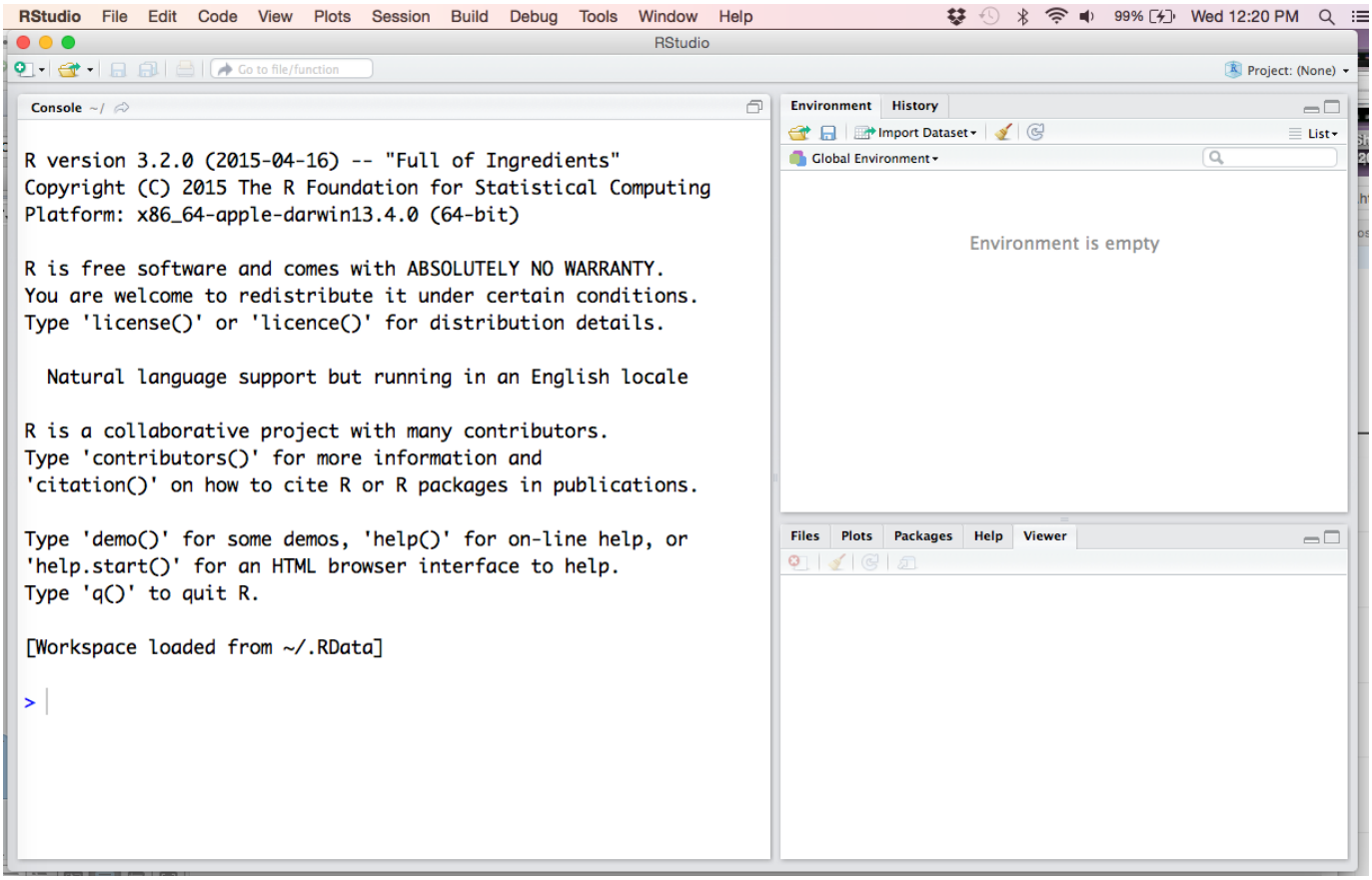
Stat087 -- Introduction to coding in R

We will be coding in the R language. We'll use a kind of 'delivery system' for R, called R Studio. Technically, R Studio is an *Integrated Development Environment* or IDE. Think of R Studio as a nice comfortable driver's seat that you get to sit in and drive, while R is the engine of the car. That is, when you use R Studio, R is the driving force behind your code. You could use R without R Studio (though it would be less 'comfortable'), but you could not operate R Studio without having R on your computer, too.

When you open R Studio for the first time, it looks something like this:



Think of R Studio as a window with different panes. The pane on the top right shows your Environment – basically a list of objects, such as data sets, that you have been working on. The pane on the bottom right serves several purposes. Among other things, it will display the plots you create, and it will show the results of requests you make for Help. (more on Help later). The large pane on the left is called the Console. You can type code into the Console (type after the > prompt), press return or enter, and you'll see the results of your computations just below on the console.

Typing on the Console. Try typing 3 + 5, and pressing return (or enter). You should see this:

```
> 3 + 5
[1] 8
```

The answer is 8, and it's preceded by [1]. For now, just ignore the [1]….

Try some other expressions: `13 - 8,` `4*5,` `2^3,` `2*3 - 1,` `sqrt(16)`

Clearly, R can be used as a calculator. But it is much more useful than that. Try saving some of your computations, by **assigning** them a name: For example, type (using a 'less than' and a dash):

```
>  answer <- 3 + 5
```

This takes the result of 3+5 and assigns it a placeholder – a 'variable' -- called **answer**. You will not automatically see the value on the console, but you may have noticed that **answer** has shown up in the Environment pane on the upper left. If you want to see the value of answer on the console, just type answer, and press return:

```
>  answer
[1]  8
```

We call the arrow (<-) that puts the value of 8 into answer an **assignment operator**. Try creating a few more variables:

```
>  x <- sqrt(16)
>  y <- 3*2^3 - 4
```

You can even use variables to create new variables:

```
>  w <- x * 10
>  w
```

Troubleshooting tip: escaping the + prompt ***(Worth remembering for later – you WILL need it!)***

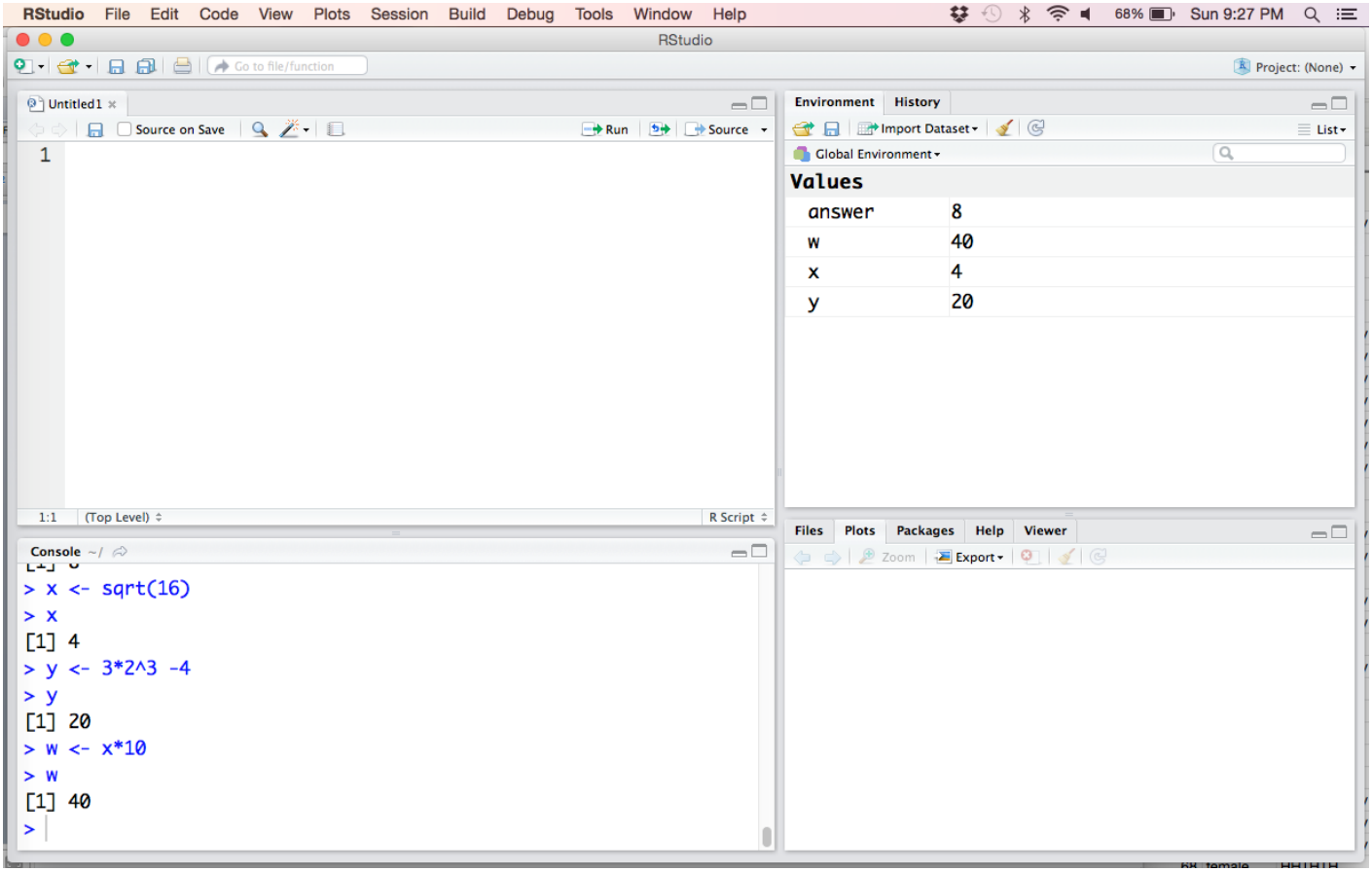
R will prompt you with the > sign, when it is ready for your next line of code. If you type only part of a line (for example, if you typed `x <- sqrt(16`, forgetting the ending parenthesis), R would not give you a > sign. Instead, you'd see a + sign. The + sign means that R is expecting more code to complete the statement. If you just type the closing parenthesis, **)**, it will run as if you typed it all at once. Try it out!

Now, sometimes, you may get the + prompt (instead of the >), and you may not know what is missing from your statement. That is, you may not know what R is waiting for. (hint: often it is a closing parenthesis) If you don't know, but you still want to get on with your coding, just hit the escape key (esc), and you will abort the unfinished statement, and you can start over.

Writing code in an R script

As you're trying out code in R, you might want to save your work for later. A useful way to save it is to write your code in a program or 'script'. To do so, click **File** on the top tool bar, then choose New File, then R Script. A pane should appear on the upper left side of your R Studio window, and the number 1 indicates where to type the first line of your script (see figure below). You can type code into this pane, and click **File** and **Save as** to give it a name and a place to save it.

To run code that you have just written in a script, highlight the code you want to run, then press the Run key on the upper right in the same pane. Try putting code from the next section in a script.



Creating Vectors

It is often useful to store several values of one type in an object. For example, suppose you have the heights of several students, and you would like to store them together. A **vector** can be used; enter it like this:

```
>  Height <- c(68, 62, 70, 74, 65)
>  Height
[1]  68  62  70  74  65
```

The function c() 'connects' (or 'combines' or 'concatenates' …) several values of one type, and puts them in the object Height, a vector. In this case, the 'type' of the Height data is **numeric**. You could also enter **character** data like this (you may use single or double quotation marks):

```
>  Gender <- c('F','F','M','M','F')
```

Another type of data is **logical** data, consisting of TRUE and FALSE values. For example:

```
>  Passing <- c(TRUE, TRUE, TRUE, FALSE, TRUE)
```

Note that there are no quotation marks around the values of TRUE and FALSE. They are not character data.

There are some shortcuts for entering data. For example, to enter consecutive numbers:

```
>  intgr <- 1:5
```

Here's a way to enter repetitive data. (Look at your results -- What is the rule for repeating?)

```
>  repeat1 <- rep(1,5)
>  repeat2 <- rep(1:3, 5)
```

Often, it's useful to generate random data in a vector. This prints ten random numbers between 0 and 1:

```
>  runif(10)
```

This prints the results of ten tosses of a fair coin, where 1 is heads, and 0 is tails:

```
>  rbinom(10, 1, .5)
```

We've talked about three data types: Numeric, Character, and Logical data. We'll talk about Factors and Date/Time, other types, later. We will not use Complex data in this course.

Some basic functions

Most of the programming you will do in R consists of using **functions** that act on **objects**. You can think of a function as a **verb** that acts on a **noun** (the object). For example, consider the sentence "Eat your dinner." "Eat" is the verb, acting on the noun "dinner." When you entered the R "sentence" **`sqrt(16),`** the function **`sqrt()`** was the verb, acting on the object, 16.
Here a few more examples of simple functions:

```
>  round(5.63)

>  log10(1000)
```

Note: The log10 of a number is the power to which you'd need to raise 10, in order to get the number. So log10(1000) is 3, since $10^3 = 1000$. Now try log(1000). (How does **this** function work?)

There are many functions that are useful for acting on **vector** objects. Here are a few to try:

```
>  mean(Height)
>  median(Height)
>  sum(Height)
```

What do the functions sqrt and log10 do when acting on a vector?

```
> sqrt(Height)
> log10(Height)
```

Comments

If you type a line in R, and it has a # in the first column, R will ignore what follows the # in the line. That means you can type whatever you like: your name, the date, descriptions of the code, your train of thought in creating the code, reminders to yourself, etc. Using comments is highly recommended!

Missing values

Suppose you were going to enter weight data for the same 5 students, but you were missing the data for the third student. (In some situations, you could just skip that student, but there may be reasons to keep the information that the third student is 'missing'. For example, you may have data on other variables for this student that you would like to use.)

R stores missing values as **NA** (think of it as 'not available')**.** To enter the weight vector, you'd do this:

```
>  Weight <- c(160, 110, NA, 200, 140)
```

Note that there are no quotation marks around the NA.

Try calculating the mean of the Weights, using **`mean(Weight).`** Note that the answer is NA. This means that R was not able to calculate the mean, because the vector Weight has missing data. If you want R to calculate the mean of the values that are not missing, add the argument **na.rm=TRUE:**

```
> mean(Weight, na.rm=TRUE)
```

The argument na.rm=TRUE works with many R functions. Try the functions **`sum()`** and **`median()`** with this Weight vector. Note that some functions will not require you to use na.rm=TRUE. You will learn which functions require it with experience.

Ugh. Why does R make me use na.rm=TRUE? It would be a lot easier if it just gave the mean of the non-missing values by default. One good reason for it is that it requires you to take a look at your data, and note when missing values are present, and why they are there. For example, suppose patients in a clinical trial have missing data because they dropped out of the study, due to bad side effects. You would want to incorporate that in your data analysis!

Getting Help

If you need help with a function, a package, or a stored data frame, use the help function in R. You can get help on, say, the mean function by typing either one of the following:

```
> help(mean)
> ?mean
```

As an alternative, you may click Help in the bottom right pane, and type the word mean in the blank on the right.

Help for a function will give you a Description, Usage, Arguments, and Examples (these are often very useful).

Accessing Data Frames

While it is often useful to work with one vector of data at a time, we usually work with sets of many vectors of data. These are called data sets, or **data frames**. Here are three ways to get data frames:

1. **Create your own data frame by putting vectors together**… Like this:

```
>  roster  <- data.frame(Height, Weight, Passing)
>  roster
```

2. **Data Frames already stored in R.** There are many…. Use the View function to look at a few:

```
>  View(mtcars)
>  View(women)
>  View(iris)
```

(Note that your current script is now covered up by the printout of the data. Use the tabs at the top of the upper left pane to get back to your script).

Also, try typing **`help(mtcars),`** and you'll see a description of each of the variables in the data frame in the Help pane on the bottom right.

3. **Reading data from a csv file**. A common way to store and send the raw data for data frames is using csv files. These files have the extension .csv (stands for 'comma separated values'). They are an efficient file for storage, since they take less space than, say, an Excel data file (.xls or .xlsx), but you can open them up with Excel. Often, csv files have a first line that contains the names of the vectors. This first line is called a 'header'. Data can also be sent using other types of files, e.g., txt files, but we will use csv files most of the time. Here is one way to do read in a csv file:

```
>  S <- read.csv(file.choose())
```

You will be prompted to browse for the csv file that you want to read in as a data frame. Once you've selected the csv file, it will be read into R as a data frame called S. You can then use `View(S)` to look at it.

Working Directory: Another way to access csv files is to have them stored in a folder on your computer, then "set your working directory" in R to that folder. To set the working directory, give R a path to the folder. Suppose you have your data files in a folder called **Stat087 files**. Your path might look something like this (of course, it will be different for you):

```
> setwd('C:/Users/sweaver/Desktop/Stat087 files/')
```

Henceforth, you can access files by name, like this:

```
>  S <- read.csv('datafile.csv')
```

*Here are a few useful arguments when reading in csv files (we'll use these later in the semester):

`na.strings = '?',` tells R that question marks should be considered NA.
`stringsAsFactors = FALSE,` makes sure that character data gets read in as character data
`header = FALSE,` is used when the first row of your data is NOT a list of the variable names

Working with data frames

If you want to work with a vector in a data frame, say, the mpg (miles per gallon) values in mtcars, you can't just type mpg. For example, try this:

```
> mean(mpg)
```

You should get the error: **Error in mean(mpg) : object 'mpg' not found**
You need to tell R that mpg is in the data frame, mtcars, like this:

```
> mean(mtcars$mpg)
```

That is, to refer to a vector within a data frame, type dataframename$vectorname.

The package, dplyr, gives us another way to access vectors in data frames; we'll use dplyr a little later.

Tidy Data

A dataset is said to be tidy if it satisfies the following conditions:

- observations are in rows
- variables are each in one column
- it is contained in a single dataset.

Generally, we think of tidy data as being as "narrow" as it can be. It should not have rows or columns that are just summaries of other rows or columns.

Look at the three data sets below. For each one: Is it tidy? If not, why is it considered untidy? Could you change it to make it a tidy data set?

Class	Name	Quiz1	Quiz2	Quiz3	Total
Z1	Anna	10	9	8	27
Z1	Tiancheng	7	9	10	26
Z1	Zack	8	10	9	27
Average		8.333333333	9.333333333	9	
Z2	Brian	8	10	7	25
Z2	Cody	7	9	6	22
Z2	Mengyuan	10	9	10	29
Z2	Sasha	9	10	9	28
Average		8.5	9.5	8	

	Pregnant	Not pregnant
Male	0	5
Female	1	4

pregnant	sex	n
no	female	4
no	male	5
yes	female	1
yes	male	0

Base Package graphs

There are many ways to make graphs in R. A quick way to start is to use functions like **`hist()`** and **`plot()`.**

For example, to create a histogram of miles per gallon values for the cars in the data set mtcars, run: **`hist(mtcars$mpg).`**

To create a scatterplot of mpg by weight of car, run: **`plot(x = mtcars$wt, y = mtcars$mpg)`** (Note that when we say mpg BY weight, we mean that weight should be the x variable, and mpg the y variable.) In fact, while it is still recommended, you don't really **need** to specify which variable is x and y – If you put the x variable first, and the y variable second, R will understand, and make the same plot: **`plot(mtcars$weight, mtcars$mpg)`**

When you give the names of the 'arguments' of a function, as in the first **`plot()`** above, these are called **named arguments.** When you just plan to put them in the right places (x first, y second), because you know what order R expects, these are called **positional arguments.** Positional arguments make the code a little shorter; however, named arguments are more descriptive, and result in more readable code.

Later in the semester, we will take a more sophisticated approach to plotting, using the R package **ggplot2**. This package is based on a 'Grammar of Graphics' that presents a unified approach to data visualizations – more later!

Accessing packages

So far, everything we have done in R has used simple functions in the 'base package,' that you have installed on your laptop. There are many, many more packages available on the internet. A 'package' is a collection of functions, data, and compiled code. Users find and install the packages that are useful to the particular programming problems that they want to solve. In this course, two very important packages we will use are **dplyr**, and **ggplot2**.

For example, to install **dplyr**, click **Packages** in the bottom right pane of R Studio. Select Install, and type **dplyr**, and click **Install**. Once you have installed a package, its name should show up in your list of packages in this pane. It is now on your laptop, and you should not have to install it again (unless you want to update it). However, each time you open a new session in R Studio, you will have to load it. Do so by clicking the box next to the package name on your list, OR by running the code **library(dplyr)** .

One way to make life a little easier, is to install a collection of packages, called **tidyverse.** Do this now: Install, and load tidyverse. It will automatically install and load dplyr, ggplot2, and a few other useful packages. In the future, you'll only have to use **library(tidyverse)** to load what you need.

A brief introduction to dplyr

dplyr is a package intended to make it easy to 'wrangle' data. There are five main 'data verbs' (functions) in dplyr (filter, summarise, group_by, mutate, arrange), and many additional ones. Recall the idea that a **`function(argument)`** statement is like a sentence (e.g., "Drink milk." applies the verb or function, Drink, to the argument, milk.) In dplyr, the sentence structure is slightly more complex. Say you want to take the data frame called 'women' (a data frame stored in R already) and calculate the mean of all of the heights. In dplyr, you would type:

```
women    %>%
     summarise(mean(height))
```

The %>% is called a 'chaining' or 'piping' symbol. You can read it as the word 'THEN.' That is, it means that you should take the data set, women, THEN perform the function following the %>% on that data set. You could read the code above as a sentence like this:

"Take the data set called 'women', **then** summarise it by finding the mean of all of the height values."

You can perform more tasks in succession by adding more %>% symbols. Here's a two-function example:

```
women    %>%
     filter(height >= 60)   %>%
     summarise(mean(height))
```

This means "Take the data set women, **then** filter the observations so that you only include women at least 60 inches tall. **Then** take the new data set, and find the mean height of all of these women."

Filter and summarise are just two of the main data verbs in dplyr. There are also many more very useful verbs (e.g., select, sample_n, left_join, rename). Here is a nice, short introduction to dplyr: http://sharpsightlabs.com/blog/2014/12/11/dplyr-intro-data-manipulation-with-r/

A brief introduction to ggplot2

As noted above, ggplot2 is based on a 'Grammar of Graphics' (hence the 'gg' in ggplot2). You can think of a plot as conveying information about variables using **aesthetics**. Examples of aesthetics:

- Position along the x axis
- Position along the y axis
- Size of a point or other element
- Shape of an element
- Color of an element

The first step in ggplot2 is to identify which variables will be represented by which aesthetics. That is, you **map your variables onto the aesthetics**. For example, if you want to produce **`plot(x =mtcars$wt, y=mtcars$mpg)`** from above, you start with:

```
ggplot(data = mtcars, mapping = aes(x = wt, y=mpg))
```

This says to start with the mtcars data frame, and let wt be plotted on the x-axis, and mpg on the y-axis. In fact, the first two aesthetics will be assumed to be x and y, unless you specify otherwise, so you could just type the following (though it's recommended to leave in the argument names):

```
ggplot(mtcars,  aes(wt, mpg))
```

This two lines are intended to 'set the frame' in place, but it will not produce a plot by itself (if you run it as is, you'll get an empty plot!). We haven't told ggplot2 exactly how to represent the values yet (points, lines, smoothed curves, etc.) The name for the representation choice in ggplot2 is the 'geometry,' or '**geom**.' If you want to plot points, you will specify the 'point' geom, like this:

```
ggplot(data = mtcars, mapping = aes(x = wt, y=mpg)) +
     geom_point()
```

Now, try **`geom_smooth()`** instead. Next, try **`geom_line()`** (is it what you expected?) You can even layer geom's over one another in many cases. Try

```
ggplot(data = mtcars, mapping = aes(x = wt, y=mpg)) +
     geom_point() +
     geom_smooth()
```

There are many, many geom's available (see http://docs.ggplot2.org/current/ for some of them.)

The geoms may also have their own aesthetics, or options: e.g., point size or shape, line thickness or type, and these can be specified, too. The **scales** of the axes may be modified; **positions** of geoms may be specified; titles and labels can be changed, as well, by adding more layers. Much more on this later!

```
ggplot(data = mtcars, mapping = aes(x = wt, y=mpg)) +
     geom_point() +
     labs(title = "This is my first ggplot", x = "Weight of vehicle")
```

Note that you can also save graph instructions as an object, then add layers to it. For example, you could save the original graph as S:

```
S <- ggplot(data = mtcars, mapping = aes(x = wt, y=mpg)) +
     geom_point()
```

Then you can add layers like this:

```
S +
     labs(title = "This is my first ggplot", x = "Weight of vehicle")
```

The R 'cheat sheet' for ggplot2 (https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf) uses this method as a shortcut for demonstrating different layers.

Readable Programming

Some people dislike the term 'coding,' because it sounds as though you are creating computer instructions that are cryptic and hard to read. In fact, the scripts that you write should be **as readable as possible – and by a human, as well as a computer.** There are many reasons for this:

1. It will help other people who would like to use your script understand how it works.
2. It will make your research results 'reproducible,' since others could verify your usable code.
3. It will make it easier for the TA and me to read and grade your work. ☺
4. Most importantly, it will make it easier for **you** to remember what you did, when you go back to your script a month, a week, or even a day later. (believe me, we all forget..)

How do you achieve **readable code**? Here are some tips:

1. **Use a lot of comments.** Write comments documenting, describing, explaining your train of thought. Remember, these are for others and for yourself. (Shortcut: You can turn a bunch of lines into comment lines by highlighting them, and typing **shift-command-C** – or **shift-control-C**)

2. **Do use named arguments.** Readable code that's a little longer is preferable to short code.

3. **Use spaces freely.** Don't jam all your code without spaces between things. Some rules of thumb: Place spaces around operators such as (=, +, -, <-, etc.). The same rule applies when using = to state arguments. Always put a space after a comma, and never before (just like in regular English). Place a space before left parentheses, except in a function call (e.g., write **`mean(Height),`** not **`mean (Height).`**

4. **Keep density low.** Strive to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font.

5. **Indent blocks of code, and skip a line between blocks.** (Shortcut: Use **command-I** – or **control-I** to indent blocks of code)