First R Session

First R Commands

Open RStudio on your computer.

Then, enter the following command at the R prompt (i.e. right after > on the console) and then press Enter/Return. You can either type it in manually or copy and paste it from this document. (Upper/lower case does matter, but single vs. double quotes does not. Sometimes quotes don't copy correctly, so if copying and pasting doesn't work, try typing the command yourself.)

```
install.packages('Stat2Data')
```

This command instructs R to install an R package called 'Stat2Data'. Once you install a package on a machine, it is saved on that computer. You only need to run this once.

Now, in order for us to get access to the information in that package during this R session, type

```
library('Stat2Data')
```

This load all of the functions and data sets stored in the package 'Stat2Data'. You can see what is inside the package by using help():

```
help(package = 'Stat2Data')
```

The help file shows that we have a variety of datasets that were compiled for a class on regression. We'll use `HighPeaks`. To load the dataset, you may use

```
data('HighPeaks')
```

and then you can look at it using the command

```
View(HighPeaks)
```

A data viewer will appear in your scripting pane. It should resemble spreadsheet software like Excel. Explore a bit, looking at the variable names and the types of values in each column. Now click the X in that tab and click back to the Global Environment. Type the following,

```
m <- mean(HighPeaks$Elevation)
```

You should see an object called 'm' assigned a value 4405.283 appear in the Environment pane on the right. As you interact with R, you will use a variety of objects. Sometimes you load them as data sets (from packages, files, internet, etc), and sometimes you create them yourself as the byproduct of a computation or some analysis you have performed. Once objects are created, they will be listed up in that environment section.

Now, take a second and realize that you submitted your first R commands to the computer! You installed a package, loaded a package, and calculated the mean of a variable. Congratulations!

R Markdown

In this session, we will be typing a lot of commands. To keep track of the great work you will do, open a new R Markdown file (Go to File → New File → R Markdown, press OK). Now, on the left side, the panel is split into two. On the top, you have your R Markdown (Rmd) file in the "scripting pane" and on the bottom is the console that you have been working in. Think of this as files to save your code and your notes. Change the title, author, and date information at the top to "First R Session", your name, and today's date. This is metadata at the top delineated by dashes. Next, you'll see a section delineated by "'{r}. This is an R chunk for R code. Then you'll see ## RMarkdown and then some text. The hashtag indicates that RMarkdown is a header and the other text is just regular text. Change the header to ## First Commands. Delete the rest of the text and press Insert (at the top of the window) > R to insert an R chunk.

Rather than typing commands directly into the console (as we did originally), you are going to type them in this file and save them (think of it like a Word document, but for R). Type or copy/paste our first command in that R chunk, `library(Stat2Data)`. Then outside the R chunk, after the ending "', write a note to yourself of what that code does.

By saving your R Markdown file (File → Save or Command-S), you will have a record of all the commands you used. This allows you to pick up where you left off if you need to stop in the middle of completing a homework. Think of this file as a complete record of your analysis so that you can come back to it and review it at a later time.

You might think that I have created more work for you in that you have to copy and paste commands to the R script and then from the R Markdown file to the console. **There is a shortcut!** You can run the code that you typed in the R chunk by highlighting the line or lines of code and then pressing the Run icon at the top of the panel or use the keyboard shortcut: Command-Enter on Mac, Ctrl-Enter on Windows. For those of us who want to use the mouse as little as possible, if your curser is on the line of code you want to run, then press Command/Ctrl-Enter to run that line of code in the console. Try it out on our first command.

The Data: HighPeaks

Now back to the data! The HighPeaks data in the package we installed is data on hiking trails in the Adirondack mountains. For each trail, the dataset includes the name of the mountain, its elevation, the difficulty rating of the trail (1-7), the vertical ascent (feet), the length of the trail (miles), and the estimated trip time of the hike (hours).

Rather than using the data viewer, we can see the first few lines of the data set with the function `head`.

```
head(HighPeaks)
```

```
##              Peak Elevation Difficulty Ascent Length Time
## 1     Mt. Marcy      5344          5   3166   14.8 10.0
## 2 Algonquin Peak      5114          5   2936    9.6  9.0
## 3   Mt. Haystack      4960          7   3570   17.8 12.0
## 4   Mt. Skylight      4926          7   4265   17.9 15.0
## 5 Whiteface Mtn.      4867          4   2535   10.4  8.5
## 6       Dix Mtn.      4857          5   2800   13.2 10.0
```

What you should see is the first 6 rows of data. Each row represents a trail's data. The columns represent different variables (characteristics measured or collected). (You'll see NA if any data is missing; this dataset doesn't have any missing values.) This object with rows and columns of data is called a *data frame* in R. If you check the class or type of object it is with:

```
class(HighPeaks)
```

```
## [1] "data.frame"
```

you'll see that R calls this data structure a "data frame". You can see the dimensions of this data frame by typing:

```
dim(HighPeaks)
```

```
## [1] 46  6
```

This command prints out a vector of two values: the first number indicates there are 46 rows and the second number indicates there are 6 columns (we'll get to what the [1] means in a bit), just as it says next to the object in the Environment (upper right pane). You can see the names of these columns (or *variables*) by typing and running the following command in your R script.

```
names(HighPeaks)
```

```
## [1] "Peak"       "Elevation"  "Difficulty" "Ascent"     "Length"
## [6] "Time"
```

You should see that the data frame contains information such as `Peak`, `Elevation`, `Difficulty`, and more.

At this point, you might notice that many of the commands in R look a lot like functions from math class; that is, invoking R commands means supplying a function with some number of input **arguments**, which then does something and may produce some output. The `dim` and `names` commands, for example, each took a single argument, the name of a data frame.

Data Exploration

With any new data set, we should first examine the data a little more closely. We can access a single variable/column of a data frame separately using a command like:

```
HighPeaks$Elevation
```

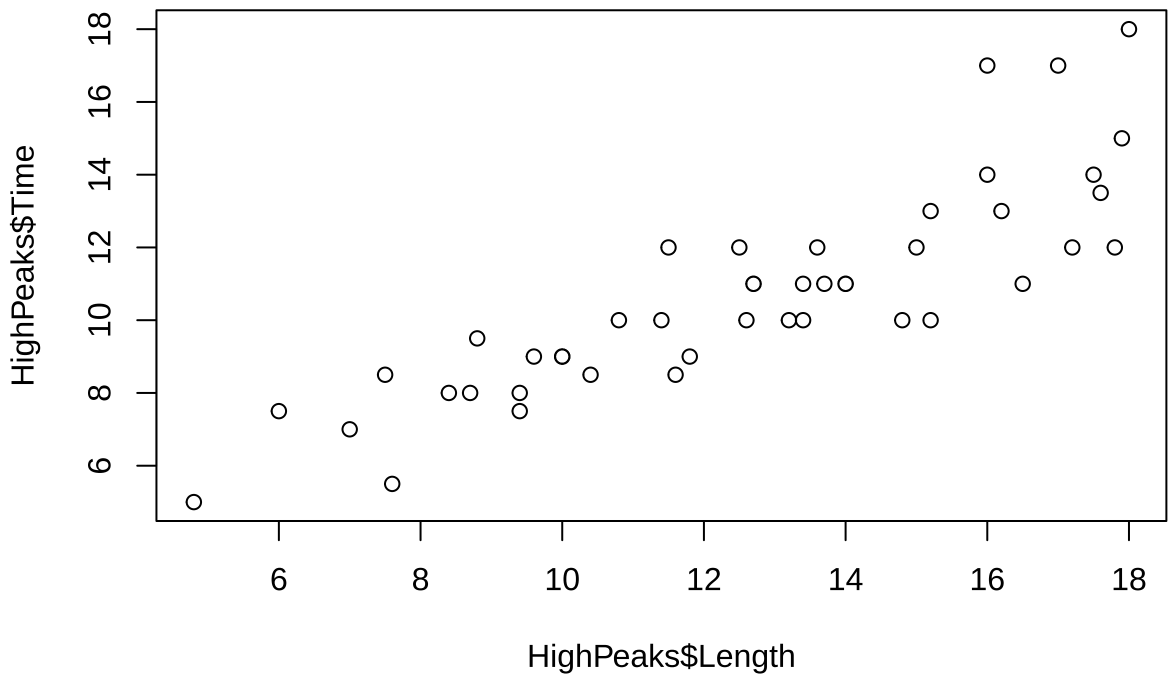
```
##  [1] 5344 5114 4960 4926 4867 4857 4840 4840 4827 4736 4714 4627 4620 4607 4606
## [16] 4580 4515 4442 4427 4420 4405 4400 4400 4361 4360 4340 4240 4240 4185 4175
## [31] 4166 4161 4140 4120 4100 4098 4060 4059 4057 4040 4020 4012 3960 3960 3895
## [46] 3820
```

This command will show the elevation of the trails. The numbers in brackets at the beginning of each line index the values, so for example we know that the 16th hike in the dataset has an elevation of 4580 feet. Note the command includes the name of the data frame (`HighPeaks`) and the name of the column/variable (`Elevation`) with a dollar sign in between (`$`). What command would you use to extract just the Difficulty of hikes?

Notice the way R has printed these values. When we looked at the complete data frame, we saw 46 rows, one on each line of the display. The data for one variable are no longer structured in a table with other variables, so they are displayed one right after another. Objects that print out in this way are called **vectors** in R; they represent a set of numbers or strings which are a sequence of characters like 'hello world'.

R has some powerful functions for making graphics. We will learn which graphics are appropriate for different types of data. For the moment, let's create a simple plot of the Time in hours (on the y-axis) by Length in miles (on the x-axis) with the command

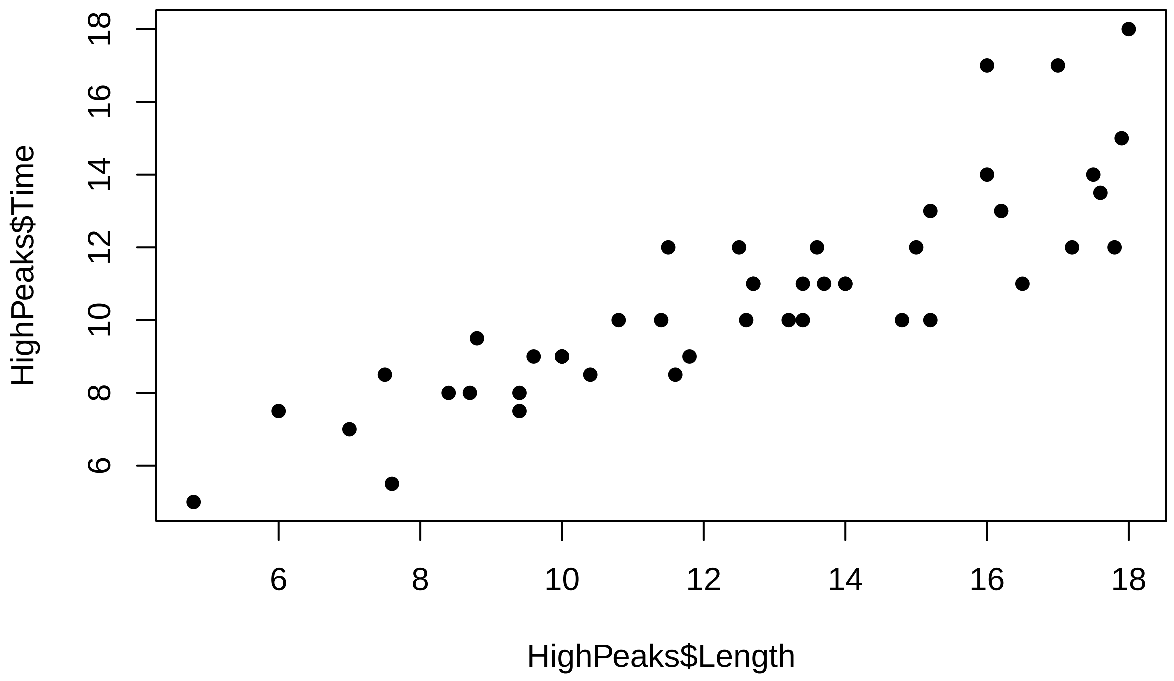
```
plot(x = HighPeaks$Length, y = HighPeaks$Time)
```



By default, R creates a **scatterplot** with each x,y pair indicated by an open circle. The plot itself may appear right below the R chunk (or it will appear under the Plots tab of the lower right panel of RStudio if you ran the code from the console). Notice that the command above again looks like a function, this time with two arguments separated by a comma. The first argument in the plot function specifies the variable for the x-axis and the second for the y-axis.

If we wanted to change the plotting symbol, we could add another argument to the function,

```
plot(x = HighPeaks$Length, y = HighPeaks$Time, pch = 16)
```



You might wonder how you are supposed to know that it was possible to add that third argument. Thankfully, the creators of R have documented all of its functions extensively. To read what a function does and learn the arguments that are available to you use, just type in a question mark followed by the name of the function that you're interested in. Try the following.

```
?plot
```

Notice that the help file replaces the plot in the lower right panel. You can toggle between plots and help files using the tabs at the top of that panel.

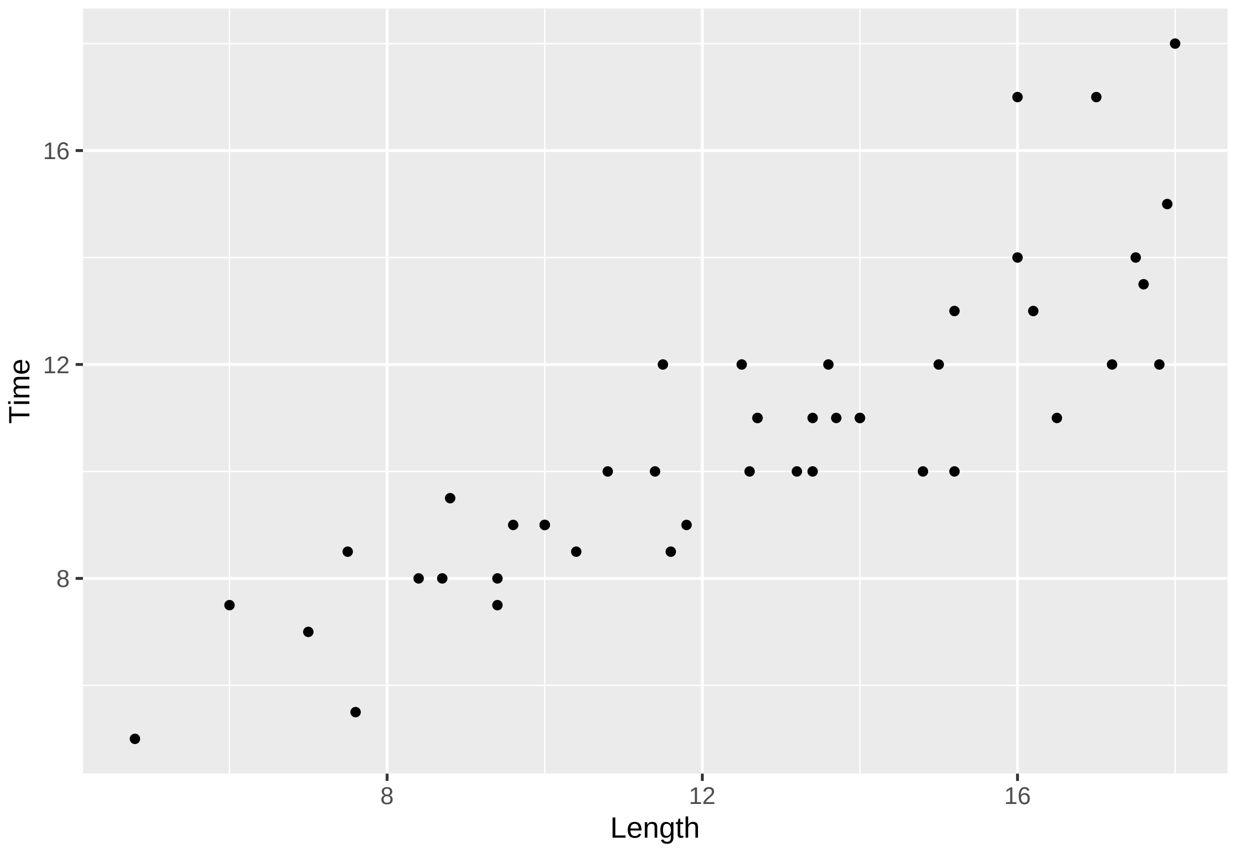
Now, let's say we want to look at this data for each difficulty level (1-7) separately. We'll use a nice graphics package called `ggplot2`, which you can install using

```
install.packages('ggplot2')
```

```
library(ggplot2)
```

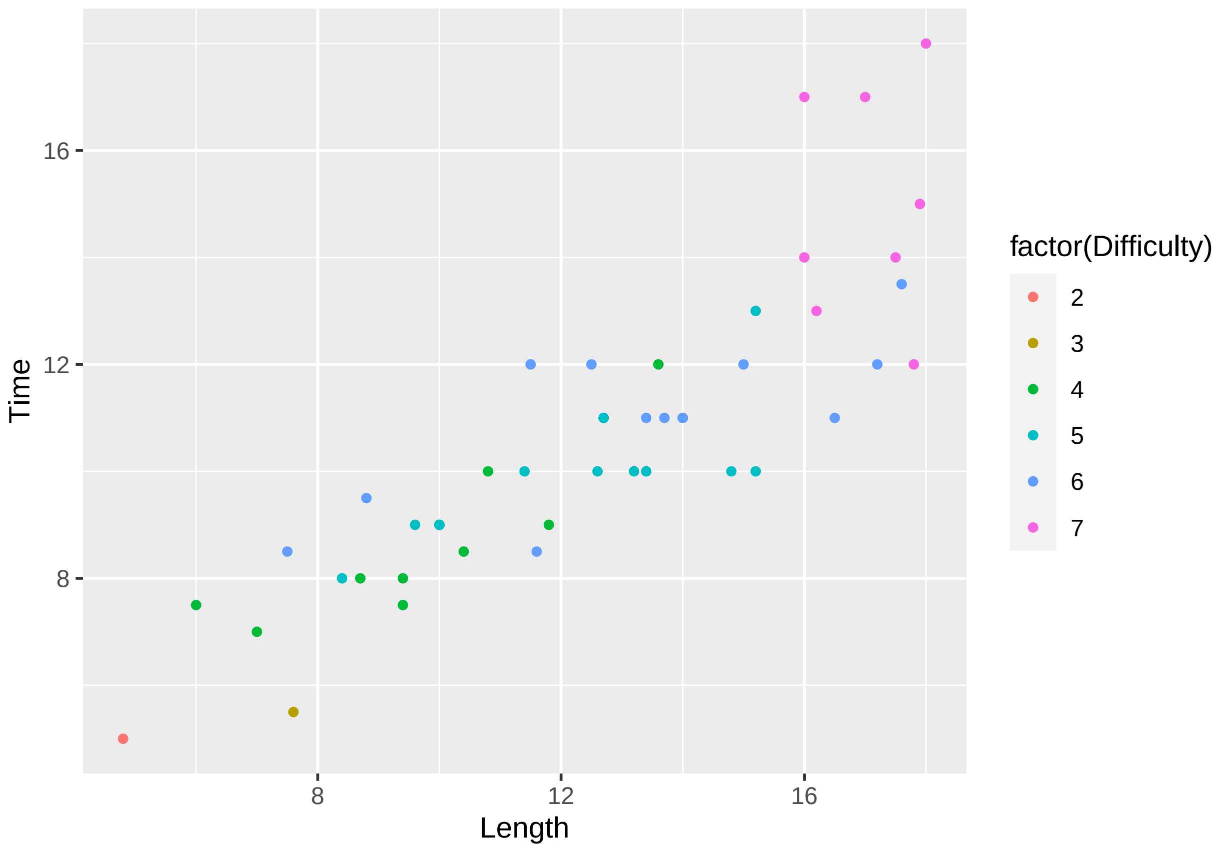
You can make the scatterplot above with ggplot using

```
ggplot(data = HighPeaks, aes(x = Length, y = Time)) +
  geom_point(shape = 16)
```



and then color by difficulty using

```
ggplot(data = HighPeaks, aes(x = Length, y = Time, color = factor(Difficulty))) +
  geom_point(shape = 16)
```



We have the output of the function `aes` as an input to the function `ggplot`. You'll see this nesting of functions fairly often.

Another strange thing is the `+` here. The plus signs in the `ggplot` context allow you to add information to the plot. First you specify the aesthetics (`aes`) by specifying which variable is on the x and y axis, and which variable we want to use for colors. Then you tell it the type of plot you want; here we want a scatter plot (`geom_point`) with filled-in circles for points (`shape=16`). Don't get caught up with this code – we'll see more specific functions for longitudinal data during the module. This is just a warm-up.

Now, suppose we want to calculate the Length in kilometers (rather than miles). To compute this, we could use the fact that R is really just a big calculator. We can type in mathematical expressions like

```
14.8*1.609
```

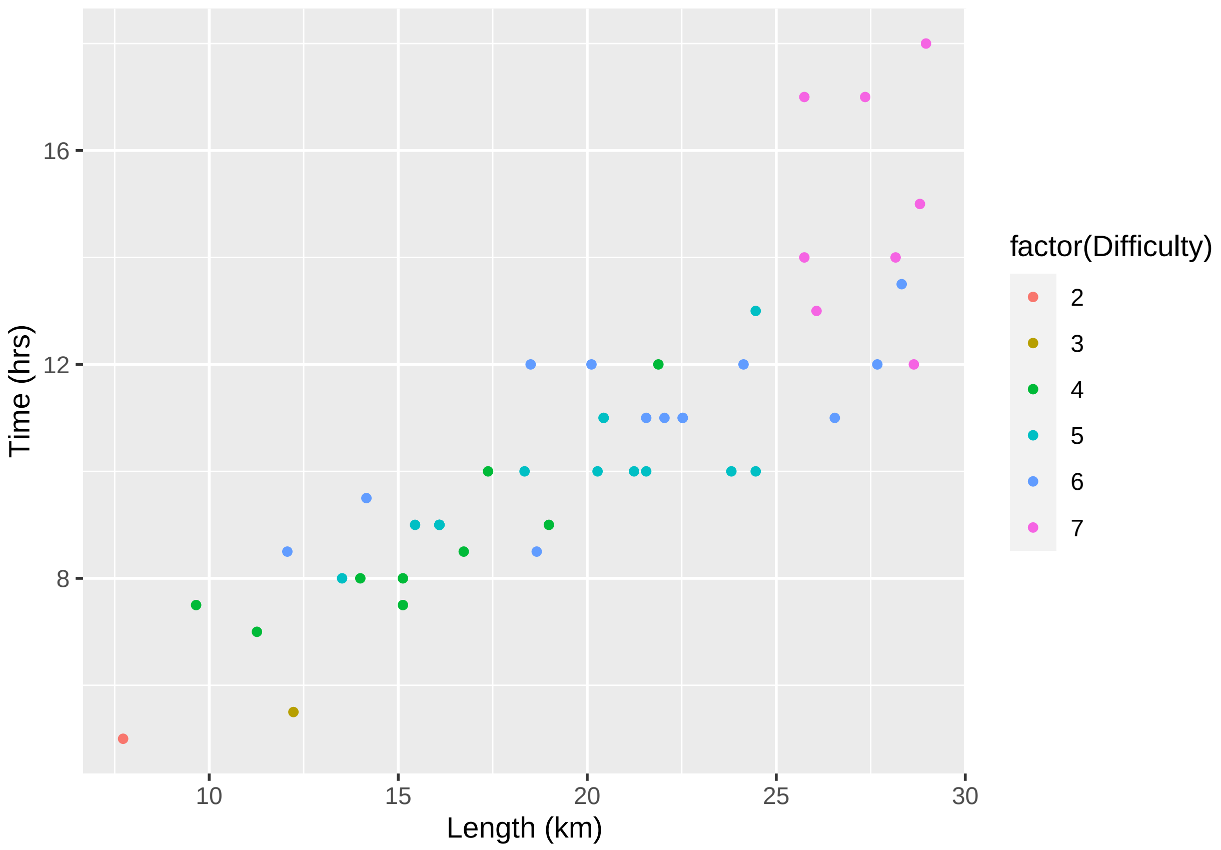
```
## [1] 23.8132
```

to convert the Length of Mt. Marcy from 14.8 miles to kilometers. We could repeat this for every hike, but there is a faster way. R is full of shortcuts! If we multiply the vector of lengths by 1.609, R will carry out that multiplication on every Length value simultaneously. We can also create a new variable in HighPeaks by assigning (`<-`) the variable name LengthKM to the new vector,

```
HighPeaks$LengthKM <- HighPeaks$Length * 1.609
```

Let's recreate the plot with the new Length variable and add a label for the x- and y-axis.

```
ggplot(data = HighPeaks, aes(x = LengthKM, y = Time, color = factor(Difficulty))) +
  geom_point(shape = 16) +
  labs( y = 'Time (hrs)', x = 'Length (km)')
```



This time, note that we left out the names of the first two arguments for aes (we left off `x =` and `y =`). We can do this because the help file shows that the first argument of aes should be the x-axis variable and the second argument should be the y-axis variable (order matters). We have to include the name of other arguments if they are not in the order specified.

Similarly to how we converted the length to kilometers, we could combine two variables. We could calculate feet of elevation gain per mile by taking Elevation and dividing it by Length.

```
head(HighPeaks$Elevation/HighPeaks$Length)
```

```
## [1] 361.0811 532.7083 278.6517 275.1955 467.9808 367.9545
```

Note that with R, as with your calculator, you need to be aware of the order of operations (they are the same as you learn in math, and you can use parentheses to enforce a certain order just like you do in math).

Tip: If you use the up and down arrow keys when your cursor is at the console, you can scroll through your previous commands, your command history.

When you are done with a R session, to exit RStudio you can click the X in the corner of the window. You will be asked if you to save your workspace. If you click 'save', RStudio will save the history of your commands and all the objects in your workspace so that the next time you launch RStudio, you will see and you will have access to the commands you typed in your previous session. This can be handy (to pick up where you left off), but it can also be dangerous (if you don't realize your R Markdown or R script file depends on something in your saved environment). Use with caution, and make sure your work is saved in an R Markdown file or R script for reproducibility.

Helpful Notes

In general, you can refer to the help files (use the ?) or the R reference card (https://cran.r-project.org/doc/contrib/Short-refcard.pdf) for a list of helpful commands.

Feel free to browse around the websites for R http://www.r-project.org and RStudio http://rstudio.org if you're interested in learning more on your own. The `swirl` R package is another way to learn R from within R; you can `install.packages("swirl")`, and then after loading it (`library(swirl)`), run `swirl()` at the console to start your interactive session. (The escape key will end the interactive session.)