

# The Very Basics of R (Windows)



**Statistical Computing Group @ Research Data Services  
University of Pennsylvania  
Last modified: 06/13/2008**

This online workshop is, as titled, about VERY BASICS. It's an overview designed to help you get started with R. If you:

- (1) have heard about R once or twice before but are wondering what it is anyway, or
- (2) are interested in R but have never touched it before, or
- (3) are trying/wanting to learn R, have it installed, have launched the program once or twice, but don't know where to go from there or what to do,

...then this is probably a good workshop for you.

Indeed, when we are beginners, it is very often just one of those “easy little things” that seem to take us forever before we (feeling exhausted) finally reach the stage of analysis that we are really interested in. The goal of this workshop is just to help make that first stage smoother.

R is very different from commercial statistical packages, such as SAS, Stata, and SPSS, in its mindset. As such, you need to learn a little different attitude from those of your familiar stats packages. I will make as much comparison as possible along the way so you can see similarities and differences among them and get some good idea what of SAS, Stata or SPSS you can/should not carry over into learning R.

Click [here](#) for how to obtain and install R and its libraries, and click [here](#) for how to use R-help, and what resources there to be.

## **Contents**

1. What's R, anyway? .....	2
2. Getting started.....	2
3. Assignments, Objects, and Functions .....	5
4. Data Entry and Attach and Detach.....	8
5. Save and Load R Objects & Save R Data in Text Format.....	20
6. Indexing and Conditional Selection.....	24
7. Sorting Cases .....	30
8. Factors: How R Handles Categorical Information.....	33
9. Recoding .....	42
10. Stacking (Appending) And Merging Data Frames .....	45

## 1. What's R, anyway?

This question can also be put another way, like “Is it for me?” or “When would I need it?”

[The official description on the R-Project site](#) is good. As it says there, it is “a language and environment for statistical computing and graphics.” For this reason, R is quite different from widely used standard commercial statistical packages, such as SAS, Stata, and SPSS.\* R is not so much a statistical package as it is a statistical programming environment.

\* For more comparison among different statistical computing systems, see [John Marcotte's presentation](#).

As such, R has both pros and cons, like:

### **PROs**

- Similar functionalities to those of Insightful Corporation's S-Plus... and it is **FREE!**
- Very flexible, powerful programming environment built around “objects” (more later).
- A wide variety of strong free packages added by third parties to R's functionality.
- Highly interactive.
- [Extensive graphics capabilities](#)

### **CONs**

- It's not easy; quite a bit of learning curve.
- It's not easy; users need to be familiar with matrix manipulation to fully benefit from R's programming flexibility.
- And so a big tradeoff between free of charge in dollars and costly in time and energy to learn and solve problems (free means no customer support, and no user-friendly documentation).

It really depends on what analysis you want to perform, what data you want to work on, and whether R offers you solutions to what you cannot accomplish in SAS, Stata, or SPSS. Typically, most of what people want to do in social science research can be done with much greater ease by using standard statistics packages, such as SAS, Stata, and SPSS. There are some procedures out there, however, that those standard statistical packages cannot implement but flexible programming environments can handle, and that is where R really fits in. In other words, there would otherwise be few strong reasons for most of us to take R over those standard packages.

## 2. Getting started

Let's get our feet wet, slowly...

First, launch R by clicking on the R shortcut or selecting R 2.6.0 from the Start Menu.  
**Start > Programs > R > R 2.6.0 (or whatever version you have)**

Observe your **R Console**.

Now, let's start working with R. It's always a good idea to have one working directory for one project. SAS users use `-libname-` and Stata users use `-cd-` to specify/change working directories. In R, we can use either its pull-down menu or command to do the equivalent. First, create a directory in your home directory or wherever you want it to be. Let's call it `mywd` (my working directory) for now. Note the path to this directory. Then set it to your current directory in R.

From the pull-down menu of the R Console, go **File > Change dir...**,

OR,

From the command line of your R Console (following `>`), type in the followings (hit enter to execute each line).

```
> ## Set up working directory.
> setwd("C:/mywd")
>
> ## Make sure where we are.
> getwd()
[1] "C:/mywd"
>
```

Note that in R, **use the / (slash)** instead of `\` (backslash) when specifying your directory/filename. This is the case even if you are working in the Windows environment. `setwd("working directly here")` lets you set up your directory. To know where you are working now, type in `getwd()` and R tells you. It's equivalent to Stata's `-pwd-`.

Now, how can we tell R what you want it to do? There are two approaches.

- **Command line approach** (similar to Stata's interactive command window)
- **Script file approach** (similar to Stata `.do` file, SAS `.sas` file, SPSS `.sps` file)

**Some important note, before moving on:**

- Expressions and commands are **case-sensitive**.
- You can add comments following `#`.
- The `.` (period) in a name is treated just as a character. For example, you can have a variable name `"pre.test," "post.test,"` etc.

Let's try **command line approach**. Just for warming-up, type in `"blah blah,"` hit Enter, and see what happens.

```
> ## Just play around.
> blah blah
Error: syntax error, unexpected SYMBOL, expecting '\n' or ';' in "blah blah"
>
```

After issuing an error (of course!), R prompts you to enter a new command (>). Suppose you still want to keep trying “blah blah.” Now don’t type it again. Instead, hit the up-arrow key of your keyboard, and you can see R recalls the previous command for you. You can scroll through the previous commands by using the up/down arrow keys, so you don’t have to retype the same command yourself.

Now, you give up on “blah blah,” and type in 3+3, and what appears on the screen is:

```
> 3+3
[1] 6
>
```

This way you can use R as a calculator (just like you do by -display- in Stata). The [1] is an index saying “first requested element will follow” (did you notice that we’ve already seen this when we did `getwd()` above?). Try typing in “`runif(8)`”.

```
> runif(8)
[1] 0.86747941 0.76475565 0.17968006 0.63989202 0.01918572
[6] 0.75834585 0.03121560 0.03209969
>
```

R says that `0.86747941` is the 1<sup>st</sup> and `0.75834585` is the 6<sup>th</sup> element of the requested list (your numbers are most certainly different; `runif()` generates random numbers).

You can use the command line interactively for data check and exploration like this. But in actual research projects, you often need to keep your scripting work for documentation and replication purposes, like you probably are doing in .do files (Stata), Enhanced/Program editor files (SAS), or .sps files (SPSS) already. R has the **Script File** that you can use for this purpose. From the pull-down menu of the R Console, go **File > New Script** to open a new script file (to load an existing script file, **File > Open Script**). Let’s save it as “introR” for now.

Now with the script window active, take a look at the five icons at the upper left. The third one, “Run line or selection,” is to run your script.



Notice that it says “Run line or selection.” Let’s play with it to see how it works.

In your script file, simply type  $4*5$  (or whatever) and hit this “Run line or selection” icon.  
Then 1928-854 (or whatever), and hit this icon.  
Put the pointer back to the first line, and hit this icon.  
Select (highlight) both the lines, and hit this icon.

So, exactly what it says. R runs the line where the cursor is (no need to select the line by highlighting), OR the whole highlighted (selected) portion of your script.

Alternatively, you can use the Ctrl + R keys, instead of hitting the “Run line or selection” button, to execute your scrip.

### 3. Assignments, Objects, and Functions

Now, let’s talk about

- **Assignments**
- **Objects**
- **Functions**

You may need to take a little new attitude about them down the line (though there are some similarities to macros in Stata and SAS).

#### Assignments

R allows for symbolic variables to represent values, so that you don’t have to retype again and again. Let’s take a look at a simple example. Type this in.

```
> ## Assignments and Objects Basics
>
> x<-3+3
>
```

Here, **<-** is called the **assignment operator**, meaning “**is assigned to**” or “**is defined as.**” You don’t see any immediately visible result now, but now R knows that x represents the value  $3+3=6$ , and you can use it in subsequent expressions. Following the above, type in  $x + x$  and see what you get.

```
> ## Assignments and Objects Basics
>
> x<-3+3
> x+x
[1] 12
>
```

## Objects

Now, in R, this “x” above is called an “**object.**” R creates objects to represent values (including vectors, computation results, functions, matrices, and data structure) whenever a name is assigned to them (`<-`) and store them in your *workspace* (the memory space in R where your objects are all stored).

One of the core ideas with **R is it is an object-driven system**, that is, **ALL** R entities (values, functions and data structures) exist as objects, and analysis in R is structured around them. This idea should be clearer and clearer as we move on. For now, keep in mind the format is:

*object.name <- values* (functions, data, etc).

One (important) tip. You can name your objects whatever you want, but **never use command names as object names!** (The reason will be clear to you as you go along, but suffice it to say for now that it would invite unnecessary confusion).

## Functions

Many things in R are done by using **functions**. The basic usage is:

*function.name()*.

Example: We just now mentioned above about the storage space in R called *workspace*. Try the function **ls()** (abbrev of **list**) to see what are stored in your **workspace**. You should find the object “x” we created above is there. Then, **remove** this object from your workplace by typing in **rm(x)** and hit Enter.

```
> ## Example: Functions
>
> # See that the object just created is stored in my workspace.
> ls()
[1] "x"
> # Remove the object "x" from my workspace.
> rm(x)
> # Now the workspace should be empty.
> ls()
character(0)
>
```

You can remove all the objects in the workspace by typing **rm(list=ls())**.

Let’s see another example to understand how assignments, R objects, and functions work. Suppose we have a small weight (lb) and height (ft/in) data for 5 people. Suppose also that using this information, we want to calculate Body Mass Index (BMI).

Here's the weight/height data we have.

Height (ft/in)	Weight (lb)
5'3''	102
6'0''	124
5'9''	138
6'4''	171
6'8''	246

Obvious first step is to read the weight and height information into R. To do this, we use the **c()** **function** (note **c** is lower-case), c as in combine or concatenate. This is probably one of the most frequently used functions in R. With **c()**, R can create a vector or a list in various situations, and further, R can handle such data vectors as **objects** (this is one strength of R, which should be clearer to you as you go on).

Type the following in and run.

```
> ## Example: Assignments, Objects and Functions.
>
> # data vector for height (ft).
> hgtft<-c(5,6,5,6,6)
> # data vector for height (in).
> hgtin<-c(3,0,9,4,8)
>
> hgtft
[1] 5 6 5 6 6
> hgtin
[1] 3 0 9 4 8
>
```

Here, I assign the object names “hgtft” and “hgtin” to the data vectors of the five people’s height in feet/inch. Then I display the contents of the **objects** “hgtft” and “hgtin” to check.

Let’s continue and do the same for the weight data.

```
> # data vector for weight.
> wgtlb<-c(102,124,138,171,246)
>
> # What's the 3rd person's weight?
> wgtlb[3]
[1] 138
>
> # Now the three data vectors should be in my workspace, check.
> ls()
[1] "hgtft" "hgtin" "wgtlb"
>
```

Here, we first assigned the object name “wgtlb” to the data vector of the same people’s weight in pound. Then, instead of listing all the vector elements, we asked R to displaying the 3rd observation by using brackets **[3]** following the object name “wgtlb.” This data selection by

using brackets is called **indexing**. In R, you can **index a vector, matrix, array, and data frame** with `[ ]` (we'll talk about data frame soon).

We also use `ls()` to tell R to list those vectors now stored in the workspace as objects.

Now, using those data vector objects, let's calculate BMI (formula:  $[\text{weight}/\text{height}^2]*703$ ) where weight is in pound and height is inch.

Let's do this step by step for the sake of practice:

- First, because our height data is measured as feet/inch, we will convert it into inch.
- Then use this  $[\text{weight}/\text{height}^2]*703$  formula on the converted data vectors.

```
> # First Step: Data vector for hight in inch (1ft = 12in).
> hgtinch<-12*hgtft+hgtin
> hgtinch
[1] 63 72 69 76 80
>
```

Notice that the operation (multiplication in this example) was carried out element-wise, i.e., each element of the vector "hgtft" was multiplied by 12, and the summation was of the two corresponding elements from the two vectors. Also notice that rather than just display the converted result, I stored it in the workspace as a new data vector object, which I named "hgtinch." This object is available for use for the second step calculation.

```
> # Second Step: Get BMI
> bmi<-(wgtlb/hgtinch^2)*703
> bmi
[1] 18.06652 16.81559 20.37681 20.81250 27.02156
> ls()
[1] "bmi"      "hgtft"   "hgtin"   "hgtinch" "wgtlb"
```

(How can we understand these numbers? [This site](#) is for those of you who are interested).

Let's explore the data just a bit by using some basic functions.

```
> # What's the mean value of those 5 guys' BMI?
> sum(bmi)/length(bmi)
[1] 20.61860
> # Of course this is the same as
> mean(bmi)
[1] 20.61860
>
```

## 4. Data Entry and Attach and Detach

We just saw how R handles data vectors and how we can use them for calculations. But for our data management and analysis work, we more often use data set as an array of columns of data. In R, what in other statistical packages would be called data set is called **data frame**. Again, **R stores a data frame as an object in your workspace**.

In this section, we will learn:

- **Data Entry**
  - How to manually enter data to create an R data frame.
  - How to read data from external text files (.txt, .csv, etc) and data files from some of the widely used statistical packages, such as Stata, SAS, and SPSS (.dta, .xport, .sps) to create an R data frame.
- **Attach and Detach Data Frame**

## Data Entry

### Data Entry, Manually:

The function **data.frame()** lets you input data vectors directly at the command line or a script file and create a data frame. Let's put our data vector objects (wgtlb, hgtinch, and BMI) together to create a data frame. Let's add a temporary id variable there, too.

```
> ## Create a data frame by manually entering data.
>
> wgtlgbt<-data.frame(id=c("A","B","C","D","E"),wgtlb,hgtinch,bmi)
> wgtlgbt
  id wgtlb hgtinch   bmi
1  A   102     63 18.06652
2  B   124     72 16.81559
3  C   138     69 20.37681
4  D   171     76 20.81250
5  E   246     80 27.02156
>
> ls()
[1] "bmi"      "hgtft"    "hgtin"    "hgtinch"  "wgtlgbt"  "wgtlb"
```

In the above, I first assigned an object name “wgtlgbt” to a new data frame to be created. Then using the function **data.frame()**, I arrayed the three data vectors and a new data vector named “id” to create a data frame “wgtlgbt.” Then display the data frame I just created. You can see the new data frame is stored in the workspace as an object.

Suppose we don't have the weight, height, and bmi information as objects in the workspace yet. From the way I use the **c()** function above to add the id variable, you may have already seen you can also do the below to create the same data frame.

```
> wgtlgbt<-data.frame(id=c("A","B","C","D","E"),
+ wgtlb=c(102,124,138,171,246),
+ hgtinch=c(63,72,69,76,80),
+ bmi=c(18.06652,16.81559,20.37681,20.8125,27.02156))
```

```

>
> wgtght
  id wgtlb hgtinch    bmi
1  A  102     63 18.06652
2  B  124     72 16.81559
3  C  138     69 20.37681
4  D  171     76 20.81250
5  E  246     80 27.02156
>

```

The `+` sign above is called the **continuation prompt**, indicating that R realizes you haven't written a complete expression yet and prompts you to type its remainder (hence R is not ready for you to type in a new command!). In the above case, I inserted returns at the end of each the variables just for an easy view of the whole expression in the script file, without completing my command expression. Continue typing in your unfinished expression. When/if this happens inadvertently, get out by pressing **ESC**.

After you have created a weight and height data frame, perhaps you might need to make small changes to it. One way would be to use the function `edit()` to bring up the **Data Editor** (a spreadsheet-like environment). Examine the following.

```

> ## How to edit the data frame?
>
> # Keep "wgtght" intact, assign new object name to the edited one.
> wgtght.new<-edit(wgtght)
>
> # If I simply want to make some changes to "wgtght"?
> fix(wgtght)
>
> # If want to enter directly from spreadsheet interface.
> wgtght.frmblnk<-edit(data.frame())
>

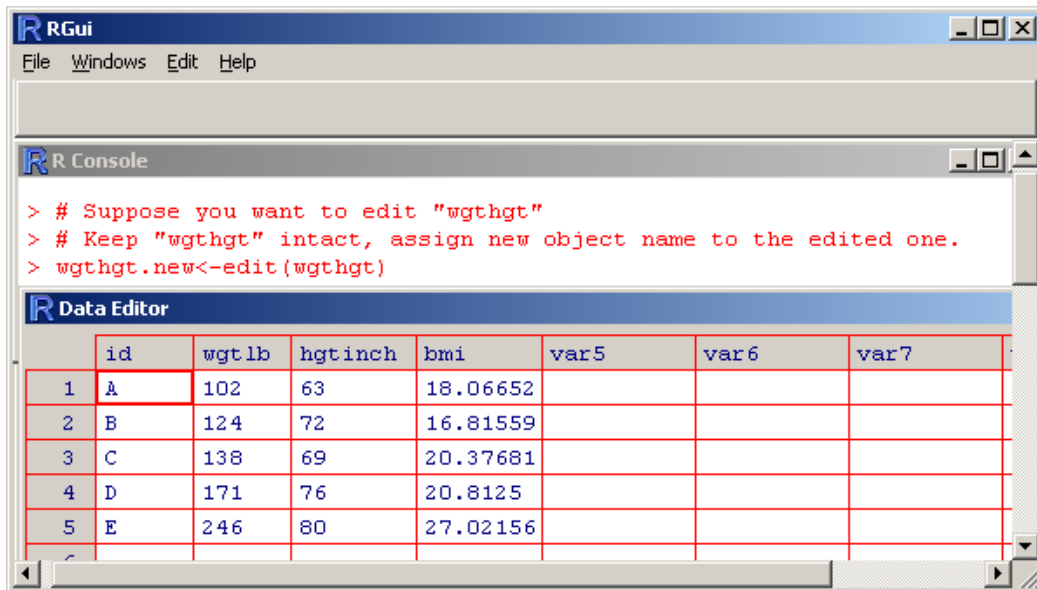
```

The command `new<-edit(old)` allows you to edit the *old* data frame via the spreadsheet interface (see the below picture) and upon edit completion the changed data frame object is assigned to a *new* object. That's what's done by `wgtght.new<-edit(wgtght)`. I now have both `wgtght.new` and the original `wgtght` in my workspace.

If you simply want to change the original data frame *old*, the simplest way is to use the function `fix(old)`. The data editor pops up for the data frame *old* so you can edit it. You could follow the same logic as `new<-edit(old)` and do `old <- edit(old)` of course (it is just cumbersome).

Finally, the command `new <- edit(data.frame())` opens a brand-new blank spreadsheet (named *new*) for you to enter new data into via the spreadsheet interface.

The below is what the data editor looks like in R.



### Data Entry, From External Files:

In actual research settings, we often need to read data from files. R's data reading functions read your data file and return it as a data frame. Here are some frequently used ones.

1. Text file (ASCII data format file) readers
  - **read.table()** reads text files (most versatile).
  - **read.csv()** reads comma separated values files.
  - **read.delim()** reads tab delimited files.
  - **read.fwf()** reads fixed width format files.
2. "Foreign" (stats package) data readers
  - **read.dta()** reads Stata .dta files (Note: as of 10/18/07, R ver.2.6.0, does not handle Stata10 format).
  - **read.spss()** reads SPSS .sav files.
  - **read.xport()** reads SAS XPORT format library.

We are going to use [these data files](#) in the examples throughout this workshop from now on. Download and place them in your working directory. In this section, we use data files borrowed from Stata's built-in file "auto" which is probably familiar to most of the Stata users here, as well as a small particle from General Social Survey.

#### 1. Text file readers

The comma separated values (.csv) format is one of the most widely used text formats. Here, we have this comma delimited file "autocsv\_head.csv." The first five observations look like this:

```
make,price,mpg,rep78,headroom,trunk,weight,length,turn,displacement,gear_ratio,foreign
AMC Concord,4099,22,3,2.5,11,2930,186,40,121,3.58,0
```

```
AMC Pacer,4749,17,3,3,11,3350,173,40,258,2.53,0
AMC Spirit,3799,22,,3,12,2640,168,35,121,3.08,0
Buick Century,4816,20,3,4.5,16,3250,196,40,196,2.93,0
Buick Electra,7827,15,4,4,20,4080,222,43,350,2.41,0
```

Let's read this data file into R by using the function `read.table()`. `read.table()` is useful and most versatile to read in rectangular data stored in ASCII format.

```
> ## Example: Data entry, from external
>
> # Read .csv data by read.table(), need specify header/sep.
> autocsv1<-read.table("autocsv_head.csv", header=T, sep=",")
>
> # Check the variable names we have in this data frame.
> names(autocsv1)
[1] "make"           "price"          "mpg"            "rep78"
[5] "headroom"       "trunk"          "weight"         "length"
[9] "turn"           "displacement"  "gear_ratio"     "foreign"
>
```

The option `header=(TRUE/FALSE)` to tell R whether the first row is the header with the variable names. The default header setting is `header=FALSE`, but here we have the variable name header, so this option must be set to `TRUE`. You can abbreviate them by just typing T or F.

R also needs to know what type of delimiter (separator) is used in your file. The default delimiter setting is blank space, so if your file use any other separator you must specify it for R by using the option `sep="delimiter type"`. Ours is comma-separated, so I specified it as `sep=","` above.

Also, the function `names()` is used to list the variable names of the data frame "autocsv2" we just created. This function is used to get or set the names of an object.

Let's look at another example. We have the same auto data, and the only difference is this time it is tab-delimited ("autotab.txt"), another common ASCII data format.

```
> # Read the same data but tab-delimited format.
> autotab1<-read.table("autotab.txt", header=T, sep="\t")
>
```

See the delimiter type is specified as `sep="\t"`. That's how you read tab delimited files. Likewise, if your data is semicolon-delimited, it would be `sep=";"` and so and so forth.

Let's check if the data is read correctly, so I list the first five observations for autocsv1 and autotab1.

```
> # Print obs 1 through 5 to check the data.
> autocsv1[1:5,]
  make price mpg rep78 headroom trunk weight length turn
1 AMC Concord 4099 22 3 2.5 11 2930 186 40
2 AMC Pacer 4749 17 3 3.0 11 3350 173 40
3 AMC Spirit 3799 22 NA 3.0 12 2640 168 35
4 Buick Century 4816 20 3 4.5 16 3250 196 40
5 Buick Electra 7827 15 4 4.0 20 4080 222 43
```

```

displacement gear_ratio foreign
1      121      3.58      0
2      258      2.53      0
3      121      3.08      0
4      196      2.93      0
5      350      2.41      0
> autotab1[1:5,]
      make price mpg rep78 headroom trunk weight length turn
1  AMC Concord  4099  22     3      2.5   11  2930   186   40
2  AMC Pacer   4749  17     3      3.0   11  3350   173   40
3  AMC Spirit  3799  22    NA      3.0   12  2640   168   35
4 Buick Century 4816  20     3      4.5   16  3250   196   40
5 Buick Electra 7827  15     4      4.0   20  4080   222   43
displacement gear_ratio foreign
1      121      3.58      0
2      258      2.53      0
3      121      3.08      0
4      196      2.93      0
5      350      2.41      0
>

```

Remember the brackets `[ ]` is for **indexing** (if you are like “What?” go back [there](#)). This time, R works on the data frame and indexes the rows and the columns by using `[ ]` two-dimensionally, i.e. `[index.ROW, index.COLUMN]`. Now `1:5` is placed before the comma, R is doing something with the rows of the data frame “autocsv1” and “autotab1.” In this case, R selects the first five cases (the rows) to print. **1:5 means 1 through 5**. Colon `:` specifies the range. The *COLUMN* part is left blank because I want to see all the variables so do not have to index any.

So far we used the `read.table()` function to read both “autocsv\_head.csv” and “autotab.txt” in R. There are functions specific to the comma separated values and tab delimited formats, `read.csv()` and `read.delim()`, respectively. You will get the same results.

```

> # Read .csv file by read.csv()
> autocsv2<-read.csv("autocsv_head.csv")
>
> # Read tab-delimited file by read.delim()
> autotab2<-read.delim("autotab.txt")

```

Try listing yourself some cases from these two data frames to make sure if their contents are identical to those of autocsv1 and autotab2.

`read.csv()` is actually identical to `read.table()` except for the default setting. The default for delimiters is, of course, comma (this is what this function is for!), and the header default is **header=TRUE**. In this case we should go with the defaults for both and don’t need to specify them on our part.

Likewise, `read.delim()` is identical to `read.table()` except that the delimiter default is **sep=“\t”** and the header default is **header=TRUE**. This function is intended specifically for reading tab-delimited files.

Now, we are going to read the “autocsv\_nohead.csv” file, which does not include the variable name header. So here are two things we want to do. First, read the file into an R data frame object, and second, add the variable name header to it.

```

> # Read "autocsv_nohead.csv" (w/o colmn name)
> autocsv3<-read.table("autocsv_nohead.csv", sep=",")
> # See, there's no header.
> autocsv3[1:5,]
      V1      V2 V3 V4  V5 V6   V7  V8 V9 V10  V11 V12
1  AMC Concord 4099 22  3 2.5 11 2930 186 40 121 3.58  0
2    AMC Pacer 4749 17  3 3.0 11 3350 173 40 258 2.53  0
3    AMC Spirit 3799 22 NA 3.0 12 2640 168 35 121 3.08  0
4 Buick Century 4816 20  3 4.5 16 3250 196 40 196 2.93  0
5 Buick Electra 7827 15  4 4.0 20 4080 222 43 350 2.41  0
>
> # Assign variable names
> names(autocsv3)<-c("make","price","mpg","rep78","headroom","trunk",
+ "weight","length","turn","displacement","gear_ratio","foreign")
>
> # List obs 1 through 5 to check the data
> autocsv3[1:5, ]
      make price mpg rep78 headroom trunk weight length turn
1  AMC Concord 4099 22    3     2.5    11  2930   186   40
2    AMC Pacer 4749 17    3     3.0    11  3350   173   40
3    AMC Spirit 3799 22   NA     3.0    12  2640   168   35
4 Buick Century 4816 20    3     4.5    16  3250   196   40
5 Buick Electra 7827 15    4     4.0    20  4080   222   43
      displacement gear_ratio foreign
1             121         3.58        0
2             258         2.53        0
3             121         3.08        0
4             196         2.93        0
5             350         2.41        0
>

```

As we just now learned, the header default of `read.table()` is `header=F`, so in this case we just went with the default. The delimiter is comma. The variable names are unspecified in the data frame “autocsv3”; they are just V1, V2, ... V12 at this point.

To set names for the data frame “autocsv3,” we used the function `names()` (see [the above](#)). Note that `c()` is used to define a character vector that contains the variable names. Again, I printed the first five observations to see if the data reading was successfully done.

This is the final example in this sub-section. We have a fixed width format data file, “gssfix.txt,” which comes with a codebook. Suppose that we want to read this data in R but we do not need to read the variable “Father’s prestige score.”

```

1765732161616110
2416237171613010
378 66189712010
4526630161818110
551 32209712110

```

Variable Name	Column #
Temporary ID	1-2
Prestige score	3-4
Father’s prestige score	5-6

6451729131316000	Age	7-8
7623729181810000	Education	9-10
834 211297 8000	Father's education	11-12
9525138141412000	Mother's education	13-14
10505620121212010	Sex	15
11362354 79897110	Race	16
12 20139799100	Region	17
1350 38149714000		

To read this type of data, we use the function `read.fwf()`, and set names for the data frame object by using the function `names()`.

```
> # Read "gssfix.txt" (fixed format) Suppose we don't read papa's prestige
> gssfix <- read.fwf("gssfix.txt", width = c(2,2,-2,2,2,2,2,1,1,1))
> names(gssfix)<-c("id","prestige","age","ed","paed",
+ "maed","sex","race","region16")
>
> gssfix
  id prestige age ed  paed maed sex race region16
1  1         76 32 16   16   16  1  1         0
2  2         41 37 17   16   13  0  1         0
3  3         78 66 18   97   12  0  1         0
4  4         52 30 16   18   18  1  1         0
5  5         51 32 20   97   12  1  1         0
6  6         45 29 13   13   16  0  0         0
7  7         62 29 18   18   10  0  0         0
8  8         34 21 12   97    8  0  0         0
9  9         52 38 14   14   12  0  0         0
10 10        50 20 12   12   12  0  1         0
11 11        36 54  7   98   97  1  1         0
12 12        NA 20 13   97   99  1  0         0
13 13        50 38 14   97   14  0  0         0
>
```

The only real difference between `read.fwf()` and the aforementioned reader functions is the option `width` by which to specify the widths of the fixed-width fields (notice a vector is defined for the widths by the `c()` function yet again). Negative-width fields are used to skip columns. In this example, we wanted to skip the variable “father’s prestige score” so used the negative number (-2) to skip that column.

Notice the “NA” for obs #12’s prestige score. This is the way R indicates missing values.

A quick summary below.

ASCII format file reader	Format	Header default	Separator default
<code>read.table()</code>	ASCII data formats general	FALSE	“ ” (blank space)
<code>read.csv()</code>	Comma separated values	TRUE	“,”
<code>read.delim()</code>	Tab delimited	TRUE	“\t”
<code>read.fwf()</code>	Fixed width	FALSE	“\t”

## 2. “Foreign” (other stats package) data readers

There may be some cases where we want to read data from standard statistical packages, such as SAS, Stata, and SPSS. To directly read those “foreign” formats in R, we need to have installed the *foreign* package first. This package allows us to use its foreign data reader functions.

You can see what packages you have available in R by running `library()`. If you do not find “foreign” in the list, that means it is not installed and ready to load. In that case, see [here](#) for how to download and install R packages.

Once “foreign” is ready to go, we first load it onto R and then read a foreign data file into a R data frame object. In this example, we first read a Stata .dta file “autostata.dta” and then read a SAS xport file. It is actually pretty easy and straightforward to read those foreign files once you have the package “foreign” ready.

```
> ## Example: Data entry, from foreign
>
> # Load library "foreign" to read foreign datasets
> library(foreign)
>
> # Read "autostata.dta" (Stata data file).
> autostata<-read.dta("autostata.dta")
>
> # Read "autosas.xpt" (SAS transport file)
> autosas<-read.xport("autosas.xpt")
>
```

List some cases from this new data frame “autostata” and “autosas” to check it is properly read into R (you know how to do it now!).

I listed [three reader functions](#) for probably most widely used foreign data formats at the beginning of this subsection. The package “foreign” allows you to read more foreign formats, however. This below is how to get information for this and other packages.

```
> # Can find help for packages this way.
> library(help=foreign)
>
```

`help(package=package name)` also works, but NOT `help(package name)`.

### Attach and Detach

We just now read the Stata file “autostata.dta” above, which is now stored in our workspace as a data frame object named “autostata.” Suppose we want to explore this data and want to get the average auto price. To do so, we use the function `mean()`.

```
> ## Example: Attach, detach, and some manipulation.
>
> # Try to get mean of price, get an error.
```

```
> mean(price)
Error in mean(price) : object "price" not found
>
```

Why did we get an error? What happened is we meant to tell R to get the mean of the *variable* named “price,” but R instead searched for an *object* named “price,” which does not exist in our workspace (check by `ls()`). This is why R returned that error message. You need to tell R this “price” is a *variable inside a data frame object*, specifically in this case it comes from the data frame “autostata.” One way to do so is to use the \$ notation.

```
> # Tell R where "price" comes from: data frame "autostata."
> mean(autostata$price)
[1] 6165.257
>
```

As you can see, **Data.Frame.Name\$Variable.Name** lets you access to variables inside a data frame specified. It would often be cumbersome, however, to type in **Data.Frame.Name\$Variable.Name** every time, again and again. Using the function `attach()`, you can tell R which data frame you want it to find variables without typing longish commands with full of the \$ notation. See below.

```
> # Attach autostata so R knows where to find "price" w/o $.
> attach(autostata)
>
> # See, therefore no error this time.
> mean(price)
[1] 6165.257
>
```

We should understand how this actually worked: with `attach(autostata)`, R creates a new environment on its system’s **search list (search path)** and copies the data frame “autostata” into that new environment (note, despite the function name, R does not actually attach the real data frame; it just creates a new environment to copy an object into). Let’s take a look at the search list by using the function `search()`.

```
> # See what happens to "search list", notice where autostata is.
> search()
[1] ".GlobalEnv"           "autostata"           "package:foreign"
[4] "package:stats"       "package:graphics"   "package:grDevices"
[7] "package:utils"       "package:datasets"   "package:methods"
[10] "Autoloads"           "package:base"
>
```

Notice that **R has copied the data frame “autostata” as No.2 on the search path**, meaning R searches *in* “autostata” second. Before we read the Stata file, we loaded the package “foreign” and notice that now its copy is placed as #3 in the path. Meanwhile, “.GlobalEnv” is your **workspace** and it *always* stays as #1 in the search path, meaning that R always looks into your workspace first for *objects* stored there. And “**package:base**” is the system library where all the standard functions are defined.

The function **detach()** does the opposite of **attach()**, i.e., deletes a data frame or a package copy from the search list. Leaving the parenthesis blank automatically removes No.2 in the search path. “.GlobalEnv” and “package:base” cannot be detached.

You can attach more than one data frame or packages. The most recently attached is copied as No.2 in the search path and searched second (try **attach(gssfix)**, then **search()** yourself, for an example). For this reason, **you should be careful not to give the same object/variable names to your data frames, vectors, variables, etc. to avoid confusion.** This is important. It is a good idea to avoid keep attaching many data frames or packages without detaching ones you no longer use.

Here is a quick example to give you a clearer idea why you should avoid the same object/variable names as much as possible. You can see how this can cause big and possibly critical confusions in actual analysis where you deal with bigger data frames.

```
> ### Example: Why avoid the same object/variable names.
>
> ### We first create two mini data frames both of which have a
> ### variable named "a"
>
> dta.x <- data.frame(a=1:8, b=9:16)
> dta.x
  a b
1 1 9
2 2 10
3 3 11
4 4 12
5 5 13
6 6 14
7 7 15
8 8 16
> dta.y <- data.frame(a=rnorm(6), c=letters[2:7])
> dta.y
      a c
1 -0.8683198 b
2 -0.3362823 c
3  0.8626605 d
4 -0.2053273 e
5  0.5773746 f
6 -0.3494458 g
>
> attach(dta.x)
> a
[1] 1 2 3 4 5 6 7 8
> ### We are seeing a variable "a" from a data frame "dta.x"
> ### Now let's attach dta.y without detaching dta.x.
>
> attach(dta.y)

The following object(s) are masked from dta.x :

a
```

```

> ### See R issues a warning that the data frame dta.x has a
> ### variable with the same name "a"!
> ### (Note: R says "the following OBJECT(S)" but in this context
> ### this should mean "variable(s)"
>
> a
[1] -0.8683198 -0.3362823  0.8626605 -0.2053273  0.5773746
[6] -0.3494458
> ### See, now this variable "a" is from dta.y, because the
> ### most recently attached data frame is placed as #2 in
> ### the search path. Try search() and you can see it.
> ### "a" in #2 is searched, found, and accessed.
> ### Meanwhile the variable "a" in dta.x is "masked."
>
> search()
[1] ".GlobalEnv"      "dta.y"           "dta.x"
[4] "autostata"       "package:foreign" "package:stats"
[7] "package:graphics" "package:grDevices" "package:utils"
[10] "package:datasets" "package:methods"  "Autoloads"
[13] "package:base"
> ### dta.y is closer to your workspace (.GlobalEnv) than dta.x
>
> detach(dta.y)
> a
[1] 1 2 3 4 5 6 7 8
> ### We detach dta.y, now we can see "a" from dta.x (now #2).
>
> detach(dta.x)
> a
Error: object "a" not found
> ### Now nothing is attached, and of course there's no object
> ### named "a" in the workspace now.
>
> ### Now, though, what if there's an object "a" INSIDE your workspace
> ### (meaning inside .GlobalEnv)?
> ### Let's create an object "a" there and try attaching dta.x.
> a<-135
> attach(dta.x)

      The following object(s) are masked _by_ .GlobalEnv :

      a

> ### Now, you have a warning... a bit different from the previous one.
>
> a
[1] 135
> ### See, R accesses and prints an object "a" which is located
> ### INSIDE .GlobalEnv. This means (because it's *INSIDE* .GlobalEnv)
> ### the object "a" is the closest to .GlobalEnv (#1).
> ### Thus, "a" from dta.x (#2) is masked by .GlobalEnv.
>
> ### The point is, this can cause confusions where you are working
> ### on a variable which you don't mean to.
>
> detach(dta.x)
> rm(a,dta.x,dta.y)

```

```
>
```

## 5. Save and Load R Objects & Save R Data in Text Format

Now that we have created data frames and other objects in R, we want to save them for later continued work. In this section, we will go over:

- How to save and load R objects as an external R file.
- How to save R objects in text format.

### Saving and Loading R Objects

We now have quite a few objects stored in our workspace. To save objects (data frame, vectors, matrices, functions, etc) as an external R file (.RData), or to load them from R data files, we use the following functions. In either case, **remember to add the .RData extension** to the file path (R will not do it for you).

- **save.image(): to save the whole workspace.**  
You don't have to specify your R file name (file=""); by default an R file named ".RData" is saved in your working directory.
- **save(object.name.to.be.saved, file="R.file.name.needed.to.be.specified.RData"): to save selected objects.**  
Unlike save.image(), you must specify file="". You can save one or more selected objects in one external R file. RData is saved in your working directory.
- **load("R\_file\_name.RData"): to load a workspace image.**

Here's a quick example.

```
> ## Save .RData workspace image.
> # Save the whole workspace
> save.image()
>
> # Save selected data frame and vector objects
> save(autocsv1, gssfix, bmi, wghtgt, file="mydta.RData")
>
> ## Load .RData workspace imate onto R. Try removing and loading.
> rm(list=ls())
> ls()
character(0)
>
> load(".RData")
> # My workspace is back in!
```

```

> ls()
[1] "autocsv1"   "autocsv2"   "autocsv3"   "autostata"
[5] "autotab1"   "autotab2"   "bmi"         "gssfix"
[9] "hgtft"      "hgtin"      "hgtinch"    "wgthgt"
[13] "wgthgt.new" "wgtlb"
>
> # Clear everything out of my workspace.
> rm(list=ls())
>
> # Load the objects I selected and saved.
> load("mydta.RData")
>
> # Now they are in my workplace.
> ls()
[1] "autocsv1" "bmi"       "gssfix"    "wgthgt"
>

```

In the example above, the function **save.image()** writes an external representation of my whole workspace to an .RData file. I also select and save the data frame and vector objects “autocsv1” “bmi” “gssfix” and “wgthgt” to an external file “mydta.RData” (you may want to see those two .RData files are now in your working directory). I clear my workspace, I load the .RData file that contains all my objects. They are now all back in the workspace. Again, with my workspace empty, I load “mydta.RData.” You see my workspace now has two selected and saved data frame objects.

This demands you to take a bit different attitude than standard statistical packages; in R, we do not save a “dataset” as a one, single, distinct spreadsheet-like file (e.g., .dta in Stata), but **we store objects in our workspace and save one or more of them as a workspace image (= an external .RData file)**. It can include a data frame(s), but do not necessarily have to. You call objects back into your workspace by loading the saved workspace image. Again, **R is an object-driven program**.

### Saving Data in Text Format

After you work on data in R, we sometimes need to save it in a different format readable by other programs. There are a couple of functions to accomplish this, but saving it in text format is the most versatile, flexible way in this regard. The function **write.table()** does it for you. It is just the opposite to **read.table()**.

**write.table(object.name.to.be.saved, file=“saved.text.file.name”)**

In the following example, we will see how to tweak output text files by adding options to this very basic format above. We will save the data frame object “autocsv1” in two types of ASCII format, i.e., space delimited and comma separated values.

Let's first browse the data frame we are working on by printing the first five cases. Notice R prints its own row names (highlighted numbers in the left, 1, 2, ...5) in addition to our data frame variables.

```
> ## Save data in other format (ASCII)
> # First browse the data, first 5 observation.
> autocsv1[1:5,]
      make price mpg rep78 headroom trunk weight length turn
1  AMC Concord 4099 22   3      2.5   11  2930  186  40
2   AMC Pacer 4749 17   3      3.0   11  3350  173  40
3   AMC Spirit 3799 22  NA      3.0   12  2640  168  35
4 Buick Century 4816 20   3      4.5   16  3250  196  40
5 Buick Electra 7827 15   4      4.0   20  4080  222  43
 displacement gear_ratio foreign
1           121         3.58      0
2           258         2.53      0
3           121         3.08      0
4           196         2.93      0
5           350         2.41      0
>
```

First, let's try the basic format of `write.table()` as is. Run the following line and then open to check the text file "myauto.txt" saved in the working directory.

```
> #1 Save autocsv1 in text format (blank space), keep row.names + no head.
> write.table(autocsv1, "myauto.txt")
>
```

```
"make" "price" "mpg" "rep78" "headroom" "trunk" "weight" "length" "turn" "displacement"
"gear_ratio" "foreign"
"1" "AMC Concord" 4099 22 3 2.5 11 2930 186 40 121 3.58 0
"2" "AMC Pacer" 4749 17 3 3 11 3350 173 40 258 2.53 0
"3" "AMC Spirit" 3799 22 NA 3 12 2640 168 35 121 3.08 0
"4" "Buick Century" 4816 20 3 4.5 16 3250 196 40 196 2.93 0
"5" "Buick Electra" 7827 15 4 4 20 4080 222 43 350 2.41 0
```

Notice that the row names of the R data frame are also outputted (italicized above in the left, "1" "2" "3" "4" "5"...), and as a result, we now have 12 column names for 13 data columns. These R row names are most certainly unnecessary in many cases. So, we want to suppress R's row names. To do so, the option `row.names=` needs to be set **FALSE** (the default is TRUE).

```
> #2 Save autocsv1 in text format (blank space), no row.names.
> write.table(autocsv1, "myauto.txt", row.names=F)
>
```

```
"make" "price" "mpg" "rep78" "headroom" "trunk" "weight" "length" "turn" "displacement"
"gear_ratio" "foreign"
"AMC Concord" 4099 22 3 2.5 11 2930 186 40 121 3.58 0
"AMC Pacer" 4749 17 3 3 11 3350 173 40 258 2.53 0
"AMC Spirit" 3799 22 NA 3 12 2640 168 35 121 3.08 0
"Buick Century" 4816 20 3 4.5 16 3250 196 40 196 2.93 0
"Buick Electra" 7827 15 4 4 20 4080 222 43 350 2.41 0
```

The row names (“1” “2” “3”...) are now gone in our output text file. We get 12 column names for 12 data columns.

Conversely, if you need to keep R’s row names, use the option **col.names=NA** to add a blank column name (“”) for R’s row names. Now we have 13 column names for 13 data columns.

```
> #3 Save autocsv1 in text format (blank space), keep row.names + blank head
> write.table(autocsv1, "myauto.txt", col.names=NA)
>
```

```
"" "make" "price" "mpg" "rep78" "headroom" "trunk" "weight" "length" "turn" "displacement"
"gear_ratio" "foreign"
"1" "AMC Concord" 4099 22 3 2.5 11 2930 186 40 121 3.58 0
"2" "AMC Pacer" 4749 17 3 3 11 3350 173 40 258 2.53 0
"3" "AMC Spirit" 3799 22 NA 3 12 2640 168 35 121 3.08 0
"4" "Buick Century" 4816 20 3 4.5 16 3250 196 40 196 2.93 0
"5" "Buick Electra" 7827 15 4 4 20 4080 222 43 350 2.41 0
```

Finally, let’s learn how to save a data frame in the .csv format.

```
> #4 Save autocsv1 in csv format, not keep row names, missing blank.
> write.table(autocsv1, "myauto.csv", sep="," , row.names=F, na="")
>
```

```
make,price,mpg,rep78,headroom,trunk,weight,length,turn,displacement,gear_ratio,foreign
AMC Concord,4099,22,3,2.5,11,2930,186,40,121,3.58,0
AMC Pacer,4749,17,3,3,11,3350,173,40,258,2.53,0
AMC Spirit,3799,22,,3,12,2640,168,35,121,3.08,0
Buick Century,4816,20,3,4.5,16,3250,196,40,196,2.93,0
Buick Electra,7827,15,4,4,20,4080,222,43,350,2.41,0
```

The delimiter default of **write.table()** is **sep= “”** (blank space). In the above example, this option is set to **“,”** to save the data frame in the comma separated values format. Also, because we do not need a column of the row names of the R data frame, we use **row.names=FALSE**. Finally, notice that in the first three examples, missing values are represented by **NA**, which is the default (**na= “NA”**). This time we want to leave missing elements blank, so we set the option **na= “”** (see the bold-faced part of the output data).

Quick summary of this sub-section.

ASCII format file writer	Format	row.names default	col.names default	Separator default	Missing default
<b>write.table()</b>	ASCII data	TRUE	TRUE	“” (blank space)	“NA”

Now that you must have gotten the feel of the function **help()**, do **help(write.csv)**, and **help(package=foreign)** to learn how to write a foreign data file.

## 6. Indexing and Conditional Selection

Oftentimes, you may want to limit your sample to a subset of research interests (e.g., select Hispanics only from a large national sample to study changes in Hispanic communities over time, etc). In this section, we will learn:

- How to subset your data by indexing and conditional selection.
- How to select your data by using the subset function.

### Indexing and Conditional Selection

Actually, we have already got a glimpse of the very basic of indexing vectors and data frames to obtain their subset (see [here](#) and [here](#)).

For vector/variable, [*index.k.th element*] ( $k = 1, 2, 3 \dots n$ )

For data frame, [*index.ROW, index.COLUMN*]

We use the same format to select data on logical expression(s).

For vector/variable, [*select.element(s).conditionally*]

For data frame, [*select.ROW.conditionally, select.COLUMN.conditionally*]

Let's first go over and summarize how to extract elements from a vector. In this example, we use the data vector "bmi" in our workspace. Remember bmi=1:5 (five people's body mass index data).

```
> ## Example: Element selection from a vector "bmi"
>
> # First, print them all
> bmi
[1] 18.06652 16.81559 20.37681 20.81250 27.02156
>
> # Number of elements? (not indexing, just warming up for later use)
> length(bmi)
[1] 5
>
> # 3rd element?
> bmi[3]
[1] 20.37681
>
> # Indexing can be used to make changes to the data
> bmi[3]<-20.66251
> bmi
[1] 18.06652 16.81559 20.66251 20.81250 27.02156
>
> # All but 3rd element?
> bmi[-3]
```

```

[1] 18.06652 16.81559 20.81250 27.02156
>
> # First 3 elements?
> bmi[1:3]
[1] 18.06652 16.81559 20.66251
>
> # Odd number elements?
> bmi[seq(1,5,2)]
[1] 18.06652 20.66251 27.02156
>
> # Selected elements 1, 3, and 5?
> bmi[c(1,3,5)]
[1] 18.06652 20.66251 27.02156
>
> # Last 3 elements?
> bmi[(length(bmi)-2):length(bmi)]
[1] 20.66251 20.81250 27.02156
>
> # Elements whose bmi greater than 20?
> bmi[bmi>20]
[1] 20.66251 20.81250 27.02156
>
> # Elements whose bmi 18.5 to 24.9 ("Normal" weight status, per CDC)?
> bmi[bmi>=18.5&bmi<=24.9]
[1] 20.66251 20.81250
>
> # Elements whose bmi's are not "normal"?
> bmi[bmi<18.5|bmi>24.9]
[1] 18.06652 16.81559 27.02156
>
> # Elements whose bmi is the smallest?
> bmi[bmi==min(bmi)]
[1] 16.81559
>

```

In R, the logical test “Equal to” is `==` (the logical “Not equal to” is `!=`), and the logical OR is `|` (the logical AND is `&`). Also, two new functions are used here. The function `seq()` is, like `c()`, used to create vectors in various situations. `min()` returns the minimum value. Use `help()` to find how these functions work.

Let’s also see how to subset data frames. Remember we did indexing and just printing selected part of the data [before](#). Just some extension of that below.

```

> ## Examples: Subsetting a data frame, indexing.
>
> # First attach the data frame.
> attach(autocsv1)
>
> # Print obs 1 through 5.
> autocsv1[1:5, ]
      make price mpg rep78 headroom trunk weight length turn
1  AMC Concord 4099  22     3      2.5    11  2930   186   40
2   AMC Pacer  4749  17     3      3.0    11  3350   173   40

```

```

3   AMC Spirit  3799  22   NA    3.0  12  2640  168  35
4 Buick Century 4816  20    3    4.5  16  3250  196  40
5 Buick Electra 7827  15    4    4.0  20  4080  222  43
  displacement gear_ratio foreign
1           121         3.58      0
2           258         2.53      0
3           121         3.08      0
4           196         2.93      0
5           350         2.41      0
>

```

Of course, we can create and save a subset of this data frame in our workspace by assign a new object name to it.

```

> # Subset data frame, create a small data frame.
> autocsv1.1z5<-autocsv1[1:5,]
>
> # Print and take a look.
> autocsv1.1z5
      make price mpg rep78 headroom trunk weight length turn
1  AMC Concord 4099  22    3     2.5   11  2930   186   40
2   AMC Pacer 4749  17    3     3.0   11  3350   173   40
3   AMC Spirit 3799  22   NA     3.0   12  2640   168   35
4 Buick Century 4816  20    3     4.5   16  3250   196   40
5 Buick Electra 7827  15    4     4.0   20  4080   222   43
  displacement gear_ratio foreign
1           121         3.58      0
2           258         2.53      0
3           121         3.08      0
4           196         2.93      0
5           350         2.41      0
>

```

Let's subset the data frame and create a small data frame that has case1, 3, and 5 only.

```

> # Create a small dframe inc. obs 1,3,5.
> autocsv1.135<-autocsv1[c(1,3,5), ]
>
> # Print and take a look.
> autocsv1.135
      make price mpg rep78 headroom trunk weight length turn
1  AMC Concord 4099  22    3     2.5   11  2930   186   40
3   AMC Spirit 3799  22   NA     3.0   12  2640   168   35
5 Buick Electra 7827  15    4     4.0   20  4080   222   43
  displacement gear_ratio foreign
1           121         3.58      0
3           121         3.08      0
5           350         2.41      0
>

```

We can further create subset data frames on specific criteria. Let's create a small data frame that has only domestic cars (the variable foreign is coded as domestic = 0, foreign = 1). Then check if the foreign car cases are all dropped in the new data frame.

```
> # Examples: Even more on subsetting, conditional selection.
>
> # Subset dframe inc. US car only.
> autocsv1.dmstc<-autocsv1[foreign==0,]
>
> # Check the data
> table(autocsv1$foreign)

 0  1
52 22
> table(autocsv1.dmstc$foreign)

 0
52
>
```

Let's create a small data frame that has the info of foreign cars' makers and prices.

```
> # First check the varnames and order.
> names(autocsv1)
 [1] "make"          "price"          "mpg"            "rep78"
 [5] "headroom"      "trunk"          "weight"         "length"
 [9] "turn"          "displacement"  "gear_ratio"    "foreign"
>
> # subset dframe inc. foreign car, their makers & prices.
> autocsv1.frgn<-autocsv1[foreign==1,c(1,2)]
> autocsv1.frgn
      make price
53   Audi 5000 9690
54   Audi Fox 6295
55   BMW 320i 9735
56   Datsun 200 6229
57   Datsun 210 4589
58   Datsun 510 5079
59   Datsun 810 8129
60   Fiat Strada 4296
61   Honda Accord 5799
62   Honda Civic 4499
63   Mazda GLC 3995
64   Peugeot 604 12990
65 Renault Le Car 3895
66   Subaru 3798
67   Toyota Celica 5899
68 Toyota Corolla 3748
69 Toyota Corona 5719
70   VW Dasher 7140
71   VW Diesel 5397
72   VW Rabbit 4697
73   VW Scirocco 6850
74   Volvo 260 11995
```

```
>
```

Let's create a small data frame that has the info of makers, prices, and mpg of foreign cars less than \$5,000.

```
> # subset dframe inc. frn cars, < $5K, maker, price and mpg.
> autocsv1.frgncheap<-autocsv1[foreign==1 & price<5000,c(1:3)]
> autocsv1.frgncheap
      make price mpg
57  Datsun 210  4589 35
60  Fiat Strada 4296 21
62  Honda Civic 4499 28
63   Mazda GLC 3995 30
65 Renault Le Car 3895 26
66   Subaru    3798 35
68 Toyota Corolla 3748 31
72   VW Rabbit 4697 25
>
```

### Subsetting by subset() function

Sometimes, using brackets to subset data frames can be cumbersome. In that case, using the function **subset()** may be a little easier way to subset your data frame. Basic usage is:

**subset(data.frame.name, logical.expression, select = expression)**

where:

- *logical expression* works on rows, evaluated in the data frame. You can refer columns by variable names (instead of column numbers as we just did above).
- **select = expression** allows you to select/drop columns, can do by variable names.

Let's subset autocsv1, but this time by using **subset()**. Create a subset data frame that has foreign cars less than \$5,000 with their makers, prices, mpg, weights, and lengths information.

```
> # Examples: Subsetting by the function subset().
>
> # Do similar selection as we just did by [].
>
> autocsv1.sub1<-subset(autocsv1, foreign==1 & price<5000,
+ select=c(make,price,mpg,weight,length))
> autocsv1.sub1
      make price mpg weight length
57  Datsun 210  4589 35   2020   165
60  Fiat Strada 4296 21   2130   161
62  Honda Civic 4499 28   1760   149
63   Mazda GLC 3995 30   1980   154
65 Renault Le Car 3895 26   1830   142
```

```

66      Subaru  3798  35  2050  164
68 Toyota Corolla  3748  31  2200  165
72      VW Rabbit  4697  25  1930  155
>

```

We can select consecutive variable names quickly.

```

> # All variables from make through length.
> autocsv1.sub2<-subset(autocsv1, foreign==1 & price<5000,
+ select=make:length)
> autocsv1.sub2
      make price mpg rep78 headroom trunk weight length
57   Datsun 210  4589  35    5     2.0    8  2020   165
60   Fiat Strada  4296  21    3     2.5   16  2130   161
62   Honda Civic  4499  28    4     2.5    5  1760   149
63   Mazda GLC   3995  30    4     3.5   11  1980   154
65 Renault Le Car  3895  26    3     3.0   10  1830   142
66     Subaru   3798  35    5     2.5   11  2050   164
68 Toyota Corolla  3748  31    5     3.0    9  2200   165
72     VW Rabbit  4697  25    4     3.0   15  1930   155
>

```

We can also “de-select” variables, i.e., select all the variables *other than some you specify*, by using the negative sign (-). This works on a similar logic as this one [here](#). In the example below, we leave in the variables other than “weight,” “length,” “trunk,” and “foreign” in our subset data frame.

```

> # Drop four vars.
> autocsv1.sub3<-subset(autocsv1, foreign==1 & price<5000,
+ select=c(-weight,-length,-trunk,-foreign))
> autocsv1.sub3
      make price mpg rep78 headroom turn displacement
57   Datsun 210  4589  35    5     2.0   32         85
60   Fiat Strada  4296  21    3     2.5   36        105
62   Honda Civic  4499  28    4     2.5   34         91
63   Mazda GLC   3995  30    4     3.5   33         86
65 Renault Le Car  3895  26    3     3.0   34         79
66     Subaru   3798  35    5     2.5   36         97
68 Toyota Corolla  3748  31    5     3.0   35         97
72     VW Rabbit  4697  25    4     3.0   35         89
      gear_ratio
57         3.70
60         3.37
62         3.30
63         3.73
65         3.72
66         3.81
68         3.21
72         3.78
>

```

Feel free to take a peek at your workspace (`ls()`) and see those subset data frames are stored there.

## 7. Sorting Cases

In this section, we learn:

- How to sort a vector/variable.
- How R sorts a data frame.

The way R handles sorting may seem a little abstract before you get used to it, but it is in fact a powerful way to handle this. Here again, indexing, one of R's “heart and soul” mindsets, plays an important role.

### How to sort a vector/variable

In the example below, we sort the data *vector* object “bmi” (actually, I only have kept this object in the workspace till now for this example!)

```
> ## Example: Sorting a single vector.
>
> # Data vector "bmi" looks like this.
> bmi
[1] 18.06652 16.81559 20.66251 20.81250 27.02156
>
> # sort it
> sort(bmi)
[1] 16.81559 18.06652 20.66251 20.81250 27.02156
>
> # Sort descending.
> sort(bmi, decreasing=T)
[1] 27.02156 20.81250 20.66251 18.06652 16.81559
>
> # Remove the object bmi from the workspace.
> rm(bmi)
>
```

The function **sort()** sorts a *vector* (notice the emphasis) into ascending or descending order. The default of the option “decreasing (=ascending)” is **decreasing=FALSE**, so to sort your data descending, this option needs to be set to TRUE.

This sounds pretty straightforward. However, emphasized just above, this function works on a single *vector* (or a variable). It sorts a data vector and returns a sorted vector.

Now, in actual data management and analysis settings, we more often need to sort a *data frame*, i.e., order more than one variables according to the values of one or more other variables, than sort and print a single vector. So let's try sorting a data frame. If you are a Stata user, you might

assume that something like `sort(varlist)` does the job, and if you are a SAS user, it may sound like R's `sort()` does the same thing as “`proc sort data = ; by ;`” Is that the case?

```
> ## Example: Try sort() for a data frame.
>
> # First, detaching the one currently #2 in the path.
> detach(autocsv1)
>
> # Attach wgtght.
> attach(wgtght)
>
> # Try to sort this data frame, default is ascending.
> sort(bmi)
[1] 16.81559 18.06652 20.37681 20.81250 27.02156
>
> # Does this do sorting to the whole dataframe?
> wgtght
  id wgtlb hgtinch    bmi
1  A   102     63 18.06652
2  B   124     72 16.81559
3  C   138     69 20.37681
4  D   171     76 20.81250
5  E   246     80 27.02156
> # Not!
>
```

As you can see above, `sort()` does not sort the data frame for us.

### How R sorts a data frame

To sort a *data frame*, we rely on the power of the function `order()` instead. Note, however, this function does *not* let you directly sort a data frame. What it does is to compute an *ordering* of a sorting variable and returns a vector of the ordered variable. We then use this ordering information for indexing (yet again!) to sort a data frame.

Let's first learn what the function `order()` does.

```
> ## Example: How to sort dataframe?
>
> # The function order().
> # Compute an ordering of your sorting var(s).
> # Let's see what happens when order() orders a variable.
> order(bmi)
[1] 2 1 3 4 5
>
> # What does this do to the whole data frame?
> wgtght
  id wgtlb hgtinch    bmi
1  A   102     63 18.06652
2  B   124     72 16.81559
3  C   138     69 20.37681
4  D   171     76 20.81250
```

```

5 E 246 80 27.02156
>
> # So, at this point the data frame is not yet sorted.
> # order() alone does not do sorting to a data frame.
>

```

The result of `order(bmi)` is a vector of R's row.names (the row numbers 1 to 5, highlighted in the above output) ordered according to the size of the variable "bmi," that is **2 1 3 4 5**.

In the following example, we use this ordering information to index the data frame "wgtght."

```

> # Index the data frame "wgtght" by using the vector containing
> # the ordering of "bmi" will do the job.
> # Create another data frame object for a sorted data & compare.
> wgtght.ascnd<-wgtght[order(bmi),]
>
> wgtght
  id wgtlb hgtinch  bmi
1  A  102     63 18.06652
2  B  124     72 16.81559
3  C  138     69 20.37681
4  D  171     76 20.81250
5  E  246     80 27.02156
> wgtght.ascnd
  id wgtlb hgtinch  bmi
2  B  124     72 16.81559
1  A  102     63 18.06652
3  C  138     69 20.37681
4  D  171     76 20.81250
5  E  246     80 27.02156
>

```

The function `order()` is placed before the comma in the square brackets for indexing the data frame "wgtght" (`wgtght[order(bmi),]`), which of course means that we are working on the rows (cases) of the data frame, i.e., we are sorting the cases.

Compare the pre- and post-sorted data frames. Again, I highlight the row.names for you so you can see how the vector rearranged by `order()` works there.

As you can see, the data frame "wgtght.ascnd" is ascendant-sorted, because the default option is `decreasing=FALSE`, just like in the case of `sort()`. Set this option to `decreasing=TRUE` to do descending sorting.

**The point is that by indexing (`[ROW, COLUMN]`) your data frame with this ordered row.names vector, the other variables can be ordered (sorted) by the same criterion, because each of R's row.name corresponds to each of the whole row.**

You can sort a data frame by more than one criterion by simply adding arguments to `order()`. Say, for example, suppose you have a data frame of the time-series cross-sectional structure,

`order(country, year)` gives a division into each country and then order it by year within each country. Let's just do a quick exercise.

```
> # 1 First domestic/foreign, then price from cheap to expensive.
> autocsv1.ascnd1<-autocsv1[order(foreign,price),]
> autocsv1.ascnd1[1:10,c("make","price","foreign")]
      make price foreign
34  Merc. Zephyr  3291      0
14  Chev. Chevette 3299      0
18   Chev. Monza  3667      0
 3   AMC Spirit   3799      0
29  Merc. Bobcat  3829      0
19   Chev. Nova   3955      0
20   Dodge Colt   3984      0
21 Dodge Diplomat 4010      0
46   Plym. Volare 4060      0
10  Buick Skylark 4082      0
>
> # 2 Then the price from expensive to cheap.
> autocsv1.dscnd2<-autocsv1[order(mpg,-rep78),]
> autocsv1.dscnd2[1:10,c("make","price","foreign","mpg","rep78")]
      make price foreign mpg rep78
26 Linc. Continental 11497      0 12      3
27   Linc. Mark V 13594      0 12      3
30   Merc. Cougar  5379      0 14      4
33   Merc. XR-7   6303      0 14      4
11   Cad. Deville 11385      0 14      3
28 Linc. Versailles 13466      0 14      3
12   Cad. Eldorado 14500      0 14      2
64   Peugeot 604 12990      1 14     NA
 5   Buick Electra  7827      0 15      4
31   Merc. Marquis  6165      0 15      3
>
```

What is new here is how to do ascending and descending sort in this case. The default is ascending. To do descending sort, use “-” (negative sign) . So in all the examples, the data frame is first sorted according to the order of the foreign/domestic coding (domestic = 0, foreign = 1). Then, within each, it is ordered ascendently (example 1) and descendently (example2) by price.

## **8. Factors: How R Handles Categorical Information**

Very often we have in data files categorical variables that indicate some subdivision of the population, such as gender, religion, levels of agreement measured by a Likert scale, and so forth. In many cases, such variables are entered into data as numeric codes, but of course, they do not have any direct numerical meaning, and they (except for 0/1 dummy coding) cannot be used in regression analyses as they are. Meaningful names should be assigned to those categories so that the data structure of categorical variables is distinguished from that of numeric variables.

R stores such information as **factors**. There are some good things about factors. When it comes to statistics, R uses factors correctly, i.e., as non-numeric, in models with the correct number of degrees of freedom assigned (which is beyond the scope of this workshop, but if you are a Stata user, it does something similar to `-xi-` in regression analyses). You will probably find factors greatly useful in wide variety of R graphics as factors handle the value labels and their meaningful order. On a more technical side, storing character variables as factors lets you use memory more efficiently.

So, in this section, we will learn:

- Factors: How R handles categorical variables (or how you have R handle categorical variables appropriately).

## Factors

**Factors** are vector objects used to define a discrete grouping of the elements of other vectors of the same length. Simply, they store categorical information. Among categorical variables, nominal variables do not have an intrinsic order in it and hence factors that store such information are called “**unordered factors**.” On the other hand, ordinal variables have a clear ordering, so “**ordered factors**” store such information. Categories in factors are called “**levels**.” **Factors are stored as a character vector of levels (this can be numbers or characters) and a numeric vector of indices.**

1. Unordered factors
2. Ordered factors
3. Adding and dropping levels in factors

**In R, you should explicitly specify categorical variables as factors.** Once you define a given variable as a factor, R in no way whatsoever treats it as anything numeric and never allows for any value other than the levels you defined, until you explicitly redefine them. To create a factor, we use the function **factor()**. You include an argument in this function, which in this case is a vector of values.

### **Unordered factors:**

Let's start with a simple example for unordered factors.

```
> ## What is a factor variable?
> # (Assuming autocsv1 is still attached; if not, attach it)
>
> foreign
[1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[33] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
[65] 1 1 1 1 1 1 1 1 1 1
>
> # We know this is actually a nominal variable.
> # But as shown below R doesn't know it's nominal.
```



```
> # Notice the warning sign!  
>
```

Let's go over what we just now did. In the data frame "autocsv1" (which is still attached as #2), we have a variable named "foreign" which indicates if a given car is a domestic or foreign product. We know this is a nominal variable (i.e., "Is this a domestic car or a foreign car?"), but that does not necessarily mean R does too.

To see what R thinks of this variable, we used the functions **is.factor()** and **is.numeric()**. These functions return TRUE or FALSE to whether the argument is a factor/numeric. In the above example, R thought that the variable "foreign" was numeric, not a factor. And for this very reason, R computed this nominal variable's average. Not that the information is entirely useless in this particular 0/1 binary case, as it tells you the proportion of the foreign cars out of the whole sample. It is meaningless as an "average" figure, however.

By using the **factor()** function, we told R to create a new variable "foreign.f1" and treat it as a factor (notice I detached and re-attached the data frame "autocsv1" right after I created this variable, because when you modify a data frame stored in the workspace, the attached [= copied on the search path] version does not change automatically). When we printed the newly created factor "foreign.f1," R told you the factor had two **levels** (0 and 1), which means "categories" in R. Now that it knew it was a factor, R did not compute a mean value of this variable that does not have a numerical meaning. Instead, it returned an "NA" and issued an warning message that goes: "**Warning message: argument is not numeric or logical: returning NA in: mean.default(foreign.f1).**"

Each of these numbers, 0 and 1, now represent an attribute of car type, and in this case, the 0 corresponds to domestic and the 1 to foreign. We want to add a character vector of these levels so we can label them. Let's try it below.

```
> ## Example: Labeling levels  
>  
> # Set the attributes (variable labels)  
> autocsv1$foreign.f2 <- factor(foreign, levels=0:1,  
+ labels = c("domestic", "foreign"))  
>  
> detach(autocsv1)  
> attach(autocsv1)  
>  
> # Factor stored as a character vector.  
> levels(foreign.f2)  
[1] "domestic" "foreign"  
>  
> foreign.f2  
[1] domestic domestic domestic domestic domestic domestic domestic  
[8] domestic domestic domestic domestic domestic domestic domestic  
[15] domestic domestic domestic domestic domestic domestic domestic  
[22] domestic domestic domestic domestic domestic domestic domestic  
[29] domestic domestic domestic domestic domestic domestic domestic  
[36] domestic domestic domestic domestic domestic domestic domestic
```

```

[43] domestic domestic domestic domestic domestic domestic domestic
[50] domestic domestic domestic foreign foreign foreign foreign
[57] foreign foreign foreign foreign foreign foreign foreign
[64] foreign foreign foreign foreign foreign foreign foreign
[71] foreign foreign foreign foreign
Levels: domestic foreign
>
> # This factor is also stored as a numeric vector of indices.
> as.numeric(foreign.f2)
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[33] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
[65] 2 2 2 2 2 2 2 2 2 2
>

```

I specified two options in `factor()` above. First, I specified levels of the factor, `levels=0:1`, which means that the input coding uses the values 0 and 1. By the next option `label=`, I created a character vector of these levels in the same order.

After re-attaching the data frame, I printed the levels (categories) of this new variable by using the function `levels()`. R now told me that there were two categories “domestic” and “foreign.” I also printed this new factor. In the first example without category labels, R just printed 0 and 1, but in this example, it printed “domestic” and “foreign.” As I mentioned at the beginning of this section, factors are stored as a character vector of levels and a numeric vector of indices. R by default prints the former if it is defined (if not, R simply prints the levels as labels, which is what happens in the first example). It is also internally stored as a numerical coding, which is extracted by using the function `as.numeric()`. Notice, the original input coding 1 and 0 disappears and it is instead coded as 1 and 2 now. R always uses numbers starting at 1 as its internal representation of a factor.

Actually, I didn’t have to specify the option `levels=` in this example we just did above, because by default R uses the values of the variable “foreign” and sorts it for you (i.e., 0, 1). However, I would recommend that you make it a habit to specify this `levels=` option regardless. Here is the reason why.

```

> ## Example: Why better habit specify the option "levels="
>
> # In this example,
> # "std" = strongly disagree
> # "sod" = somewhat disagree
> # "nth" = neither agree nor disagree
> # "soa" = somewhat agree
> # "sta" = strongly agree
>
> # Create a character data vector "opinion"
>
> opinion<-c("std","sta","soa","soa","sod","nth","nth","sta","sod")
> opinion
[1] "std" "sta" "soa" "soa" "sod" "nth" "nth" "sta" "sod"
>
> # A character variable per se is NOT a factor. You must specify.
> opinion.f1<-factor(opinion)
> opinion.f1

```

```

[1] std sta soa soa sod nth nth sta sod
Levels: nth soa sod sta std
> as.numeric(opinion.f1)
[1] 5 4 2 2 3 1 1 4 3
> # Does this make sense? No.
>
> # Now, we specify the option levels.
> opinion.f2<-factor(opinion, levels = c("std","sod","nth","soa","sta"))
> opinion.f2
[1] std sta soa soa sod nth nth sta sod
Levels: std sod nth soa sta
> as.numeric(opinion.f2)
[1] 1 5 4 4 2 3 3 5 2
>

```

See what happened here in the first set of R's messages in blue: **Levels: nth soa sod sta std**. Obviously, R sorted the vector according to its default, because we did not specify the option **levels=** this time. However, **sorting is alphabetical for character variables!** Of course, this sorting is substantively meaningless and unsuitable in this case. This is the reason why you should make it your habit to specify levels for factors always.

So we ordered the input coding in a meaningful way in the option **levels=**. Now, the levels are sorted as they should be: **Levels: std sod nth soa sta**.

### Ordered factors:

Thus far, we did not differentiate unordered and ordered factors. In fact, though, the first “foreign” example is a nominal variable and the second “opinion” example is an ordinal variable. We know it and sort the vector “opinion” according to the ordering of the Likert scale just above. But R, of course, does not know about it unless you tell it.

```

> ## Example: Ordered factor for ordered (ordinal) variable.
>
> # Does R know "opinion.f2" is an ordinal variable?
> is.ordered(opinion.f2)
[1] FALSE
> # R does not know there is an intrinsic order with this variable.
>

```

I used the function **is.ordered()** to confirm R does not know this variable is ordinal. So, let it know. There are two ways, i.e., add option **ordered=TRUE** to the function **factor()** or use another function **order()**.

```

> # Let's tell R this is ordered
> opinion.o1<-factor(opinion,
+ levels = c("std","sod","nth","soa","sta"), ordered=TRUE)
> opinion.o1
[1] std sta soa soa sod nth nth sta sod
Levels: std < sod < nth < soa < sta
> is.ordered(opinion.o1)
[1] TRUE
>

```

As for the example above that uses the function `factor()`, the default setting of the option `ordered=` is `ordered=is.ordered(vector.you.are.working.on)` which means `FALSE`. Thus, this one needs to be set to `TRUE` (to understand the logic, remember what the function `is.ordered()` did and what it returned [in the just above ## Example](#)).

The same result can be obtained by using the function `ordered()`. This function has the same arrangement as the function `factor()`, but you of course do not have to specify the option to make ordered factors. This function is designed for that purpose.

```
> # This does the same, shorter command.
> opinion.o2<-ordered(opinion, levels = c("std","sod","nth","soa","sta"))
> opinion.o2
[1] std sta soa soa sod nth nth sta sod
Levels: std < sod < nth < soa < sta
> is.ordered(opinion.o2)
[1] TRUE
>
```

Notice the different way R printed the levels for you: `Levels: std < sod < nth < soa < sta`.

### Adding and dropping levels in factors:

Suppose that you think the 6<sup>th</sup> person in the “opinion.f2” data vector is miscoded and it is actually in the “refused to answer (rd)” category, not “neither agree nor disagree.” You may want to change the value for that case accordingly. In the section of [indexing](#), we learned we could make changes to data vectors/data frames by using indexing. For your review, here is what we did in that section.

```
> ## Example: Change factors (add level).
> # Intro
> detach(autocsv1)
> attach(wgthgt)
> bmi
[1] 18.06652 16.81559 20.37681 20.81250 27.02156
> # Change the 3rd case.
> bmi[3]<-20.66251
> bmi
[1] 18.06652 16.81559 20.66251 20.81250 27.02156
>
```

So you might think the same way should work to change your factor and try it. However, R issues a warning message.

```
> # Attempt to do the same, but fail.
> opinion.f2[6]<-"rd"
Warning message:
In `[<-factor`(`*tmp*`, 6, value = "rd") :
  invalid factor level, NAs generated
>
```

What happened? What does the warning mean to our factor variable opinion.f2?

```
> # What does the warning mean to say?
> opinion.f2
[1] std sta soa soa sod <NA> nth sta sod
Levels: std sod nth soa sta
>
```

The 6<sup>th</sup> person is now replaced with a missing value **NA**. Why did this happen? Remember, when we defined the factor “opinion.f2,” we did not include any such factor level as “rd.” So simply, it is not there for our use. As mentioned at the beginning of this subsection “Factor,” R does not allow for any other factor level than those you define, until you explicitly redefine it. You cannot use levels that are not present in your defined factor levels, and otherwise R returns “NA.” This is quite different from many of the commercial statistical packages, and the way R handles this may sound cumbersome. But it is actually good—R safeguards against error entries of impossible values. It forces you to be always clear about your categorical variables.

So below, we will redefine our factor level definition by adding a new factor level “rd” so it is ready and available for use. Then we will replace the value of the 6<sup>th</sup> case with this newly added factor level.

```
> # Change factor def by adding a new level
> opinion.f2<-factor(opinion.f2, levels=c(levels(opinion.f2), "rd"))
> # Now your factor has the new level
> opinion.f2[6]<-"rd"
> levels(opinion.f2)
[1] "std" "sod" "nth" "soa" "sta" "rd"
> opinion.f2
[1] std sta soa soa sod rd nth sta sod
Levels: std sod nth soa sta rd
>
```

Suppose that then you realize the previous factor definition was actually just fine and the 6<sup>th</sup> person can be categorized into the “neither agree nor disagree” group. In this case, you can simply replace “rd” with “nth” for the 6<sup>th</sup> case, because the defined factor levels has the level “nth” in it. However, the level for “rd” is still present in the defined factor levels; it is now simply unused. If you would like to drop this completely, then you can use indexing to drop this unused factor level to fix your factor. See the example below.

```
> ## Example: Change factors (remove level).
>
> # First, change the 6th case's value back to "nth"
> opinion.f2[6]<-"nth"
> opinion.f2
[1] std sta soa soa sod nth nth sta sod
Levels: std sod nth soa sta rd
> # the 6th is now "nth" but the level "rd" remains in the factor, unused.
>
> # Drop this now unused factor level "rd" completely
> opinion.f2<-opinion.f2[drop=TRUE]
```

```

> opinion.f2
[1] std sta soa soa sod nth nth sta sod
Levels: std sod nth soa sta
> # Now "rd" is removed from the factor "opinion.f2"
>

```

The only new thing here is the option **drop=** in indexing, which removes unused level(s). It is set to **TRUE** as we want to drop the unused “rd.”

Factors help you get a clearer idea about your data by providing a meaningful ordering of variables and value labels. Let’s see very simple frequency tables for an example. Suppose you want to cross-tabulate the opinion by party affiliation. In the below example, I create a data vector of party affiliation that corresponds to the nine cases in the opinion data vector, then cross-tabulate the two variables, but first by using the non-factored versions, and then their factors.

```

> ## Example: Some usefulness of factor
>
> # Data vector, 0 = Republican, 1 = Democrat
> party<-c(0,1,1,0,1,0,1,1,0)
>
> # Create a small factor from a vector "party"
> party.f1 <- factor(party, levels=0:1, labels = c("Rep", "Dem"))
> party.f1
[1] Rep Dem Dem Rep Dem Rep Dem Dem Rep
Levels: Rep Dem
> # the variables are not factors
> table(opinion,party)
      party
opinion 0 1
      nth 1 1
      soa 1 1
      sod 1 1
      sta 0 2
      std 1 0
> # the variables are factors
> table(opinion.f2,party.f1)
      party.f1
opinion.f2 Rep Dem
      std    1  0
      sod    1  1
      nth    1  1
      soa    1  1
      sta    0  2
>

```

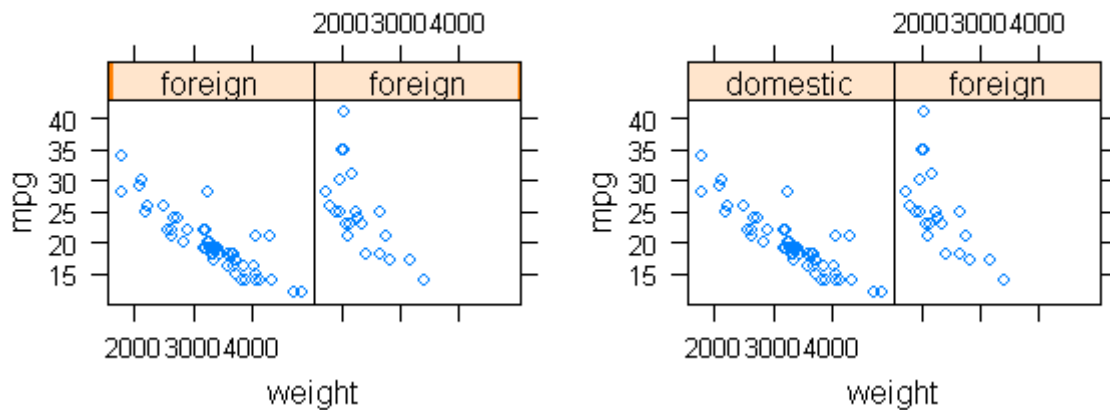
Compare the two tables. The latter one is the better and clearer presentation.

Similarly, it helps with a clear presentation in graphics. Let’s plot cars’ weight and mpg by domestic/foreign (I will use the package “**lattice**” for graphics, which is available to download at: <http://cran.r-project.org/src/contrib/Descriptions/lattice.html>. After download the zip file, from the RGui window, go **Packages >**

Install package(s) from local zip file(s).... to install. Then load it by using the function `library()`. You should also see [here](#) how to install packages in general).

```
> # lattice: http://cran.r-project.org/src/contrib/Descriptions/lattice.html
> # weight and milage by foreign, scatter.
> detach(wgthgt)
> attach(autocsv1)
> library(lattice)
> # The var foreign is not a factor.
> xyplot(mpg ~ weight | foreign)
> # The var foreign.f2 is a factor labeled.
> xyplot(mpg ~ weight | foreign.f2)
>
```

The left one below does not use a factor version of “foreign,” the right one uses a factor version with value labels.



## 9. Recoding

In this section, we will learn:

- How to recode (or create) variables by indexing.
- How to recode variables by the function `cut()`
- How to recode variables by the function `recode()`

In fact, there is not so particularly new thing here. It is just another application of how to use indexing and functions.

### Recode (create) variables by indexing

Let's continue to use the data frame "autocsv1" (it should still be attached, but if not, attach it now). Let's first think of recoding the continuous "price" variable into another categorical variable (low, middle, high). For the sake of example simplicity here, I use the 1<sup>st</sup> and 3<sup>rd</sup> quartile points of "price" as my cut-off lines.

```
> ## Example: Recoding by indexing
>
> # Summarize just to get 1st and 3rd quartile.
> summary(price)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 3291   4220   5006   6165   6332   15910
>
> # First, create a blank factor levels (define levels)
> autocsv1$pricerangel<-factor(NA, levels = c("Low","Middle","High"),
ordered=T)
> autocsv1$pricerangel
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[13] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[25] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[37] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[49] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[61] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
[73] <NA> <NA>
Levels: Low < Middle < High
>
> # Then fill in the blank levels with actual values based on the criteria.
> # It's just to apply the "how-to-index" we already learned.
> autocsv1$pricerangel[price<=4220]<-"Low"
> autocsv1$pricerangel[price>4220 & price<=6332]<-"Middle"
> autocsv1$pricerangel[price>6332]<-"High"
> autocsv1$pricerangel
 [1] Low   Middle Low   Middle High  Middle Middle Middle High
[10] Low   High  High  High  Low   Middle Middle Middle Low
[19] Low   Low   Low   Middle High  Middle Low   High  High
[28] High  Low   Middle Middle Middle Middle Low   High  Middle
[37] Middle Middle Low   Low   High  Middle Middle Middle High
[46] Low   Middle Middle Middle Middle Middle Low   High  Middle
[55] High  Middle Middle Middle High  Middle Middle Middle Low
[64] High  Low   Low   Middle Low   Middle High  Middle Middle
[73] High  High
Levels: Low < Middle < High
>
> detach(autocsv1)
> attach(autocsv1)
>
> table(pricerangel)
pricerangel
  Low Middle  High
   19    36    19
>
```

The function `cut()` is also used to define the same factor as the above. It divides the range of continuous variables into sub-range groupings and codes them according to which grouping they

fall under. The basic usage of this function is: **cut(variable.to.be.divided, breaks)** (feel free to do **help(cut)**). The grouping breaks criteria is defined as a vector (we use **c()**). The leftmost grouping corresponds to level one.

```
> # Example: Do the same by using cut()
> autocsv1$pricerange2<-cut(price, breaks=c(0,4220,6332,16000))
>
> detach(autocsv1)
> attach(autocsv1)
>
> table(pricerange2)
pricerange2
      (0,4.22e+03] (4.22e+03,6.33e+03] (6.33e+03,1.6e+04]
                19                    36                    19
>
> is.factor(pricerange2)
[1] TRUE
> # See, the new variable is a factor.
> levels(pricerange2)
[1] "(0,4.22e+03]"      "(4.22e+03,6.33e+03]"
[3] "(6.33e+03,1.6e+04]"
>
```

By default, the intervals are constructed as “(a, b]” “(b,c]” etc. If the option **right** is set to **right = FALSE** (default = **TRUE**), the intervals are specified “[a,b)”, “[b, c)” etc.

Let’s put on the same value labels on the groupings as those of the first example. Here, we use the function **levels()**. We already learned this function as how to access to the levels attribute of a factor, but here you also learn this function can be used with the usage **levels()<- value** to specify the levels and put labels on them.

```
> # Let's put labels on the levels.
> levels(pricerange2)<-c("Low", "Middle", "High")
> factor(pricerange2)
 [1] Low      Middle Low      Middle High      Middle Middle Middle High
[10] Low      High      High      High      Low      Middle Middle Middle Low
[19] Low      Low       Low       Middle High      Middle Low      High      High
[28] High     Low       Middle Middle Middle Middle Low      High      Middle
[37] Middle Middle Low      Low      High      Middle Middle Middle High
[46] Low      Middle Middle Middle Middle Middle Low      High      Middle
[55] High     Middle Middle Middle High      Middle Middle Middle Low
[64] High     Low       Low       Middle Low      Middle High      Middle Middle
[73] High     High
Levels: Low Middle High
>
> pricerange2<-ordered(pricerange2)
> factor(pricerange2)
 [1] Low      Middle Low      Middle High      Middle Middle Middle High
[10] Low      High      High      High      Low      Middle Middle Middle Low
[19] Low      Low       Low       Middle High      Middle Low      High      High
[28] High     Low       Middle Middle Middle Middle Low      High      Middle
[37] Middle Middle Low      Low      High      Middle Middle Middle High
[46] Low      Middle Middle Middle Middle Middle Low      High      Middle
[55] High     Middle Middle Middle High      Middle Middle Middle Low
```

```
[64] High Low Low Middle Low Middle High Middle Middle
[73] High High
Levels: Low < Middle < High
>
```

I assign different object names to recode variables into separate factors, but you of course can recode variables into the same variables simply by using the same object names.

## 10. Stacking (Appending) And Merging Data Frames

In this final section, we will discuss:

- How to stack (append) data frames
- How to merge data frames

This is yet another straightforward thing. All you need to do is use appropriate functions appropriately.

### Stack data

To stack data, use the function **rbind()**, which combines rows of data. The very basic usage to combine data frame objects is:

```
rbind(dframe1, dframe2...)
```

So we are getting started with a very simple example. We have two auto data files, one of which has information of domestic cars (foreign = 0) and the other of which includes observations where cars are foreign-made (foreign = 1).

```
> ## Example: Stacking files
>
> # Simple example
> detach()
> auto.domestic <- read.csv("auto_domestic.csv")
> auto.foreign <- read.csv("auto_foreign.csv")
>
> auto.append <- rbind(auto.domestic, auto.foreign)
> fix(auto.append)
> # Check if the append worked successfully.
>
```

Now, let's take a look at another example to understand how R handles factors when stacking data frames. As we just learned above, R sticks to factor definitions we set up until we explicitly redefine them. The question, then, is "What if two data frames we want to stack each have a factor named the same but differently defined?"

Here in this example, we have two data frames, gss94.wrk1 and gss94.wrk2. Let's see details of those data frames.

```
> # Example: What happens to factors?
> load("ex.stacking.RData")
>
> ls()
[1] "auto.append"    "auto.domestic" "auto.foreign"  "gss94.wrk1"
[5] "gss94.wrk2"
>
> str(gss94.wrk1)
'data.frame':  20 obs. of  6 variables:
 $ wrkstat: Factor w/ 4 levels "Working full time",...: 1 1 2 2 1 3 1 3 3 4 ...
 $ marital: Factor w/ 5 levels "Married","Widowed",...: 1 1 5 5 1 4 1 4 1 1 ...
 $ age    : int  NA 59 21 22 34 71 38 59 28 61 ...
 $ educ   : int  12 8 13 15 16 12 13 12 12 15 ...
 $ race   : Factor w/ 3 levels "White","Black",...: 2 1 2 2 1 2 2 2 2 1 ...
 $ sex    : Factor w/ 2 levels "Male","Female": 2 1 2 2 1 2 2 2 2 2 ...
> str(gss94.wrk2)
'data.frame':  20 obs. of  6 variables:
 $ wrkstat: Factor w/ 4 levels "Retired","In school",...: 1 NA 3 4 1 3 3 3 4 1 ...
 $ marital: Factor w/ 5 levels "Married","Widowed",...: 1 5 3 3 5 4 3 1 5 1 ...
 $ age    : int  61 54 33 58 65 38 55 30 39 67 ...
 $ educ   : int  12 9 12 12 16 12 16 16 13 11 ...
 $ race   : Factor w/ 3 levels "White","Black",...: 2 2 2 1 1 2 2 1 3 2 ...
 $ sex    : Factor w/ 2 levels "Male","Female": 1 2 2 2 1 2 2 2 1 1 ...
>
> levels(gss94.wrk1$wrkstat)
[1] "Working full time"
[2] "Working part-time"
[3] "With a job but not at work b/c of illness, vacation, or strike"
[4] "Unemployed, laid off, looking for work"
> as.numeric(gss94.wrk1$wrkstat)
 [1] 1 1 2 2 1 3 1 3 3 4 1 4 3 1 1 2 2 1 1 4
> levels(gss94.wrk2$wrkstat)
[1] "Retired"          "In school"        "Keeping house"    "Other"
> as.numeric(gss94.wrk2$wrkstat)
 [1] 1 NA 3 4 1 3 3 3 4 1 2 3 1 2 1 4 NA 3 2 1
> # So the wrkstat variable is internally coded the same way.
>
```

We first use the function `str()` to obtain the basic data structure information. As you can see, both of the two data frames have a work status factor “wrkstat.” However, this factor is defined differently between the two data frames. `gss94.wrk1` has respondents who are in labor force and the factor `wrkstat` is defined as having four associated levels in labor force. The `gss94.wrk2` data frame has cases out of labor force and the factor `wrkstat` is defined as having a different set of four associated levels. We display the internal numeric expression of this factor by using `as.numeric()` and find that the `wrkstat` factor is internally coded the same way across the two data frames, ranging 1 to 4, despite the fact that the actual category contents differ. So, for example, in `gss94.wrk1`, the level 1 is “Working full time” whereas the level 1 is “Retired” in `gss94.wrk2`.

Now, the question. What happens to this “wrkstat” factor if we stack these two data frames?

```

> # Stack the data frames as they are.
> wrkstat.append <- rbind(gss94.wrk1, gss94.wrk2)
>
> # What happened to wrkstat?
> levels(wrkstat.append$wrkstat)
[1] "Working full time"
[2] "Working part-time"
[3] "With a job but not at work b/c of illness, vacation, or strike"
[4] "Unemployed, laid off, looking for work"
[5] "Retired"
[6] "In school"
[7] "Keeping house"
[8] "Other"
>
> as.numeric(wrkstat.append$wrkstat)
[1] 1 1 2 2 1 3 1 3 3 4 1 4 3 1 1 2 2 1 1 4 5
[22] NA 7 8 5 7 7 7 8 5 6 7 5 6 5 8 NA 7 6 5
> str(wrkstat.append)
'data.frame': 40 obs. of 6 variables:
 $ wrkstat: Factor w/ 8 levels "Working full time",...: 1 1 2 2 1 3 1 3 3 4 ...
 $ marital: Factor w/ 5 levels "Married","Widowed",...: 1 1 5 5 1 4 1 4 1 1 ...
 $ age : int NA 59 21 22 34 71 38 59 28 61 ...
 $ educ : int 12 8 13 15 16 12 13 12 12 15 ...
 $ race : Factor w/ 3 levels "White","Black",...: 2 1 2 2 1 2 2 2 2 1 ...
 $ sex : Factor w/ 2 levels "Male","Female": 2 1 2 2 1 2 2 2 2 2 ...
>
> # So the factors have their levels expanded as necessary!
>

```

So, what happened is the factor wrkstat has its levels expanded as needed when you stack those two data frames. That is how R handles factors in stacking data frames. Here, R makes the factor wrkstat an eight-level factor and assigns internal numeric coding 5 to 8 to those levels for out-of-labor-force statuses.

## Merge data

Now, finally, let's learn how to merge data frames. To perform matched merge, we use the function **merge()**. The basic usage is:

***merge(dframe.x, dframe.y, by = the same by | by.x = by, by.y = by, all = FALSE)***

In common statistics applications, such as SAS and Stata, you merge data by the same by-variable shared by multiple data files. R, of course, can do the same. See the following example.

```

> ## Example: Merging files
>
> # Read mycensus1.csv
> detach()
> mycensus1 <- read.csv("mycensus1.csv")
>
> # Read mycensus2.csv

```

```

> mycensus2 <- read.csv("mycensus2.csv")
>
> # First 1 obs.
> mycensus1[1:10,]
  stateno      state region      pop poplt5 pop5_17  pop18p  pop65p
1         2    Alaska   West  401851  38949   91796  271106  11547
2         50  Wyoming   West  469557  44845  100708  324004  37175
3         45  Vermont    NE   511456  35998  109320  366138  58166
4          8  Delaware  South  594338  41151  125444  427743  59179
5         34 N. Dakota N Cntrl 652717  54752  136239  461726  80445
6         41 S. Dakota N Cntrl 690768  58446  147160  485162  91019
7         26  Montana   West  786690  64455  167440  554795  84559
8         28  Nevada    West  800493  56132  159667  584694  65756
9         44  Utah      West 1461037 189962  350143  920932 109220
10        27  Nebraska N Cntrl 1569825 122946  324224 1122655 205684
> mycensus2[1:10,]
  stateno popurban medage death marriage divorce
1         8   419819   29.8   5123     4437    2313
2        45  172735   29.4   4587     5226    2623
3         2  258567   26.1  1604     5361    3517
4        34  318310   28.3  5596     6094    2142
5        50  294639   27.1  3215     6868    4003
6        39  824004   31.8  9300     7490    3606
7        26  416402   29.0  6664     8336    4940
8        41  320777   28.9  6523     8800    2811
9        29  480325   30.1  7594     9251    5254
10       11  834592   28.4  4849     11856   4438
>
> # Example: by-variable with the same name
> # We have "stateno" as common identifier.
>
> census.same.id <- merge(mycensus1, mycensus2, by="stateno", all=T)
>
> census.same.id[1:10,]
  stateno      state region      pop  poplt5  pop5_17  pop18p
1         1    Alabama  South  3893888  296412  865836  2731640
2         2    Alaska   West   401851   38949   91796   271106
3         3    Arizona   West  2718215  213883  577604  1926728
4         4    Arkansas  South  2286435  175592  495782  1615061
5         5  California   West  23667902 1708400 4680558 17278944
6         6    Colorado   West  2889964  216495  592318  2081151
7         7 Connecticut    NE   3107576  185188  637731  2284657
8         8    Delaware  South  594338   41151  125444  427743
9         9    Florida   South  9746324  570224 1789412  7386688
10        10   Georgia   South  5463105  414935 1231195  3816975
  pop65p popurban medage death marriage divorce
1  440015 2337713   29.3  35305   49018   26745
2  11547  258567   26.1  1604     5361    3517
3  307362    NA     NA     NA     NA     NA
4  312477 1179556   30.6  22676   26513  15882
5 2414250 21607606   29.9 186428   210864  133541
6  247325    NA     NA     NA     NA     NA
7  364864 2449774   32.0  26005   26048  13488
8   59179  419819   29.8   5123     4437    2313
9 1687573  8212385   34.7 104190  108344  71579
10 516731  3409081   28.7  44230   70638   34743
>

```

Simple, isn't it?

There are a couple of things I want to draw your attention to here, though. First, notice that I didn't sort the two data frames before merging operation. Unlike Stata or SAS, there is no need to sort each data frame by the by-variable beforehand; simply specify the common identifier and R takes care of the rest.

Second, don't forget to specify the option **all = TRUE** to keep all the observations from each of the data frames, rather than keep the observations from both of the data frames only (**all = FALSE**). The default setting is **FALSE**, so you need to explicitly specify this option to **TRUE** every time (you might find this annoying, but this is actually because of R's merge flexibility; it allows for various database join operations, and **all = TRUE** is just one of them).

And finally, notice that the merged data frame is already sorted by the id-variable for you. That's the default setting (**sort = TRUE**).

And here is another thing. In R, the common identifier actually does not have to have the same name across multiple data frames to be merged. You just need to specify the common columns following **by.x =** and **by.y =** inside **merge()**. See the example below—we are merging `mycensus1` and `mycensus3`, which have differently named id variables, respectively.

```
> # Read mycensus3.csv
> mycensus3 <- read.csv("mycensus3.csv")
>
> # First 10 obs.
> mycensus3[1:10,]
  stateid popurban medage death marriage divorce
1      8  419819  29.8  5123   4437   2313
2     45  172735  29.4  4587   5226   2623
3      2  258567  26.1  1604   5361   3517
4     34  318310  28.3  5596   6094   2142
5     50  294639  27.1  3215   6868   4003
6     39  824004  31.8  9300   7490   3606
7     26  416402  29.0  6664   8336   4940
8     41  320777  28.9  6523   8800   2811
9     29  480325  30.1  7594   9251   5254
10    11  834592  28.4  4849  11856   4438
>
> # So, mycensus1 has the id named "stateno" but mycensus3 has the
> # id named "stateid." They just have different names.
>
> # Example: by-variable with different names
>
> census.diff.id <- merge(mycensus1, mycensus3,
+ by.x="stateno", by.y="stateid", all=T)
> census.diff.id[1:10,]
  stateno      state region      pop  poplt5  pop5_17  pop18p
1        1  Alabama  South 3893888 296412  865836 2731640
2        2  Alaska   West  401851  38949  91796  271106
3        3  Arizona  West 2718215 213883 577604 1926728
4        4  Arkansas South 2286435 175592 495782 1615061
```

```

5      5  California  West 23667902 1708400 4680558 17278944
6      6  Colorado   West 2889964  216495  592318  2081151
7      7  Connecticut NE  3107576  185188  637731  2284657
8      8  Delaware   South 594338  41151  125444  427743
9      9  Florida    South 9746324 570224 1789412 7386688
10     10  Georgia    South 5463105 414935 1231195 3816975
      pop65p  popurban  medage  death  marriage  divorce
1  440015      NA      NA      NA      NA      NA
2  11547      258567  26.1  1604      5361  3517
3  307362  2278728  29.2  21226  30223  19908
4  312477  1179556  30.6  22676  26513  15882
5  2414250 21607606  29.9 186428  210864 133541
6  247325  2329869  28.6  18925  34917  18571
7  364864  2449774  32.0  26005  26048  13488
8   59179  419819  29.8   5123   4437  2313
9 1687573  8212385  34.7 104190 108344  71579
10 516731  3409081  28.7  44230   70638  34743
>

```

Finally (and this is sort of a sidebar), R does not generate an equivalent of Stata's `_merge` variable after merging, an indicator of which data frame each observation comes from. If you need such a variable, therefore, create one yourself. Various ways to do that, and the below is just one simple example.

```

> # Example: Stata _merge like indicator variable.
>
> mycensus1$merge1 <- 1
> mycensus2$merge2 <- 1
>
> census.mergevar <- merge(mycensus1, mycensus2, by="stateno", all = T)
>
> attach(census.mergevar)
> census.mergevar$merge[merge1 == 1 & merge2 == 1] <- 3
> census.mergevar$merge[is.na(merge1)] <- 2
> census.mergevar$merge[is.na(merge2)] <- 1
>
> census.mergevar <- subset(census.mergevar, select=c(-merge1, -merge2))
>> census.mergevar[1:15,]
  stateno      state  region      pop  poplt5  pop5_17  pop18p
1       1  Alabama  South 3893888  296412  865836  2731640
2       2  Alaska   West  401851   38949   91796  2711106
3       3  Arizona   West 2718215  213883  577604  1926728
4       4  Arkansas  South 2286435  175592  495782  1615061
5       5  California West 23667902 1708400 4680558 17278944
6       6  Colorado   West 2889964  216495  592318  2081151
7       7  Connecticut NE  3107576  185188  637731  2284657
8       8  Delaware   South 594338  41151  125444  427743
9       9  Florida    South 9746324 570224 1789412 7386688
10      10  Georgia    South 5463105 414935 1231195 3816975
11      11      <NA>  <NA>      NA      NA      NA      NA
12      12      <NA>  <NA>      NA      NA      NA      NA
13      13  Illinois  N Cntrl 11426518  842241 2400796  8183481
14      14  Indiana  N Cntrl  5490224  418764 1199554  3871906
15      15  Iowa    N Cntrl  2913808  221628  604245  2087935

```

	pop65p	popurban	medage	death	marriage	divorce	merge
1	440015	2337713	29.3	35305	49018	26745	3
2	11547	258567	26.1	1604	5361	3517	3
3	307362	NA	NA	NA	NA	NA	1
4	312477	1179556	30.6	22676	26513	15882	3
5	2414250	21607606	29.9	186428	210864	133541	3
6	247325	NA	NA	NA	NA	NA	1
7	364864	2449774	32.0	26005	26048	13488	3
8	59179	419819	29.8	5123	4437	2313	3
9	1687573	8212385	34.7	104190	108344	71579	3
10	516731	3409081	28.7	44230	70638	34743	3
11	NA	834592	28.4	4849	11856	4438	2
12	NA	509702	27.6	6753	13428	6596	2
13	1261885	9518039	29.9	102230	109823	50997	3
14	585384	3525298	29.2	47300	57853	40006	3
15	387584	1708232	30.0	26348	27474	11854	3
>							

This is the end of The Very Basics of R. Thanks for playing!