Stat 422/722: Recitation 1

Justin Khim

September 5th, 2017

1 Installation

For this course, you will need to install R and the package `dplyr`. We also suggest using the IDE (integrated developing environment) RStudio. I'll try to use RStudio to make all of our lives simpler, particularly with regard to scripts and the working directory.

1.1 Installing R

You can download R at the following website: `https://cran.r-project.org/` Follow the instructions for your computer to install R. This should give you a program for R on your computer, which you could use for this course. It also installs the language R, which can subsequently be used by RStudio.

1.2 Installing RStudio

You can download the (free version) of RStudio at `https://www.rstudio.com/products/rstudio/download/#download` Again, follow the instructions.

1.3 Installing Packages in R

Now, open RStudio. You should have a number of available panes of information. Go to the one called the "Console". This is where we run our R code. To install `dplyr`, we use the function `install.packages`. The whole command is

```
install.packages('dplyr')
```

Alternatively, one could install the whole `tidyverse`, which is a collection of packages containing `dplyr`. RStudio may ask you for a mirror from which to download `dplyr`. I suggest picking one that is geographically close to your current location.

Note that for our current session, we haven't loaded `dplyr` into the console. All we have done is install it on our computer. To load it, use the command

```
library(dplyr)
```

Then, `dplyr` commands are available for our current session. If you close RStudio, then you should use the command again upon reopening.

We may use additional packages in the future. If we do, just follow these instructions and replace any instance of 'dplyr' with the package in question.

2 Basics of R

Now that we've got R installed, we'll want to play around with it a bit. First, we'll think about some basic "data" types. Here, "data" refers to the computer-science version of data, which you can think of as variables. Next, we'll talk about how to make our results easily reproducible via scripts.

2.1 R as a Calculator

The data we're interested in analyzing probably contains numbers in some form. Let's see some basic number things then:

```
2 + 2

## [1] 4

2 / 2

## [1] 1

3.5 * 4

## [1] 14

3^2

## [1] 9
```

Note that for each of these, the text after ## signififes the output for writing the preceding line in the console and then hitting enter.

You may notice a [1] right before the answer, and this stands for the index. In general, R thinks of numbers as vectors of numbers, and it just so happens that the length of these vectors is 1. Hence, the [1] helpfully tells us that our answer is the first index.

We can define vectors ourselves.

```
c(2, 2) + c(3, 4)

## [1] 5 6
```

```
c(1, 5) - c(2, 6)

## [1] -1 -1

1 : 5

## [1] 1 2 3 4 5

rep(1, 5)

## [1] 1 1 1 1 1

rep(c(0, 1), 4)

## [1] 0 1 0 1 0 1 0 1

1:50

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50
```

If we want to concatenate numbers into a vector, we use `c`. If we want an interval, we can use the colon. If we want to repeat a vector a certain number of times, then we can use the `rep` function. Finally, observe that when the output is too long to fit on one line, R helpfully puts the index of each line in brackets, as with [1].

This is nice, but to begin to get to the level of functionality of a TI-83 or TI-84 Plus calculator, we also need to be able to store variables. We can do that with either the equals sign or with an arrow.

```
x = c(1, 2)
x

## [1] 1 2

y <- 1:3
y

## [1] 1 2 3
```

Now, observe that we can do some arithmetic or concatenate vectors themselves.

```
x + c(4, 5)

## [1] 5 7
```

```
c(x, y)

## [1] 1 2 1 2 3
```

One last interesting thing that we'll note is that scalars (and more generally, smaller vectors) get reused in order to make lengths match (if possible). For example, here we add 1 to `x`. Mathematically, this doesn't really make sense because `x` is a vector with 2 entries.

```
x + 1

## [1] 2 3
```

Here, R just adds 1 to each element of the vector. This can be nice for, say, standardizing a vector. However, it's one of the many ways in which R will try to do something that doesn't exactly make sense, which sometimes gives you weird and wrong answers. If that was unclear, the basic idea is that R may not give you an error, but that doesn't mean the code is doing what you think it should do.

Well, that was enough fun with numbers for the day. How about we think about non-numeric "data"? The next obvious thing is words or, to be more precise, strings. We see that we can define strings using quotes.

```
"Hello, world!"

## [1] "Hello, world!"
```

Now, let's put some strings together.

```
x = "Hi, my name is "
name = "Justin"
z = "."
paste(x, name, z, sep = "")

## [1] "Hi, my name is Justin."

paste(x, name, z, sep = "     ")

## [1] "Hi, my name is      Justin     ."
```

Note that the argument `sep` just controls what goes between strings. Easy enough, right? If you're wondering why strings are important, think about them being used to label data. For instance, if we have a table of medical records, we might label the rows with patient names and the columns with things like height, weight, marital status, etc.

Finally, we should at least mention Boolean data, ie `TRUE` or `FALSE` data.

```
x = 5
x > 1

## [1] TRUE

x == 0

## [1] FALSE

x != 3

## [1] TRUE
```

In the first line, we're assigning `x`, and in the subsequent lines, we're doing comparisons. We also have an operator for AND and an operator for OR.

```
4 < x  & x < 7

## [1] TRUE

x == 5 | x == -2

## [1] TRUE
```

We'll go over these again as necessary, but you should know these exist. Note that vectors can only contain one type of "data".

2.2 Scripts

Now, we need to talk about saving our code so we can run it over again to reproduce our results. Alternatively, we may need to make minor tweaks, and we wouldn't want to have to type everything into the console again. This is where scripts come in. You can open one by going to "File", "New File", and then "`R` Script" on a Mac. There should be something similar on Windows computers that essentially gives you a new pane with a blank file. Here, you can type in lines of code, except if you hit enter, you just go to a new line, like in Word or Notepad. Suppose we write the following lines in the script.

```
x = c(1, 2)
y = 1000 * x
x + y
```

How do we get this to the console? One way is by copying and pasting, but this is tedious. If you hit command and enter on a Mac or control and R (for "run") on a PC, then whatever line your cursor is on is sent to the console. Finally, if you hit the "Source" button on the upper-right side of the script pane, then the entire file runs, line by line. So, things have to be in the correct order to get the output you want.

You should always use scripts when working in R. The top line or two of your script is also a good place to load in `dplyr`.

There's one last thing to note, particularly if you are looking at code that you did not write. A # on a line of code, that signals the start of a comment. A comment isn't evaluated by the console; so they're often used to write something to someone who is actually reading the code. Here's an example.

```
x = 5 # Set x
x     # Evaluate x.
```

It's good coding practice (or "coding style") for you to write comments. This is helpful for your instructors if you need help. Additionally, if you wait six months without using a file, you'll probably forget what the file was meant to do, the variable names, etc. Thus, you should comment your code for your future self.

In general, good coding style is supposed to make everyone's life easier. It's especially important if you're ever going to code as a part of a team. If you want to look at a style guide, then I suggest the Google R style guide, which can be found at `https://google.github.io/styleguide/Rguide.xml`

3 Data

Now, we come to the most important part of using R: data! We need to talk about how data is stored in R. So, we'll start with small examples where we input the data ourselves, just to get used to data frames. Then, we'll talk about how to load data from files. For the latter task, we'll use the King County house data file from class.

3.1 Data Frames

A data frame is R's version of a table or spreadsheet. Rows contain instances, and columns contain attributes. Here's a basic example. Let's consider a data frame consisting of three people: Alice, Bob, and Eve. The data frame will contain names, gender, height, and weight. First, we create vectors for each attribute.

```
nms = c("Alice", "Bob", "Eve")
gender = c("F", "M", "F")
height = c(65, 70, 64)
weight = c(120, 170, 115)
```

Note that we cannot create vectors for each person because the attributes have different types. Now, let's create our data frame.

```
df = data.frame(nms, gender, height, weight)
df
```

```
##     nms gender height weight
## 1 Alice      F     65    120
## 2   Bob      M     70    170
## 3   Eve      F     64    115
```

We can see that we are given a table of the data as output. This is what we would expect. Note that the column names are the given vector names.

3.2 The Working Directory

Now, we need to talk about the working directory. If I want you to load a file of data into R, how would you go about doing it? It is not an Excel spreadsheet that you can simply open from Finder or Explorer. You have to load it in the console in R yourself. Let's consider the King County file.

```
kc.df = read.csv("kc_house_data.csv")
```

If you try this, you'll probably get an error saying that this directory contains no such file. What this means is that console can't find the file. The reason it can't find the file is that, much like using a Finder or Explorer window, it is only looking in one folder at a time. This is called the working directory.

How can you change the working directory? There's a command to do it from the console, `setwd`. It requires typing the path, slashes and all, to the folder that you want to go. Alternatively, we can have RStudio help us out. Note that there is a pane, usually in the bottom right, that has lots of tabs. The tabs are "Files", "Plots", "Packages", "Help", and "Viewer", at least for me. Click on "Files". This shows you the files in your working directory. Now, on the right side there is an elipses button "...". Click this, and it will allow you to change open a new folder instead. Change this to the folder that you would like to use, presumably the one containing the data. Then, there's a button to the left with a blue gear and "More" written. Click this, to give the option to set the current folder as the working directory, and hit this as well. There, you've (hopefully) changed your working directory! You generally know it worked because it generates and runs the necessary `setwd` command.

3.3 Reading Data

At this point, we can load the data. We'll use the `head` function to look at the head of the data frame, which is just the first few rows.

```
kc.df = read.csv("kc_house_data.csv")
head(kc.df)

##           id year month day   price bedrooms bathrooms sqft_living
## 1 7129300520 2014    10  13  221900        3      1.00        1180
## 2 6414100192 2014    12   9  538000        3      2.25        2570
```

```
## 3 5631500400 2015     2  25  180000        2      1.00         770
## 4 2487200875 2014    12   9  604000        4      3.00        1960
## 5 1954400510 2015     2  18  510000        3      2.00        1680
## 6 7237550310 2014     5  12 1230000        4      4.50        5420
##   sqft_lot floors waterfront view condition grade sqft_above sqft_basement
## 1     5650      1          0    0         3     7       1180             0
## 2     7242      2          0    0         3     7       2170           400
## 3    10000      1          0    0         3     6        770             0
## 4     5000      1          0    0         5     7       1050           910
## 5     8080      1          0    0         3     8       1680             0
## 6   101930      1          0    0         3    11       3890          1530
##   yr_built yr_renovated zipcode     lat     long
## 1     1955            0   98178 47.5112 -122.257
## 2     1951         1991   98125 47.7210 -122.319
## 3     1933            0   98028 47.7379 -122.233
## 4     1965            0   98136 47.5208 -122.393
## 5     1987            0   98074 47.6168 -122.045
## 6     2001            0   98053 47.6561 -122.005
```

Since there are a lot of columns and the columns have long entries or names, there's still a lot of output. Anyways, this gives us a sense of what the data looks like, at least as far as manipulating it in R. Speaking of data manipulation, that's the next section. Hopefully you loaded the data, otherwise the next section will be hard to follow.

4 Data Manipulation

4.1 Basic Commands

Let's start by looking at our old example data frame of Alice, Bob, and Eve. What are some basic things we might want to do? For one, we might want to get columns, elements, or rows back from the table. From the data table, one can access the vector of the column in three ways. Here, we'll illustrate this for the names.

```
df[, 1]
```

```
## [1] Alice Bob   Eve
## Levels: Alice Bob Eve
```

```
df[, "nms"]
```

```
## [1] Alice Bob   Eve
## Levels: Alice Bob Eve
```

```
df$nms
```

```
## [1] Alice Bob   Eve
## Levels: Alice Bob Eve
```

The first way is to use a bracket and row-column indexing. Leaving the row index blank gives all the rows, and then we're interested in the first column. The second way is to refer to the column directly by its name. This is useful in large data frames where you may not know the exact index or don't want to count it. The final way is just "syntactic sugar". By using the dollar sign and then the column name, we can more easily access the column.

If we wanted just the first name in the names column, we can just use our row-column index:

```
df[1, 1]

## [1] Alice
## Levels: Alice Bob Eve

df[1, "nms"]

## [1] Alice
## Levels: Alice Bob Eve
```

This provides two ways of obtaining the first name.

Now, what if I want to do fancier manipulations? Suppose I only want the names of the females. Then, I could write code like this:

```
df[which(df[, "gender"] == "F"), "nms"]

## [1] Alice Eve
## Levels: Alice Bob Eve
```

We've essentially covered all the code necessary to understand this with the exception of the `which` function. The problem is that this is somewhat complicated, and somehow we had to call the data frame `df` twice even though we only care about really looking at it once. Making this manipulation easier is why we're using `dplyr`.

Getting rows is essentially the same as getting columns.

```
df[1, ]

##     nms gender height weight
## 1 Alice      F     65    120
```

However, note that the output is a data frame, not a vector. This can be useful for getting a subset of the data to work with.

4.2 Using `dplyr`

Now, we need to learn the basic `dplyr` commands. I've collected all of them in the next subsection if you just need a reference. But, simply continue reading this for a walkthrough.

For starters, we need to load `dplyr`.

```
library(dplyr)

##
## Attaching package:  'dplyr'
## The following objects are masked from 'package:stats':
##
##    filter, lag
## The following objects are masked from 'package:base':
##
##    intersect, setdiff, setequal, union
```

The first command we'll use is `select`. We'll first use this to select a few columns of our data to look at, since all the columns just clutter the console. Let's select the id, year, month, day, price, bedroom, and bathrooms.

```
kc.small = select(kc.df,
                  id,
                  year,
                  month,
                  day,
                  price,
                  bedrooms,
                  bathrooms)
head(kc.small)

##           id year month day   price bedrooms bathrooms
## 1 7129300520 2014    10  13  221900        3      1.00
## 2 6414100192 2014    12   9  538000        3      2.25
## 3 5631500400 2015     2  25  180000        2      1.00
## 4 2487200875 2014    12   9  604000        4      3.00
## 5 1954400510 2015     2  18  510000        3      2.00
## 6 7237550310 2014     5  12 1230000        4      4.50
```

One new thing to note is that I can break the command input over multiple lines to make it easier to read. Generally, all of the arguments to the function should line up. The next thing to note is that we did indeed get rid of many of the columns, and now we have a new data frame to work with.

There is an alternative syntax we could have used. We could have written

```
kc.small.2 = kc.df %>% select(id,
                              year,
                              month,
                              day,
                              price,
                              bedrooms,
```

```
                              bathrooms)
head(kc.small.2)

##           id year month day   price bedrooms bathrooms
## 1 7129300520 2014    10  13  221900        3      1.00
## 2 6414100192 2014    12   9  538000        3      2.25
## 3 5631500400 2015     2  25  180000        2      1.00
## 4 2487200875 2014    12   9  604000        4      3.00
## 5 1954400510 2015     2  18  510000        3      2.00
## 6 7237550310 2014     5  12 1230000        4      4.50
```

Here, `kc.small` and `kc.small.2` have the same data, but we used the select command differently to get `kc.small.2`. Also, we had to use the command `%>%`, which is called the pipe operator. It feeds the output of one function into another, or rather, it pipes it along. This seems to be the standard for `dplyr`, so I'll suggest using it. It becomes useful when you want to perform multiple operations on data, and it becomes easier to read than using functions in the usual way.

Now, let's suppose that we're interested in sales in which there were at least two bedrooms and two bathrooms. So, we can define a new data frame using the `filter` function.

```
kc.filtered = kc.small %>% filter(bedrooms >= 2, bathrooms >= 2)
head(kc.filtered)

##           id year month day   price bedrooms bathrooms
## 1 6414100192 2014    12   9  538000        3      2.25
## 2 2487200875 2014    12   9  604000        4      3.00
## 3 1954400510 2015     2  18  510000        3      2.00
## 4 7237550310 2014     5  12 1230000        4      4.50
## 5 1321400060 2014     6  27  257500        3      2.25
## 6 3793500160 2015     3  12  323000        3      2.50
```

And we see that we did indeed filter the data. Note that the commands in `filter` must correspond to the column names as they are given in the data frame. Otherwise, R won't know what to do with them.

Next, suppose we want to rearrange the rows of the data frame by increasing price. We can use the `arrange` function for this.

```
kc.arranged = kc.filtered %>% arrange(price)
head(kc.arranged)

##           id year month day  price bedrooms bathrooms
## 1 7129304540 2014    12  20 133000        5       2.0
## 2 1823049182 2014     9  15 147400        3       2.0
## 3 2976800749 2014    10  31 150000        4       2.0
## 4  818500490 2014    10   9 153503        2       2.5
```

```
## 5 3356403304 2014    10  16 154000        3       3.0
## 6 7697870600 2014     9   9 158000        3       2.5
```

Thus, by specifying the column `price`, we arranged the data frame in increasing order of price. We could have also done this in decreasing order of price using the descend command, `desc`.

```
kc.descending = kc.filtered %>% arrange(desc(price))
head(kc.descending)

##           id year month day   price bedrooms bathrooms
## 1 6762700020 2014    10  13 7700000        6      8.00
## 2 9808700762 2014     6  11 7060000        5      4.50
## 3 9208900037 2014     9  19 6890000        6      7.75
## 4 2470100110 2014     8   4 5570000        5      5.75
## 5 8907500070 2015     4  13 5350000        5      5.00
## 6 7558700030 2015     4  13 5300000        6      6.00
```

Now, houses with different numbers of bedrooms really are different in the number of people they can comfortably house. So, let's group the data by number of bedrooms using the `group_by` command.

```
kc.grouped = kc.arranged %>% group_by(bedrooms)
head(kc.grouped)

## # A tibble: 6 x 7
## # Groups:   bedrooms [4]
##           id  year month   day  price bedrooms bathrooms
##        <dbl> <int> <int> <int>  <dbl>    <int>     <dbl>
## 1 7129304540  2014    12    20 133000        5       2.0
## 2 1823049182  2014     9    15 147400        3       2.0
## 3 2976800749  2014    10    31 150000        4       2.0
## 4  818500490  2014    10     9 153503        2       2.5
## 5 3356403304  2014    10    16 154000        3       3.0
## 6 7697870600  2014     9     9 158000        3       2.5
```

This gives us something that looks a little bit different. Nothing appears to be grouped, but this is finally where we can do some descriptive statistics. Using the `summarize` command, we can compute summary statistics for each group.

```
analysis = kc.grouped %>% summarize(avg_price = mean(price),
                                    sd_price  = sd(price),
                                    avg_bathrooms = mean(bathrooms))
head(analysis)
```

```
## # A tibble: 6 x 4
##   bedrooms avg_price sd_price avg_bathrooms
##      <int>     <dbl>    <dbl>         <dbl>
## 1        2  508522.1 251898.9      2.292321
## 2        3  531518.8 300043.1      2.445915
## 3        4  677504.9 408914.9      2.652078
## 4        5  836451.3 622784.5      2.978924
## 5        6  862650.6 833533.0      3.243852
## 6        7  999400.5 751675.4      3.850000
```

Thus, at a glance, we can see on average what the price of a house is by the number of bedrooms. Additionally, we can tell about how many bathrooms a house has given the number of bedrooms. Neat.

Earlier, I said pipe notation can be useful. Well, we could have done everything in one step.

```
analysis.2 = kc.df %>%
                     select(id,
                            year,
                            month,
                            day,
                            price,
                            bedrooms,
                            bathrooms) %>%
                      filter(bedrooms >= 2, bathrooms >= 2) %>%
                      arrange(price) %>%
                      group_by(bedrooms) %>%
                      summarize(avg_price = mean(price),
                                sd_price  = sd(price),
                                avg_bathrooms = mean(bathrooms))
head(analysis.2)

## # A tibble: 6 x 4
##   bedrooms avg_price sd_price avg_bathrooms
##      <int>     <dbl>    <dbl>         <dbl>
## 1        2  508522.1 251898.9      2.292321
## 2        3  531518.8 300043.1      2.445915
## 3        4  677504.9 408914.9      2.652078
## 4        5  836451.3 622784.5      2.978924
## 5        6  862650.6 833533.0      3.243852
## 6        7  999400.5 751675.4      3.850000
```

This gives us the same output we got earlier. Additionally, this is much easier than nested functions.

4.3 Important `dplyr` Commands

As a summary, here are the functions that we used.

```
select()
filter()
arrange()
mutate()
group_by()
summarize()
```

If you ever need help with a function, using Google or using the help function of R is a good idea. To get the documentation for a function, use a question mark, eg `?select`.

5 Basic Visualization

We also need to do some basic visualization. Here, I'll show you bar charts, box plots, histograms, and scatter plots.

5.1 Bar Charts

Let's make a barplot of the number of houses sold by year. We'll use the `kc.arranged` data frame.

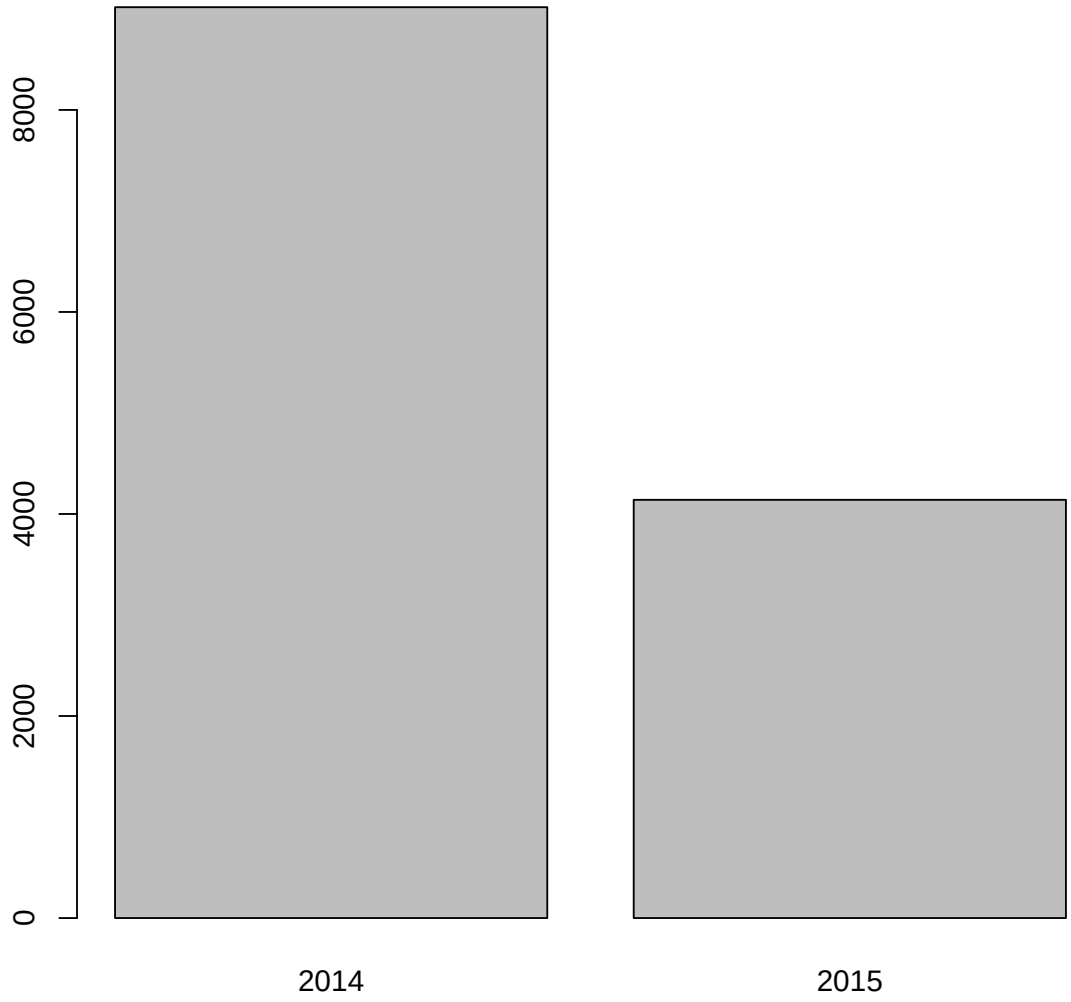
```
year.data = kc.arranged %>%
             group_by(year) %>%
             summarize(sales = length(id))
year.data
```

```
## # A tibble: 2 x 2
##    year sales
##   <int> <int>
## 1  2014  9017
## 2  2015  4140
```

```
year = year.data$year
sales = year.data$sales
```

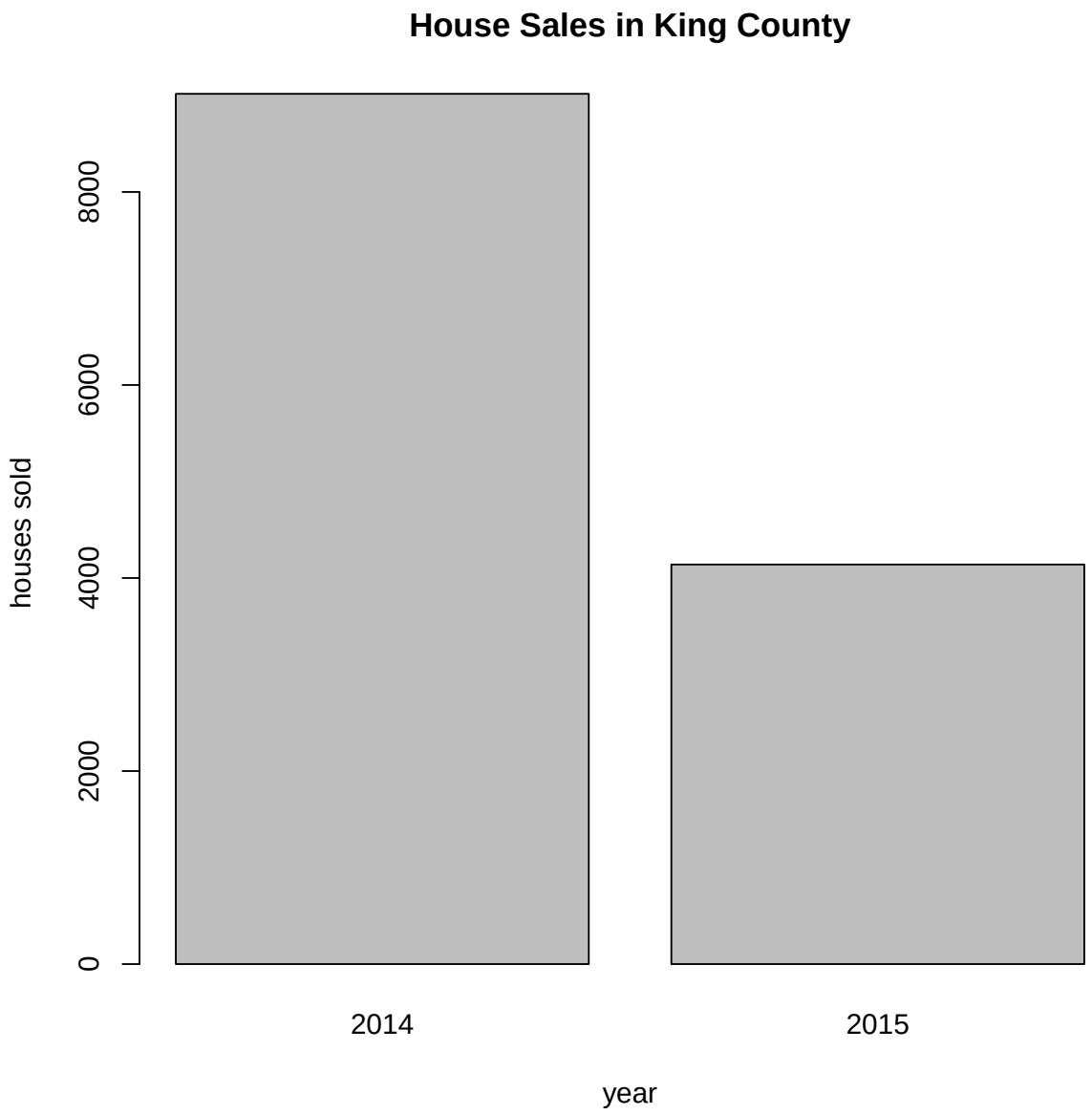
Now, we have vectors of the years and the sales. From here, we can make a barplot.

```
barplot(sales, names.arg = year)
```



This gives us a reasonable start for a bar plot. However, we want these plots to look nice and professional rather than like a 5th grade science project. So, we should add axis labels and a title.

```
barplot(sales,
        names.arg = year,
        main = "House Sales in King County",
        xlab = "year",
        ylab = "houses sold")
```

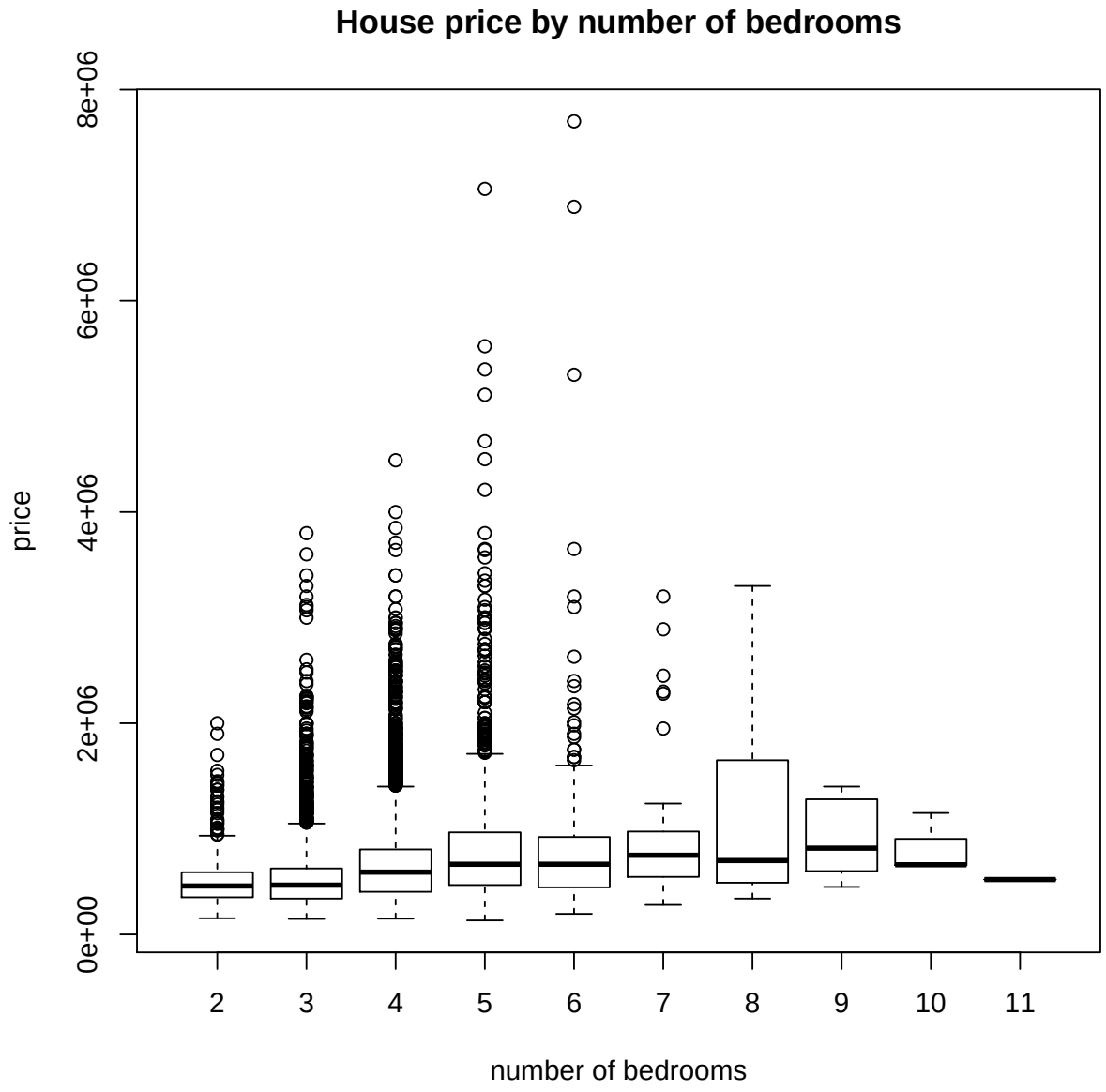


This looks better. There are other optional parameters for playing around with fonts, but let's leave this where it is right now. Also, you may have seen side-by-side bar plots before. For example, for each year, we may have broken down the sales by the number of bedrooms in the house. To make this bar plot, you have to use matrices. It's a bit beyond the scope of what we want to do now, so we'll continue on.

5.2 Box Plots

The next type of plot we'll look at is box plots. Box plots are one way of showing the distribution of a random variable in one dimension. They're nice because you can easily compare multiple distributions side-by-side. For example, suppose we want to look at the distribution of house price by number of bedrooms in our `kc.arranged` data frame.

```
price.by.room = kc.arranged %>%
                  group_by(bedrooms) %>%
                  summarize(price.list = list(price))
price.list = price.by.room$price.list
bedrooms   = price.by.room$bedrooms
boxplot(price.list,
        names = bedrooms,
        main = "House price by number of bedrooms",
        xlab = "number of bedrooms",
        ylab = "price")
```



This was a bit complicated, so perhaps we can look at some of the smaller pieces. First,

let's look at the head of our summary.

```
head(price.by.room)
```

```
## # A tibble: 6 x 2
##   bedrooms    price.list
##      <int>        <list>
## 1        2   <dbl [573]>
## 2        3 <dbl [5,325]>
## 3        4 <dbl [5,581]>
## 4        5 <dbl [1,376]>
## 5        6   <dbl [244]>
## 6        7    <dbl [35]>
```

Note that in the bedrooms column, we see the "¡int¿" header, which helpfully tells us that the column consists of integer data. For the second column, we see "¡list¿" instead. This is another data type that we haven't discussed. Essentially, a list is an ordered connection of any type of variable. Here, we see that it consists of double (precision floating point numbers), or rather, vectors of doubles of various lengths. We can look at an element if we want

```
price.list[[8]] # Note the double brackets for a list to get the element.
```

```
## [1]  450000  599999  700000  934000 1280000 1400000
```

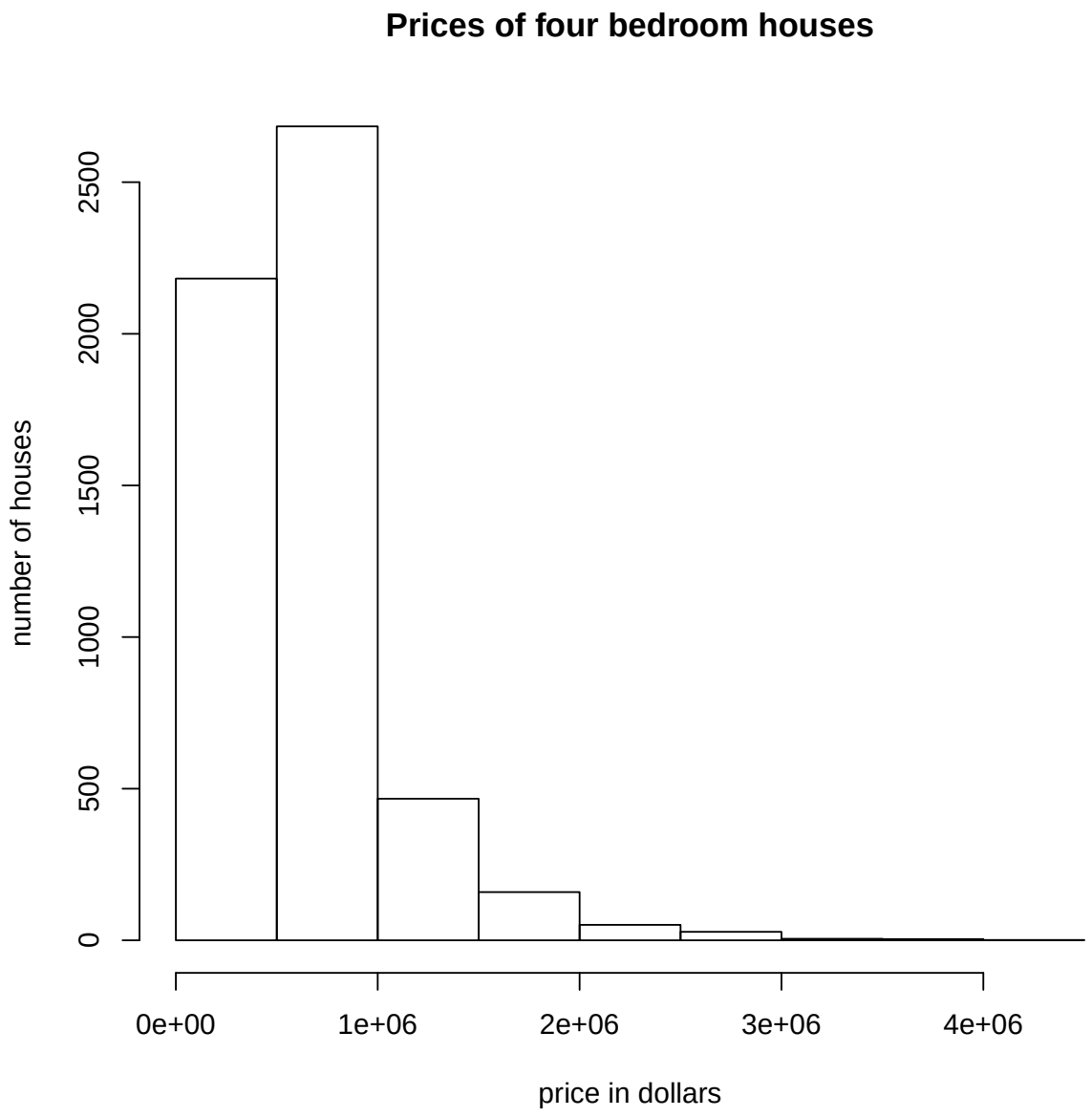
This provides us with the six house prices for the houses that sold with 9 bedrooms (since we start counting at 2 bedrooms, remember).

Long story short, how does the boxplot function work again? You give it a list of vectors, and then it plots each vector in a boxplot. Simple enough, hopefully.

5.3 Histograms

Histograms are another way of showing the distribution of a one-dimensional variable. The upside is that you get a better sense of the distribution of the variable than with a boxplot, but the downside is that it is harder to compare multiple distributions. Additionally, there's the problem of bin sizes, which are arbitrary. In any case, let's make a histogram of the prices of houses with four bedrooms (and at least two bathrooms).

```
four.bedrooms.prices = price.list[[3]] # Again, 3 because
                                       # we exclude 1 bedrooms.
hist(four.bedrooms.prices,
     main = "Prices of four bedroom houses",
     xlab = "price in dollars",
     ylab = "number of houses")
```



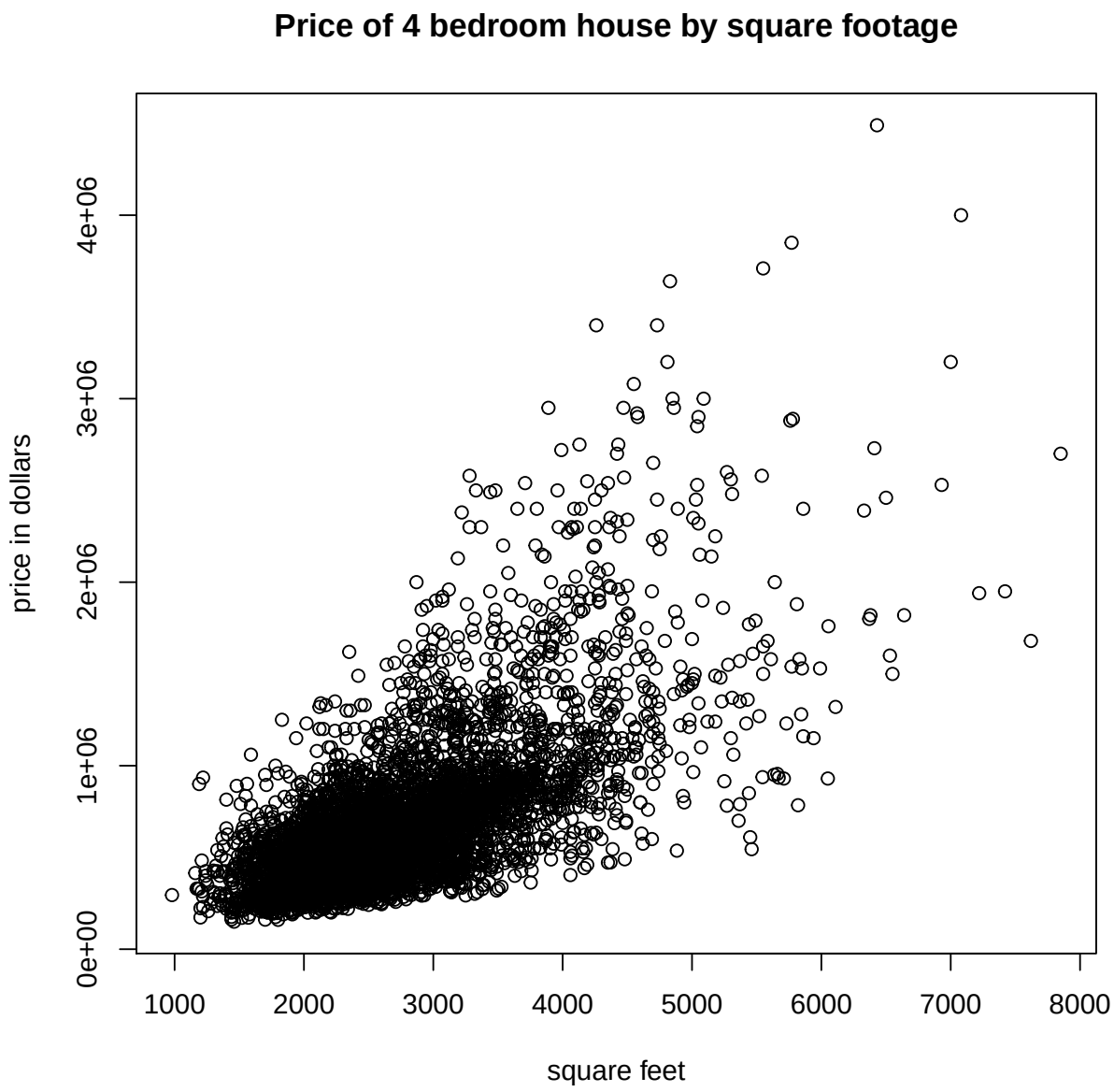
Amazing, right?

5.4 Scatter Plots

Now, let's make a scatter plot. Here, we'll just consider the price of a four bedroom (with at least two bathrooms) by square footage. We'll need to define a new data frame.

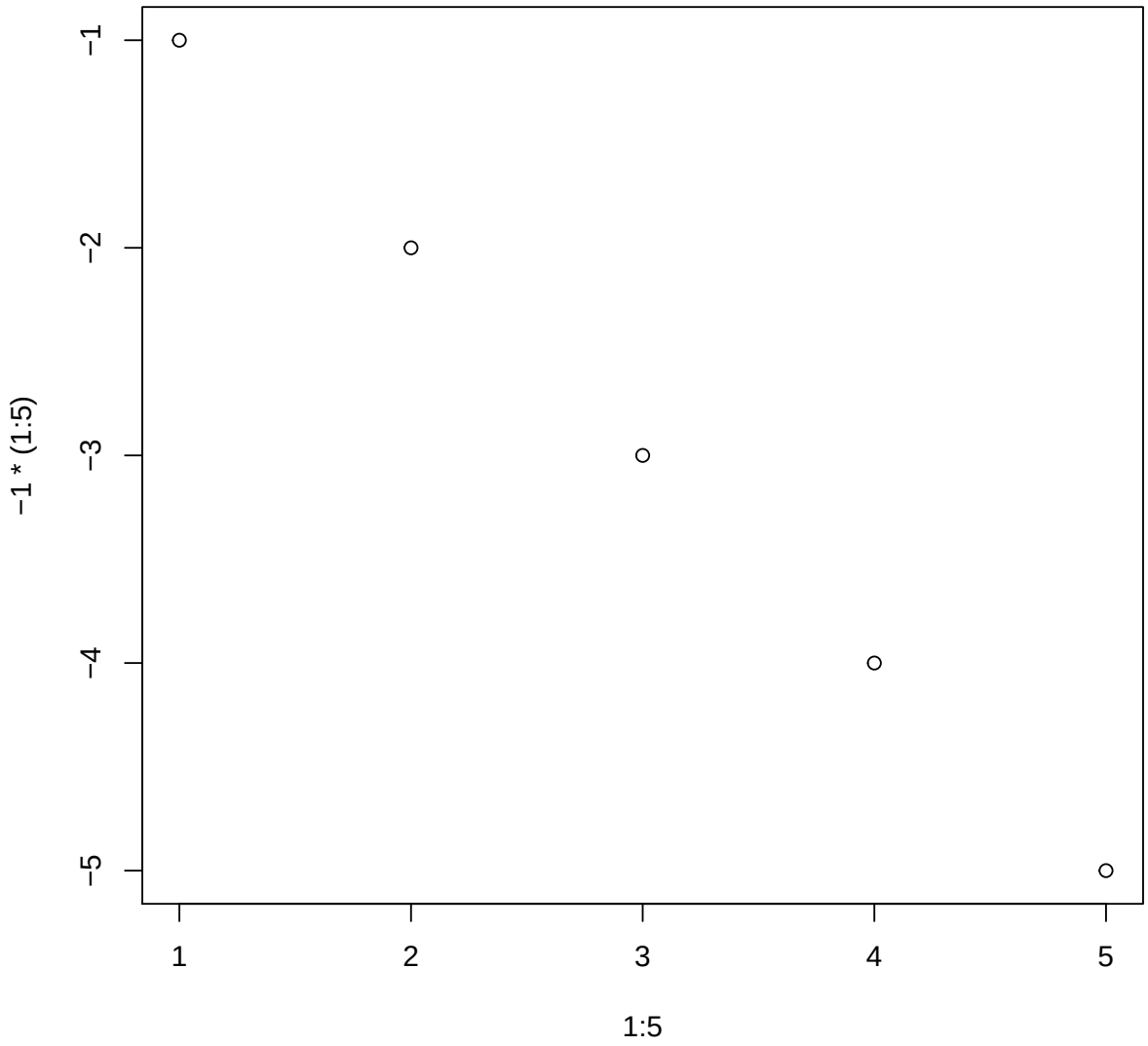
```
four.bedrooms = kc.df %>% filter(bedrooms == 4,
                                 bathrooms >= 2) %>%
                          select(sqft_living, price)
plot(four.bedrooms$sqft_living,
     four.bedrooms$price,
```

```
     main = "Price of 4 bedroom house by square footage",
     xlab = "square feet",
     ylab = "price in dollars")
```



There we go! Now we have a scatter plot. So, what exactly did we do here? Well, we gave the `plot` function two vectors of numerical data of equal length, and the points were plotted. We could easily come up with a simpler, more transparent example.

```
plot(1 : 5,
     -1 * (1 : 5))
```



This is hopefully simple enough.

5.5 Saving Plots

One last thing. I haven't told you how to save plots yet, but it's easy enough to do in RStudio. Simply click the "Export" command in the plotting pane, and save the file as an image or a pdf. Or, save it to your clipboard to paste into your printed assignment.

There are ways to save plots in R directly from the console. Also, using these more advanced options, it's possible to change the font sizes and margins of the plots. This is probably too much for us right now though.

6 Practice

If you would like some practice with `dplyr`, then there's a short set of practice questions (and solutions) at `https://www.r-bloggers.com/lets-get-started-with-dplyr/`. If you would like some additional plotting work, then you can also look at my Stat 111 recitation 2 notes. The necessary data is also available on my webpage, `http://kiefer.wharton.upenn.edu/~jkhim/teaching.html`.