Introduction to R (with candy!)

Dan Chitwood, Donald Danforth Plant Science Center (St. Louis, MO)

March 27, 2016

Introduction

The point of this script is to serve as an introduction to computational statistics in R. It was prepared for the graduate course Bio5702, "Current Approaches in Plant Research", at Washington University in Saint Louis, given March 30, 2016 by Dan Chitwood and Chris Topp (Donald Danforth Plant Science Center, St. Louis). It is the first of a series of lectures focusing on quantitative biology, image processing, morphometrics, statistics, and phenotyping methods. Candy will be used as an example in these lectures, and the dataset presented will be expanded upon as further data is collected.

There will be many needs during your careers for statistics, especially in a "Big Data" context, for which computational statistics is indispensable. Among them, they include:

- QTL (Quantitative Trait Loci) analysis
- Performing GWAS (Genome Wide Association Studies)
- Analyzing RNA-Seq data (a quantitative analysis of gene expression using next-generation sequencing technologies)
- Phenotyping and using image processing and morphometric techniques
- Phylogenetic, evolutionary, and ecological analyses
- Many more unique applications of statistics in biology too numerous to list here. Perhaps you will invent...
- an entirely new field of statistical biology!

Most importantly, intuitively understanding statistics and how to format and manipulate large datasets in a quantitative way will give you a toolset to ask new questions in novel ways, and further your abilities to be critical about quantitative data. Statistics in biology, especially quantitative genetics, is a time-honored tradition. Many of the techniques we will learn, such as ANOVA (Analysis of Variance), were either developed or popularized by famous biologists, such as Sir Ronald Fisher.

About the candy

Yes, candy will be analyzed during this course. Candy types range from chocolate and peanut butter to gummi and sugar-based candies. As proof, here is a picture of the 75 candy types to be analyzed:

There is one rule: *no candy shall be eaten until completely phenotyped!* This is your incentive to learn multivariate statistics and image analysis as applied to measuring phenotype. What do we mean by phenotype? We mean all attributes of an organism (or, in this case candy). Phenotype is infinite: not only is it the sum attributes from molecular to cellular to organismal levels of organization, but it varies plastically in response to innumerable environments an organism may encounter and is expressed dynamically over the course of an organism's development or lifecycle. We can never measure the true phenotype of living things—or even *candy*—but we will try to measure as many attributes of our 75 distinct candy species as we possibly can. This introduction to R tutorial includes the nutrition label information from the 75 distinct candy types we will be analyzing. **But it up to you** over the next couple weeks to:

1) Take images of thousands of individual candy components
2) Use computational image analysis methods to segment and isolate individual candy components
3) Perform morphometric and colorimetric analysis on thousands of candy pieces from 75 candy types
4) Analyze the multivariate data collected
5) Be able to apply the high-throughput phenotyping methods and multivariate statistical techniques introduced to you in this course to any biological question of interest to you, whether bacterial or plant, whether biochemical, genetic, organismal, or ecological.

The use of candy was inspired by a recent video from Emily Graslie prompting taxonomists at the Field Museum to classify candy types. The classification of organisms is a long-standing and still highly-debated problem central to biology, touching on disciplines as diverse as evolution and quantitative genetics. Just like the taxonomists in this video, your task is to classify the candies. Our approach will be to measure as many features of the candy as possible, and you will be given full freedom to come up with any "traits" you would like to measure. The multivariate analyses you will perform are relevant to many computational statistics questions you will face during your careers, whether the analysis of metabolomic data or gene expression, or measuring the ecological relationships between organisms or mapping the genetic basis of a complex trait.

Be sure to check out the video! Also, you can follow the documentation of this course on twitter with the hashtag #CandyPhenotyping.

How does R work?

If you are reading this, having opened this script in R, you have already downloaded R. R is a free platform, which is important because it enables people—just like you—to use "packages" (suites of custom functions and programs) for specialized analyses or even for you to write your own programs to share with others. In short, R is open source software that enables a community of biological statistics. With time, you will also learn the language and syntax of R, its grammar. And once you can do this, you will be able to do very powerful analyses and visualizations of statistical and quantitative data.

You may find it interesting that instructions you are reading can be written right into the script like this. The first thing to know about code and scripts is that the hashtag symbol `#` at the beginning of a line tells R to ignore any code or text after it. In just a few minutes you will "execute" your first lines of R code. If that code were to be preceded by a hastag `#`, it would be ignored and not executed.

So, let's try and execute our first lines of R code. If you opened up this script using the R program you downloaded, this script should be one window, and there should be another window, labeled the `R Console`. There should be text in this window, and it should begin like this, or something similar based on your computer and operating system:

```
R version 3.2.4 (2016-03-10) -- "Very Secure Dishes"

Copyright (C) 2016 The R Foundation for Statistical Computing

Platform: x86_64-apple-darwin13.4.0 (64-bit)
```

Additionally, there should be a little "greater than" symbol, like this, as the last line: `>`. This is the prompt, and it tells you what the computer is doing, and it also tells you the instructions that your computer has received.

You use this window (the script) to communicate to your computer (as displayed on the console). By doing things this way, the script serves as a record of your programming activities. Keeping good scripts that are well-organized and can be understood by you and others in the future is one of the first principles of statistical computing. In a way, keeping good scripts is like keeping a good lab notebook. It allows for reproducibility.

OK, let's execute our first function! One of the most simplest functions in R and most other programs is `print()`. The `print()` function will print text onto your console. The traditional first code to execute in almost all languages is to print the text `Hello world!`. Functions are followed by a pair of parentheses. Within these parentheses, you put "arguments": arguments are variables that tell a function what to do. In our case, we will execute the print function with the following arguments so that it will print `Hello world!` in the console: `print("Hello world!")`

To do this, either highlight the below line of code, or put your cursor at the end of the line. Then, if you are on a Mac, press "command+enter". If you are on a PC, then press "control+r" (Note: I am using a Mac, so what I suggest for PC might be wrong. Please email me and let me know if this is the case and we'll figure it out together! Or consult the internet simply by googling your questions: you'll be surprised how often you find the answer!).

```
print("Hello world!")
```

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

If you ever need help with a function in R, just type `?` in front of the function and execute the code. Try it below to learn about the print function:

```
?print
```

How to read data into R

We won't be formalistic for this course, because we don't have much time. Two weeks is simply not enough time. So let's jump to the chase, and start using R like you would in real life.

In real life, the first thing you usually want to do in R is to read in your data. R has many different types of data formats. Each is different and each has its own peculiarities. Some of these formats include: vectors, matrices, tables, arrays, and dataframes. The dataframe is probably the most popular format and the one we'll be using. A dataframe is essentially like an excel file: there are rows and columns. The rows and columns have names and you can refer to them as such.

You first need to tell R where to find the data! R assumes you are working from a particular path, a specific folder, and this is called the "working directory". Good practice is to put an R script plus the associated datafiles you will be using with it in the same folder. For example, this script, "Intro_to_R.R", should be in the same folder as the data file we'll be reading in, "candy_nutrition.txt". If you are on a Mac, if you first open R from this script (e.g., by dragging the script into the R icon on the dock, or opening R from the script itself), in the folder with the datafile, your "working directory" will be the folder and you can read in the data.

Let's check what the current working directory is. You can get the current working directory with the function `getwd()` below, and it will appear in your console. Try it!

```
getwd()
```

```
## [1] "/Users/bratsche/Desktop/R_Bio5702"
```

What if this is not the folder where this script and "candy_nutrition.txt" is? If on a Mac, you can change the working directory from the "Misc" subheading in the menu bar and select "Change Working Directory. . . " If on a PC, I believe you go to the "File" subheading in the menu bar and select "Change dir. . . " (again, I might be wrong about the PCs. Please inquire and get help if you have problems!). After selecting this option, you will be prompted to browse and select the folder that R should be working from. You can also do this using the `setwd()`. In my case (yours will be different), I am telling R to find my data in a folder called "R_Bio5702" on my Desktop. Please try to change your working directory from the menu bar first. If that doesn't work, try using `setwd()`. Please contact me if you have trouble with this step.

Just to make sure, double check you're in the folder with the data, by executing . . .

```
getwd()
```

```
## [1] "/Users/bratsche/Desktop/R_Bio5702"
```

Great! Let's read in some data! I personally save my files as tab-delimited .txt files, and one way to read in this type of data is using the `read.table()` function. One of the arguments, `header`, tells R to expect column names, which are present in our data file "candy_nutrition.txt", so we set header to `TRUE` as below:

```
data <- read.table("./candy_nutrition.txt", header=TRUE)
```

We should cover objects, because we just made one! Did you see that `data <-` part of the code above? What we did there is one of the most powerful parts of R. We created an object, called `data`. We could have called it anything we wanted to! But the little `<-` symbol is very intuitive, and should be read as an arrow, which it looks like. It means, "for the output of the `read.table()` function, put that output into an object that we're going to call `data`. Because we we're reading in the contents of a file, the data in that file will now be accessible in the object `data`. Again, you could have called the object whatever you wanted to, `dans_data`, `chris_data`, and `ursulas_data` are all examples! But I used `data` here.

Great! We have some data now! But what is the data? How do we look at it and know what we're working with?

Checking your data

In previous years, we had worked with phenotypic data from domesticated tomato and wild tomato relatives. But we thought we should do something different this year, because

1. not everyone in this class studies plants,
2. in real life Chris and I do study plants and maybe we want to study something different, and
3. you should collect your own data, because arguably the most important part of data analysis is collecting the data itself!

So, in that spirit, we will be phenotyping... **CANDY!!!**

This is only an initial dataset. We will be looking at 75 candy types, ranging from chocolate and peanut butter to the more sugar-based candies and gummis. This is a dataset to get you started with information from the nutritional labels, but over the next couple weeks, you will be "phenotyping" this candy for an impressive array of "traits", including:

1) morphometric information derived from image analysis,
2) colorimetric information,
3) sensory evaluation (how does it taste!), and most importantly
4) any traits that you can think of!!!

The topic of candy was chosen because the elements you should take away from this lecture have very little to do with subject matter, and ultimately you will be applying what you learn to your own problems. Rather, it is more important that you learn high-throughput phenotyping methods and how to statistically analyze large amounts of data.

So now that we laid out the data we will be collecting, how do we start analyzing the data we just loaded into R?

The first thing you should do when you read in your data is check what it is and if it was read in correctly!

Let's use the `names()` function to look at the column names of our data

```
names(data)
```

```
##  [1] "id"                "name"              "company"
##  [4] "class"             "serving_size_g"    "calories"
##  [7] "calories_fat"      "total_fat_g"       "saturated_fat_g"
## [10] "cholesterol_mg"    "sodium_mg"         "total_carb_g"
## [13] "dietary_fiber_g"   "sugars_g"          "protein_g"
## [16] "primary_ingredient"
```

It tells us the name of the first column is `id`, the second `name`, the third `company`, etc.

Here's a brief run down of what all this data is:

- `id`: a unique id number given to each candy type
- `name`: name of candy
- `company`: company that makes the candy
- `class`: general class of candy
- `serving_size_g`: the serving size in grams
- `calories`: calories per serving
- `calories_fat`: calories from fat
- `total_fat_g`: the total fat in grams
- `saturated_fat_g`: the saturated fat in grams
- `cholesterol_mg`: chloesterol in mg
- `sodium_mg`: sodium in mg
- `total_carb_g`: total carbs in grams
- `dietary fiber`: dietary fiber in grams
- `sugars`: sugars in grams
- `primary ingredient`: the first ingredient listed on the label

But what if we want more? Let's use the function `head()` to look at the first 6 lines of our data:

```
head(data)
```

```
##     id                     name     company     class serving_size_g
## 1 id_1                mini_eggs     cadbury chocolate             40
## 2 id_2    soft_eating_liquorice darrell_lea liquorice             42
## 3 id_3              raspberries      haribo     sugar             39
## 4 id_4               candy_corn        nice     gummi             41
## 5 id_5           crawlers_minis      trolli      sour             40
## 6 id_6 strawberry_shortcake_mms        mars chocolate             42
##   calories calories_fat total_fat_g saturated_fat_g cholesterol_mg
## 1      190           70           8               5              5
## 2      140           10           1               0              0
## 3      140            0           0               0              0
## 4      160          160           0               0              0
## 5      130            0           0               0              0
## 6      210          100          10               6              5
##   sodium_mg total_carb_g dietary_fiber_g sugars_g protein_g
## 1        30           28             0.5       27         2
## 2        40           30             0.0       16         1
## 3         0           36             0.0       29         1
## 4        75           39             0.0       32         0
## 5        35           31             0.0       24         1
## 6        40           29             0.0       28         2
##   primary_ingredient
```

```
## 1          chocolate
## 2              syrup
## 3              sugar
## 4              sugar
## 5              syrup
## 6          chocolate
```

Now, you should see numbers and data underneath those columns! If you want to look at, let's say 10 lines, you can add the following argument (which you can learn about from `?head`)

```
head(data, n=10)
```

```
##       id                     name     company      class serving_size_g
## 1   id_1                mini_eggs     cadbury  chocolate             40
## 2   id_2    soft_eating_liquorice darrell_lea  liquorice             42
## 3   id_3              raspberries      haribo      sugar             39
## 4   id_4               candy_corn        nice      gummi             41
## 5   id_5           crawlers_minis      trolli       sour             40
## 6   id_6 strawberry_shortcake_mms        mars  chocolate             42
## 7   id_7           milk_chocolate     hershey  chocolate             43
## 8   id_8                milk_duds     hershey  chocolate             39
## 9   id_9       marshmallow_chicks   just_born      sugar             42
## 10 id_10    crazy_beans_starburst     wrigley jelly_bean             40
##    calories calories_fat total_fat_g saturated_fat_g cholesterol_mg
## 1       190           70           8             5.0              5
## 2       140           10           1             0.0              0
## 3       140            0           0             0.0              0
## 4       160          160           0             0.0              0
## 5       130            0           0             0.0              0
## 6       210          100          10             6.0              5
## 7       210          110          13             7.0              5
## 8       170           60           6             3.5              0
## 9       140            0           0             0.0              0
## 10      130            0           0             0.0              0
##    sodium_mg total_carb_g dietary_fiber_g sugars_g protein_g
## 1         30           28             0.5       27       2.0
## 2         40           30             0.0       16       1.0
## 3          0           36             0.0       29       1.0
## 4         75           39             0.0       32       0.0
## 5         35           31             0.0       24       1.0
## 6         40           29             0.0       28       2.0
## 7         35           26             2.0       22       3.0
## 8        105           27             0.0       20       0.5
## 9         15           36             0.0       34       1.0
## 10         0           33             0.0       26       0.0
##    primary_ingredient
## 1           chocolate
## 2               syrup
## 3               sugar
## 4               sugar
## 5               syrup
## 6           chocolate
## 7               sugar
```

```
## 8               syrup
## 9               sugar
## 10              sugar
```

To look at the last 6 lines, use `tail()`:

```
tail(data)
```

```
##       id                           name   company         class
## 70 id_70              marshmallow_eggs just_born         sugar
## 71 id_71            tropical_starburst   wrigley    jelly_bean
## 72 id_72             reeses_miniatures   hershey peanut_butter
## 73 id_73 reese_peanut_butter_eggs_large   hershey peanut_butter
## 74 id_74            original_starburst   wrigley    jelly_bean
## 75 id_75            marshmallow_chicks just_born         sugar
##    serving_size_g calories calories_fat total_fat_g saturated_fat_g
## 70             32      110            0           0               0
## 71             40      140            0           0               0
## 72             44      220          110          13               5
## 73             34      170           90          10               3
## 74             40      140            0           0               0
## 75             42      140            0           0               0
##    cholesterol_mg sodium_mg total_carb_g dietary_fiber_g sugars_g
## 70            0.0        10           28               0       26
## 71            0.0         0           34               0       27
## 72            2.5       130           26               1       23
## 73            0.0       135           18               1       16
## 74            0.0         0           34               0       27
## 75            0.0        15           34               0       32
##    protein_g primary_ingredient
## 70         1              sugar
## 71         0              sugar
## 72         4          chocolate
## 73         4            peanuts
## 74         0              sugar
## 75         1              sugar
```

Most powerful is the `summary()` function. Let's get a summary for the entire dataset:

```
summary(data)
```

```
##        id                         name         company             class
##  id_1   : 1   marshmallow_chicks   : 4   hershey  :17   chocolate    :26
##  id_10  : 1   marshmallow_bunnies  : 2   mars     :11   gummi        : 9
##  id_11  : 1   milk_chocolate       : 2   just_born: 8   jelly_bean   : 5
##  id_12  : 1   peanut_mms           : 2   wrigley  : 8   liquorice    : 6
##  id_13  : 1   soft_eating_liquorice: 2   nestle   : 6   peanut_butter: 7
##  id_14  : 1   baby_ruth            : 1   haribo   : 4   sour         : 7
##  (Other):69   (Other)              :62   (Other)  :21   sugar        :15
##  serving_size_g     calories      calories_fat  total_fat_g
##  Min.   : 7.00   Min.   : 25.0   Min.   :  0   Min.   : 0.000
##  1st Qu.:39.50   1st Qu.:140.0   1st Qu.:  0   1st Qu.: 0.000
```

```
##  Median :40.00   Median :150.0   Median : 15   Median : 1.500
##  Mean   :38.56   Mean   :158.2   Mean   : 39   Mean   : 4.167
##  3rd Qu.:42.00   3rd Qu.:190.0   3rd Qu.: 80   3rd Qu.: 8.500
##  Max.   :45.00   Max.   :220.0   Max.   :160   Max.   :13.000
##
##  saturated_fat_g cholesterol_mg     sodium_mg       total_carb_g
##  Min.   :0.000   Min.   : 0.000   Min.   :  0.00   Min.   : 6.00
##  1st Qu.:0.000   1st Qu.: 0.000   1st Qu.: 10.00   1st Qu.:26.00
##  Median :1.000   Median : 0.000   Median : 30.00   Median :31.00
##  Mean   :2.347   Mean   : 1.567   Mean   : 37.73   Mean   :29.63
##  3rd Qu.:5.000   3rd Qu.: 2.500   3rd Qu.: 50.00   3rd Qu.:34.00
##  Max.   :8.000   Max.   :10.000   Max.   :180.00   Max.   :39.00
##
##  dietary_fiber_g     sugars_g       protein_g    primary_ingredient
##  Min.   :0.0000   Min.   : 6.00   Min.   :0.00   chocolate:21
##  1st Qu.:0.0000   1st Qu.:20.50   1st Qu.:0.00   dextrose : 2
##  Median :0.0000   Median :24.00   Median :1.00   peanuts  : 1
##  Mean   :0.3667   Mean   :23.63   Mean   :1.42   sugar    :37
##  3rd Qu.:1.0000   3rd Qu.:27.00   3rd Qu.:2.00   syrup    :14
##  Max.   :2.0000   Max.   :35.00   Max.   :5.00
##
```

Some of the columns, like `id`, `name`, `company`, `class`, and `primary_ingredient`, list how many times a particular word is found. Others, like `calories` and `total_fat_g`, list minimums, maximums, means, and medians, as well as 1st and 3rd quartiles (the 25th and 75th percentiles). Variables in R are either "factors" or "numerics". Basically, these are categorical and continuous variables. Categorical variables (factors, such as `company`) are counted for each level ("level" = type, or category) in the summary, whereas continuous variables (numerics, such as `calories` or `serving_size_g`) are summarized by min, max, mean, median, etc.

Dealing with dataframes

Great! We've read in data and checked it. Let's play around with it some. Figuring out how to manipulate your data in R is the quickest way to start being productive and doing what you need to get done.

Let's talk about some of the syntax of R for dataframes, which will allow you to perform powerful manipulations of the data. Let's say you only wanted to look at rows 5-7. Or let's say you only wanted to look at specific columns. You can refer to a data frame in the following format: data[rows,columns]. Let's say we only want to look at rows 5-7, then we could execute the following syntax:

```
data[5:7,]
```

```
##     id                     name company     class serving_size_g calories
## 5 id_5           crawlers_minis  trolli      sour             40      130
## 6 id_6 strawberry_shortcake_mms    mars chocolate             42      210
## 7 id_7           milk_chocolate hershey chocolate             43      210
##   calories_fat total_fat_g saturated_fat_g cholesterol_mg sodium_mg
## 5            0           0               0              0        35
## 6          100          10               6              5        40
## 7          110          13               7              5        35
##   total_carb_g dietary_fiber_g sugars_g protein_g primary_ingredient
## 5           31               0       24         1              syrup
## 6           29               0       28         2          chocolate
## 7           26               2       22         3              sugar
```

You should see only rows 5-7 in your console. Note the use of colon to say "5 through 7". Note also the column information after the comma is blank, so all columns are displayed. Let's get fancy and retrieve rows 5-7 plus row 10:

```
data[c(5:7,10),]
```

```
##       id                      name company      class serving_size_g
## 5   id_5           crawlers_minis  trolli       sour             40
## 6   id_6 strawberry_shortcake_mms    mars  chocolate             42
## 7   id_7           milk_chocolate hershey  chocolate             43
## 10 id_10    crazy_beans_starburst wrigley jelly_bean             40
##    calories calories_fat total_fat_g saturated_fat_g cholesterol_mg
## 5       130           0           0               0              0
## 6       210         100          10               6              5
## 7       210         110          13               7              5
## 10      130           0           0               0              0
##    sodium_mg total_carb_g dietary_fiber_g sugars_g protein_g
## 5         35           31               0       24         1
## 6         40           29               0       28         2
## 7         35           26               2       22         3
## 10         0           33               0       26         0
##    primary_ingredient
## 5               syrup
## 6           chocolate
## 7               sugar
## 10              sugar
```

Now we have rows 5,6,7, and 10. When we start doing more complicated things, we add the `c()`, which means to "combine" its arguments into one. You can get as complicated as you like, for example:

```
data[c(5:7,10,17,74,100000),]
```

```
##       id                      name company      class serving_size_g
## 5   id_5           crawlers_minis  trolli       sour             40
## 6   id_6 strawberry_shortcake_mms    mars  chocolate             42
## 7   id_7           milk_chocolate hershey  chocolate             43
## 10 id_10    crazy_beans_starburst wrigley jelly_bean             40
## 17 id_17        milky_way_caramel    mars  chocolate             22
## 74 id_74       original_starburst wrigley jelly_bean             40
## NA  <NA>                     <NA>    <NA>       <NA>             NA
##    calories calories_fat total_fat_g saturated_fat_g cholesterol_mg
## 5       130           0         0.0               0              0
## 6       210         100        10.0               6              5
## 7       210         110        13.0               7              5
## 10      130           0         0.0               0              0
## 17      100          40         4.5               3              5
## 74      140           0         0.0               0              0
## NA       NA          NA          NA              NA             NA
##    sodium_mg total_carb_g dietary_fiber_g sugars_g protein_g
## 5         35           31               0       24         1
## 6         40           29               0       28         2
## 7         35           26               2       22         3
## 10         0           33               0       26         0
```

```
## 17        25           15               0       13         1
## 74         0           34               0       27         0
## NA        NA           NA              NA       NA        NA
##    primary_ingredient
## 5               syrup
## 6           chocolate
## 7               sugar
## 10              sugar
## 17          chocolate
## 74              sugar
## NA               <NA>
```

Note that I kind of got silly, trying to ask for row 100000, because no row 100000 exists. R gave `NA`s back for that, which is standard terminology for "not available" or "does not compute". `NA` reads "missing data".

Remember! You can create an object at anytime in R, with any name, and as many as you like! For example:

```
silly_object <- data[c(5:7,10,17,74,100000),]
```

And then we can look at the contents of silly object,

```
silly_object
```

```
##       id                     name company      class serving_size_g
## 5   id_5           crawlers_minis  trolli       sour             40
## 6   id_6 strawberry_shortcake_mms    mars  chocolate             42
## 7   id_7           milk_chocolate hershey  chocolate             43
## 10 id_10    crazy_beans_starburst wrigley jelly_bean             40
## 17 id_17        milky_way_caramel    mars  chocolate             22
## 74 id_74       original_starburst wrigley jelly_bean             40
## NA  <NA>                     <NA>    <NA>       <NA>             NA
##    calories calories_fat total_fat_g saturated_fat_g cholesterol_mg
## 5       130           0         0.0               0              0
## 6       210         100        10.0               6              5
## 7       210         110        13.0               7              5
## 10      130           0         0.0               0              0
## 17      100          40         4.5               3              5
## 74      140           0         0.0               0              0
## NA       NA          NA          NA              NA             NA
##    sodium_mg total_carb_g dietary_fiber_g sugars_g protein_g
## 5         35           31               0       24         1
## 6         40           29               0       28         2
## 7         35           26               2       22         3
## 10         0           33               0       26         0
## 17        25           15               0       13         1
## 74         0           34               0       27         0
## NA        NA           NA              NA       NA        NA
##    primary_ingredient
## 5               syrup
## 6           chocolate
## 7               sugar
## 10              sugar
## 17          chocolate
```

```
## 74              sugar
## NA               <NA>
```

Or even summarize it

```
summary(silly_object)
```

```
##        id                        name       company         class
##  id_10  :1   crawlers_minis       :1   mars    :2   chocolate :3
##  id_17  :1   crazy_beans_starburst:1   wrigley :2   jelly_bean:2
##  id_5   :1   milk_chocolate       :1   hershey :1   sour      :1
##  id_6   :1   milky_way_caramel    :1   trolli  :1   gummi     :0
##  id_7   :1   original_starburst   :1   airheads:0   liquorice :0
##  (Other):1   (Other)             :1   (Other) :0   (Other)   :0
##  NA's   :1   NA's                :1   NA's    :1   NA's      :1
##  serving_size_g     calories      calories_fat     total_fat_g
##  Min.   :22.00   Min.   :100.0   Min.   :  0.00   Min.   : 0.000
##  1st Qu.:40.00   1st Qu.:130.0   1st Qu.:  0.00   1st Qu.: 0.000
##  Median :40.00   Median :135.0   Median : 20.00   Median : 2.250
##  Mean   :37.83   Mean   :153.3   Mean   : 41.67   Mean   : 4.583
##  3rd Qu.:41.50   3rd Qu.:192.5   3rd Qu.: 85.00   3rd Qu.: 8.625
##  Max.   :43.00   Max.   :210.0   Max.   :110.00   Max.   :13.000
##  NA's   :1       NA's   :1       NA's   :1        NA's   :1
##  saturated_fat_g cholesterol_mg   sodium_mg      total_carb_g
##  Min.   :0.000   Min.   :0.0    Min.   : 0.00   Min.   :15.00
##  1st Qu.:0.000   1st Qu.:0.0    1st Qu.: 6.25   1st Qu.:26.75
##  Median :1.500   Median :2.5    Median :30.00   Median :30.00
##  Mean   :2.667   Mean   :2.5    Mean   :22.50   Mean   :28.00
##  3rd Qu.:5.250   3rd Qu.:5.0    3rd Qu.:35.00   3rd Qu.:32.50
##  Max.   :7.000   Max.   :5.0    Max.   :40.00   Max.   :34.00
##  NA's   :1       NA's   :1      NA's   :1       NA's   :1
##  dietary_fiber_g     sugars_g       protein_g     primary_ingredient
##  Min.   :0.0000   Min.   :13.00   Min.   :0.000   chocolate:2
##  1st Qu.:0.0000   1st Qu.:22.50   1st Qu.:0.250   dextrose :0
##  Median :0.0000   Median :25.00   Median :1.000   peanuts  :0
##  Mean   :0.3333   Mean   :23.33   Mean   :1.167   sugar    :3
##  3rd Qu.:0.0000   3rd Qu.:26.75   3rd Qu.:1.750   syrup    :1
##  Max.   :2.0000   Max.   :28.00   Max.   :3.000   NA's     :1
##  NA's   :1        NA's   :1       NA's   :1
```

The above is different than the summary for the entire `data` object

You can isolate columns in the same fashion. Let's only request rows 50 through 54 for column 2, which is candy `name`

```
data[c(50:54),2]
```

```
## [1] jel_bunnies     gummy_sharks    good_and_plenty mike_and_ike
## [5] sour_patch_kids
## 68 Levels: baby_ruth butterfinger candy_corn ... whoppers_robin_eggs
```

What R returns is just a series of 5 names, the candies that are found in rows 50:54

Another very powerful way to refer to columns is by the $ symbol. The dataframe name followed by the column name will retrieve just that column. Let's look at the first 6 candy names listed using the head function together with our new $ notation:

```
head(data$name)
```

```
## [1] mini_eggs                soft_eating_liquorice
## [3] raspberries              candy_corn
## [5] crawlers_minis           strawberry_shortcake_mms
## 68 Levels: baby_ruth butterfinger candy_corn ... whoppers_robin_eggs
```

Formatting and manipulating your data

Next, we'll explore four powerful functions in R: **subset()**, **tapply()**, **cbind()**, **rbind()**

subset(): Let's say you wanted only the data from only **hershey** candies? We would use the **subset()** function to extract only those rows where **company == "hershey"**. Why did I use two **==**? In programming, the equal signs get complicated. You can use **!=** to mean "everything but" for example. Or **<=** to mean "lesser than or equal to" or **>=** to mean "greater than or equal to". So **==** essentially means "I really mean equals!"

```
just_hershey <- subset(data, company=="hershey")
everything_but <- subset(data, company!="hershey")
```

We can even subset continuous variables. Maybe we only want the candies < 150 calories or greater than or equal to 150 calories

```
lotsa_cals <- subset(data, calories>=150)
few_cals <- subset(data, calories<150)
```

If we look at the summary for just_hershey we see there are only **hershey** samples now

```
summary(just_hershey)
```

```
##        id                   name           company              class
##  id_16  : 1   good_and_plenty: 1   hershey    :17   chocolate    :8
##  id_22  : 1   kisses         : 1   airheads   : 0   gummi        :0
##  id_23  : 1   krackel        : 1   brachs     : 0   jelly_bean   :0
##  id_36  : 1   milk_chocolate : 1   cadbury    : 0   liquorice    :4
##  id_39  : 1   milk_duds      : 1   darrell_lea: 0   peanut_butter:5
##  id_40  : 1   mr_goodbar     : 1   haribo     : 0   sour         :0
##  (Other):11   (Other)        :11   (Other)    : 0   sugar        :0
##  serving_size_g     calories      calories_fat     total_fat_g
##  Min.   :34.00   Min.   :130.0   Min.   :  0.00   Min.   : 0.000
##  1st Qu.:40.00   1st Qu.:170.0   1st Qu.: 50.00   1st Qu.: 5.000
##  Median :40.00   Median :190.0   Median : 90.00   Median :10.000
##  Mean   :40.18   Mean   :181.8   Mean   : 70.88   Mean   : 8.206
##  3rd Qu.:43.00   3rd Qu.:210.0   3rd Qu.:110.00   3rd Qu.:13.000
##  Max.   :44.00   Max.   :220.0   Max.   :110.00   Max.   :13.000
##
##  saturated_fat_g cholesterol_mg     sodium_mg       total_carb_g
##  Min.   :0.000   Min.   : 0.000   Min.   : 35.00   Min.   :18.00
```

```
##  1st Qu.:3.000   1st Qu.: 0.000   1st Qu.: 35.00   1st Qu.:25.00
##  Median :5.000   Median : 0.000   Median : 65.00   Median :26.00
##  Mean   :4.618   Mean   : 2.206   Mean   : 79.12   Mean   :27.12
##  3rd Qu.:7.000   3rd Qu.: 5.000   3rd Qu.:105.00   3rd Qu.:31.00
##  Max.   :8.000   Max.   :10.000   Max.   :180.00   Max.   :35.00
##
##  dietary_fiber_g     sugars_g       protein_g     primary_ingredient
##  Min.   :0.0000   Min.   :16.00   Min.   :0.500   chocolate:4
##  1st Qu.:0.0000   1st Qu.:19.00   1st Qu.:0.500   dextrose :0
##  Median :1.0000   Median :22.00   Median :3.000   peanuts  :1
##  Mean   :0.8235   Mean   :21.12   Mean   :2.206   sugar    :8
##  3rd Qu.:1.0000   3rd Qu.:23.00   3rd Qu.:3.000   syrup    :4
##  Max.   :2.0000   Max.   :28.00   Max.   :4.000
##
```

Alternatively, to be more precise, we can look at the summary of just the `company` column in the object `just_hershey`

```
summary(just_hershey$company)
```

```
##     airheads       brachs      cadbury  darrell_lea       haribo
##            0            0            0            0            0
##      hershey    just_born         mars       nestle         nice
##           17            0            0            0            0
##       reeses     smarties   sour_patch       target tootsie_roll
##            0            0            0            0            0
##       trolli      wrigley
##            0            0
```

'tapply()': is a more complicated function, but very powerful. Let's say that you wanted to find the mean saturated fats in grams for each of the different `primary_ingredient`. `tapply()` will take a variable, like `saturated_fat_g`, and index it by a factor, like `primary_ingredient`, to find the values of a function, like `mean()`, for each level of that factor (in this case, each type of `primary_ingredient`). You can read how the function works with `?tapply`, but to summarize here, `tapply(value, index, function)`, for example:

```
ingredient_means <- tapply(data$saturated_fat_g, data$primary_ingredient, mean)
```

We then check the results and see that indeed, means for `saturated_fat_g` were calculated for each `primary_ingredient`

```
ingredient_means
```

```
## chocolate  dextrose   peanuts     sugar     syrup
## 4.8571429 0.0000000 3.0000000 1.6891892 0.6071429
```

Success! *Be careful* with things primarily made of chocolate and/or peanuts!

Let's try some more advanced formatting. Let's say you wanted to calculate the mean, median, variance, standard deviation, minimum, maximum, sum for each `class` for `total_fat_g`, then:

```
class_mean_fat <- tapply(data$total_fat_g, data$class, mean, na.rm=TRUE)
class_median_fat <- tapply(data$total_fat_g, data$class, median, na.rm=TRUE)
class_var_fat <- tapply(data$total_fat_g, data$class, var, na.rm=TRUE)
class_sd_fat <- tapply(data$total_fat_g, data$class, sd, na.rm=TRUE)
class_min_fat <- tapply(data$total_fat_g, data$class, min, na.rm=TRUE)
class_max_fat <- tapply(data$total_fat_g, data$class, max, na.rm=TRUE)
class_sum_fat <- tapply(data$total_fat_g, data$class, sum, na.rm=TRUE)
```

But that's awkward to refer to each function as a different object! Let's use the function `rbind()` to put them all together. The concept of `rbind()` is to literally bind the rows into one object, as below:

```
class_fat_rbind <- rbind(class_mean_fat, class_median_fat, class_var_fat, class_sd_fat, class_min_fat, c
```

Let's check the rbind object

```
class_fat_rbind
```

```
##                   chocolate gummi jelly_bean liquorice peanut_butter
## class_mean_fat     8.519231     0          0   0.75000     11.285714
## class_median_fat   8.000000     0          0   1.00000     11.000000
## class_var_fat      8.569615     0          0   0.17500      2.238095
## class_sd_fat       2.927391     0          0   0.41833      1.496026
## class_min_fat      3.000000     0          0   0.00000      9.000000
## class_max_fat     13.000000     0          0   1.00000     13.000000
## class_sum_fat    221.500000     0          0   4.50000     79.000000
##                       sour     sugar
## class_mean_fat   0.4285714 0.3000000
## class_median_fat 0.0000000 0.0000000
## class_var_fat    0.5357143 0.3857143
## class_sd_fat     0.7319251 0.6210590
## class_min_fat    0.0000000 0.0000000
## class_max_fat    1.5000000 1.5000000
## class_sum_fat    3.0000000 4.5000000
```

Success! All of the data is together now. But look at those row names! We can do better than that. Let's rename the row name with `rownames()`:

```
rownames(class_fat_rbind) <- c("mean","median","variance","stand_dev","minimum","maximum","sum")
```

Now, if we check the `class_fat_rbind` object again, the row names should be easier to read:

```
class_fat_rbind
```

```
##            chocolate gummi jelly_bean liquorice peanut_butter      sour
## mean        8.519231     0          0   0.75000     11.285714 0.4285714
## median      8.000000     0          0   1.00000     11.000000 0.0000000
## variance    8.569615     0          0   0.17500      2.238095 0.5357143
## stand_dev   2.927391     0          0   0.41833      1.496026 0.7319251
## minimum     3.000000     0          0   0.00000      9.000000 0.0000000
## maximum    13.000000     0          0   1.00000     13.000000 1.5000000
## sum       221.500000     0          0   4.50000     79.000000 3.0000000
```

```
##               sugar
## mean      0.3000000
## median    0.0000000
## variance  0.3857143
## stand_dev 0.6210590
## minimum   0.0000000
## maximum   1.5000000
## sum       4.5000000
```

We should note here: `cbind()` and `colnames()` work analogously to `rbind()` and `rownames()`, binding together and naming columns instead of rows

Let's do one more dataframe manipulation and...

1) transpose the dataset and
2) create a new column with information derived from the other columns.

The function to transpose a dataframe is appropriately called `t()`:

```
transposed_fat <- t(class_fat_rbind)
```

And let's check if rows and columns have been transposed:

```
transposed_fat
```

```
##                    mean median  variance stand_dev minimum maximum   sum
## chocolate      8.5192308      8 8.5696154 2.9273905       3    13.0 221.5
## gummi          0.0000000      0 0.0000000 0.0000000       0     0.0   0.0
## jelly_bean     0.0000000      0 0.0000000 0.0000000       0     0.0   0.0
## liquorice      0.7500000      1 0.1750000 0.4183300       0     1.0   4.5
## peanut_butter 11.2857143     11 2.2380952 1.4960265       9    13.0  79.0
## sour           0.4285714      0 0.5357143 0.7319251       0     1.5   3.0
## sugar          0.3000000      0 0.3857143 0.6210590       0     1.5   4.5
```

Indeed they have. But you have to be careful when using transformations to dataframes in R. Remember, there are alternative data formats to the `data.frame`, including vectors, tables, matrices, and arrays. Let's see which type of object `transposed_fat` is:

```
is.vector(transposed_fat)
```

```
## [1] FALSE
```

```
is.table(transposed_fat)
```

```
## [1] FALSE
```

```
is.matrix(transposed_fat)
```

```
## [1] TRUE
```

```
is.array(transposed_fat)
```

```
## [1] TRUE
```

```
is.data.frame(transposed_fat)
```

```
## [1] FALSE
```

`transposed_fat` seems to be an array and matrix object. We need it to be a dataframe. One way to do this is the the `as.data.frame()` function:

```
transposed_fat_df <- as.data.frame(t(class_fat_rbind))

is.data.frame(transposed_fat_df)
```

```
## [1] TRUE
```

Great! Now `transposed_fat_df` is a data frame! Now, we can manipulate the columns more easily. As a test, let's create a new column. The new column will be called `sq_var`, and it will contain the square root of the variance, which should be equal to standard deviation. We will also create a column called `range`, which will be the maximum minus the minimum value. Note how the added columns are added automatically once they are defined, even though technically they don't even exist yet before you execute this line of code!

```
transposed_fat_df$sq_var <- sqrt(transposed_fat_df$var)
transposed_fat_df$range <- transposed_fat_df$maximum - transposed_fat_df$minimum
```

Let's check if the new columns, with their new information, were added correctly:

```
transposed_fat_df
```

```
##                    mean median  variance stand_dev minimum maximum   sum
## chocolate      8.5192308      8 8.5696154 2.9273905       3    13.0 221.5
## gummi          0.0000000      0 0.0000000 0.0000000       0     0.0   0.0
## jelly_bean     0.0000000      0 0.0000000 0.0000000       0     0.0   0.0
## liquorice      0.7500000      1 0.1750000 0.4183300       0     1.0   4.5
## peanut_butter 11.2857143     11 2.2380952 1.4960265       9    13.0  79.0
## sour           0.4285714      0 0.5357143 0.7319251       0     1.5   3.0
## sugar          0.3000000      0 0.3857143 0.6210590       0     1.5   4.5
##                  sq_var range
## chocolate     2.9273905  10.0
## gummi         0.0000000   0.0
## jelly_bean    0.0000000   0.0
## liquorice     0.4183300   1.0
## peanut_butter 1.4960265   4.0
## sour          0.7319251   1.5
## sugar         0.6210590   1.5
```

We created our own dataframe, and analyzed our own data, in R! Now, what if we want to use our data outside of R? How do we save our new data frame? We use the `write.table()` function, the arguments for which are the data object to write out and the name you want to give that object (again, consult `?write.table`). Note: the file will be saved in the working directory

```
write.table(transposed_fat_df, "my_fat_summary.txt")
```

Be sure to email me the saved file (plus the homework below) to show me you got through the tutorial!

Loading in packages, calling up libraries

One of the most powerful aspects of the R community is being able to use the tremendous amount of packages others have written. You can check out the multitudes of packages at the Comprehensive R Archive Network (CRAN), Bioconductor (a bioinformatics-specific suite of R software), or many developers that make their packages available through GitHub.

Let's install a package and call it up. My favorite package is ggplot2, by Hadley Wickham. It is a powerful visualization package. You should not be using excel for anything after learning R, but especially for your visualizations, which `ggplot2` makes very aesthetic. We will be going over how to make `ggplot2` graphs over the next few lectures. But for now, let's make sure we have it installed to save time.

First, try executing the code below, which hopefully will do the trick. You should see in the console the download process begin if it works, or get an error message if it doesn't.

```
install.packages("ggplot2")
```

If that doesn't work, you can also use the menu bar to initiate the installation process. On a Mac, go to `Packages & Data` in the menu bar, and select `Package Installar`. You will be prompted to search for the package (type `ggplot2` in the search bar), then click `Get List`. Select `ggplot2` then (**important**) click the `Install Dependencies` box so it is checked, and then click `Install Selected`. The download process should begin. You may be prompted to select a mirror location. If so, select one of the options (it doesn't matter) near you location wise.

On a PC, I believe you select `Packages` from the menu, then select `Install Packages`. You will be prompted to select a mirror site (just select the one nearest you). You should then be provided a list of packages and you can select and click `ggplot2` and then the `OK` button. The download and installation process should then start.

Once you have downloaded and installed a package, you need to load it into your current R session in order to use it. You do this using the `library()` function, as below:

```
library(ggplot2)
```

Data visualization with ggplot2

Once you have loaded your package, you can use the functions that are specific to it. Don't worry about the code below, we'll go over `ggplot2` more (below) and in class as well. But do execute the lines below, make sure you get a graph, and if you do, email it to me, so that I know you successfully loaded in the `ggplot2` package and can execute its functions!

If you want to learn more about `ggplot2`, browse over its amazing documentation. Scroll through all the different types of graphs and if something catches your eye that you think might be useful, check out the examples and make your own graph!

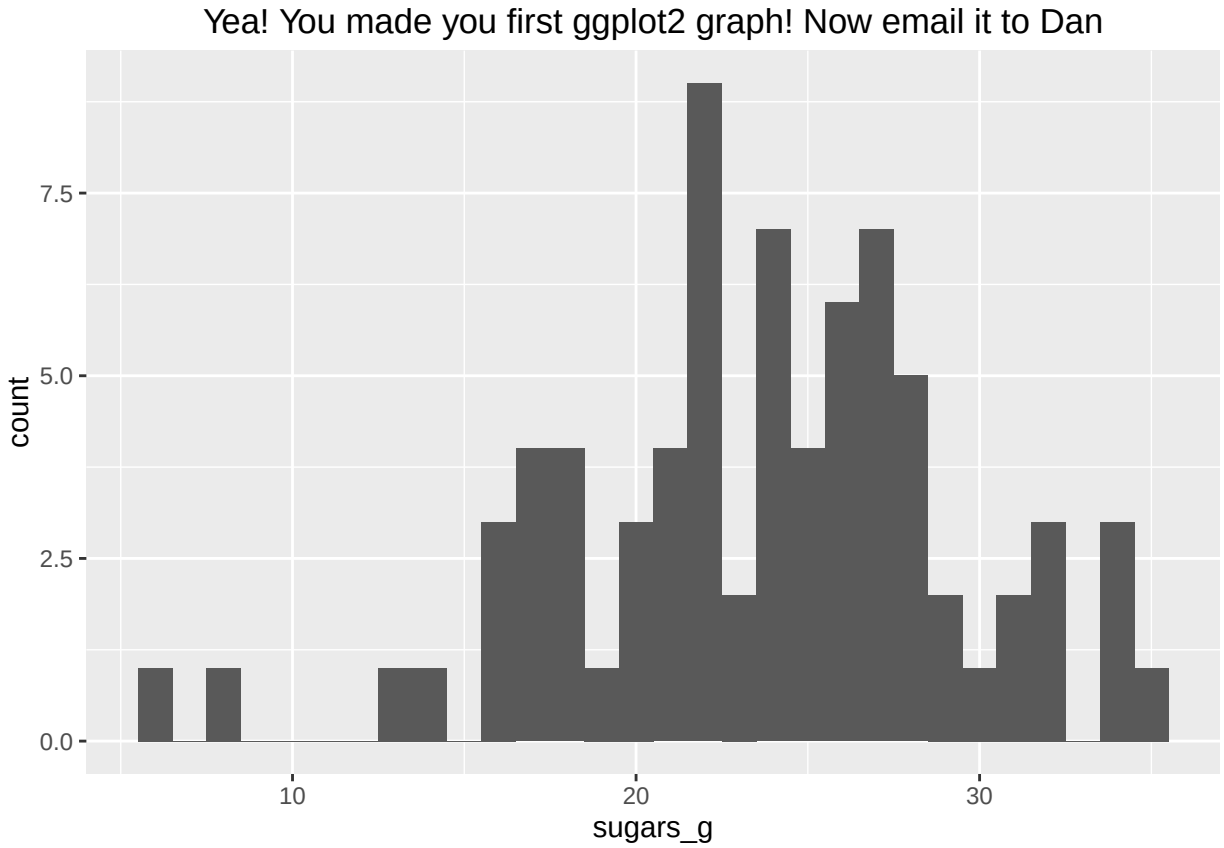
Let's make a graph!!!

```
p <- ggplot(data=data, aes(x=sugars_g))

p + geom_histogram() +

ggtitle(label="Yea! You made you first ggplot2 graph! Now email it to Dan")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



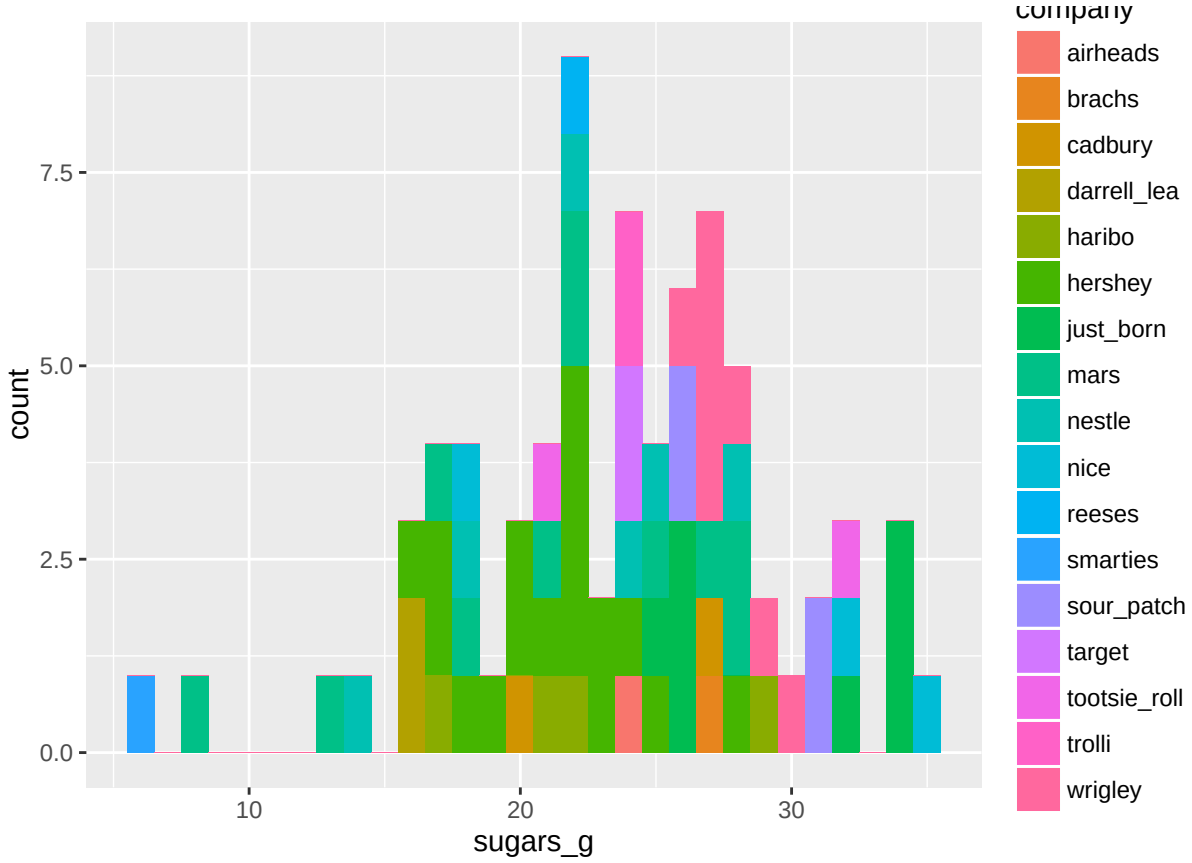
Let's play around a little more with `ggplot2` graphs and explore their functions. The above is a histogram, which provides an overview of the distribution of values for a trait, `sugars_g` in this case. The `ggplot()` function has a formulaic layout: we place into an object `p` the `ggplot()` function output, which specifies data and then variables used in the graph using another function, `aes()`, which stands for "aesthetics". In the above case, we specify that x will equal `sugars_g`. Once we create the object `p`, we can then add to it "geom"s" which are types of graphs. `geom_histogram()` makes a histogram. There are many other functions, like `ggtitle()` which adds a title. If you need help, try `?geom_histogram` or even better, look up the geom at http://docs.ggplot2.org/current/.

We can also add `fill` and `colour` attributes to the `aes()` function. For example, lets add `fill=company` to the graph we just made.

```
p <- ggplot(data=data, aes(x=sugars_g, fill=company))

p + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



One nice attribute of ggplots is their beautiful aesthetics. We can customize the color scheme used using the `scale_fill` and, if working with colour rather than fill, `scale_colour` functions. Let's color our histogram by class of candy, for which there are 7 classes. We can specify colours by name. Note that the order of levels will be assigned the order of colors you provide. The class order is alphabetical:

```
summary(data$class)
```

```
##     chocolate         gummi    jelly_bean     liquorice peanut_butter
##            26             9             5             6             7
##          sour         sugar
##             7            15
```

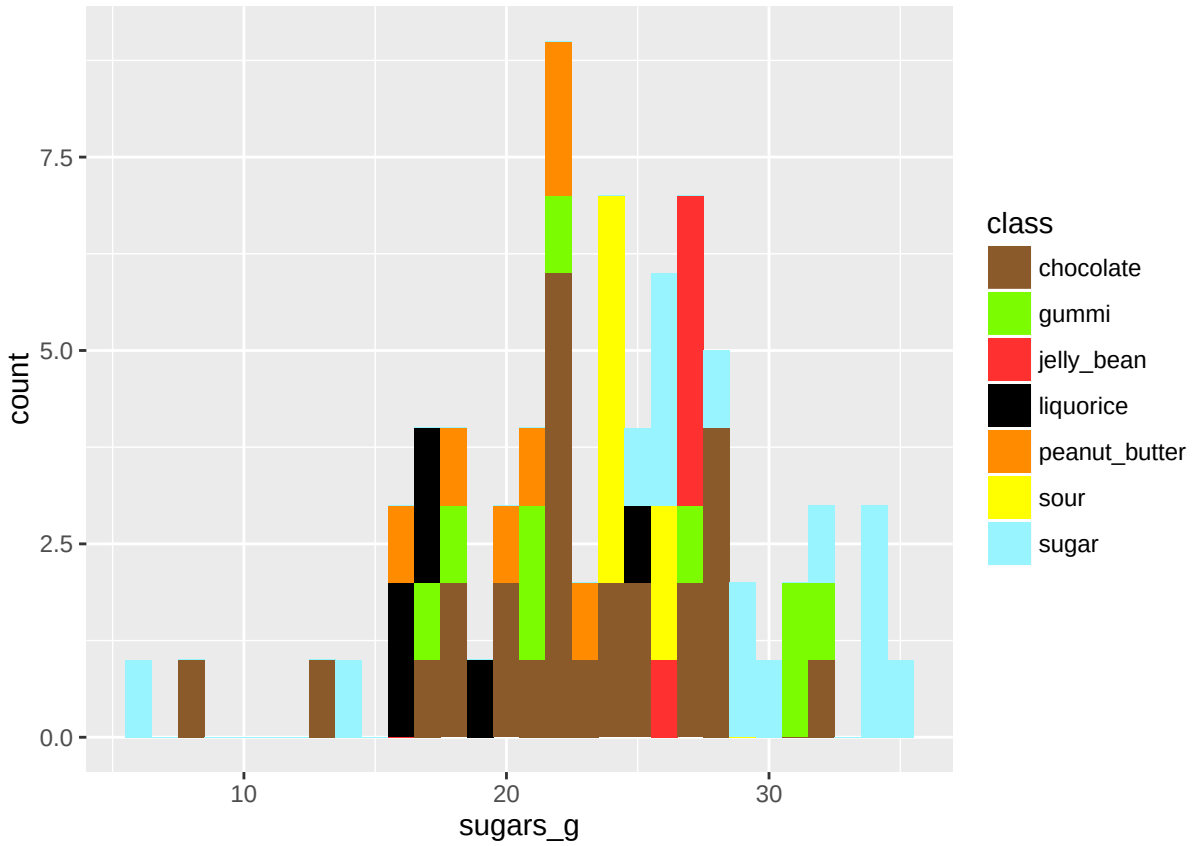
Let's try to assign appropriate colours for each class:

```
p <- ggplot(data=data, aes(x=sugars_g, fill=class))

p + geom_histogram() +

scale_fill_manual(values=c("tan4","lawngreen","firebrick1","black","darkorange","yellow","cadetblue1"))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



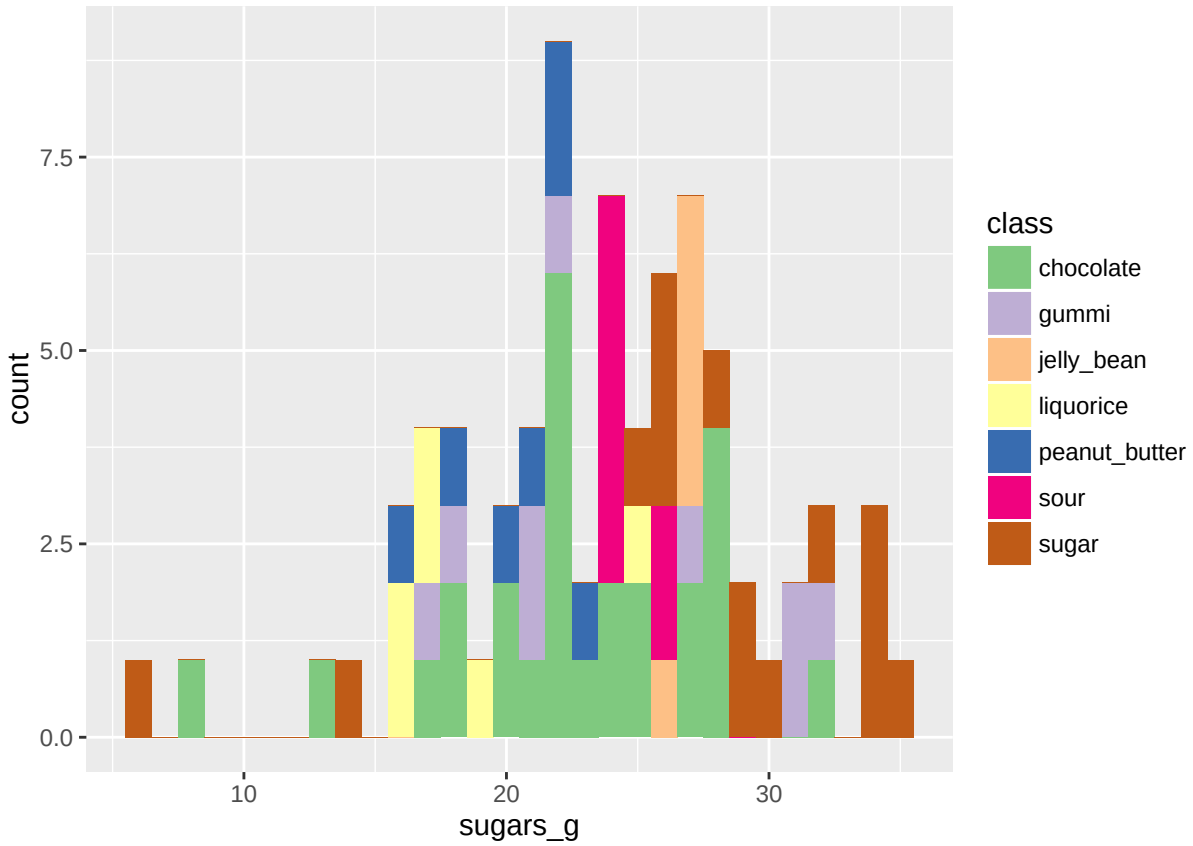
Or, if you would like very aesthetic colours, you should check out the color brewer website and the colour brewer function `scale_fill_brewer()`. Note that colour schemes at colour brewer are for sequential, divergent, and qualitative data types. Also, be sure to check out the `scale_fill_gradient` options at the ggplot2 website!

```
p <- ggplot(data=data, aes(x=sugars_g, fill=class))

p + geom_histogram() +

scale_fill_brewer(type="qual", palette=1)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



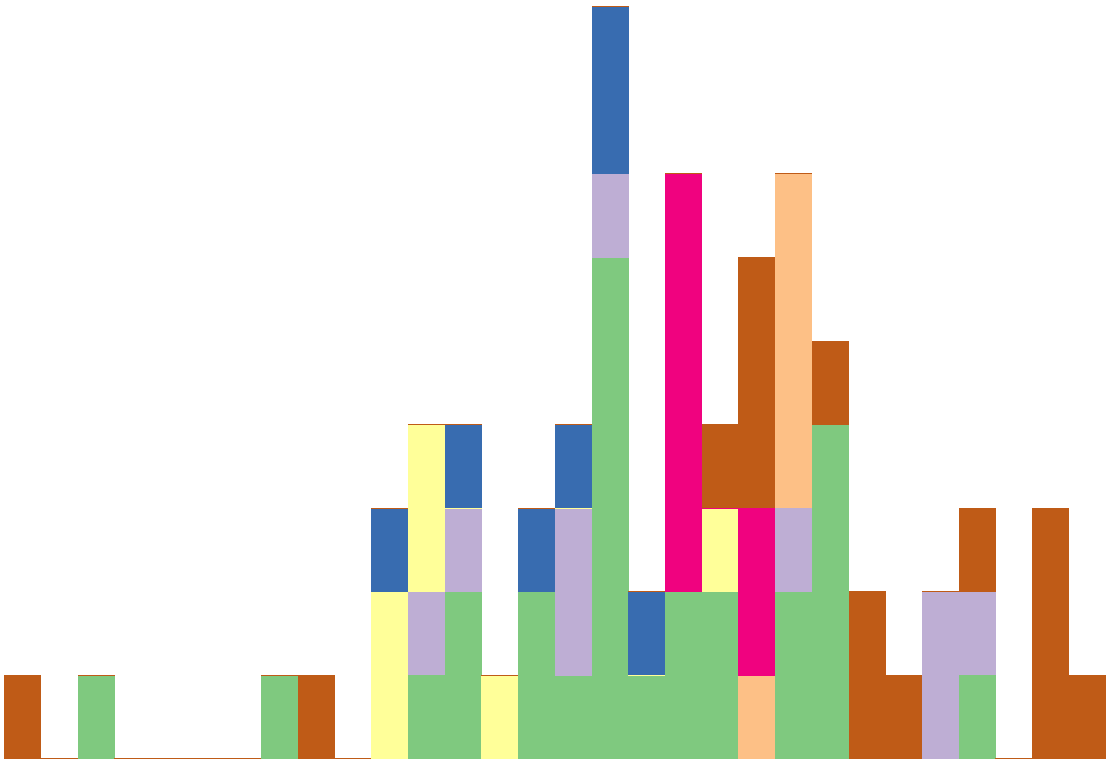
We can also manipulate other features of our graph using the `theme()` function. To see all the things we can change, maybe it's better to first take them all away:

```
p <- ggplot(data=data, aes(x=sugars_g, fill=class))

p + geom_histogram() +

scale_fill_brewer(type="qual", palette=1) +
theme(axis.line=element_blank(),
      axis.text.x=element_blank(),
      axis.text.y=element_blank(),
      axis.ticks=element_blank(),
      axis.title.x=element_blank(),
      axis.title.y=element_blank(),
      legend.position="none",
      panel.background=element_blank(),
      panel.border=element_blank(),
      panel.grid.major=element_blank(),
      panel.grid.minor=element_blank(),
      plot.background=element_blank())
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



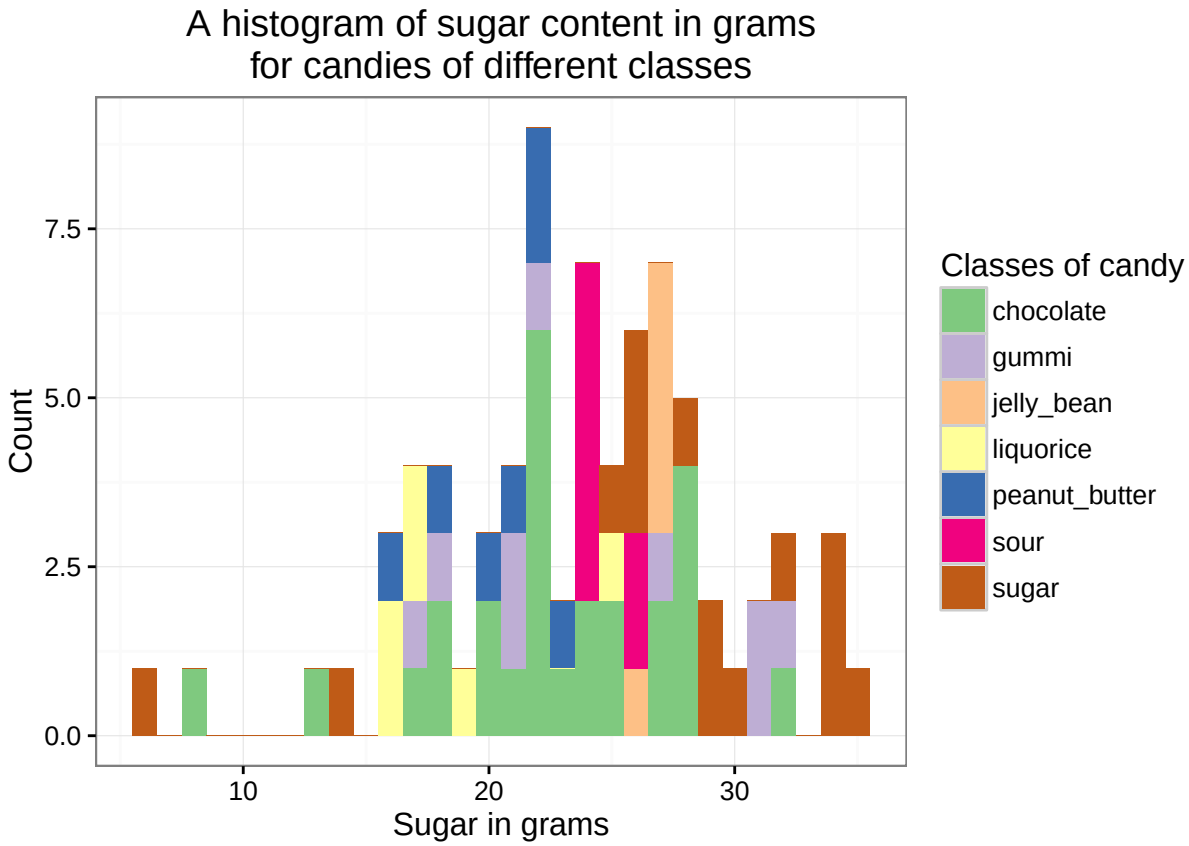
Nobody wants *nothing* in their graph! So let's just add titles for the axes using `xlab()` and `ylab()` functions and remove the grey background with `theme_bw()`. Note also the change of legend title and adding two lines to the title with `ggtitle` using `\n`, which creates a new line.

```
p <- ggplot(data=data, aes(x=sugars_g, fill=class))

p + geom_histogram() +

scale_fill_brewer(type="qual", palette=1, name="Classes of candy") +
xlab(label="Sugar in grams") +
ylab(label="Count") +
ggtitle("A histogram of sugar content in grams\nfor candies of different classes") +
theme_bw()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

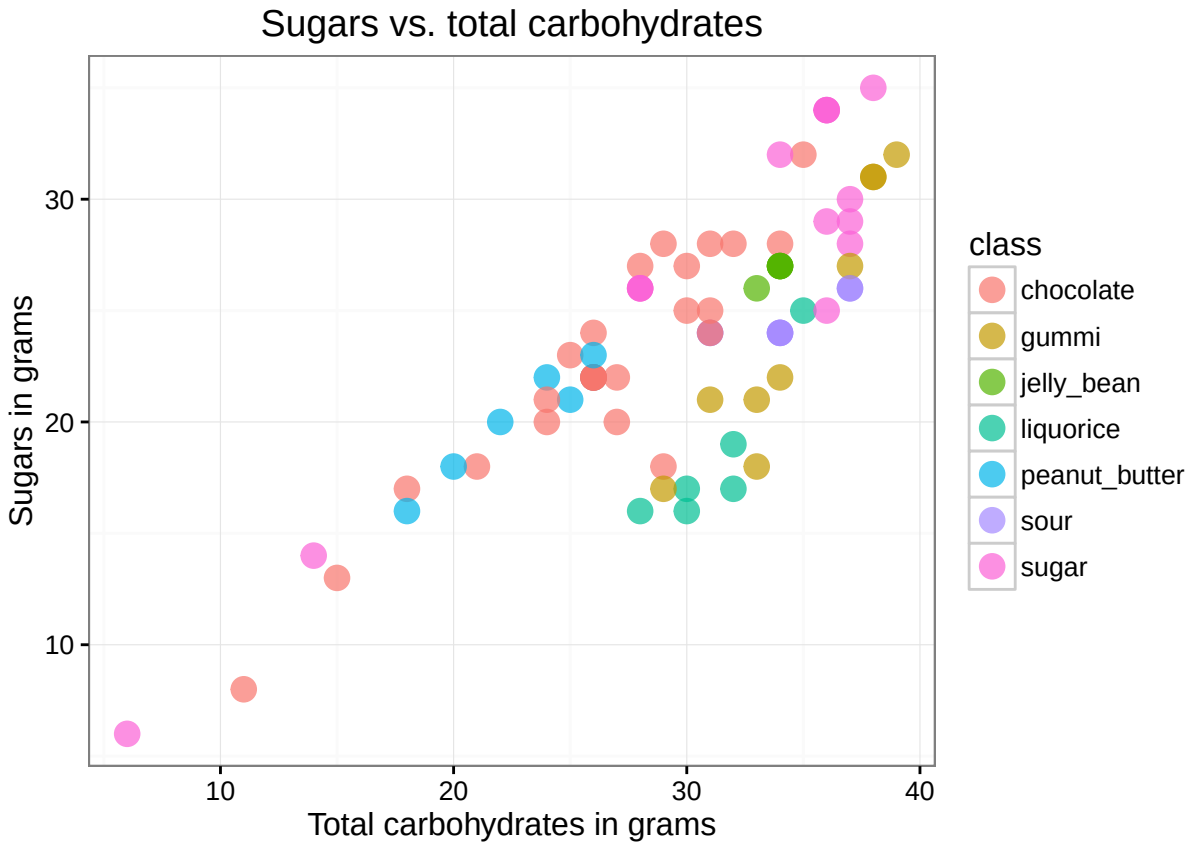


Let's try some other basic graphs, like a scatterplot. Note the use of `size` and `alpha` for the points (`alpha` is transparency, where "1" is opaque)

```
p <- ggplot(data=data, aes(x=total_carb_g, y=sugars_g, colour=class))

p + geom_point(size=4, alpha=0.7) +

scale_fill_brewer(type="qual", palette=1, name="Classes of candy") +
xlab(label="Total carbohydrates in grams") +
ylab(label="Sugars in grams") +
ggtitle("Sugars vs. total carbohydrates") +
theme_bw()
```



Wouldn't be nice to know the identity of each candy in this plot? And if two candies overlap, wouldn't it be nice to see both names? Let's install the `ggrepel` package and used the function `geom_repel_text`

```
install.packages("ggrepel")
```

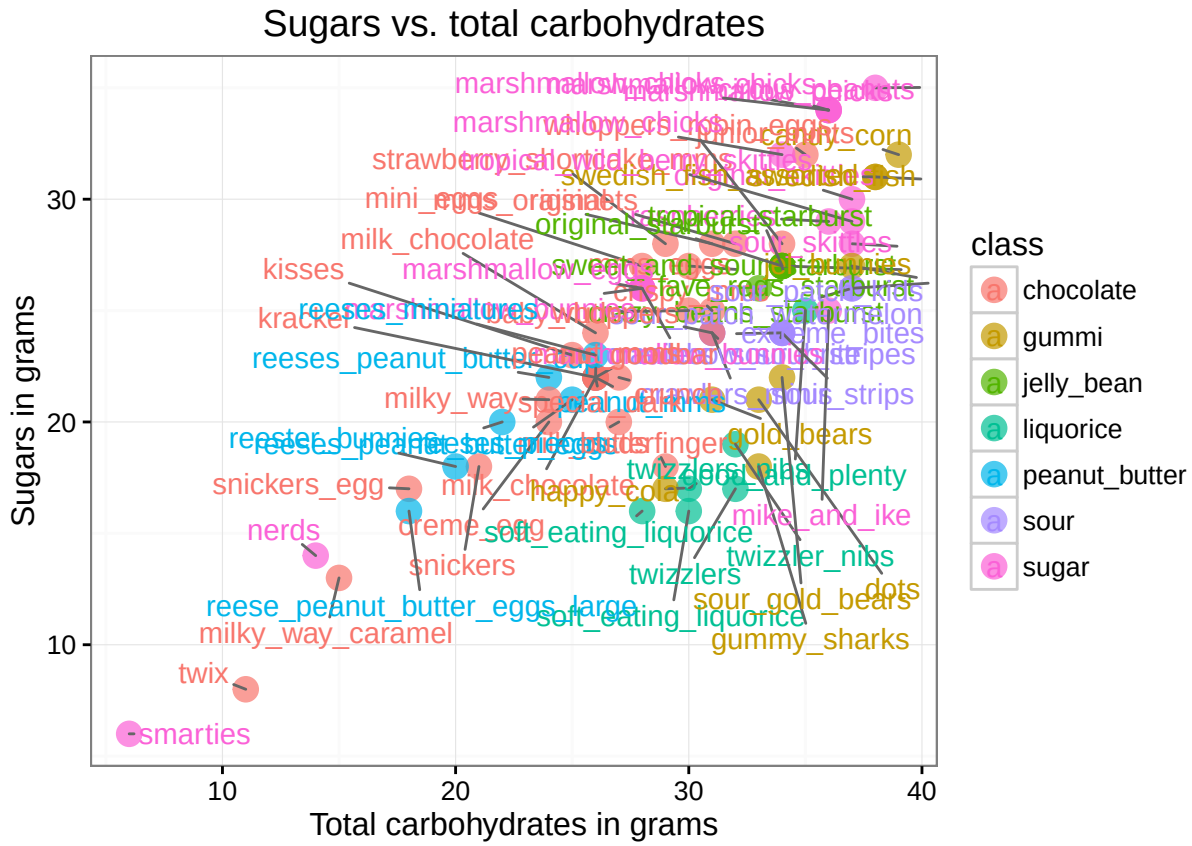
```
library(ggrepel)
```

And now lets make the graph using `geom_text_repel`:

```
p <- ggplot(data=data, aes(x=total_carb_g, y=sugars_g, colour=class))

p + geom_point(size=4, alpha=0.7) +

geom_text_repel(data=data, aes(x=total_carb_g, y=sugars_g, label=name)) +
scale_fill_brewer(type="qual", palette=1, name="Classes of candy") +
xlab(label="Total carbohydrates in grams") +
ylab(label="Sugars in grams") +
ggtitle("Sugars vs. total carbohydrates") +
theme_bw()
```

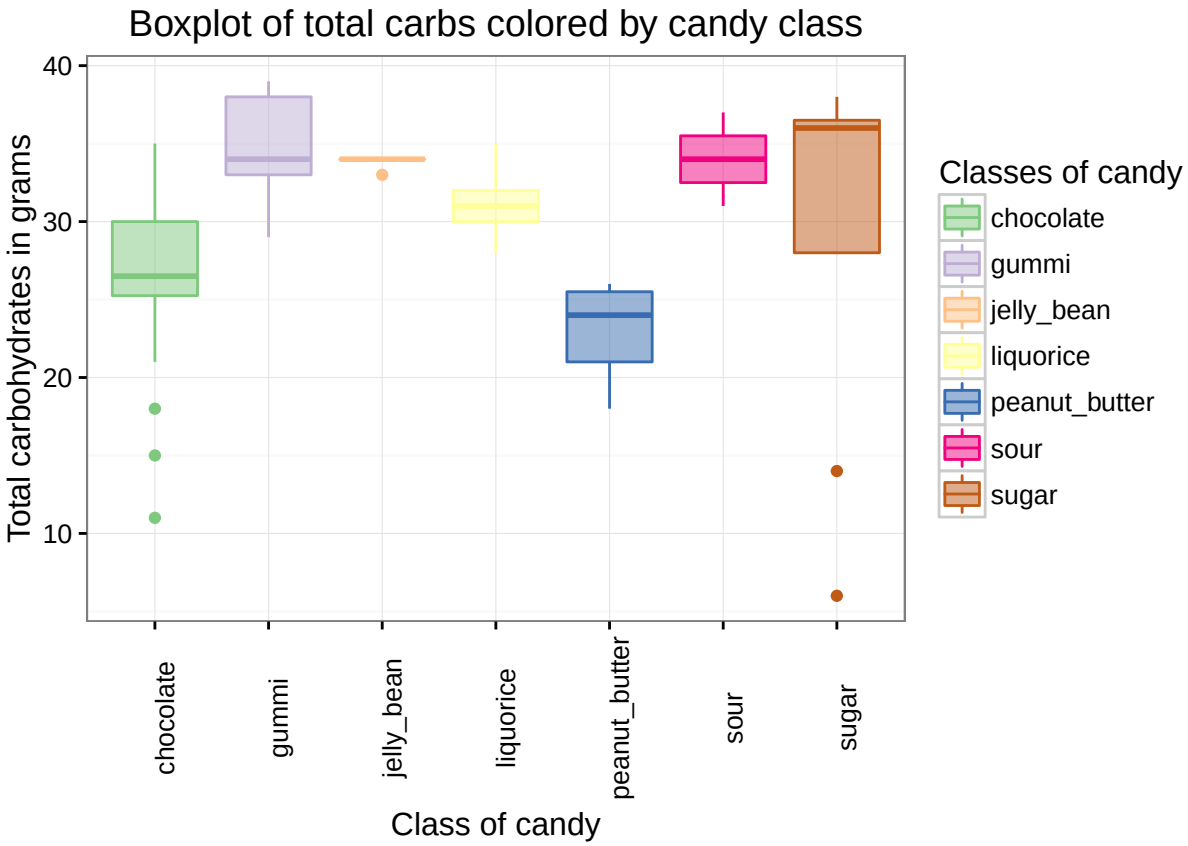


Just one last graph type, a boxplot!

```
p <- ggplot(data=data, aes(x=class, y=total_carb_g, colour=class, fill=class))

p + geom_boxplot(alpha=0.5) +

scale_fill_brewer(type="qual", palette=1, name="Classes of candy") +
scale_colour_brewer(type="qual", palette=1, name="Classes of candy") +
xlab(label="Class of candy") +
ylab(label="Total carbohydrates in grams") +
ggtitle("Boxplot of total carbs colored by candy class") +
theme_bw() +
theme(axis.text.x=element_text(angle=90))
```



OK, you should now be able to start making your own ggplot2 graphs!!!

Your homework assignment is to make *three graphs* of your choice using the "candy_nutrition.txt" dataset and email them to me. We will go over everybody's results together next lecture.

Explore! Be creative! Go the ggplot2 website and try out different types of graphs! Consider subsetting and looking at specific attributes of the data. Consider creating new variables (for example, normalize the nutrition variables by dividing them by `serving_size_g`). Look at distributions, scatterplots and correlations, and boxplots. Try some plots we didn't go over today and see what you come up with! Most importantly: use data visualization as a *tool*. Use it to know your data, the relationships and correlations between variables, the distributions of variables. All these are important prerequisites to data analysis. Use your graphs to *express* a hypothesis, idea, or conjecture you have about your data. Use data visualization to *investigate* patterns you wouldn't otherwise know about and use it to *challenge* your underlying assumptions about your data.

The ability to visualize data is at the heart of data exploration and *knowing* your data. The skills we learned today will be used to look at the data you'll be collecting in the next couple weeks: image analysis, morphometrics, colorimetrics, and multivariate statistics.