194.5   3rd Qu.:127.0   3rd Qu.:115.00
 Max.   :45.00   Max.   :449   Max.   :245.0   Max.   :136.0   Max.   :134.00
```

A final basic descriptive function is the `table` function. With one argument, it simply reports frequences. Two arguments generates a cross-tabulation, and three or more arguments will represent output broken down into multiple tables.

```
> table(sex)

sex
 0  1
14 11

> table(age, sex)

    sex
age  0 1
  7  1 1
  8  2 1
  9  1 0
  11 0 1
  12 1 1
  13 1 1
  14 1 1
  15 0 1
  16 0 1
  17 2 2
  19 1 1
  20 1 0
  23 3 0

> table(height, age, sex)
```

3.1 Statistical Tests

Manipulating data, formatting tables, and tweaking graphics output can all be somewhat complicated in R, as shown elsewhere in these sessions. But, for the most part, the application of statistical tests is quite easy and straightforward. After all, this is what R was built to do. In most cases, all we need to do is find R's name for the function that applies a particular test. The function can be run without any parameters to produce sensible default output, or it can be tweaked via the specification of those parameters.

Let's try this for a one-sample t test. We won't go into any discussion of the statistics behind the t test or when it is valid. We'll just use it to illustrate how R works. The R function is simply `t.test`.

```
> t.test(pemax)

        One Sample t-test

data:  pemax
t = 16.3173, df = 24, p-value = 1.713e-14
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
  95.31792 122.92208
sample estimates:
mean of x
   109.12

> mean(pemax)

[1] 109.12

> t.test(pemax, mu = 100)

        One Sample t-test

data:  pemax
t = 1.3638, df = 24, p-value = 0.1853
alternative hypothesis: true mean is not equal to 100
95 percent confidence interval:
  95.31792 122.92208
sample estimates:
mean of x
   109.12
```

In this data, `pemax` is the "maximum expiratory pressure", a measure of lung capability. The default `t.test` checks whether `pemax` is significantly different than zero, which it clearly is in this case. Since the mean of `pemax` is 109.12, it might make more sense to test something closer to that. By specifing `mu`, we provide the value to test for. In this case, the true mean of `pemax` could be 100, since 100 lies within the confidence interval. An alternative method of specifying the t.test is to use the conf.level argument.

```
> t.test(pemax, mu = 90, conf.level = 0.99)

        One Sample t-test

data:  pemax
t = 2.8591, df = 24, p-value = 0.008651
alternative hypothesis: true mean is not equal to 90
99 percent confidence interval:
  90.4158 127.8242
sample estimates:
mean of x
   109.12
```

This tests whether 90 is within the 99% confidence interval for `pemax`, which it is not (just barely!). Other tests of this type work in the same way. You can use the Wilcoxon signed rank test, for example, with similar arguments, by using the R function `wilcox.test`. Or you can apply tests like Fisher's and χ^2 with commands like `fisher.test` and `chisq.test`.

It is easy to use the built-in help to inspect the parameter options for functions. Simply type `?functionname`, e.g., `?chisq.test`.

3.2 Regression

Regression is also easy to apply in R. Simple linear regression is handled by the `lm` function (think linear model). Again, use the ~ operator. The basic form is `lm(x~y)` where x is the dependent and y is the independent variable (x is a function of y). Let's try it out on `pemax` as a function of `tlc`, total lung capacity.

```
> lm(pemax ~ tlc)

Call:
lm(formula = pemax ~ tlc)

Coefficients:
(Intercept)          tlc
   149.9191      -0.3579
```

Note that the default output is scanty, providing us only with the coefficients of the equation. The rest of the information is there, waiting to be extracted. One way to do this is with the `summary` function. We can also easily store our regression output for further manipulation.

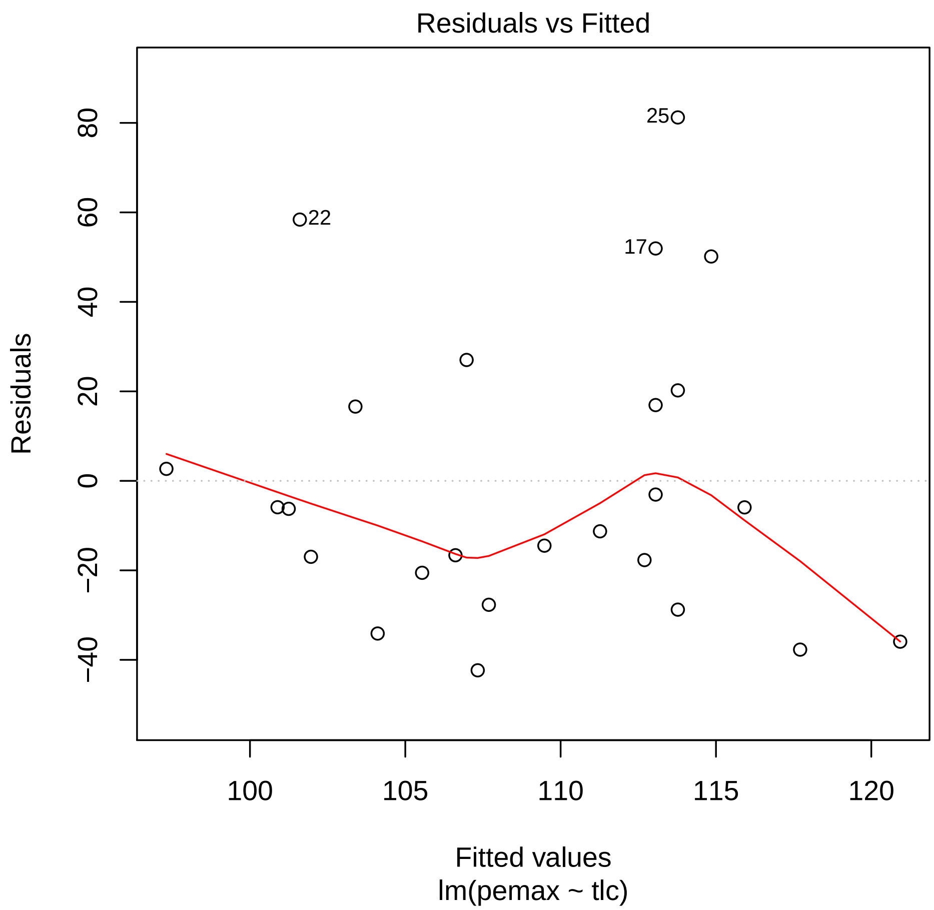
```
> summary(lm(pemax ~ tlc))

Call:
lm(formula = pemax ~ tlc)

Residuals:
    Min      1Q  Median      3Q     Max
-42.331 -20.541  -6.246  16.943  81.227

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 149.9191    46.5501   3.221  0.00379 **
tlc          -0.3579     0.4041  -0.886  0.38493
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 33.59 on 23 degrees of freedom
Multiple R-squared: 0.03298,        Adjusted R-squared: -0.00906
F-statistic: 0.7845 on 1 and 23 DF,  p-value: 0.3849

> regoutput <- lm(pemax ~ tlc)
> names(regoutput)

 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

> regoutput$residuals

         1          2          3          4          5          6          7
 -5.888596 -16.962258   2.690275 -20.541129 -17.698871 -27.688452 -42.330564
         8          9         10         11         12         13         14
 -3.056758 -34.109580  -6.246483  -5.919855 -16.614790 -11.267323 -37.709291
        15         16         17         18         19         20         21
 20.227467  27.027323  51.943242  16.606194  16.943242 -35.930275 -28.772533
        22         23         24         25
 58.395630  50.153806 -14.477887  81.227467
```

The `names` command shows us the component objects in the regression output, which can then be individual displayed or manipulated.

There are lots of other built in functions for working with regression output in R, like `predict`, `anova`, `cor.test`, and more.

We can also `plot` our regression output, which produces four default graphics: a plot of residuals vs. fitted output, a Q-Q plot, a scale-location plot, and a residuals vs. leverage plot.

```
> plot(regoutput)
```



Let's leave further discussion of graphics to the next section.

We can also perform multiple regression. This is as simple as adding additional arguments to the regression function, as follows:

```
> summary(lm(pemax ~ tlc + age + sex))

Call:
lm(formula = pemax ~ tlc + age + sex)

Residuals:
    Min      1Q  Median      3Q     Max 
-55.310 -12.248   6.415  17.037  47.612 

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  25.4115    54.3429   0.468  0.64488
tlc           0.2435     0.3704   0.657  0.51810
age           4.2342     1.2596   3.361  0.00295 **
sex         -12.1769    11.1098  -1.096  0.28547
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 27.13 on 21 degrees of freedom
Multiple R-squared: 0.4238,        Adjusted R-squared: 0.3415
F-statistic: 5.149 on 3 and 21 DF,  p-value: 0.007968
```

As you might expect, there are also methods for stepwise regression, logistic regression, and essentially any methodology you might care to apply. But we will stop here. This is, after all, only an introduction!

4 Graphics: three methods

Only a few of the graphs generated by the code are included in this PDF, in order to keep the file size down. Run the code on your own to view all of the examples.

Base R has the `graphics` package built in. This package is powerful enough to create many kinds of graphical output, but it requires the explicit passing of many parameters to accomplish tasks. You must state what you want to draw and where to draw it. Complex plots can be built up in layers using the results from several different programming steps. This can be an advantage or a disadvantage, depending on the situation. The creation of multi-panel comparison graphs using `graphics` can be cumbersome.

The `lattice` package makes the creation of multi-panel graphs easy. It also implements more sophisticated management of parameters and default settings, with the result that "pretty" output is usually easy to obtain.

The `ggplot2` package incorporates some features of lattice, and applies the Grammar of Graphics approach to creating graphics (Wil05). This approach breaks the creation of the graphic into conceptual parts. The language used to create the graphic on the computer mirrors the logical process of converting data to visualization. As applied in `ggplot2`, this approach leads to graphics that are easier to program and modify than the default approach. And yes, `ggplot2` is a revision of the original `ggplot`. Because the names of the graphics commands in each of these packages are different, it is possible to load them simultaneously and invoke selected commands from each package according to the task at hand.

This section will provide representative illustrations of these three techniques. Note that these are in no way exhaustive. R is noted for its power and flexibility in graphical output, and there are many other different packages and approaches to creating graphs. A good place to explore more is the R Graph Gallery at the *Addicted to R* website, which contains both sample graphs and R code examples.

4.1 Base `graphics`

For our initial examples, we will use the diamond dataset in ggplot2. Load it and take a look at the description of variables, including carat, cut, colour, clarity, and price. Note that this dataset has more than 50,000 observations. We will see that R handles this amount of data with ease.

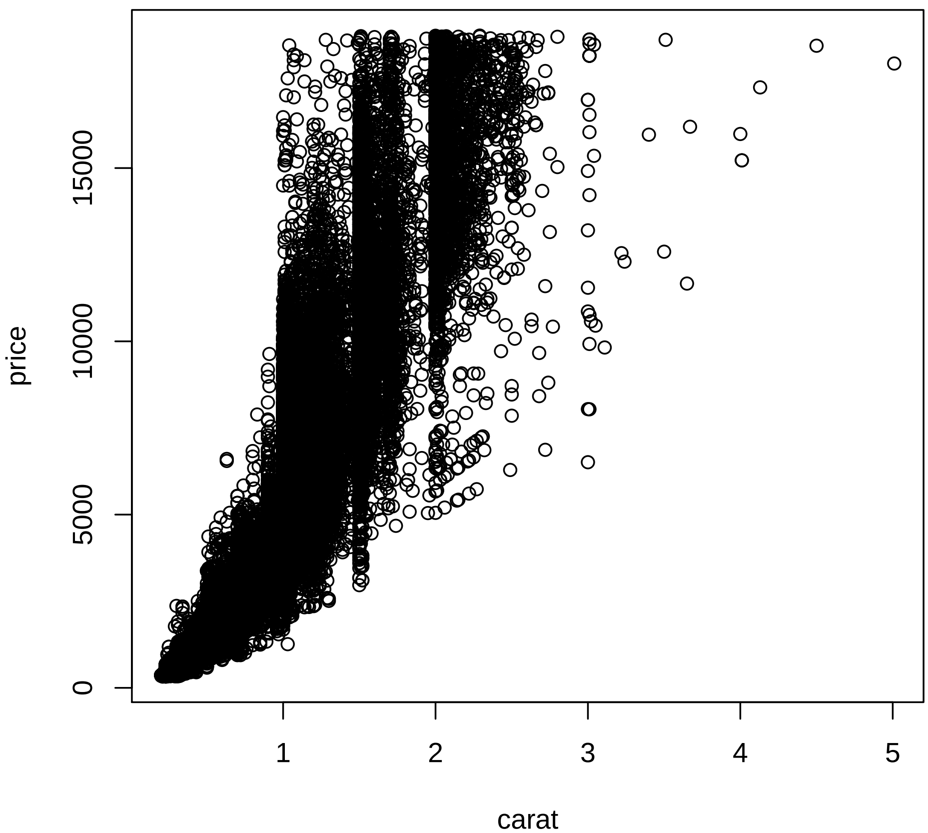
```
> library(lattice)
> library(ggplot2)
> data(diamonds)
> `?`(diamonds)
> attach(diamonds)

The following object(s) are masked from 'diamonds (position 4)':

    carat, clarity, color, cut, depth, price, table, x, y, z
```

```
The following object(s) are masked from 'diamonds (position 7)':

    carat, clarity, color, cut, depth, price, table, x, y, z
```

For convenience, we attached the dataset so that we do not have to worry about explicitly calling the dataset at all times.

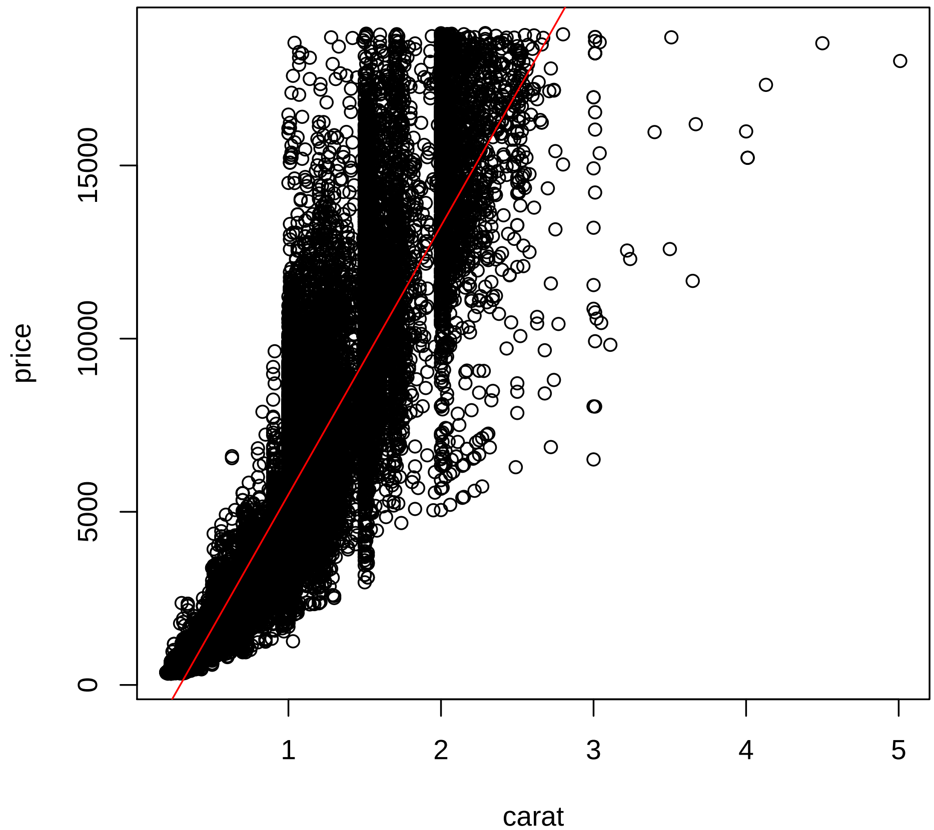
Now let's investigate our data by plotting a scatterplot. We check price versus carat (weight). In `graphics`, the command is simply `plot`.

```
> plot(price ~ carat)
```



If we want to plot our data, then draw a regression line on top, this is accomplished in two steps. The `abline` function draws a straight line. Note that we can include an R function (`lm`) as a argument to the `abline` function. Additional steps are layered on top of the original plot. If you make a mistake, you must start again: there is no way to erase a line.

```
> plot(price ~ carat)
> abline(lm(price ~ carat), col = "red")
```



If we want to dress up our graph by labeling axes, changing colors, and such, this can be accomplished by setting various parameters. See the following example:

```
> plot(price ~ carat, col = "steelblue", pch = 3, main = "Diamond Data",
+     xlab = "weight of diamond in carats", ylab = "price of diamond in dollars",
+     xlim = c(0, 3))
```

4.2 lattice

While the `lattice` package can replicate most of the graphs we have already seen, it excels at something that is difficult to do without manipulating data in base `graphics`: `lattice` makes it easy to produce multiple graphs based on grouping characteristics. For example, the scatterplot syntax in `lattice` is quite similar to base `graphics`, but we can use | to condition our graph on any variable. We can even get quite elaborate by also using the `groups` option, which plots points differently by group. It can even do this in three dimensions with `cloud`.

```
> xyplot(price ~ carat | clarity)
> xyplot(price ~ carat | cut, groups = clarity, auto.key = list(space = "right"))
> cloud(price ~ carat * table | clarity)
```

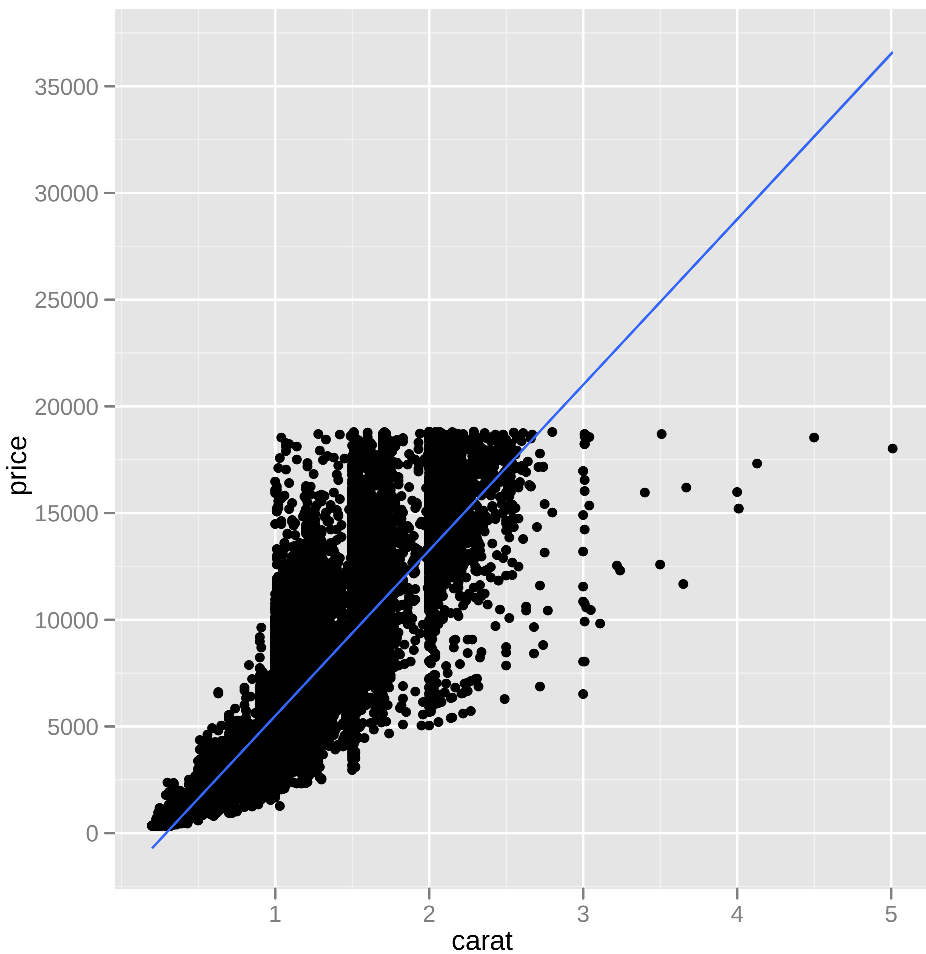
4.3 ggplot2

The `ggplot2` package is versatile, and can produce many kinds of graphs, including grouped output similar to `lattice`. It cannot, however, produce 3D plots at this time. Where `ggplot2` differs is in its unique syntax, that exploits the "Grammar of Graphics" to break commands into different conceptual components (aesthetics, geometries, scales, coordinates, options, etc.). This enables modifications to be easily made to one aspect of the graph without affecting the rest. This is a bit difficult to understand in the abstract, so let's look at some examples, starting with a bar chart. Here `aes` specifies the data set and x-axis variable and `geom_bar` tells `ggplot2` to draw a bar chart. Options that affect one component are enclosed in the parentheses for that part.

```
> ggplot(diamonds, aes(cut)) + geom_bar(position = "stack")
> ggplot(diamonds, aes(clarity)) + geom_bar(position = "stack")
> ggplot(diamonds, aes(clarity)) + facet_grid(. ~ cut) + geom_bar(position = "dodge")
```

Note the use of the `facet_grid` argument in the final example. And this example illustrates drawing a regression line:

```
> print(ggplot(diamonds, aes(carat, price)) + geom_point() + geom_smooth(method = lm))
```



For a fully "tweaked" graph in ggplot2, see the following example. Note the new `labs` and `opts` parameters being specified. Also, `geom_point()` is the way to specify a scatterplot in the general `ggplot` function.

```
> ggplot(diamonds, aes(carat, price)) + xlim(0, 3) + geom_point(colour = "steelblue",
+     pch = 3) + labs(x = "weight of diamond in carats", y = "price of diamond in dollars") +
+     opts(title = "Diamond Data")
```

4.4 Exporting graphical output

Creating presentation-ready graphs for insertion into documents is really quite easy in R. While the default graphical output is routed to the screen, it is simple to turn on a graphics device that routs output to a file. Any graphics calls will then create file output until this feature is explicitly turned off. The files can be PDF, PNG, JPG or other formats that can be easily inserted into documents. Also, if you are writing documents using `Sweave`, you can automatically include graphics output. This document was written using that capability in `Sweave`.

Here's how you would create PDF output:

```
> pdf(file = "output.pdf")
> xyplot(price ~ carat | clarity)
> xyplot(price ~ carat | cut, groups = clarity, auto.key = list(space = "right"))
> cloud(price ~ carat * table | clarity)
> dev.off()
```

```
X11cairo
       2
```

You will get a sometimes cryptic confirmation message that the graphics output device has been turned off. Once you have executed the code, check your home directory to see the PDFs there. Note that `ggplot2` also has an explicit `ggsave` function that can export a graphic in a single step.

That concludes this brief introduction to R graphics. Now let us turn to data manipulation techniques.

5 Data Manipulation

This section is all about getting data into R and transforming it so that it is suitable for analysis. As in other sections, we will give illustrative examples, but often skim over some of the details of the code. Again, the more detailed modules at the RutgersR Libguideprovide a lengthier explanation.

Let's create a small practice dataset to get our feet wet. Base R has a bare bones data editor and viewer. If you need more functionality in this area, some of the add-on GUIs and editors for R will let you do much more (RCmdr, Deducer, RKWard). In Windows, you can use the Data Editor from the menu. Or, we can start the data editor from the command line. To do this, we must first create the data object that we will edit. Here's how we do it:

```
> testdata <- data.frame()
> testdata
```

```
data frame with 0 columns and 0 rows
```

We are telling R to create an object called `testdata`. The arrow (composed by typing the less-than sign and a hyphen) is the "assignment" operator and assigns to `testdata` the value of whatever comes after the arrow. You can reverse the arrows direction, or use the equals sign. However, the arrow is preferred in R because there are a few situations where the equals sign is syntactically ambiguous. The equals sign is routinely used for setting parameter values in functions, as you can see from the above example. So, stick with the `<-` arrow for assignment, unless you find it too troublesome. Here, we set up `testdata` as a data frame.

By typing `testdata` at the end, we are actually issuing a command equivalent to `print(testdata).` Rather than typing `print(testdata)`, we just use the name as a shortcut.

A *data frame* is a special data construct in R, and is the closest equivalent to the typical rectangular dataset produced by SAS or SPSS typically used for social science data. In R, the columns are called vectors, variables, or just columns, while the rows are cases, observations, or just rows. But in R, the data frame is only one of many data types.

The `class` of an object describes its structure. A *vector* is a one-dimensional list of entries of the same mode, and can be of arbitrary length. R will automatically set the vector to the least restrictive mode.

So, `mytest<-c("M", 1, 3)` has mode "character". Other classes are *list*, a group of objects that can be of different modes; *matrix*, an object of dimension dim(x,y) whose elements must be of the same mode; and *array*, which is like a data frame but in higher dimensions. Now we can be more precise about a data frame too. The *data frame* is an object of dimension dim(x,y), whose elements can be of different modes, but whose rows all have the same length. Although this variety is initially a bit confusing, the ability to manipulate many different kinds of data objects in the R workspace is another source of R's power.

Let's fill in our data frame in the editor. Just experiment with typing in data, then close the editor. If you type `testdata` again, you will see the data you entered.

Essentially these data structures behave as you would expect, and R will gracefully handle many of the details automatically. Unlike other software, R does not mask these complexities entirely, and you will need to keep in mind that certain operations will only work with certain data structures. An understanding of data structures will help you to design and debug your R programs.

5.1 Importing and Converting Data

If the only way to get data was to type it in manually, we wouldn't get very far. Fortunately R provides convenient ways of importing data in a variety of formats.

The `read.table` command is the Swiss army knife of file importing in R, and can handle any kind of delimited file. In its simplest form, `read.table` needs only a filename as an argument. In the following examples, we access some sample data files using a complete URL in the `read` command.

```
> importdata <- read.table("http://www.rci.rutgers.edu/~rwomack/R/Spring2011/myfile.txt")
```

However, this will only work correctly if all of the default assumptions are met. R will correctly read a tab or space delimited file that has variable names in the header row and a first line with a length one shorter than subsequent lines (that is, a single blank cell in the upper left hand corner of the matrix). The default representation for a missing value is NA, and R will not correctly read SAS or SPSS files that use two consecutive tabs to represent a missing value.

We can enter parameters to the command to adjust for any unique characteristics of our data. For example,

```
> importdata2 <- read.table("http://www.rci.rutgers.edu/~rwomack/R/Spring2011/myfile2.txt",
+     header = TRUE, sep = ";", row.names = "id", na.strings = "..",
+     stringsAsFactors = FALSE)
```

This tells R to read a data file with a header row and semi-colon separated data. R will use the variable named "id" as the identifier for the rows in the R data frame. R will convert the .. characters in the original data to NA in R. We can also use parameters to force the reading of certain variables as characters or integers. Here we tell R not to convert strings into factors. In general, that is a good idea for things like names and addresses, where there is no real use for the "levels" that a factor representation of the variable would generate.

There are also functions `read.csv`, `read.delim`, `read.csv2` (for euro-style separators (; and . replacing , in numbers) with reasonable default values for typical files in these formats. There is no difference between these functions and `read.table`, except that using them saves the entry of several extra parameter options. Also, note that you can access data placed on the clipboard with `read.table("clipboard", header=T)`.

When you are ready to export data from your R environment, you can use `write.table`, `write.csv`, and so on. These functions have all the same parameters as their `read` versions, but create delimited tables.

R also has a package called `foreign` which eases working with other data formats, such as SAS, SPSS, and Stata.

Here is an example of importing a SAS file, then an SPSS file. The documentation for `foreign` explains further options. We are also detaching the package once we are finished using it, just to keep our workspace tidy and avoid potential function conflicts. Note that for the SAS file, we have to do some things differently in order to import the file correctly. This cannot be handled in one step. We use `download.file` to bring the file into the local workspace, setting the parameter `mode="wb"`, which indicates we are downloading a binary file. There is a warning when importing the SPSS file, since not all of the attributes of an SPSS file are specified in the simplified example file used here. But the data is imported correctly. Note how the SPSS import differs in structure from SAS and other formats.

```
> library(foreign)
> download.file("http://www.rci.rutgers.edu/~rwomack/R/Spring2011/mydata.xpt",
+     "mydata.xpt", mode = "wb")
> importdata3 <- read.xport("mydata.xpt")
> importdata4 <- read.spss("http://www.rci.rutgers.edu/~rwomack/R/Spring2011/mydata.sav")
> detach(package:foreign)
```

For further examples, you can try using the `foreign` package on any ICPSR dataset of your choosing, or try the Pew Foundations free SPSS data downloads.

Just as in the case of `read.table` and `write.table`, `foreign` allows you to write datasets in other formats using the `write.foreign` command.

Finally, for working with Excel files, there are additional packages; `xlsx` for Excel 2007 and later formats (XML-based .xlsx), and `xlsReadWrite` for earlier formats (.xls). Note that due to software limitations, `xlsReadWrite` is available for Windows only. The package `gdata` can import both .xlsx and .xls files, using the command `read.xls` for both file formats. The number in the `read` command refers to the sheet from the Excel spreadsheet that will be read to create the data frame. See "Reading Excel spreadsheets" for additional advice. The following example uses `gdata` to import two Excel files.

```
> library(gdata)
> importdata5 <- read.xls("http://www.rci.rutgers.edu/~rwomack/R/Spring2011/mydata.xlsx",
+     1)
> importdata6 <- read.xls("http://www.rci.rutgers.edu/~rwomack/R/Spring2011/mydata.xls",
+     1)
> detach(package:gdata)
```

5.2 A Few Data Manipulation Techniques

For these examples, we will use live data from the World Bank Open Datasite. For this workshop, we'll treat the initial download and processing of the data as a canned example and just execute the code. Again, consult the R Libguide for step-by-step explanations. We will use two of these datasets, *Gender Statistics* and the *Millenium Development Indicators* (MDI). In our example, we can think of a researcher who is interested in isolating variables associated with fertility and gender differences and comparing them with indicators of the availability of modern communications technology. For now, this researcher is only interested in the three most populous countries in the world: China, India, and the United States. We will create a customized data extract to meet these needs.

Note that for the *World Development Indicators*, there is a special R package, `WDI`, that simplifies the process of extracting data from this source. We will, however, use the generic methods in R that will work for any sort of data set.

```
> download.file("http://databank.worldbank.org/databank/download/Gender_Stats_csv.zip",
+     "Gender.zip")
> unzip("Gender.zip")
> genderstats <- read.csv("Gender_Stats_Data.csv")
> download.file("http://databank.worldbank.org/databank/download/MDG_csv.zip",
+     "MDI.zip")
> unzip("MDI.zip")
> MDstats <- read.csv("MDG_Data.csv")
```

5.2.1 `Subset` and `Select`

In order to create a subset of our data, use the `subset` command. We will use the logical operator $==$ to represent equality. The double equal distinguishes it from the use of $=$ as an assignment in R, and is a persistent source of typos in code! Other logical operators are and (&), or (|), not(!), and the usual >,<,>=,<=, and not equal to (!=). We'll create a subset of the gender data and a subset of MDI data, and assign those subsets to new objects as follows. The `table` command lets us check the results.

```
> gscountry <- subset(genderstats, Country_Name == "China" | Country_Name ==
+     "India" | Country_Name == "United States")
> MDcountry <- subset(MDstats, Country.Name == "China" | Country.Name ==
+     "India" | Country.Name == "United States")
> table(gscountry$Country_Name)
> table(MDcountry$Country.Name)
```

Notice that since all of the levels are inherited from the parent object, the names of all other countries are still present, but with 0 data elements. In general, this won't cause any harm.

Now we will further refine our subsets by selecting a few variables of interest, while at the same time limiting the number of years that we want the data for. Using `subset` in combination with `select` allows us to do both at once.

```
> myMDI <- subset(MDcountry, Series.Name == "Mobile cellular subscriptions (per 100 people)" |
+     Series.Name == "Internet users (per 100 people)", select = c(Country.Name,
+     Series.Name, X2000:X2008))
> myMDI

      Country.Name                                    Series.Name      X2000
3458         China Mobile cellular subscriptions (per 100 people)  6.7524918
3511         India Mobile cellular subscriptions (per 100 people)  0.3521030
3628 United States Mobile cellular subscriptions (per 100 people) 38.7983329
3897         China                Internet users (per 100 people)  1.7819736
3950         India                Internet users (per 100 people)  0.5413796
4066 United States                Internet users (per 100 people) 43.9448280
          X2001     X2002     X2003     X2004     X2005     X2006     X2007
3458 11.3865629 16.089112 20.952576 25.833690 30.175652 35.167883 41.529116
3511  0.6334303  1.239700  3.165168  4.836434  8.235100 14.962005 20.770155
3628 45.0747504 49.269642 55.329868 63.068258 72.019557 80.979795 87.207374
3897  2.6496835  4.615745  6.170444  7.252667  8.579043 10.601042 16.130450
3950  0.6779836  1.581094  1.736192  2.037563  2.466693  2.901395  4.089663
4066 50.0989217 60.052769 63.100013 66.255455 69.573900 70.571101 73.520801
         X2008
3458 48.407322
3511 30.429882
3628 88.870637
3897 22.496424
3950  4.539613
4066 75.771663
```

Let's move on to the Gender data. We'll choose a few more variables here, relating to fertility and education. The technique is identical to the first statement.

```
> mygender <- subset(gscountry, Indicator_name == "Expected years of schooling, female" |
+     Indicator_name == "Expected years of schooling, male" | Indicator_name ==
+     "Adolescent fertility rate (births per 1,000 women ages 15-19)" |
+     Indicator_name == "Fertility rate, total (births per woman)",
+     select = c(Country_Name, Indicator_name, X2000:X2008))
> mygender

       Country_Name
4312          China
```

```
4313          China
4354          China
4364          China
10691         India
10692         India
10746         India
10756         India
24705 United States
24706 United States
24783 United States
24793 United States
                                                    Indicator_name    X2000
4312                            Expected years of schooling, female       NA
4313                              Expected years of schooling, male       NA
4354  Adolescent fertility rate (births per 1,000 women ages 15-19)  9.94550
4364                       Fertility rate, total (births per woman)  1.76700
10691                           Expected years of schooling, female  7.29924
10692                             Expected years of schooling, male  9.38709
10746 Adolescent fertility rate (births per 1,000 women ages 15-19) 87.89300
10756                      Fertility rate, total (births per woman)  3.28000
24705                           Expected years of schooling, female 15.67437
24706                             Expected years of schooling, male 14.76851
24783 Adolescent fertility rate (births per 1,000 women ages 15-19) 47.64300
24793                      Fertility rate, total (births per woman)  2.05600
         X2001    X2002    X2003    X2004    X2005    X2006    X2007    X2008
4312        NA       NA 10.26641       NA       NA 11.03888 11.34453 11.63204
4313        NA       NA 10.37785       NA       NA 10.77219 10.98305 11.15502
4354   9.88850  9.83150  9.79630  9.78290  9.76950  9.75610  9.74270  9.74330
4364   1.76200  1.75900  1.75800  1.75900  1.75900  1.76100  1.76200  1.76500
10691  7.34272  7.59879  8.54005  9.05620  9.32073  9.42251  9.79339       NA
10692  9.35196  9.45295  9.71347 10.27065 10.50751 10.62709 10.85564       NA
10746 84.48300 81.07300 78.24530 75.99990 73.75450 71.50910 69.26370 67.11510
10756  3.21000  3.14000  3.07000  3.00000  2.93000  2.86300  2.79800  2.73800
24705 15.90585 16.00107 16.22459 16.24171 16.37938 16.37430 16.49133 16.58228
24706 14.86189 14.88676 14.99040 14.98156 14.95467 14.93431 15.02104 15.14060
24783 45.81900 43.99500 42.36290 40.92270 39.48250 38.04230 36.60210 34.95750
24793  2.03400  2.01300       NA  2.04500  2.05400  2.10000  2.11320  2.10000
```

5.2.2 Merge

Now we'd like to combine the two datasets. To do this is quite simple. There are two functions, `rbind` (for row bind) and `cbind` (for column bind), that allow you to quickly paste together data objects. R is generally pretty good about matching observations and variables. Since we have similar variables in the columns, we simply want to add together our observations.

There is also another way to combine data frames, using `merge`. The `merge` function does more robust checking to make sure that data frames align, so it is preferable to `rbind` or `cbind` in more complex situations. The `all=TRUE` option is necessary to include all observations from each data frame. Without this, `merge` would select only observations whose variable values matched between the two data frames (an empty set in this case). Here we reverse the order of the combination so that we can see the change in the output [not displayed in the text]. Note that we have an additional `names` step to fix the issue with variable names that do not quite match between our two data sets.

```
> names(mygender) <- c("Country.Name", "Series.Name", "X2000",
+     "X2001", "X2002", "X2003", "X2004", "X2005", "X2006", "X2007",
+     "X2008")
```

```
> mydata <- merge(mygender, myMDI, all = TRUE)
> mydata
```

5.2.3 Split

Another useful technique is to `split` the data. Any grouping variable can be used to separate a data frame into separate subframes. So, if we wanted to be able to easily access all of the data for one country, we could do the following:

```
> mysplit <- split(mydata, mydata$Country.Name, drop = TRUE)
> mysplit$China
```

```
  Country.Name                                                    Series.Name
1        China Adolescent fertility rate (births per 1,000 women ages 15-19)
2        China                            Expected years of schooling, female
3        China                              Expected years of schooling, male
4        China                        Fertility rate, total (births per woman)
5        China                                Internet users (per 100 people)
6        China                    Mobile cellular subscriptions (per 100 people)
     X2000     X2001     X2002     X2003     X2004     X2005    X2006    X2007
1 9.945500  9.888500  9.831500  9.796300  9.782900  9.769500  9.75610  9.74270
2       NA        NA        NA 10.266410        NA        NA 11.03888 11.34453
3       NA        NA        NA 10.377850        NA        NA 10.77219 10.98305
4 1.767000  1.762000  1.759000  1.758000  1.759000  1.759000  1.76100  1.76200
5 1.781974  2.649684  4.615745  6.170444  7.252667  8.579043 10.60104 16.13045
6 6.752492 11.386563 16.089112 20.952576 25.833690 30.175652 35.16788 41.52912
     X2008
1  9.74330
2 11.63204
3 11.15502
4  1.76500
5 22.49642
6 48.40732
```

The arguments to `split` are the data frame, the grouping variable or variables, and options. In this case, we dropped all of the countries for which there are no data from the `mysplit` data frame using the `drop=TRUE` argument.

5.3 Exporting output and saving the workspace

We have accomplished our goal of creating a useful extract from the World Bank data. We'll export the mydata file in .csv and SPSS format, using the obverse of the read functions used at the beginning:

```
> write.csv(mydata, "mydata.csv")
> library(foreign)
> write.foreign(mydata, datafile = "mydata.sav", codefile = "mydata.sps",
+     package = "SPSS")
```

And finally, save the workspace as an RData file. This stores all of the data structures that we have created in this session.

```
> save.image("mydata.RData")
```

Now you can truly say that you can handle data in R!

6 More R Exploration (GUIs, Editors, Integration, Reproducible Research)

After this introduction to R, you can explore the links at theRutgers Libguide for R to find more R tutorials, guides to finding packages, ways to get R help and news, and download books on R topics. Also, remember that in the beginning, we downloaded the package `Rcmdr`. `Rcmdr` is a general purpose graphical user interface. You can launch it by invoking `library(Rcmdr).` Other GUIs for R include `JGR, RStudio, rattle, Deducer`, and more.

If you become a regular R user, you will probably find it beneficial to choose an editing environment that supports your R programming. Many text editors will support syntax highlighting, and it is easy enough to write and then cut and paste into your R session. On Linux systems, the ESS extension to Emacs is often recommended (if you are willing to deal with Emacs). I have found RKWard works well as an R editing environment on Linux. It is clean, easy, and does not require learning additional commands. On Windows systems, Tinn-R fulfills the same function as a purpose-built R editor. Others may prefer incorporating R into an environment they use for other programming, such as Eclipse (with StatET). RStudio is a newly released R development environment, but is very slick and has become quite popular.

The beauty of R is that it is open source and open-ended, so you can choose any approach that gets the job done for you. R has packages that integrate with websites, databases, and document publishing. For example, these handouts and associated R scripts were prepared using `Sweave`. `Sweave` allows R code to be inserted into a document. The code is then executed and evaluated, with results inserted into the final document. This is a great time saver, and allows the incorporation of graphics and constantly updated data into reports. Most importantly, it allows for *reproducible research*. The code needed to create the statistical output is always present, and the data is always pulled directly from the source, eliminating any typos or ambiguities. `Sweave` in fact requires that the code run without errors. And the fact that R is open source means that any reader of a research paper can obtain the software necessary to run the analysis for themselves.

References

[Dal08] Dalgaard, Peter, *Introductory Statistics with R*, 2nd ed., Statistics and Computing, Springer, 2008.

[Mue09] Muenchen, Robert A., *R for SAS and SPSS Users*, Statistics and Computing, Springer, 2009.

[Sar08] Sarkar, Deepayan, *Lattice: Multivariate Data Visualization with R*, Use R!, Springer, 2008.

[Spe08] Spector, Phil, *Data Manipulation with R*, Use R!, Springer, 2008.

[Wic09] Wickham, Hadley, *ggplot2: Elegant Graphics for Data Analysis*, Use R!, Springer, 2009.

[Wil05] Wilkinson, Leland, *The Grammar of Graphics*, 2nd ed., Statistics and Computing, Springer, 2005.