the star lab's Introduction to R: A Short Course

Prepared for the starlab by J.P. Olmsted

Updated: July 21, 2011

Contents

Introduction to this Short Course

About. This document was prepared by the author during his tenure as **the star lab** Fellow for the incoming graduate student cohort in the Political Science department at the University of Rochester. Previous versions of the short course and the corresponding materials are available at **the star lab**'s website (`http://www.rochester.edu/college/psc/thestarlab/main/index.php`). This document is an adaptation of Arthur Spirling's version. In addition to the supplemental files used for the various exercises included in this version, this document is available from both the starlab's website and the author's website (`http://www.rochester.edu/college/gradstudents/jolmsted/`).

Comments. If you have any comments, questions, or concerns regarding this document please contact Jonathan Olmsted (`jpolmsted@gmail.com`).

Description of Course. This short course was covered in five days. Parts 1 and 2 are covered in one day and the remaining parts are covered on their own days. Part 1 discusses the whats and whys of `R`. Part 2 is a brief introduction to the fact that everything in `R` is an object and that objects have traits like *classes* and *modes*. Part 3 is the first journey into practice with the `R` language. We walk through an example of how to create and manipulate various objects in a way related to

Part I
An Introduction to Using R

This part of the short course introduces you to R and provides a basic overview of its use.

1 What is R?

- a statistical scripting language (i.e. it was created with data in mind and you can use the language interactively)
- a statistical environment (i.e. it is a program you open up and interact with)
- open-source, free, compatible with most platforms, and constantly updated
- the most common platform for quantitative work in political science (or at least I am claiming this)

You can find out more about R at `http://www.r-project.org/`.

2 Why Would You Use R?

- you will have to use it for PSC 405 and PSC 505 problem sets
- it is the "go to" program for non-standard numerical analysis in political science
- it was created with data in mind (so you aren't forcing a square peg in a round hole)
- the language and your code is as much a way of communicating a set of instructions to other researchers as it is to your computer
- the graphics capabilities of R are un-rivaled among statistics software
- R can be anything you want it to be because it is so easily extended

Since you are now convinced that you would like to use R, you can download it at `http://www.r-project.org/index.html`. However, you won't need to download it if you are in **the star lab**. It is already installed on these computers.

3 R and Your Workflow

The role that R will play in your work and how it will fit into your workflow will vary. However, there is a general structure to it all.

1. come up with a numerical problem to solve
2. conceptualize a set of steps to solve this problem
3. translate these steps into the R scripting language
4. run your R code
5. debug your R code
6. produce graphics (if applicable)
7. consume you results (either textual or graphical)

Step 1. R will not tell you what you want to do. It can help you get there, but you simply must know where you want to go. The problem to be solved can range from *calculating the sampling distribution of the sample mean of samples of five random draws from a distribution* to *identifying the treatment effect of being exposed to negative campaign advertisements on voter turnout.* In general, it is a bad sign to have already opened R if you don't know what you want to do.

Step 2. If R is your first computer language, this step will likely be tricky. This is all about translating a human understanding of a problem into a machine-ready algorithm. For some people this is easier than it is for others. For all people, though, practice makes it easier.

Step 3. This will be the hardest step. As you use R, you understanding of what is and is not possible in R will evolve. The code you write will take on incredibly different styles.

Because you are writing your R code in a text-editor it will be a plain text file (like in LaTeX!).

When you write your R code, I recommend doing several things:

- **do not use** the feature-less script editor in R itself
- **do use** a feature-full editor that integrates well enough with R (e.g. RWinEDT on **the star lab** computers)
- **do use** an editor to write your code and then run it at the R prompt (this maintains transparency and reproducibility).

Step 4. Running your code can happen at an interactive `R` shell or in "batch mode". The latter option will most likely not be used by you any time soon. For our purposes we will run code interactively. But, that is different from performing operations "on the fly". If you write your code in RWinEdt, the toolbar icons provide a simple way of running your already written code at the interactive shell.

Step 5. There is a built-in debugger in R and there are some external packages which provide debugging facilities. I neither use nor recommend these. For the most part, there is nothing you can't debug with just some cleverness and the interactive command line. In reality, Steps 4 and 5 are iterative and repetitive. In it's idealized form, you might write your code in a single shot and run it without errors. In practice, this will not happen. The point is, though, that for any given project, you'll have it set up so that it can finally run without intervention or correcting. It will be as if you wrote completely correct code and sent it all to `R`.

Step 6. `R` is known for the ability to produce great graphical output. There are several different "frameworks" to produce graphics. We will only cover the *base* graphics in this tutorial.

Step 7. How you consume your output is up to you. But, consumption is not a one-shot deal. Therefore, part of this step is ensuring reproducibility. Also, you will want to "export" the product of your hard work to a non-`R` format. This can be a plain text file of your results or a `.pdf` file with a figure you created.

4 The VERY Basics of the `R` Interpreter

4.1 The Prompts

`R` tells you it is ready for input by displaying a ">". `R` knows its own language, so if you hit *enter* and you expect something to evaluate, but `R` keeps giving you a "+" (to tell you it needs more) there is a syntax error. To get out of the perpetual "+" prompts, hit *ctrl* + *c* or *esc*. When you get this problem, there is likely to be a missing "`"`", "`,`", or "`)`".

At the end of a command, you hit *enter* if you are at the command line. If you are writing your script file in a text editor, simply hit *enter* to add a line break and start a new line.

4.2 Comments

`R` interprets anything after a "`#`" as a comment. The comment ends and `R` begins to interpret code once a new line is started. Multi-line comments require a new comment character on each line.

4.3 Spaces

`R`, in general, ignores spaces between correctly entered objects and operators. Spaces are not part of the names of objects or operators. Spaces may be contained in character strings and they matter.

4.4 Multiple Commands on a Physical Line

In `R`, you can place two (or more) complete sets of commands on the same physical line (i.e. without hitting *enter*). You do this by separating them with a "`;`". However, this is not recommended and the utility of it is questionable.

5 Exercises

These short exercises are intended to familiarize you with entering commands at the prompt and the various kinds of windows that might open as a result.

1. Start `R` on the machines in **the star lab**.
2. If the `RWinEdt` package is not automatically loaded, type `library(RWinEdt)`. Hit *enter*.
3. Type `sessionInfo()` at the command prompt (i.e. `>`) and hit *enter*. Your are running the function `sessionInfo` without any arguments. `R` is "computing" the output of the function and then "printing" the output with an "appropriate" method.
4. Type `citation()` at the command prompt and hit *enter*. Recall the mention of BibTeX from the LaTeX course? `R` is "computing" the output of the function `citation` and then "printing" the output with an "appropriate" method.
5. Enter `demo(graphics)` at the prompt. Progress through the various demonstrated plots until you have seen them all. Close the plot window. `R` is "computing" the output of the function `demo` which has been passed the argument `graphics`. In fact, `graphics` is a package which contains functions for creating graphics. Would you like to know something about it? Enter `?graphics` at the command line. This displays the help page for the `graphics` package. Close this.
6. Enter the following in WinEdt. You do not need to save this R script.

```
###
### Region 1
###
# is this 2 or 3?
1 + 1 # + 1
###
```

```
###
### Region 2
###
  1+ 1   +
1
###

###
### Region 3
###
"asdf " == "asdf"
###
```

7. Evaluate each "region" one at a time. So, select the region with your cursor, and press the RWinEdt toolbar button with the word "paste" on it. This pastes the selected region into the `R` shell.

8. Type `1 +` and hit *enter*. Hit *enter* a number of times. How do you get out of this?

9. Enter `q()` at the prompt. Respond "`N`" (no).

Part II
Fundamentals of Working in R

This part of the course is an introduction to the different building blocks in `R`, objects.

6 Objects, Modes, and Attributes ... *Oh my!*

The following description is typically not covered in introductions to `R`. It may seem weird, un-welcoming, and esoteric. In some sense, it is. But, the reality is, if you can keep this in mind as you become familiar with `R` you will understand why and how things happen.

6.1 Objects

Within `R`, the basic units or entities are called "objects". The following are the basic objects that you will use as you learn `R`. There are many more.

vectors These are exactly what you think they are. A sequence of elements of the same *mode*. We'll talk about modes in a second. Vectors have order. You can have a vector of length 26 (i.e. 26 elements) where each element is a letter in the alphabet. This vector's mode will be `character`.

lists Lists are similar to vectors. They are sequences of elements. They have order. However, each component of a list need not be of the same mode. You can have a list with two elements. The first element being a vector containing the 26 characters in the English alphabet and the second element being a vector contain the 26 relative frequencies of these characters as they show up on the Political Science Department's website. These two vectors are not of the same mode and could not be combined into a single stacked vector. But, we can combined them in a list.

matrices/arrays Matrices are 2-dimensional rectangular objects. Arrays are higher dimensional "rectangular" objects. Each element must be of the same mode (e.g. everything is a character sting, logical value, numerical value, etc.)

dataframes Dataframes can be thought of as *special* matrices. However, they technically are *special* lists. In general, you will tend to store your empirical data in dataframes. These objects are two dimensional containers with the rows corresponding to "observations" and the columns corresponding to "variables".

factors These are vectors which are intended to help classify categorical data. They are handle differently than numerical, integer, or character vectors. Notice, though, factors are a different class of object, not a different mode.

functions Functions, although less obviously so, are also objects. The details of functions vary, but the general idea is simple. This object takes other objects (of any mode), performs even more basic functions on those objects and returns some final object (of some arbitrary mode).

6.2 Modes

logical elements of either `TRUE` or `FALSE`

integer integer numbers like 1, 2, or -539

numeric real numbers; don't use 1.00, 2.00, and 3.00 when 1, 2, and 3 will do

complex complex or imaginary numbers; you will most likely not use them much in applied political methodology but they can come up!

character elements made up of "text"; each character string can be some arbitrary length; `''1''` is not `1` in `R`

raw you can safely ignore this mode!

6.3 Attributes

The command `attributes()` can be used to set and extract the attributes of an arbitrary `R` object. Attributes can be accessed through this general function or through particular functions which are focused on a particular attribute. We won't cover how to do this or why we would do this, here is an example of some attributes:

1. the length of a vector of any mode (`length()`)
2. the size of a matrix or a higher dimensional array (`dim()`)
3. the column names of a dataframe (`colnames()`)

7 Assignment, References, Scope

So far, we've superficially described objects in `R`. Everything is an object! Our data are an object. The results from a linear regression are an object. The function that runs a linear regression is an object.

Once we have created an object that we need, we can store it as its own named object instead of re-computing it. So, if we want to save the value of 2π and recall it without having to continuously multiple the two factors, we can create an object called `vTwoPi` (or `BillRiker` for that matter).

After we have assigned this value to a name, we can recall it in various ways.

7.1 Assignment

The standard assignment operator is `<-`. One *could* also use `=`, but you should not. We will keep `=` for another use. Assignment can happen in the other direction, `->`. The general form of the use of the assignment operator is `name <- object`. Although quotes aren't necessary, `"name" <- object` is identical.

An object is saved with the name `name` once this command is run regardless of if an object of name `name` exists or not. You can easily overwrite an object this way. You may want to or you may not.

7.1.1 Naming Rules

However, when you name an object, you must follow certain rules.

Briefly, these are:

> Identifiers consist of a sequence of letters, digits, the period ('.') and the underscore. They must not start with a digit nor underscore, nor with a period followed by a digit.[1]

In addition `TRUE`, `FALSE`, `Inf`, `NaN`, `NA`, `NULL`, `if`, `else`, `for`, `in`, `while`, `next`, and `break` are reserved words.

I recommend the following convention for naming objects in `R`. If I want to associate the name "foo" or "Foo" with an object and it is a vector (or factor vector), I name it `vFoo`. For objects of the other classes, I use `lFoo`, `dfFoo`, and `mFoo`.

The exception to this pattern is my naming of functions. Then, I use action words and capitalization to name the function `DoFoo` or `CalcFoo`.

7.2 References

If a vector object is named `vFoo` and you would like to display or "print" the object, type `vFoo` and hit *enter*. Notice, `print(vFoo)` will do the same thing. `R` automatically calls the `print()` function when you simply enter the object name.

You can achieve the same thing with the more powerful function `get()`.

If you are at the `R` prompt, `R` will help you choose the full name of an object you are referring to. This is called tab completion. If you type the beginning of a name of an object that `R` knows about and then hit *tab* twice in succession, `R` will print out a list of object names (e.g. vectors, functions, etc.) that are possible completions of the partial name you typed. For example, type "`print`" and invoke the tab completion. Look how many objects `R` knows about that begin with these five letters.

[1] `http://cran.r-project.org/doc/manuals/R-lang.html#Identifiers`

7.3 Scope

The scoping rules of the `R` parser are slightly beyond this tutorial. But see `http://cran.r-project.org/doc/manuals/R-lang.html#Scope` to understand them. The `R` workspace is a hierarchy of layers. Objects can exist in any of these layers (or environments/frames) and the scoping rules determine which objects are being manipulated or created. `.GlobalEnv` (an object) is the user's workspace and this is where we will focus our attention for the time being.

8 Asking for Help

Asking for help in `R` can be tricky for the following reasons:

- Depending on who you ask you may not like the response (e.g. some people have been ridiculed on mailing out questions that are poorly formed). These communities have norms, if you want help, observe the norms before you start shooting off emails.
- The language you use matters when you are searching. Knowing the right language is often half of the problem.
- Not everyone is solving your exact problem. So you have to abstract away from your problem, find an applicable solution to the more general problem and then apply it to your specific case.

Follow these steps:

1. Begin looking for help/clarification in the official `R` documentation.
2. If that fails, try a general web search with google. You will most likely want to append "cran r" to whatever topic you are looking for help with instead of just "r". The latter will not filter out unrelated sites too well.
3. Next, turn to the `R` specific search engines.
4. Try re-wording your query. Read up on related topics to see if there are any hints on what might create a more successful search.
5. Ask a colleague in person. It is easier to explain these problems in person with code in front of you than communicating to someone else in the `R` community.
6. Lastly, do your homework before you post in a public setting. It is not scary and people are helpful. But, they also tend not to be too patient with people who look for help without reading the mailing list or forum posting guidelines. No need to rock the boat.

8.1 R Documentation

If you want to know about the object `foo`, you can type `help(‘‘foo’’)` or `?foo`. If `foo` is not an object in your current session with documentation, you will have no results.

If you want to search the `R` help files for a concept or topic `foo`, try `??foo` or `help.search(‘‘foo’’)`.

Next, try `help.start()`. This will bring up the HTML interface to the help facilities. Here, you can find HTML versions of the `R` manuals which are worth searching. You can also browse package-specific documentation through this interface.

8.2 R Community

Stack Overflow <`http://stackoverflow.com/questions/tagged/r`> If you have looked for an answer to how to do something and have found nothing, try posting here first. Posting to the mailing list is a bad idea and can be embarrassing. Stack Overflow tends to be more friendly.

R Meta Search <`http://search.r-project.org/`> You can search just about every official web page and mailing list from this site.

R Seek <`http://www.rseek.org/`> Less impressive meta-search engine.

R Graphics Gallery <`http://addictedtor.free.fr/graphiques/`> Site with many examples of graphs and the code that generates them. Useful for getting started with fairly complex graphical output.

inside-R <`http://www.inside-r.org/`> Community site run by Revolution Analytics. Blog posts, a forum, snippets, etc.

R-Forge <`http://r-forge.r-project.org/`> Development platform for `R` packages. You might find what you need here before it is found elsewhere.

R-wiki <`http://rwiki.sciviews.org/doku.php`> Pretty low traffic site site. However, many useful code snippets. Navigating this site can be an art.

9 Exercises

Enter all of the code in these exercises in a text file. Give it a `.R` file extensions. Add some useful information at the top about what this file by using comments. Run the code using RWinEdt’s toolbar.

1. Create two numerical vectors:

```
vA <- c(1, 3, 5, 7)
vB <- (vA + 3) / 2
```

2. Create a third vector by comparing the value of `vA` and `vB`: `vC <- vA > vB`
3. Use `print()` to inspect each vector. Now inspect each vector by referencing it on its own line.
4. What type of vector is `vC`? Try `is(vC)`.
5. What type of object is the output of `is(vC)`? Try `is(is(vC))`.
6. Coerce `vA` to a character vector with `as.character()`. What happens? If you get an error, use `?as.character`.
7. Run `is.logical(is.logical(as.character(vA)))`. Does the output make sense? What kind of object is the output?
8. Create a matrix where `vA` and `vB` are the rows using `rbind(vA, vB)`. Now do the same with the columns with `cbind(vA, vB)`. Confirm that each of these is a matrix using either `is()` or `is.matrix()`. How do the output of these two differ?
9. Search for help on `data.frame()` using `?`. Knowing how to read `R` help pages is a critical skill.
10. Create a dataframe from `vA`, `vB`, and `vC`.
11. Find two `R` packages that would help you do *ideal point estimation*.

Part III
Case One: Simple Object Creation, Manipulation, Summarizing Objects, Plotting, and Bookkeeping

This part of the short course is a guided walk-through some fictional analyses. We will create some data. Recode and summarize this data. Then, we will make some simple plots. Lastly, we learn how to save an `R` object for later use, and discuss object bookkeeping.

10 Simple Object Creation and Manipulation

10.1 Vectors

We will start this walk-through by creating a vector of names:
`vPeople <- c("Tyson", "Jonathan1", "Gary", "Peter", "Jonathan2")`.
`vPeople` is a vector of character strings. Create three more vectors as follows:

```
vEoM <- c(FALSE, TRUE, FALSE, TRUE, NA)

vAge <- c(28, 25, 25, 30, 25)

vState <- c("TX", "CT", "CA", "DE", "MO")
```

Each of these vectors is of a different mode. You can verify this with `is()`. The `c()` function lets us concantenate elements of the same mode and form a vector.

One technical thing to note is that "constants" like 1 or 4 are vectors of length 1.

Two commands you may use from time to time are `seq()` and `rep()`. Lastly, don't forget about the use of `:` to create consecutive integer sequences.

10.2 Pseudo-Random Numbers and Using Functions

To create several more "variable" vectors we will use `R`'s pseudo-RNG capabilities. Consider `?rnorm`. The most common probability distributions will have a page like this. So, for the Normal distribution we see that:

`rnorm` generates random values

`dnorm` is the density function (i.e. $f(x)$)

`pnorm` is the distribution function (i.e. $F(x)$)

`qnorm` is the quantile function

Because these are so closely related they appear on the same help page. In the case of `rnorm`, we see that the function takes three named arguments. The names are `n`, `mean`, and `sd`. The last two of these have default values (0 and 1, respectively). This means that you can evaluate a call to `rnorm` without explicitly setting a value for `mean` or `sd`.

Run `rnorm(4)` three times at the shell prompt. Then run `a <- rnorm(4)`. Now, reference `a` at the shell prompt four times. Why are these different?

Using the RNG capabilities we create three more vectors:

```
vVar1 <- rbinom(5, 2, .5)
vVar2 <- rnorm(length(vPeople), vAge)
vVar3 <- rnorm(sd = vAge, mean = vAge, n = length(vPeople))
```

Notice that when we call the function, the arguments do not need to follow the order they are listed in the help file exactly. How does `R` match arguments if they aren't always necessary and if the order can change? The `R` manuals explain the precise set of rules, but there is a shortcut to not having to worry about this. If an argument is a named argument in the help file, explicitly name it. So, run `rnorm(n = 4)` and not `rnorm(4)`.

10.3 Basic Mathematical Operators

We can perform arithmatic in the natural way. The command `4 + 6` evaluates as we would expect. Other operators like `-`, `/`, `*`, `%%`, `exp()`, `log()`, `^`, and `abs()` work, too. Create this relatively complicated object using the arithmatic operators, the objects we have already created, and another call to the RNG.

```
vVarY <- (4 * exp(vVar1) +
          abs(vVar2) + (vVar3) ^ (2) +
          runif(vPeople, -1, 1)
          )
```

We know that `vVar1` is a vector. What is `exp(vVar1)`? It too is a vector. How can you check this? `R` evaluates the `exp` function element by element and then concantenates the result. Similarly, `4 * exp(vVar1)` is formed by multiplying each element in the vector we just created by 4 and then concantenating the results. `R` adds vectors the same way, element by element.

`R` has trigonometric functions like `sin()` and `cos()`, too.

Scientific notation is achieved by typing `1e3` and `1e-1`.

10.4 Matrices

Recall that matrices must be of the same mode. How are the following two objects different?

```
cbind(vVar1, vVar2, vVar3)
cbind(vVar1, vVar2, vVar3, vState)
```

`R` trys hard to make the code you give it work. Often times, this means it will coerce objects into different modes so that the code can run. Here, numerical elements are coerced to character strings.

Create the object `mNums <- cbind(vVar1, vVar2, vVar3)`. We can get information about its dimensionality with `dim()`, `nrow()`, and `ncol()`. We can refer to just the elemnts on the main diagonal with `diag(mNums)`. Interestingly enough, we can create a square, diagonal matrix with of size 6, for example, with `diag(6)`. The arithmatic operators work on matrices just like they do on vectors (i.e. element-wise). Try a few on `mNums`.

Now, there are also matrix-specific operations. We can transpose the the matrix with `t()`. We perform matrix multiplication with `%*%`. We can calculate the determinent of a square matrix with `det()`. We can invert a square matrix with `solve()`

Sometimes the rows and columns have labels which makes life easier for users. These are stored as a vector of character strings and this is, not surprisingly, an object itself. See `?rownames` and `?colnames`.

10.5 Dataframes

Create a data frame with the following code:

```
dfData <- data.frame(vPeople, vEoM, vAge, vState,
                     vVar1, vVar2, vVar3, vVarY
                     )
```

Print out the object `dfData`. Notice that character strings become factors. The numbers remain numerical elements and are not coerced. Lastly, we can see that the logical values have remained as logicals. This happens because dataframes allow us to mix modes in a way matrices do not. See `?data.frame` for more information on creating them.

10.6 Indexing and Subsetting

We can refer to subsets of vector, matrix, and dataframe objects. Vectors have a single dimension (i.e. length). So, we simply specify the positions corresponding to the elements we wish to extract from the vector. Try:

```
vPeople[1]
vPeople[-1]
vPeople[c(1,4,2)]
```

Matrices have two dimensions. So, to refer to particular elements, specify the exact row and column as in `mNums[2, 3]` or `mNums[c(2,1), 3]`. You can refer to entire columns with `mNums[2, ]` and entire rows with `mNums[, 3]`.

Elements in dataframes can be referred to just like they can in matrices. But, there is some added functionality in this case. We can still do:

```
dfData[1,1]
dfData[-1,-1]
dfData[-1,-1]
```

However, the columns can be referred to in a special way. Try:

```
dfData$vPeople
dfData[, "vAge"]
```

To see the names of the columns returns as character vector use `names(dfData)`. Even though they appear to be similar, `dfData$vPeople` and `vPeople` are two different objects.

We can subset dataframes based on logical tests. Typically, the logical test we use will be dependent on the data, itself. See `?subset`. Then, consider `subset(dfData, vAge > 25)`. One very important point to note is that the `vAge` term in the logical test refers to the column name in `dfData` and not the object by the same name. Confirm this by replacing the object `vAge` with a vector that equals the old vector plus 100. Rerun the subset command.

Moreover, we can use any arbitrary logical vector to subset any object. Consider:

```
vPeople[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
vPeople[vAge > 25]
vPeople[((1:5) * 3) < 10]
```

10.7 Logical Operators

We just saw some logical vectors used in indexing and subsetting. In general, they will be very useful. The common logical operators are `<`, `>`, `|`, `&`, `is.*()`, `!`, `<=`, `>=` and `xor()`.

Two particularly useful functions are `any()` and `each()`. They test a vector of logicals for whether or not at least one and every element, respectively, evaluates to `TRUE`.

10.8 Simple Recoding

Combining the ways to index particular elements in an object and the set of logical operators, we are equipped to do some simple recoding of values within objects that we have already created.

If we find out that Tyson is not from Texas, but from France (hence, Chatagnier), we will want to change the entry for him in the state variable. We will leave alone the state variable in the dataframe. Instead, consider the original vector we created, `vState`. If we want Tyson's entry to display that he is from `FR` we can use `vState[1] <- "FR"` because we know the first entry corresponds to him. Confirm that this worked by considering `vState`. However, this required we know his entry was in the first row. If we didn't know this, we could use

```
vState[vPeople == "Tyson"] <- "FR"
```

However, we must be careful. This will change the state to `FR` for every Tyson there is. If we have more than one, we'd need to go about this another way.

Alternatively, we could use the `ifelse()` function. Suppose we have to censor the state of origin for all individuals under the age of 26. We could use

```
vState <- ifelse(vAge < 26, NA, vState)
```

To see why, consider `?ifelse`.

11 Summary Functions

Many times, when using `R`, we will want to describe many numbers or observations with some sort of summary value like the mean. We will use different summaries for categorical, ordinal, and interval level data. At the same time, we have functions for both univariate and multivariate summaries.

The functions `mean()`, `median()`, `mode()`, `sd()`, `sum()`, `max()`, and `min` describe the central tendencies and other aspects of the data. How these functions handle missing data (i.e. `NA`) matters greatly. Consider the following:

```
mean(vVarY)
min(vVarY)
max(vVarY)
sd(vVarY)

mean(vEoM)
mean(vEoM, na.rm = TRUE)
```

However, instead of evaluating several functions for each vector we can use `summary(dfData)` and this will give us most of the information we might have wanted. `R` wisely customize the summary for each variable based on the kind of data it is. For factors and logicals, we find out about frequencies. For numerical data, we get a description of the distribution.

Beyond these univariate measures, we can use `cor()` to calculate the correlation coefficient between two numerical vectors or `table()` to count the number of observations occurring within each unique classification between multiple factor vectors. Consider the following:

```
cor(vVarY, vVar1)
table(dfData$vAge, dfData$vState)
```

If we are interested in conditional summaries, we need to do a little more work. There are two natural ways to approach questions like, what is the average value of `vVarY` within people of the same age. We know there are three different groups because of `unique(dfData$vAge)`. What does `unique()` do?

We could subset the dataframe according to age, then run `summary` or `mean`. Or, instead, we could use a very convenient function called `tapply()`. Consider `?tapply` and then the code below.

```
mean(subset(dfData, vAge == 25)$vVarY)
tapply(dfData$vVarY, dfData$vAge, mean)
```

12 Basic Plots

Although the numerical/textual summaries of our data will be helpful, we often look to make figures to communicate some aspect of our date or some result to others. For now, we will simply go through different kinds of plots without concerning ourselves with customization.

If we have univariate data, we could use a boxplot or a histogram. Consider the following:

```
boxplot(rnorm(n = 50))
hist(rnorm(n = 50))
```

If we were so inclined, we could simply plot the values of interest against an index. Try:

```
plot(rnorm(n = 50))
```

If we have bivariate data, we might want a scatterplot with lines connecting observations in sequence. Consider:

```
plot(rnorm(n = 50), rnorm(n = 50), type = "b")
```

For three dimensional data, we can generate a surface plot. Use:

```
persp(z = matrix(data = sort(rnorm(100, sd = 3)), ncol = 10))
```

The options are many and `http://cran.r-project.org/web/views/Graphics.html` is a good link to read. If you know the kind of plot you would like to create, it will most likely either exist already or you can create it. We will cover building plots from the ground up and customization later.

13 Object Bookkeeping

Recall, just about everything in `R` is an object. And, by now, we've created a number of objects. To display all of the objects type `ls()`. What is the class of the output of this function call? If you would like to remove an object named `foo`, try `rm(foo)`. If you would like to remove every object, use `rm(list = ls())`. Some folks use this at the top of each script to make sure their `R` session is fresh. I would recommend this.

Now, let's create an object that will contain all of our work. A list is the best choice here. First, we initialize a list. Second, we will save the dataframe as one element in this list. Third, and last we will leave a note for ourselves as a character string.

```
lOutput <- list()
lOutput$data <- dfData
lOutput$comment <- "today’s work was very rewarding"
lOutput
```

Now, we can reference the elements in the list by their names and the `$` operator.
To save the object `lOutput`, first look at `?save`. Then, run:

```
save(lOutput, file = "Z:/mydata.Rdata"}
```

Part IV
Case Two: Realistic Data Management and Realistic Plotting

This part of the short course is designed to take the basic data management skills acquired in the last part and put them to use on a more realistic problem. Similarly, we will work with this data to construct polished figures describing certain aspects of our data. Lastly, as we proceed through this more realistic example, we cover topics that you will encounter when you start to work on real problems, not just problem sets: installing and loading packages, saving textual output, and saving graphical output.

Because some of the code used here is slightly more involved than an introductory level, don't worry about "getting it" immediately. The details of various functions come with time. You can always refer back to these examples and the `R` help files when you do your own work.

14 Statement of Problem

Let's consider the United States of America for this example. More specifically, we will take state level data for 2005. We are interested in the relationship between the proportion of the population that is incarcerated, debt and other financial indicators of the state, and major professional sports teams within the states. As fascinating of a dataset as this may sound to be, the reality is that the data does not come pre-loaded in `R`. Therefore, we will have to do some work to get the data in `R` and then to merge the different sources.

15 Loading the Data Files

Although finding usable data is one of the harder components of data analysis (believe it or not), the files we need for this analysis are ready and waiting.[2]

Locate `spending.csv`, `prison.xls`, and `teams.csv` on your machine. Ensure that they are in a place that is easily accessible.

When we load pre-existing data into `R`, there are two kinds of files we might work with. If we have data in a plain text format (i.e. we can open it in a text editor) our life is easy (relatively speaking). If we have data in a binary format (i.e. a MS Excel spreadsheet, a Stata dataset) our life might be easy or it might be hard. In our case, though, since we have two plain text files and a MS Excel file (which is standard) we will have no problem getting this data into `R`. Indeed, there is an entire manual on this topic.[3]

[2]These plain text data files should be available however this document was accessed, but, if not, an email to `jpolmted@gmail.com` will remedy that.

[3]`http://cran.r-project.org/doc/manuals/R-data.html`.

To read in plain text formatted data, try `?read.table`. There, we see five related commands. These different functions are based on the same `read.table()` function, but have different default argument values to facilitate reading particular kinds of data. Which one should we choose? It depends what our data look like. Because `spending.csv` and `teams.csv` are plain text files we can open them in a text editor. Go ahead and do this.

We can now see that we are interested in using `read.csv2()` because the fields are separated by a semi-colon (`;`) The first argument to this function is `file` and we need to pass the function a character string describing the location of the file on the machine. There are two ways of doing this. First, you could give `R` the absolute path to the location of the data file.. If you choose this way, every time you tell `R` where a file is, you have to start with a drive (e.g. "`C:/`", "`F:/`") and then you will work through the hierarchy of directories. Second, you tell `R` where your *working directory* is and then, afterwards, you need only specify the location of the file relative to the working directory. Let's use the second option. Each way has its advantages and disadvantages and as you advance, the two will both be easy.

```
getwd()
setwd("Z:/stuff")
```

We can ask `R` where the current working directory was and then we change it to the directory `Z:/stuff`. Be certain that the directory exists. What happens if it doesn't? Also, make sure the data files are in this directory.

Try:

```
read.csv2(file = "teams.csv", header = FALSE, skip = 5)
```

If we use one of the other functions, `R` will get structure the data incorrectly (unless we override all of the defaults). Test it out to verify this.

What is the output of this call to `read.csv2()` (Hint: use `is`)? What kind of object is the returned value from this function? What do the arguments we passed do?

This time, let's use

```
dfTeams <- read.csv2(file = "teams.csv", header = FALSE, skip = 5)
dfSpending <- read.csv2(file = "spending.csv", header = TRUE, skip = 7)
```

The second *comma separated values* file is set up slightly differently so we need to change the arguments.

The last file we need to load in is an MS Excel file. Take a moment to try to find a function to read in MS Excel files. The package `gdata` provides the function `read.xls()`. To access this functionality, we will install the package, load it, and then read the help file.

```
install.packages("gdata")
library("gdata")
?read.xls
```

After looking at the help file, try

```
dfPrison <- read.xls("prison.xls", skip = 3, header = TRUE)
```

As with the first two `read.*` functions, if we do not assign the object to an identifier the data are read into `R` and the dataframe object is printed to the display. By assigning it, we can refer the dataframe by name instead of re-importing it.

16 Cleaning and Merging the Data Files

Take a look at the dataframes we have created. If the output is a little overwhelming, try using the `head()` and `tail()` commands. How many observations are there in each dataframe? How many variables?

```
dfSpending
dfTeams
dfPrison

head(dfSpending)
head(dfTeams)
head(dfPrison)

tail(dfSpending)
tail(dfTeams)
tail(dfPrison)

dim(dfSpending)
dim(dfTeams)
dim(dfPrison)
```

Notice that `dfSpending` has more than 50 rows. Look at the dataframe. Why is this? Notice that `dfTeams` has only 26 rows. The other states not listed have 0 teams. We aren't "missing" the data, per se. It just isn't included. However, for `dfPrisons` we don't have data for two of the states. Each of these issues needs to be treated in turn so that we can produce a unified, homogeneous dataset.

Clean up the spending data with the following code:

```
dfSpending1 <- dfSpending[-c(9, 45),]
```

What are we doing here?

In order to merge these dataframes together we will use the function `merge()`. Open the help file. Let's start with merging the spending data and the sports teams data. Then, we'll clean up the variable names and address the "missing" data for states without any teams.

```
dfData1 <- merge(dfSpending1, dfTeams, by.x = "State", by.y = "V1", all.x = TRUE)
names(dfData1)[names(dfData1) == "V2"] <- "teams"
names(dfData1)[1:6] <- c("state", "sls", "sld", "gsp", "rsg", "pop")
dfData1$teams[is.na(dfData1$teams)] <- 0
```

- we merge two dataframes by specifying which variable in each dataframe should be used as a key
- the merged dataframe is assigned to a new identifier
- we manipulate the column names of the dataframe by working on the vector object that results from `names()`
- we force the value of the variable to be 0 for any observation where we previously had it coded as `NA`.

The last step is important because we know these states have 0 teams. That is different from the missingness implied by an `NA`. Observations with `NA`'s are often dropped, so replacing these with 0 is a good thing for our application.

In order to merge `dfData1` with `dfPrison`, we would need them to have a variable in common and they don't. The closest we have are the state variables. But, they are not identical, so the merge would not result in what we want. First, we code a new state variable in `dfData1` and then we merge the dataframes.

```
dfData1$state_lower <- tolower(as.character(dfData1$state))
dfData2 <- merge(dfData1, dfPrison, by.x = "state_lower", by.y = "state", all = TRUE)
```

This is the second block of code that manipulates character vectors for some purpose. This will become a useful skill.

Finally, we have a dataframe with all of our variables in it. Let's add one more variable based on whether or not the state is in the south.

```
vSouth <- c("arizona", "new mexico", "texas", "oklahoma", "arkansas",
            "louisiana", "mississippi", "alabama", "florida", "georgia",
            "tennesee", "south carolina", "north carolina", "kentucky",
            "virginia", "west virginia"
            )
dfData2$south <- is.element(dfData2$state_lower, vSouth)
```

The function `is.element()` is a helpful function. Type `?set` to see similar functions. This is a clear example of one issue many `R` beginners struggle with: *finding the name of the function that performs fairly straightforward tasks.* As you use `R` more and more, you come across an increasing number of these functions and will recall them in the future. Until then, don't hesitate to ask.

In this code, I am just tidying up some vector types and coercing my way through. You need not understand exactly what is going on here. You will in time, though.

```
dfData2[, 3:7] <- apply(dfData2[, 3:7], 2, function(X) as.numeric(as.character(X)))
```

This has been a good amount of work to create a unified dataframe from disparate sources. Moreover, this data started off in a fairly clean form.[4] In practice, you will spend a lot of time getting your data ready for analysis. Some people prefer to do these steps in other software packages like Stata. This is wholly unnecessary and is not recommended. Everything can be done in R.

17 Numerical Summaries of the Data

We have previously covered the use of simple functions to provide quantitative summaries of our data. So, for this example, instead of repeating those steps, we will focus on some more advanced functions.

In the future, the packages `reshape` and `plyr` should be used early and often. They provide some of the smartest functions related to data management in R.

```
install.packages(c("reshape", "plyr"))
library(reshape)
library(plyr)
```

We recall that `summary` gave us univariate descriptions of each of our variables. For example, there are fifteen states in the south (`south`), the median state population is 4.2 million (`pop`), and the maximum real state growth rate is 9.4% (`rsg`). Instead, suppose we wanted to discuss these same quantities, but we wanted to summarize the data as if it belonged to two different groups, those states that have professional sports teams and those that do not.

We could subset the data twice and then invoke `summary`. However, there are cleaner and more powerful ways to do this. Consider this code:

```
dfData2$teams2 <- ifelse(dfData2$teams > 0, 1, 0)
dfTemp1 <- melt(dfData2, measure.vars = c("rsg", "pop"),
                         id.vars = c("state", "south", "teams2")
                )
cast(dfTemp1, teams2 ~ ., subset = (variable == "rsg"), mean, margins = TRUE)
```

We have calculated the mean real state growth rate for those states with professional sports teams and those without.

Similarly, we can calculate the average population for states falling within the four categories obtained when we cross-classify being in the south with having a professional sports team. The average state population is larger in states in the south than those not in the south. Similarly, the average population in states with sports teams is larger than those without.

Although the dataset we put together contains all of the information to compute these values, their calculation is not immediate with any of the tools we yet know.

Lastly, consider the function `sink()`. Rerun the `melt()` and `cast()` code after trying:

[4] I did some pre-processing of the data to simplify steps in R.

```
sink(file = "foo.txt")
```

All of the R output will be sent to that file instead of the screen. After running the aggregation, run `sink()` to close the connection. Now, open up `foo.txt`. If you add the line `cat("output")` in with your code it will annotate the "sunken" output. Still, you must remember that you will not have the luxury of knowing exactly which command generated what unless you compare it to your source code.

18 Graphical Summaries of the Data

Now, we will look at how to create a polished plot and save it to an external file. There is no way to introduce all of the possible plot types that are available. The way to learn this is to focus on understanding what the plotting parameters mean. Then, at a later date, you can combine that knowledge, your experience, and the R help files together to figure it out. The `plot()` command is very generic and will try to guess the plot that makes the most sense for your data.

Create a scatter plot of *number of professional teams* against *state population.*

```
plot(dfData2$pop, dfData2$teams)
```

Load the `ggplot2` package (or install it first if it is not installed). We want the `alpha()` function from the `ggplot2` package and the below code will not work without it.

The previous plot is rather spartan and not very helpful. Instead, try

```
plot(x = subset(dfData2, south == 0)$pop, y = subset(dfData2, south == 0)$teams,
     main = "Teams by Population",
     xlab = "State Population (Millions of People)",
     ylab = "Professional Sports Teams (Count of Teams for Real Sports)",
     cex.main = 2,
     pch = 3,
     col = 3
     )
points(x = subset(dfData2, south == 1)$pop,
       y = subset(dfData2, south == 1)$teams,
       pch = 1,
       col = 5
       )
abline(v = mean(dfData2$pop),
       col = alpha(1, .3)
       )
abline(h = mean(dfData2$teams),
       col = alpha(1, .3)
       )
text(x = 15, y = 17,
     paste("Correlation", cor(dfData2$pop, dfData2$teams))
     )
legend(x = 20, y = 15,
       legend = c("Non-South", "South", "", "Marginal Avg"),
       pch = c(3, 1, NA, NA),
       col = c(3, 5, NA, alpha(1, .3)),
       lty = c(NA, NA, NA, 1)
       )
```

This figure is produced using what is known as the *base* graphics package.

- `plot()` creates the initial scatter plot based on the first subset of the data
- since `plot()` is deciding on the geometry of the figure, it will reflect only the data that it has
- overriding the automatic calculation of axis limits and other aspects is sometimes the trick to successfully plotting multiple instances of subsetted data on the same figure
- we set the title (`main`) and labels (`xlab`, `ylab`) arguments here
- `points()` allows us to add points to an already existing plot (like the one we just created with `plot()`)
- `abline()` is a general function that allows us to add straight lines
- notice that when we specify the color (`col = ...`) we can use the `alpha()` function to provide us transparency[5]
- `text()` allows us to add arbitrary text
- instead of hard coding the correlation coefficient we used the `paste()` command to turn a string and a numeric object into one longer string
- `legend()` allows us to add a legend where we want (`x, y`)
- the legend is entirely custom, so it reflects only what you tell it and the order in which you tell it things
- we customize the legend by specifying which shape, color, linetype, etc. should show up on each line, `NA` let's us "skip" one of these attributes

Alternatively, we could compare distributions of the incarceration rates among states for states in the south vs. those not in the non-south and for whites vs. blacks.

[5]For more help on choosing colors see `http://research.stowers-institute.org/efg/R/Color/Chart/`

```
boxplot(value  ~ variable + south,
        data = melt(dfData2, id.vars = c("south", "state"),
            measure.vars = c("white", "black")
            ),
        notch = FALSE,
        names = c("White, Non-South", "Black, Non-South",
            "White, South", "Black, South"
            ),
        main = "State Incarceration Rates by Region and Race",
        ylab = "Count Incarcerated per 100,000 Population"
        )
```

We can see several things: one R related, one not. First, some of the specifics change depending on the kind of plot we are making, but many things carry over directly. Second, it is no surprise this country has some race relation problems. We can't even blame the south in this instance! (Don't close the plot.)

In fact, this reality is so unbearable that you should type:

```
rect(0,0,5,5000,
     density = 1,
     angle = 135,
     lwd = 10
     )
rect(0,0,5,5000,
     density = 1,
     angle = 45,
     lwd = 10
     )
title(sub = "lock this result up!")
```

The point of this demonstration is to show you that the order in which you add things to plots matters. The items at the end are layered on top of the previous parts. Whatever you want on the bottom is what you should plot first.

Also, notice that coordinates given to specify locations are relative to the units on the axes of the figure.

Now, there are two ways to save these figures. There is the *smart* way and then the other way. The other way involves using the menu and saving the image as some inferior file format. Do not do this.

Instead, consider the figure created by

```
plot(density(replicate(1000, mean(runif(10))
                      ),
             kernel = "rectangular"
             )
     )
```

You can save this file as a `.pdf` with

```
pdf(file = "draws.pdf")

plot(density(replicate(1000, mean(runif(10))
                       ),
             kernel = "rectangular"
             )
     )

dev.off()
```

This figure is saved as a `.pdf` file to the working directory. And, in fact, everything I add to this plot will continue to go to the file until the command `dev.off()`. Note, this command is necessary to finalize the file. Similar functions exist to save figures as other filetypes: `jpeg()`, `ps()`, `png()`.

In the long run, I recommend using the `ggplot2` package to create figures and not *base*, though *base* is a good place to start. Not surprisingly, `ggplot2` is maintained by the same person who authored the `reshape` and `plyr` packages. Also, you may be interested in the `tikzDevice` package for inclusion of figures natively in LaTeX.

Part V
Writing Programs

19 Control Flow Operations

19.1 Conditionals

We often want to check a conditional statement and then do something in response to that conditional holding, or not holding. Try (writing in RWinEdt):

```
d <- runif(1)
if (d > 0.5) {
  cat("\n\n today is a good day \n\n")
}
```

Here is what is happening:

1. `d` is a random uniform number drawn from (0,1)
2. we tell `R`: if `d` is greater than 0.5, then do the operation in the curly braces.
3. the operation in the curly braces is a `cat` statement (short for 'concatenate and print') which will print the contents of the quotation marks.
4. the `\n` are simply line breaks to impose a bit of space between the prompt and our output. Here that means two breaks either side of the text.

Perhaps we need `R` to do something if the condition isn't met. No problem (note the curly braces!):

```
if(runif(1)>0.5){
  cat("\n\n the beatles are on itunes\n\n")
} else {
  cat("\n the beatles are of questionable skill \n")
}
```

A simpler, 'hard wired' alternative to this is `ifelse()` and we've already encountered this.

Suppose we want to check more than one condition. Then `&` will come in handy:

```
x <- "3"
y <- cos(3)

if (runif(1) > 0.5 & rnorm(1) < 0) {
  print(x); print(y)
} else{
  (plot(density(rnorm(100))))
}

if (!(rnorm(1) > 0) | !(runif(1) < 0.5)) {
  print(x)
  print(y)
} else{
  (plot(density(rnorm(100))))
}
```

Notice the use of ; to have R do a couple of things. Sometimes you'll see the use of &&. This means that the second conditional is only checked if the first one is true. This would make no difference to the example above. See the help file for more information.

Usefully, we can nest `if()` loops. Try the following:

```
if (r<-runif(1) > 0.5){
  cat("r is",r,"\n")
  ##
  if(r < .6){
    print("0.5 to 0.6")
  } else {
    if(.6 < r & r < .7) {
      print("0.6 to 0.7")
    } else {
      print("bigger than 0.7")
    }
  }
}  else {
  print("smaller than 0.5")
}
```

Lots of things to notice here:

1. `(r <- runif(1)) > 0.5` checks the conditional *and* assigns the number to `r` in one go.

2. `{cat("r is",r,"\n")}` reports back the value actual value of `r` that has been assigned (notice the use of the commas) *outside* the quotation marks.

3. `if(r < .6)print("0.5 to 0.6")` this occurs conditional on `r` being greater than .5, but less than .6

4. `else if (.6 < r & r < .7) {print("0.6 to 0.7")}` is checked if `r` is greater than 0.5, and it is not less than 0.6.

5. `else {print("bigger than 0.7")}` prints something if the previous statements are false (but `r`>0.5)

6. `else {(print("smaller than 0.5"))}` is the last line of the program and matches the first conditional (i.e. we are now in the case where `r` is less than 0.5)

19.2 Loops

Loops, like crystal methamphetamine, are easy to use, and easy to abuse. They are the workhorses of much of the `R`that gets written, partly because they are so straightforward to write. This is unfortunate is some ways, because they are often inefficient. Throwing caution to the wind, try the following in:

```
for (i in 1 : 1000) {
  hist(rnorm(100), col = i, main = paste("Picture", i, sep = " "))
}
```

1. this is a `for` loop: the give away is in the first line. We are saying, 'for' `i` between 1 and 1000, do the thing in the curly braces.

2. here that is: draw a histogram of 100 random normal points, color it with color number `i` and then call it `Picture i` where `i`, of course, is just a number.

3. actually there are not 1000 colors in `R`'s palette, so it recycles some.

4. see `?paste` to read the way it is used

Of course, from a programming perspective, having this loop run every time you run the program maybe annoying. One option is simply to wrap it into a function. So:

```
homework<-function(){
  for(i in 1:1000){
    hist(rnorm(100), col = i,
         main = paste("Picture", i, sep = " ")
         )
  }
}
```

Which means that the `for` loop won't run until we call it via `homework()`

Most of the time, we want to loop through a matrix (or data set) take something from that matrix and put it somewhere else.

First off, create a matrix—initially filled with missing values—to take the fruit of our labors:

```
mOutput <- matrix(NA, nrow = 30 , ncol = 1)
```

Now, suppose we have a matrix like this

```
mData <- matrix(runif(900), nrow = 30)
```

That is, 30×30 with 900 random numbers. And we want to go row by row taking the mean of the row and outputting it. This would work:

```
for (i in 1 : nrow(mData)) {
  mOutput[i] <- mean(mData[i, ])
}
```

Notice:

1. we can use pretty much anything for our index: here it is `i`; if we replaces every reference to `i` in this expression with `monkey`, that would work too
2. the end of the index is the number of rows in `mData` (which is 30)
3. now, we are assigning the mean of the i^{th} row of `mData` to the i^{th} row of `mOutput`

So what's the problem? The loop is perfectly accurate, but it is slow, and laborious to code.

We may sometimes have cause to place `for` loops within other `for` loops (but generally try to avoid). Here is an example:

```
mLetters <- matrix(data = NA, nrow = 10, ncol = 5)
for (m in 1:10) {
  for(n in 1:5){
    mLetters[m,n] <- (letters[n + m])
  }
}
```

Look at `letters`. What is it? What does this code do?

Notice that the way we are indexing the loops matters here. If we put `mLetters[n, m]` instead of `mLetters[m, n]`in our code, we'll get the dreaded

```
Error: subscript out of bounds
```

There are alternatives to `for` which are used in different circumstances. Examples are `while` and `repeat` which are often used together.

Other Types of Loops

The syntax for `while` is `while(condition) expression` which means while a particular condition holds, the `expression` will be executed. `repeat(expression)` simply repeats the `expression` operation again and again and again. This type of thing turns up a lot in monte-carlos. Consider the following:

```
n <- 1
mF <- matrix(NA, nrow = 10, ncol = n)

while (n < 36) {
  repeat{
    vF <- rbinom(10, 1, 0.5)
    if (sum(vF) %% 2 == 0) {
      break()
    }
  }
  mF <- matrix(cbind(mF[, 1 : (n-1)], vF), nrow = 10, ncol = n)
  n<-n + 1
}
```

Here, while `n` is less than 36, I need to repeatedly sample from a binomial (with a sample size of 10) until an even number of the sample are ones. So, for example, $[1,0,0,0,0,0,0,1,1,0]$ won't do, but $[1,1,0,0,1,1,0,1,1,0]$ is fine. As soon as I get a sample fulfilling my requirements, it should be stored (in `mF`). Notice that I have to increment the loop with `n + 1` or else the condition `n < 36` will be true forever, and the program will never stop.

Just to make this point, consider

```
repeat(cat("\n lake effect snow today\n"))
```

which won't end until you hit `esc` or `STOP`.

20 Sampling

Quite often we have to sample from a (posterior) distribution we created. In general, we want to sample `n` values, but we want the sample we produce to be 'weighted'—in the sense that it is proportional to the mass—for each value of our discrete support (say, the thousand values of θ we created for our homework.

We could put together the numerical cdf, but it easier to use `sample` directly.

```
vCand <- seq(0, 1, 0.001)
vP <- dchisq(vCand, df = 15)
vS1 <- sample(vCand, 1000, replace = TRUE, prob = vP)

vDraws <- rnorm(length(vCand))
vS2 <- sample(vDraws, length(vDraws), replace = TRUE)
```

Here, I'm assuming that we first took 1000 values between zero and one, and then we worked out that the posterior,$\Pr(\theta|y)$, was χ^2_{15} (which is unlikely, but anyway). Then we sampled:

1. the object of our sampling was our candidate θ values.
2. we wanted a sample of size 1000

3. we `replace`d the candidates each time we sampled
4. we set `prob = vP` which means we are sampling in proportion to the posterior we calculated.
5. What could `vS2` look like if we used `replace = FALSE`

21 Functional Programming

21.1 Basics

Creating a function is very easy and any time you use code over and over again (e.g. typed manually, in a loop, next semester) you can and should wrap that code up in a function. Functions take in arguments (objects) and return a single object.

Consider:

```
SayHello <- function()
  {
    cat("\n \n \n Hello World \n \n \n")
  }
```

Try both `SayHello` and `SayHello()`. One displays the function itself and one displays the output. That is, loosely speaking, the two are the difference between f and $f(x)$ for some fixed value x.

In this function, there are no arguments and we don't create any objects for latter use, but this need not be the case.

```
GetCos <- function (x = 0)
  {
    vC <- cos(x)

    return(vC)
  }
```

`GetCos()` takes a single argument which has a default value of 0. Then, we calculate the cosine of x and assign it to `vC`. Lastly, before the function ends, we tell `R` that the output of the function is `vC`.

Try `GetCos(1)` and `GetCos()`. Have we created any new objects in our workspace, namely `vC`? No. To save the output for latter use we would do

```
vOut <- GetCos()
```

If we have more than one object to return at the end of the function, we can wrap these objects up together in another single object.

Consider

```
GetTan <- function (x)
  {
    vT <- tan(x)

    vTT <- "your face is a tangent!"

    return(list(vT, vTT
                )
           )
  }

GetTan(1)
```

We return the output as a list (because the objects had different modes) and we use something like `lOutput2 <- GetTan(4)` to save it for future use.

If you pass a function an argument that it was not expecting based on the function definition, you will get an error.

If you don't pass a function an argument that it was expecting based on the function definition, you will get an error.

21.2 Scope

There are some very important rules to follow when using functions to keep things clear. First, realize that a function is a temporary workspace (or environment in `R`-speak). The arguments are one-way traffic onto this island. The `return` statement is one-way traffic off the island.

1. any object not created by and contained within your function should be passed to it as an argument
2. any object(s) to be used later should be returned by your function
3. functions (with the exceptions of `save`, `plot`, etc.) should not have side-effects

Consider this code:

```
vThing <- 4

GetSquare <- function()
  {
    vThing ^ 2
  }

GetSquare()

rm(vThing)

GetSquare()
```

We violated our rule of relying on other objects only through the arguments. And, we can see how easily our function becomes worthless if we don't listen to that rule.

Using the `<<-` operator is a quick way to violate our other rules.

Try:

```
BadFunc <- function()
  {
    cat("i woke up at 8am and all i got was this lousy R function")

    mLetters <<- "gone! mwahhahhah"
  }
```

Look at `mLetters` from before. We changed it even though we didn't pass the function any arguments, we didn't return any object as output, and we didn't assign output to an identifier. Had we used `<-` instead of `<<-`, we would only have altered the object `mLetters` within the function environment so nothing would have been overwritten.