Data Science 101: Breaking Up with Excel
MSACL Connect 2020
Oct 17 - 20, 2020

Shannon Haymond (SHaymond@luriechildrens.org)
Lindsay Bazydlo (LAL2S@hscmail.mcc.virginia.edu)
David Lin (dalin@luriechildrens.org)
Daniel T. Holmes, MD (dtholmes@mail.ubc.ca)

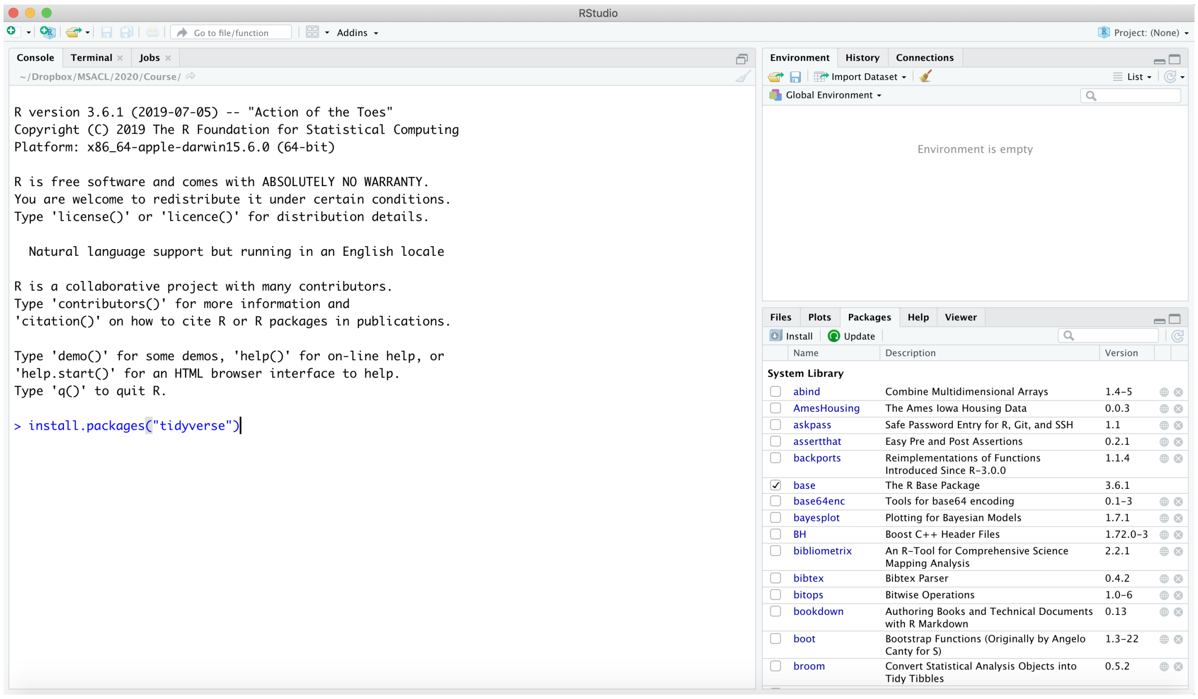
Contents

0.1 Preparation for the Course

This Short Course requires you to bring your own laptop. Plugs will be provided on the tables so that you do not have to rely on your battery life. You must install the necessary software before you arrive. It can take up to 30 mins and you do not want to be stuck doing this in the session because you get behind.

You need to install both the R programming Language and the R-Studio interface to R. While one can use R without R-Studio, we will all use it to make things uniform. The R programming language is available for download from many different places. Here are three places you can try, http://cran.stat.sfu.ca/; http://cran.wustl.edu/; http://lib.stat.cmu.edu/R/CRAN/. Choose the download that is appropriate for your laptop (whether Linux, Mac or Windows). Rstudio can be downloaded and installed free for personal use from: https://www.rstudio.com/products/RStudio/. You want the Open Source Edition.

Once you have done this, you need to install some extra software called "a package" that we will use extensively in the course. In our case, this package is called the "tidyverse". Open R Studio and type `install.packages("tidyverse")` in the console pane (left or bottom left) and then hit enter. You need to be connected to the internet and lots of stuff will install. If you get stuck, don't worry, email one of us and we can help you.



```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> install.packages("tidyverse")
```

	Name	Description	Version
	System Library		
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	AmesHousing	The Ames Iowa Housing Data	0.0.3
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.4
✓	base	The R Base Package	3.6.1
☐	base64enc	Tools for base64 encoding	0.1-3
☐	bayesplot	Plotting for Bayesian Models	1.7.1
☐	BH	Boost C++ Header Files	1.72.0-3
☐	bibliometrix	An R-Tool for Comprehensive Science Mapping Analysis	2.2.1
☐	bibtex	Bibtex Parser	0.4.2
☐	bitops	Bitwise Operations	1.0-6
☐	bookdown	Authoring Books and Technical Documents with R Markdown	0.13
☐	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-22
☐	broom	Convert Statistical Analysis Objects into Tidy Tibbles	0.5.2

Throughout this handout, we'll be showing you R code in a shaded box like this

```
R code goes here
```

and output will come be in an unshaded area after two "#" signs

```
## [1] "Here is some R output"
```

1 Basics and Data Types

1.1 What is R?

The purpose of this course is to show you that many of the data management and data analysis task you *wish* you could do in Excel are, in fact, very easy if you move to the statistical programming language R.

R is what is known as a "scripting language". This means that R does not build portable, stand–alone programs like commercial software we usually purchase. In contrast, R requires the R interpreter to be pre-installed on the computer before we can use any of its utilities. The same is true of other popular scripting languages like Python, PHP, Ruby and JavaScript. In principle this means that if you are using R on a company laptop and you don't have administrative privileges, you are going to need IT to help you install it.

R has been built for Windows, Linux and Mac, which means that R code you write can be run by other people on other systems *without any alterations to the code*, provided that they have installed the R interpreter. There are occasional exceptions to this rule but any code alterations are–most often–trivial.

The programs you write in R are saved as text or "ASCII" files that you save like any old document. You can use any text editor to write your program. This might include programs like Notepad or Notepad++ in the Windows environment or Gedit, eMacs, vi, Nano or Sublime Text in the Linux/Mac environments. In this course we will use a text editor in an environment specifically built for R called RStudio. This program has the benefit that it can both allow you to edit your text and run it without leaving the RStudio program. RStudio has many other convenient features that you will discover should you choose to continue this jou–R–ney. We could spend a lot of time on the background, but we do not have a great deal of time together so we should just jump right in.

1.2 Algebra

R can act as a calculator. It follows these basic rules or algebra.

Operation	Expression	R Code
Addition	$x + y$	`x + y`
Subtraction	$x - y$	`x - y`
Multiplication	$x \times y$	`x * y`
Division	$x \div y$	`x / y`
Exponents	x^y	`x^y` or `x**y`
Logarithm	$\log(x)$	`log(x)`
Exponential	e^x	`exp(x)`
Trig Functions	$\sin(x)$	`sin(x)`

There are some other useful built-in functions in R. These are:

- `abs(x)`
- `sqrt(x)`
- `floor(x)`
- `round(x, digits = n)`
- `signif(x, digits = n)`

1.2.1 Exercise

1. Experimentation is a great way to find out what a function does. Determine the square root of 64 with the `sqrt()` function and using the fact that the square root of 64 is really $64^{1/2}$
2. Try the following:
   - `ceiling(1.2)`
   - `ceiling(1.49)`
   - `ceiling(1)` What does ceiling do?
3. Try the following:
   - `round(1.4503,1)`
   - `round(exp(1),4)`
   - `round(pi,4)`
   - Now try typing `round(12314,-1)` and `round(12314,-2)`. What's happening here?
4. What does `trunc(5.99)` give you? What does `trunc(-3.43)` give you? How is `trunc()` different than `floor()`?

1.3 Comments

Note that anything we type on an R line that comes after the `#` sign is ignored by R. This is very useful for including comments in your code and helps to remind you (and others) what you were thinking. Because anything after the `#` is ignored, we can either put a comment on its own line:

```
# here is a comment
```

or after some code that is going to be executed

```
2 + 2  # we had better get the answer "4"
```

```
## [1] 4
```

1.4 Variables

Variables allow you to store your data so that it can be easily retrieved at a later time. For example, suppose you calculated the standard deviation of a data set and you had to use this result over and over again (and you did not want to type it our each time!). The best way to reuse this value is to store it in a variable.

```
my.sd <- 0.352
my.sd
```

```
## [1] 0.352
```

```
1.96 * my.sd
```

```
## [1] 0.68992
```

Notice that assigning the variable is performed with an "arrow" <- . The arrow can actually go the other way too but we don't do that all too much.

```
a <- 5 -> b
a
```

```
## [1] 5
```

```
b
```

```
## [1] 5
```

Getting rid of a variable is sometimes convenient, and the way to do this is with the `rm()` function.

```
rm(a)
a
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

To list all active variables type `ls()`. To remove all active variables type `rm(list = ls)`. This can also be achieved by clicking the broom icon in the **Environment** tab of the top right pane of Rstudio.

1.5 Data Types

We will encounter a variety of different data types during this course, and each has specific applications.

```
var.1 <- TRUE # a logical variable
class(var.1)
```

```
## [1] "logical"
```

```
var.2 <- 32.5 # a numeric variable
class(var.2)
```

```
## [1] "numeric"
```

```
var.3 <- "Michael" # a character variable
class(var.3)
```

```
## [1] "character"
```

Also, we can make integer variables and factor variables. R guesses which you want, and if it guesses wrong you may need to force it to assume the correct data type. You will see this later on. Don't assume that R has correctly read your mind on the data type you wanted.

```
var.4 <- 5
class(var.4)
```

```
## [1] "numeric"
```

```
var.4 <- as.integer(var.4)  # coerces the value to the class of integer
class(var.4)
```

```
## [1] "integer"
```

```
var.5 <- "4"
class(var.5)
```

```
## [1] "character"
```

```
7 * var.5                    # what happened?
```

```
## Error in 7 * var.5: non-numeric argument to binary operator
```

```
var.5 <- as.numeric(var.5)  # coerces the character into a numeric
class(var.5)
```

```
## [1] "numeric"
```

```
7 * var.5                    # ahhh no error now.
```

```
## [1] 28
```

1.6 Vectors

Vectors are a way to store multiple data points in a single variable. They are like a column from an Excel file and we will use them a lot. To define a vector you have to use the combine `c()` function to group the data.

```r
x <- c(5,3,6,4,7,2,6) # defines the variable x
class(x)              # what class will this be?
```

```
## [1] "numeric"
```

Importantly, every member of the vector must be of the same type and if they are not, R will choose a data type that will be compatible with all the elements.

```r
y <- c(5,3,6,4,7,2,"Hi There")
class(y)              # What happened?
```

```
## [1] "character"
```

```r
y
```

```
## [1] "5"        "3"        "6"        "4"        "7"        "2"        "Hi There"
```

Let's explore some algebra:

```r
x + 2   # What does it do?
```

```
## [1] 7 5 8 6 9 4 8
```

```r
x + x       # How is this different from x+2
```

```
## [1] 10  6 12  8 14  4 12
```

```r
x / 2       # What does this do?
```

```
## [1] 2.5 1.5 3.0 2.0 3.5 1.0 3.0
```

```r
x * x       # What does this do?
```

```
## [1] 25  9 36 16 49  4 36
```

```r
x / x       # What does this tell you about dividing vectors?
```

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

```r
sum(x)  # What does this calculate?
```

```
## [1] 33
```

```r
mean(x) # And this?
```

```
## [1] 4.714286
```

```r
sd(x)       # And this?
```

```
## [1] 1.799471
```

```r
length(x)   # And this? This is really useful.
```

```
## [1] 7
```

```r
c(x, x)
```

```
##  [1] 5 3 6 4 7 2 6 5 3 6 4 7 2 6
```

```r
rep(x,3)
```

```
##  [1] 5 3 6 4 7 2 6 5 3 6 4 7 2 6 5 3 6 4 7 2 6
```

What if we wanted to find out an individual value from x?

```r
x[2]  # Note the SQUARE brackets.
```

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

```r
# OK, makes sense
```

What if we wanted to know which value of x was 6, if any?

```r
which(x  ==  6)
```

```
## [1] 3 7
```

CAREFUL "==" is used to compare two values to see if they are equal; don't confuse this with and "=" or "<-", which are for assignment.

```r
x[which(x  ==  6)]    # should give us back a number of 6's of course.
```

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

```r
x  ==  6    #this gives us a logical vector answering the question "Is the value 6?"
```

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

```r
y <- x  ==  6
y
```

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

```r
x[y]    #this should just give us back the 6's
```

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

```r
x[x  ==  6]
```

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

1.6.1 Exercise

1. Define a variable, `days`, which contains all the days of the week.
2. Now imagine that you move to a planet where there are an 8th and 9th day of the week, called "Chillday" and "Sleepday". Can you use `c()` to add these days to your days variable so that you do not have to retype everything?

1.6.2 Exercise

1. Type `1:10` and see what happens.
2. Now type `x <- 1:10`. What did this do? Find out by asking R what `x` is.
3. Now type `x <- 5:10`. What did this do?
4. Type ?seq in the console to find out what the `seq()` function does. Can you replicate the results of `x <- 5:10` with `seq()`? Why is `seq()` more flexible?

1.6.3 Exercise

1. Write an expression in R that will always return the last value in a vector named `z`.
2. Test your expression's success. We'll start by generating a sequence of letters of the alphabet that terminates randomly.

```
z <- letters[1:round(runif(1,0,26),0)]  # don't worry about why this works right now.
z
```

```
## [1] "a"
```

```
z[length(z)]                            # check what the last value is
```

```
## [1] "a"
```

```
Now apply your expression to identify the last letter in the sequence you generated.
```

2 Matrices, Dataframes and Lists

2.1 Matrices

A matrix is a 2D analogue of the vector. Matrices may not seem all that important but there are certain R statistical functions requiring their use and so you are certainly going to encounter them.

```
days <- 1:28   # Generates a sequence of 28 integers
days           # confirm that you have done what you thought
```

```
##  [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 25
## [26] 26 27 28
```

```
# Note: another way to do this is with days <- seq(from = 1, to = 28, by = 1)
```

Now, let's convert this to something that looks more like the calendar layout of February.

```
matrix(days, nrow = 4, ncol = 7)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    5    9   13   17   21   25
## [2,]    2    6   10   14   18   22   26
## [3,]    3    7   11   15   19   23   27
## [4,]    4    8   12   16   20   24   28
```

```
# Huh? What happened?
#
# Let's try again
matrix(days, nrow = 4, ncol = 7, byrow = TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4    5    6    7
## [2,]    8    9   10   11   12   13   14
## [3,]   15   16   17   18   19   20   21
## [4,]   22   23   24   25   26   27   28
```

```
# Ahhh
```

2.1.1 Exercise

- February 2016 started on a Monday but was a leap-year. Make a matrix for February 2016 and fill days that are not in February with NA. We'll talk later about different places that NA pops up, but for now just know that it's R's way of denoting data that is "**N**ot **A**vailable". Hint for this exercise: start by using the `c()` command to prepend the appropriate number of NA's to the beginning and the end of variable days. Assign the name feb.2016 to your matrix.

To get a specific value of your matrix, you simply refer to the row and column.

```
feb.2016[3, 4] #Gives entry from row 3 column 4.  Should return 17 as the result.
```

```
## [1] 17
```

```
# To get all the values in the column, you simply omit the row number.
feb.2016[ ,6] # Gives us what we need: all the Saturdays.
```

```
## [1]  5 12 19 26 NA
```

```
# ...and the same trick works for columns
feb.2016[3, ] # Gives us week 3.
```

```
## [1] 14 15 16 17 18 19 20
```

2.1.2 Exercise

The raw speed data from your latest bike ride can be exported from your Garmin and brought into R. The ride happens to be exactly 38 mins duration and data is sampled every second. This means that there will be 2280 measurements in total. Execute this code to import the data from a file.

```
library(tidyverse)
speed.vec <- scan("Data_Files/speed.txt") #reads the data into a tibble
```

- Now, convert this data into a matrix called `speed.mat` of 38 columns where each column is the speed data of one minute.

2.2 Tibbles

A tibble (a tidyverse answer of the base-R dataframe) is the closest thing you are going to get to Excel-like storage of your data. When you read your data into R from a file (if you have saved it from Excel in a standard format), it will become a tibble.

Let's convert our February 2016 matrix to a tibble, and then we will look at how to import Excel-like data to a tibble. We will spend some time working with tibbles because they are going to be our bread and butter.

First, we can convert a matrix to a tibble:

```
feb.2016 <- matrix(c(NA,1:29,NA,NA,NA,NA,NA), nrow = 5,ncol = 7, byrow = TRUE)
feb.2016 <- as_tibble(feb.2016)
feb.2016
```

```
## # A tibble: 5 x 7
##      V1    V2    V3    V4    V5    V6    V7
##   <int> <int> <int> <int> <int> <int> <int>
## 1    NA     1     2     3     4     5     6
## 2     7     8     9    10    11    12    13
## 3    14    15    16    17    18    19    20
## 4    21    22    23    24    25    26    27
## 5    28    29    NA    NA    NA    NA    NA
```

```
# this has uninformative column names, but we can name the columns as we can in Excel
names(feb.2016) <- c("Sun","Mon","Tue","Wed","Thu","Fri","Sat")
feb.2016
```

```
## # A tibble: 5 x 7
##     Sun   Mon   Tue   Wed   Thu   Fri   Sat
##   <int> <int> <int> <int> <int> <int> <int>
## 1    NA     1     2     3     4     5     6
## 2     7     8     9    10    11    12    13
## 3    14    15    16    17    18    19    20
## 4    21    22    23    24    25    26    27
## 5    28    29    NA    NA    NA    NA    NA
```

```
# much better
```

Second, we can build a tibble from vectors as follows.

```
odd.nums <- c(1,3,5,7)
even.nums <- c(2,4,6,8)
numbers <- tibble(odds = odd.nums, evens = even.nums)
numbers
```

```
## # A tibble: 4 x 2
##    odds evens
##   <dbl> <dbl>
## 1     1     2
## 2     3     4
## 3     5     6
## 4     7     8
```

```
# alternatively, numbers <- tibble(rbind(odd.nums, even.nums))
```

So, the columns of a tibble could be the results of different data fields in your study, name, health number, sex, age, date of last visit, blood pressure, medications, creatinine, hemoglobin etc. Whatever you could store in an Excel sheet could be in a tibble. We often want to pull out specific columns from a tibble – usually to perform statistical tests.

Pulling out a column is easy. We can do it by the column name or we can do it by the column number.

```
# by column name
feb.2016$Tue # gives all the Tuesdays in Feb 2016
# or
feb.2016[["Tue"]]

# by column number
feb.2016[[3]] #result as vector
#or
feb.2016[,3] #result as tibble

# You can pull out data from an individual cell.
feb.2016[2,3] # keeps it as a tibble, oddly

# Or you can do the same with the $ approach because feb.2016$Tue is a vector
feb.2016$Tue[2]
# or
```

```r
feb.2016[["Tue"]][2]
# or
feb.2016[[3]][2]
```

If you need to pull more than one column out, you can do so by numbers or the names:

```r
# by number
feb.2016[,3:4]
```

```
## # A tibble: 5 x 2
##     Tue   Wed
##   <int> <int>
## 1     2     3
## 2     9    10
## 3    16    17
## 4    23    24
## 5    NA    NA
```

```r
#by name - this is particularly convenient when dealing with large dataframes
feb.2016[,c("Tue","Wed")]
```

```
## # A tibble: 5 x 2
##     Tue   Wed
##   <int> <int>
## 1     2     3
## 2     9    10
## 3    16    17
## 4    23    24
## 5    NA    NA
```

```r
#or
select(feb.2016,c("Tue","Wed"))
```

```
## # A tibble: 5 x 2
##     Tue   Wed
##   <int> <int>
## 1     2     3
## 2     9    10
## 3    16    17
## 4    23    24
## 5    NA    NA
```

Let's go back to that bike speed data we have above. We will start by turning the data, stored previously as matrix called `speed.mat`, into a dataframe.

```r
speed.mat <-  matrix(speed.vec, nrow = 60, ncol = 38, byrow = FALSE)
speed.tb <- as_tibble(speed.mat)
names(speed.tb) <- paste0("min_",1:38)
#head(speed.tb)
```

2.2.1 Exercise

- Using the speed.tb tibble, find the average speed of the 20th minute of your ride.

Note that if you try to find the average by row instead of a column, you will run into a problem that illustrates something about tibbles, namely that if you extract a row, you can't just simply do algebra on it because there is generally no guarantee that different columns of a data frame will all be numbers.

```r
min.start.avg <- mean(speed.tb[1,]) #does not work
min.start.avg <- mean(speed.mat[1,]) #does work
```

Here are some things you can do with data in a tibble. You can ask R to tell you things about your data points. For example, you could ask R for some descriptive statistics of the last minute of your ride:

```r
mean(speed.tb$min_38)
```

```
## [1] 25.15944
```

```r
median(speed.tb$min_38)
```

```
## [1] 23.103
```

```r
sd(speed.tb$min_38)
```

```
## [1] 4.482852
```

```r
summary(speed.tb$min_38) # a statistical summary
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   21.50   22.52   23.10   25.16   26.81   38.38
```

```r
quantile(speed.tb$min_38 ,probs = c(0.25,0.50, 0.75)) # specific quantiles
```

```
##     25%     50%     75%
## 22.5216 23.1030 26.8137
```

But this does not work when you try to take the grand mean:

```r
mean(speed.tb)
```

```
## [1] NA
```

But this approach does work for matrices

```r
mean(speed.mat)
```

```
## [1] 30.9961
```

Remember, we'll say more later about "NA", but for now notice that R gives us a nasty warning message. The reason that the `mean` function does not operate on data frames in this case because the data in a data frame is often not numeric.

You can also ask R to tell you which values have certain properties.

```r
# did you dip below 25 kph in the last minute of your ride?
speed.tb$min_38 < 25
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [49]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

But the code authors acknowledge that it is a pain to have to write out those `$` signs all the time, so try this:

```r
attach(speed.tb)
min_1 # now min_1 is a local variable
```

```
##  [1] 14.1984 17.2188 18.8964 20.0448 21.3732 22.4172 22.9752 23.2992 23.3424
## [10] 23.1228 22.6908 22.2876 21.9456 21.6648 21.3768 21.1140 20.9412 20.7648
## [19] 20.6244 20.5056 20.1960 19.9080 19.3608 18.9612 18.8460 18.8424 18.9324
## [28] 19.1232 19.2600 19.5696 20.3868 20.0412 19.2276 18.0828 16.9488 16.2684
## [37] 15.5304 15.0048 14.7204 14.3928 14.3172 14.5800 15.1416 15.1884 12.4380
## [46] 12.3840 12.8736 13.3128 13.6764 14.0724 14.4720 14.8104 15.1596 15.4800
## [55] 15.6384 15.8400 16.0596 16.2900 16.4952 16.6860
```

```
min_2 # ...and so is min_2!
```

```
##  [1] 17.0064 17.4456 17.8488 18.1260 18.3996 18.6264 18.9576 19.4436 18.9756
## [10] 19.3752 19.8648 20.4372 21.0564 21.6972 22.6152 23.4756 24.1200 24.5772
## [19] 24.8796 25.1820 25.5852 25.9020 26.1864 26.4996 26.8560 27.2232 27.6732
## [28] 28.2204 28.7856 29.2536 29.6244 29.9376 30.2544 30.5064 30.6576 30.8412
## [37] 31.0320 31.2156 31.3128 31.3488 31.3164 31.3200 31.3452 31.4280 31.4712
## [46] 31.3632 31.3020 31.2048 31.0608 30.9024 30.7404 30.5892 30.3804 30.0780
## [55] 29.5488 28.7388 28.0620 28.0980 28.4004 28.5876
```

Note that if you alter the attached variables, they *do not* alter the original data frame. To remove these variables we type:

```
detach(speed.tb)
```

There are lots of reasons not to use `attach`, but it is good for quick and dirty things.

2.2.2 Exercise

- We are going to read in a little more data from your bike ride. This time the data will contain time, cadence, heart rate, distance, speed and power.

```
ride.data <- read_csv("Data_Files/ride_file.csv")
head(ride.data)
```

```
## # A tibble: 6 x 6
##    secs   cad    hr      km   kph watts
##   <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl>
## 1     1     1    80 0.00347  14.2   360
## 2     2    75    81 0.00788  17.2   360
## 3     3    77    83 0.0129   18.9   255
## 4     4    91    84 0.0182   20.0   245
## 5     5    99    86 0.024    21.4   334
## 6     6   101    87 0.0302   22.4   249
```

Isolate the data from your ride for which your speed was over 65 kph.

What you have just done is called subsetting your tibble, and it is a very frequent task that we are going to be doing more of in the next hour. Because it is such a common task (e.g. pulling out all the subjects less than 40 years, pulling out all the male subjects, excluding an outlier), there is a specific command for it:

```
filter(ride.data, kph > 65)
```

```
## # A tibble: 8 x 6
##    secs   cad    hr    km   kph watts
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  1387    96   129  10.8  65.7    88
## 2  1388    96   129  10.8  66.8    88
## 3  1389    96   129  10.8  67.2    95
## 4  1390    96   129  10.8  66.4    96
## 5  1450   102   134  11.8  66.9   131
## 6  1451   102   135  11.8  67.7   131
## 7  1452   103   134  11.8  67.1   131
## 8  1453   103   134  11.9  65.1   123
```

You can build other logical constraints to isolate data of interest in vectors, matrices and data frames by using `&` (AND), `|` (OR), and `!` (NOT). For example:

```
x <- c(3,4,2,7,5,8,9,5,-2,34,15,7)
# values of x greater than 2 and less than 7. Use the & for AND.
x[x > 2 & x < 7]
```

```
## [1] 3 4 5 5
```

```
# values of x not less than 15
x[!(x < 15)]
```

```
## [1] 34 15
```

```
# which is the same as
x[x >= 15]
```

```
## [1] 34 15
```

```
# values of x less than 3 or greater than 5. Use the | for OR.
x[x < 3 | x > 5]
```

```
## [1]  2  7  8  9 -2 34 15  7
```

```
# all values except the first three: use a minus sign
x[-(1:3)]
```

```
## [1]  7  5  8  9  5 -2 34 15  7
```

and with the tibble we might ask for rows your speed was under 20 km/h but your power was over 350 watts. This would find sections where you were likely riding uphill.

```
filter(ride.data, kph < 15 & watts > 350)
```

```
## # A tibble: 11 x 6
##     secs   cad    hr      km   kph watts
##    <dbl> <dbl> <dbl>   <dbl> <dbl> <dbl>
##  1     1     1    80  0.00347 14.2   360
##  2  1282    98   162 10.1      6.88   386
##  3  1291    98   163 10.1      9.43   359
##  4  1292    98   163 10.1      9.66   356
##  5  1293    98   163 10.1      9.84   354
##  6  1294    98   164 10.1     10.1    362
##  7  1295    98   164 10.1     10.4    360
##  8  1296   100   164 10.1     10.7    360
##  9  1297   100   164 10.1     11.1    353
## 10  1298   102   165 10.1     11.5    353
## 11  1302   108   166 10.1     13.6    353
```

(Note that – just like algebra – we can use parenthesis to make absolutely clear what order we are using to evaluate the expressions)

2.3 Lists

Lists are another way of storing data in a conveniently accessible way. Usually they are used for data types that are different in structure (or store different types of data) but are related because they are part of the same analysis. For example, R frequently provides output of statistical analysis in the form of a list. Suppose you had a vector, a data frame and a matrix:

```
a <- c("Doe","a","deer")
b <- tibble(lyrics1 = c("a","female","deer"), lyrics2 = c("Ray","a","drop"))
d <- matrix(seq(from = 0, to = 100, length.out = 25), nrow = 5, ncol = 5)
```

But they were all related to some specific problem and you wanted to bundle them together in another variable. This is where you would use a list.

```
my.list <- list(a,b,d)
my.list
```

```
## [[1]]
## [1] "Doe"  "a"    "deer"
##
## [[2]]
## # A tibble: 3 x 2
##   lyrics1 lyrics2
##   <chr>   <chr>
## 1 a       Ray
## 2 female  a
## 3 deer    drop
##
## [[3]]
##           [,1]     [,2]     [,3]     [,4]      [,5]
## [1,]  0.000000 20.83333 41.66667 62.50000  83.33333
## [2,]  4.166667 25.00000 45.83333 66.66667  87.50000
## [3,]  8.333333 29.16667 50.00000 70.83333  91.66667
## [4,] 12.500000 33.33333 54.16667 75.00000  95.83333
## [5,] 16.666667 37.50000 58.33333 79.16667 100.00000
```

Components of the list are addressed using a double bracket notation. For example:

```
my.list[[2]] # gives the tibble.
```

```
## # A tibble: 3 x 2
##   lyrics1 lyrics2
##   <chr>   <chr>
## 1 a       Ray
## 2 female  a
## 3 deer    drop
```

If you happen to give the components of the list names, you can address with the $ notation:

```
my.list <- list(one = a,two = b,three = d)
my.list
```

```
## $one
## [1] "Doe"  "a"    "deer"
##
## $two
## # A tibble: 3 x 2
##   lyrics1 lyrics2
##   <chr>   <chr>
## 1 a       Ray
## 2 female  a
## 3 deer    drop
##
## $three
##           [,1]     [,2]     [,3]     [,4]      [,5]
## [1,]  0.000000 20.83333 41.66667 62.50000  83.33333
## [2,]  4.166667 25.00000 45.83333 66.66667  87.50000
## [3,]  8.333333 29.16667 50.00000 70.83333  91.66667
## [4,] 12.500000 33.33333 54.16667 75.00000  95.83333
## [5,] 16.666667 37.50000 58.33333 79.16667 100.00000
```

```
my.list$two # gives the same dataframe data
```

```
## # A tibble: 3 x 2
##   lyrics1 lyrics2
##   <chr>   <chr>
## 1 a       Ray
## 2 female  a
## 3 deer    drop
```

R very often uses lists for statistical reports. For example, the `lm()` function is used to generate a regression report by R. You can assign the output of `lm()` to a variable and this variable contains a very useful list.

```
x <- 1:10
y <- 2 * x + rnorm(10,0,1)  # generates some fake data
reg <- lm(y ~ x)
str(reg)                    # shows what variables that reg stores.

# ...but they generate a lot of output, so try it on your own

reg$residuals
reg$fitted.values
```

```
# OK, fine--we'll show one...
reg$coefficients
```

```
## (Intercept)           x
##  -0.5796645   2.1490702
```

As a final trivia point, tibbles are really just lists comprised columns, each of which must be an equal-length vector.

3 Reading in Data and Basic Sanity Checking

3.1 The Working Directory

First we need to learn how to read in Excel data. While .xls and .xlsx files can be read in in their native formats (we will show this later), usually we use the simpler approach of dealing with a .csv file. You can save your Excel spreadsheet as a .csv file by clicking “Save As” and selecting “Comma Separated Values”.

Before we do this, however, we need to start by telling R where it should consider its current working directory. You can check the current working directory by typing `getwd()`. The directory will display. You are going to need to get R to the place where you want to go. You can do this with RStudio by choosing **Session -> Set Working Directory -> Choose Directory** and then navigate where you want to go. However, if this navigation process has to be part of the code in the R Script (which it frequently does), you can get code to do the same thing.

To do the same thing in code, you need to know the *path* that you want to send R to.

- In Windows, you will right-click and then examine Properties of the folder or file.
- In Mac OS X you will use CTRL-click on the folder or file and select "Get Info" and look at the "Where" row of the General tab. You can also drag a folder into the terminal window and the full path of the directory will appear on the command line.
- In Linux you will know what to do because Linux people are never confused by navigating directories from the command line.

```
getwd() # where am I now
setwd("/Users/danielholmes/Dropbox/MSACL/2020/Course")
getwd() # see that things have changed
```

** Caution to Windows users: ** * Windows uses backslash (\) not forwardslash (/) between subdirectories. The \ is a special character in R (and many other languages) called an "escape" and it is used to denote special characters like tabs and carriage returns. To overcome this you can either turn all your \ to \\ (an escaped backslash) or you can turn all your \ to /, to be like the rest of the world.

3.2 Reading a csv File

The command `list.files()` or this fancy-pants alternative, `fs::dir_ls()`, will show you all files in the directory you are working in. To read in the data we first need to make sure that R knows what directory to read from. As mentioned earlier, we can set the working directory using the R Studio Gui under Session > Set Working Directory > Choose Directory. Alternatively we can use the `set.wd()` command as previously discussed.

from the csv file we type:

```
read_csv("Data_Files/intersalt.csv")
```

The `read_csv()` function has lots of options. Type ?read_csv to display them.

- `delim = ","` tells R that a comma is what separates different columns.
- `delim = "\t"` is another common option for tab delimited files
- `trim_ws = TRUE` is an option to prevent something like `"M"` and `"M "` and `" M "` from being considered three different things.

If you have a file from a collaborator in Quebec or Europe, you may use a comma as decimal place in your work. This means that you are not going to be using vanilla `read_csv()`. First, you will need to be sure that you do not use a comma as your column separator when you save a csv file. You will need to use something else. Most often this is a semicolon. R has a built-in function for those who use commas as decimals. This is `read_csv2()`, which is the same as `read_csv()` except that the default parameter `dec = "."` is changed to `dec = ","` and `sep = ","` is changed to `sep = ";"`.

We should store this data in a variable so that we can do some analysis of it. Since it is data from the intersalt study we can name the variable intersalt. The file contains blood pressure (bp), sodium excretion (na) and nationality data. We have deliberately inserted a bad data point for instructional purposes.

```
intersalt <- read_csv("Data_Files/intersalt.csv")
```

The `head()` function is a quick way to look at what the data looks like in a snippet.

```
head(intersalt) # gives us the first 6 lines
```

```
## # A tibble: 6 x 4
##       b bp       na country
##   <dbl> <chr> <dbl> <chr>
## 1 0.512 72    149.  Argentina
## 2 0.226 78.2  133   Belgium
## 3 0.316 73.9  143.  Belgium
## 4 0.042 61.7    5.8 Brazil
## 5 0.086 61.4    0.2 Brazil
## 6 0.265 73.4  149.  Canada
```

```
head(intersalt,10) #gives us the first 10 lines
```

```
## # A tibble: 10 x 4
##        b bp       na country
##    <dbl> <chr> <dbl> <chr>
##  1 0.512 72    149.  Argentina
##  2 0.226 78.2  133   Belgium
##  3 0.316 73.9  143.  Belgium
##  4 0.042 61.7    5.8 Brazil
##  5 0.086 61.4    0.2 Brazil
##  6 0.265 73.4  149.  Canada
##  7 0.384 79.2  184.  Canada
##  8 0.501 66.6  194.  Colombia
##  9 0.352 82.1  136.  Denmark
## 10 0.443 75    139.  East Germany
```

Notice that the tibble displays the data type of each column underneath the column name.

3.3 Surveying your data

The `str()` or structure function is where we should go next. We want to find out what assumptions R has made about the nature of the data and change them if need be.

The command `str(intersalt)` gives the following output:

```
## Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 52 obs. of  4 variables:
##  $ b      : num  0.512 0.226 0.316 0.042 0.086 0.265 0.384 0.501 0.352 0.443 ...
##  $ bp     : chr  "72" "78.2" "73.9" "61.7" ...
##  $ na     : num  149.3 133 142.6 5.8 0.2 ...
##  $ country: chr  "Argentina" "Belgium" "Belgium" "Brazil" ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   b = col_double(),
##   ..   bp = col_character(),
##   ..   na = col_double(),
##   ..   country = col_character()
##   .. )
```

The `glimpse()` function is similar to `str()`, except that it displays all columns down the page and tries to show you as much data as posisble:

```
## Observations: 52
## Variables: 4
## $ b       <dbl> 0.512, 0.226, 0.316, 0.042, 0.086, 0.265, 0.384, 0.501, 0.3...
## $ bp      <chr> "72", "78.2", "73.9", "61.7", "61.4", "73.4", "79.2", "66.6...
## $ na      <dbl> 149.3, 133.0, 142.6, 5.8, 0.2, 148.9, 184.3, 194.1, 135.6, ...
## $ country <chr> "Argentina", "Belgium", "Belgium", "Brazil", "Brazil", "Can...
```

`summary(intersalt)` conveys some summative information:

```
##        b                  bp                   na           country
##  Min.   :0.0420   Length:52          Min.   :  0.2   Length:52
##  1st Qu.:0.2930   Class :character   1st Qu.:135.0   Class :character
##  Median :0.3460   Mode  :character   Median :152.5   Mode  :character
##  Mean   :0.3502                      Mean   :148.3
##  3rd Qu.:0.4422                      3rd Qu.:172.7
##  Max.   :0.6790                      Max.   :242.1
```

This means that something is wrong with the blood pressure data. The phenomenon observed here occurs when there is non-numeric data in the data set. This is extremely common when there are measurements that are below or above a calibration range. For example, glucose < 1 mmol/L or troponin < 5 ng/L.

3.4 Coping with Non-numeric Data

So what can we do? See what happens when we type:

```
as.numeric(intersalt$bp)
```

```
##  [1] 72.0 78.2 73.9 61.7 61.4 73.4 79.2 66.6 82.1 75.0 77.5 77.4 79.2 71.7 70.7
## [16] 73.9 79.6 69.9 76.0 72.9 68.4 67.2 73.5 67.9 77.2 72.6 79.7 62.9 66.1 67.4
## [31] 70.2 75.7 77.9 78.2 71.4 72.1 72.7 68.0 75.2 75.5 73.1 71.2 72.4   NA 78.5
## [46] 72.4 73.2 81.4 76.2 74.7 73.1 75.6
```

The `as.numeric()` function converts the bp data from a character to a numeric. But, why is there now a NA in the data?

This is what all non-numerics become, even if one is < 40 and another is > 200 ("NA" stands for "not available"). So, be careful. If we want to preserve something of what was actually in the data file, we will need another approach. If we are happy to have all non-numerics display NAs, then what we have here is fine.

But if we now want to look at the statistics of the bp column, the NA results cannot contribute. It would be good to figure out what is going on.

3.4.1 Exercise

1. Define vector `x <- c("The", "story", "of", "Hansel", "and", NA, "is", "frightening")`
2. Apply the function `is.na()` to x. What do you get?
3. Use this logical vector to replace NA with "Gretel".
4. Look at the raw intersalt data and figure out what is wrong with the bp on the entry that is becoming NA.

3.5 Simple replacements with `str_replace()`

There is another supremely useful function called `str_replace()`, global substitution, which is sort of equivalent to Excel's find and replace but ultimately much more powerful. In its simplest form it is invoked on vector x with this syntax: `str_replace(string, pattern, replacement)`

For example, we already have a vector `x <- c("The", "story", "of", "Hanzel", "and", "Gretel", "is", "frightening")`. But if we wanted to invent a new story we could write:

```
x <- c("The", "story", "of", "Hanzel", "and", "Gretel", "is", "frightening")
str_replace(x, "Gretel", "Olga")
```

```
## [1] "The"         "story"       "of"          "Hanzel"      "and"
## [6] "Olga"        "is"          "frightening"
```

```
#and to change x itself
x <- str_replace(x, "Gretel", "Olga")
x
```

```
## [1] "The"         "story"       "of"          "Hanzel"      "and"
## [6] "Olga"        "is"          "frightening"
```

3.5.1 Exercise

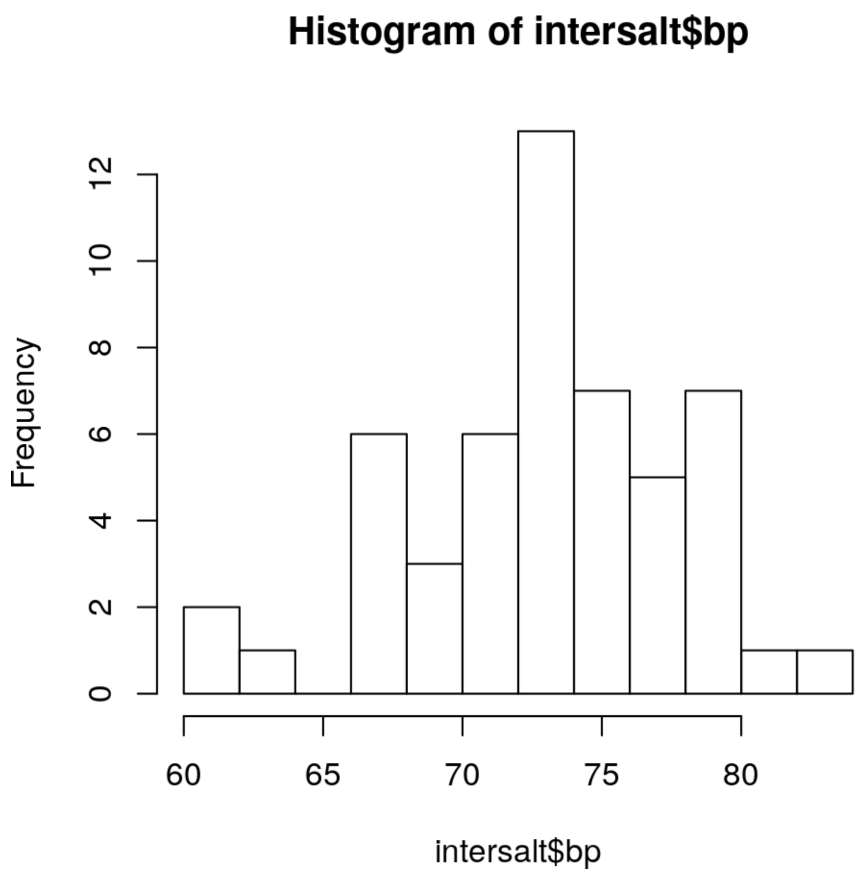
1. Read back in your original intersalt data from the csv file using the `read_csv()` command we used earlier.
   - Replace all O's in the bp column with 0's using `str_replace()`.
   - Convert the bp column to numerics as it should be
   - Now get a summary of the data and convince yourself it makes sense.

3.6 Sanity Checking your Data with Simple Visualization

3.6.1 Histograms

To produce a histogram, we use the command `hist(x)`. To increase the number of bins, we can write `hist(x, breaks = n)` where n is any number you like.

```
hist(intersalt$bp, breaks = 10)
```

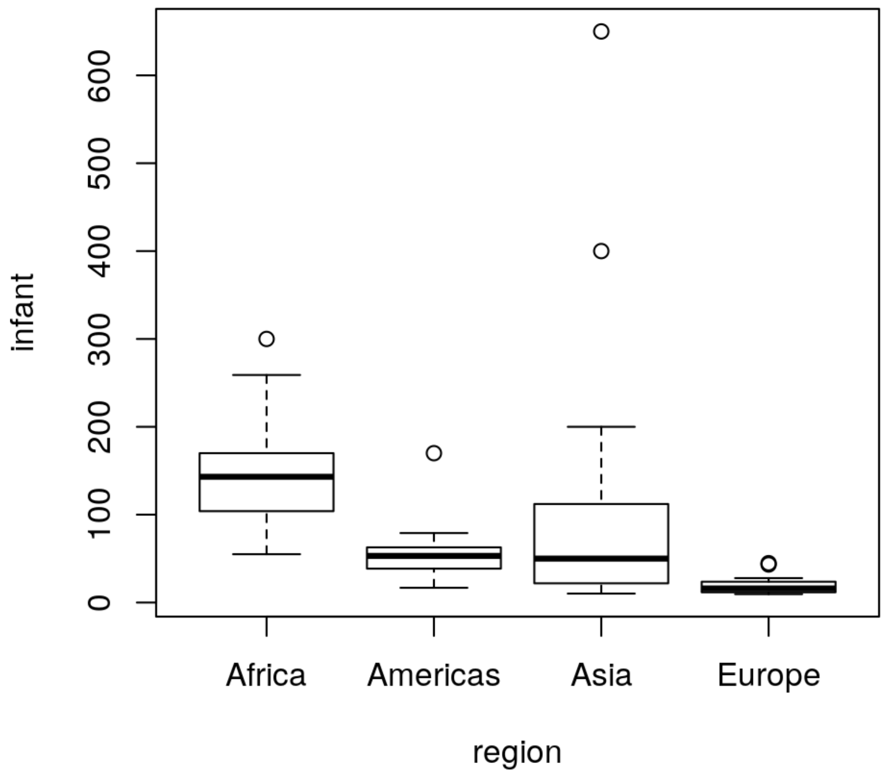


We will go into detail on beautifying our plots later, but there are many things you can do. Customization is infinite.

3.6.2 Boxplots

We can read in a toy data set in order to show how boxplots are made. This data set is comprised of infant mortality data as a function of country.

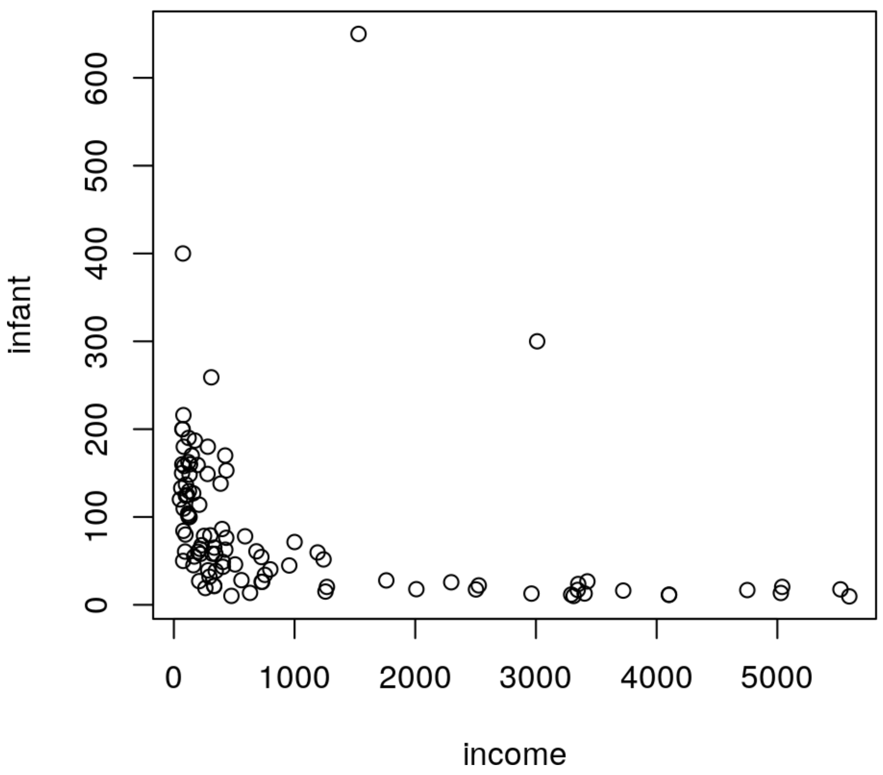
```
mortality <- read_csv("Data_Files/Leinhardt.csv")
boxplot(infant ~ region, data = mortality)
```



3.6.3 Basic Scatterplot

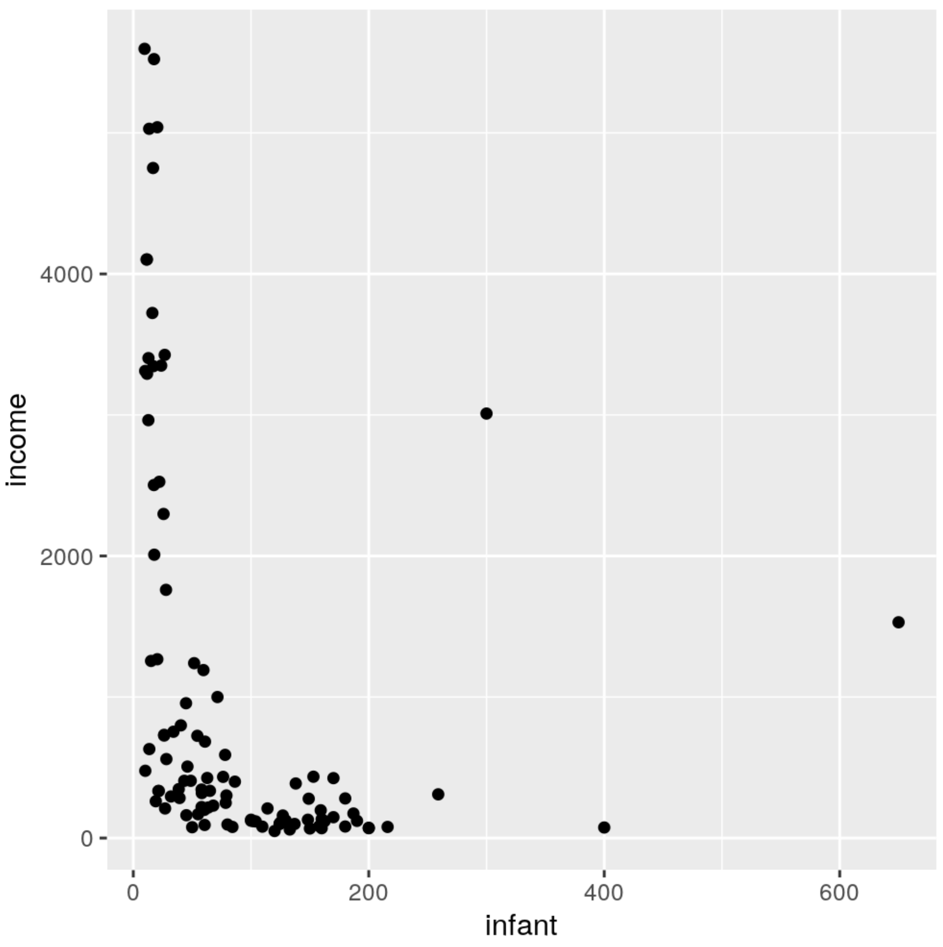
Old skule boomeR style.

```
plot(infant ~ income, data = mortality)
```



The style preferred by the cool people is:

```r
qplot(infant, income, data = mortality)
```



4 Regression

4.1 Ordinary Least Squares

In this hour we are going to look at one of our very common tasks, which is regression. We will cover ordinary least squares (OLS), Deming and Passing Bablok. These latter two forms of regression have found a neurotic devotion in the Clinical Chemistry literature, and so you have to use them when you publish. They are not available in Excel. The nice thing about Passing Bablok is that it is very resistant to the effect of an outlier, though it is certainly not the only form of robust regression.

Let's start with OLS and let's use the tube.type dataset we used in the last hour. Let's plot the PTH results of EDTA and SST against one another, since we know that there are statistical differences in the mean and median. The function we will use is called `lm()`

```
tube.data <- read_csv(file = "Data_Files/tube_data.csv")
OLS.reg <- lm(EDTA ~ SST,data = tube.data)
summary(OLS.reg)
```

```
##
## Call:
## lm(formula = EDTA ~ SST, data = tube.data)
##
## Residuals:
```

```
##      Min       1Q   Median       3Q      Max
## -1.64067 -0.90371 -0.04158  0.68455  2.44498
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.22948    0.62417   -1.97   0.0644 .
## SST          1.33713    0.08776   15.24 9.93e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.176 on 18 degrees of freedom
## Multiple R-squared:  0.928,  Adjusted R-squared:  0.924
## F-statistic: 232.1 on 1 and 18 DF,  p-value: 9.925e-12
```

```
#alternative syntax
lm(tube.data$EDTA ~ tube.data$SST)
```

```
##
## Call:
## lm(formula = tube.data$EDTA ~ tube.data$SST)
##
## Coefficients:
##   (Intercept)  tube.data$SST
##        -1.229          1.337
```

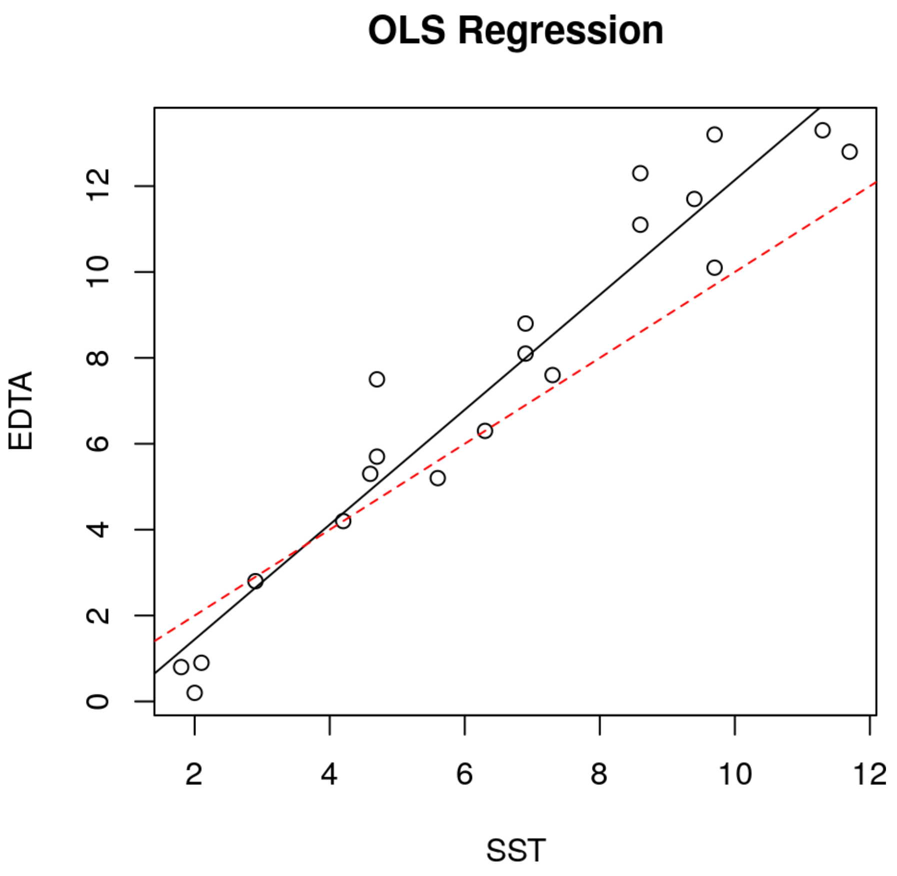
Now let's look at all of the things that R calculates:

```
str(OLS.reg)
```

```
## List of 12
##  $ coefficients : Named num [1:2] -1.23 1.34
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "SST"
##  $ residuals    : Named num [1:20] 1.459 0.379 0.803 0.36 0.83 ...
##   ..- attr(*, "names")= chr [1:20] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:20] -33.071 17.919 0.536 0.11 0.574 ...
##   ..- attr(*, "names")= chr [1:20] "(Intercept)" "SST" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:20] 11.74 4.92 8 11.34 10.27 ...
##   ..- attr(*, "names")= chr [1:20] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:20, 1:2] -4.472 0.224 0.224 0.224 0.224 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:20] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "SST"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.22 1.18
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 18
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = EDTA ~ SST, data = tube.data)
##  $ terms        :Classes 'terms', 'formula'  language EDTA ~ SST
##   .. ..- attr(*, "variables")= language list(EDTA, SST)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "EDTA" "SST"
##   .. .. .. ..$ : chr "SST"
##   .. ..- attr(*, "term.labels")= chr "SST"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(EDTA, SST)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "EDTA" "SST"
##  $ model        :'data.frame':   20 obs. of  2 variables:
##   ..$ EDTA: num [1:20] 13.2 5.3 8.8 11.7 11.1 12.3 5.2 8.1 0.8 6.3 ...
##   ..$ SST : num [1:20] 9.7 4.6 6.9 9.4 8.6 8.6 5.6 6.9 1.8 6.3 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language EDTA ~ SST
##   .. .. ..- attr(*, "variables")= language list(EDTA, SST)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "EDTA" "SST"
##   .. .. .. .. ..$ : chr "SST"
##   .. .. ..- attr(*, "term.labels")= chr "SST"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(EDTA, SST)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "EDTA" "SST"
##  - attr(*, "class")= chr "lm"
```

This is probably more information than you wanted for right now, but it's a fairly complete list of the things that you might like to know at some point. To take a look at your data and add the regression line, you can type:

```
plot(EDTA ~ SST, data = tube.data, main = "OLS Regression")
#to add the regression line
abline(OLS.reg$coefficients)
# abline(OLS.reg$coefficients[1], OLS.reg$coefficients[2]) does same thing
# abline(OLS.reg) also does the same but is less intuitive
# lines(tube.data$SST,OLS.reg$fitted.values) does the same thing
#
# to add the line of identity...
abline(0,1, col = "red", lty = 2)
```



If you enter `plot(OLS.reg)`, you can cycle through some diagnostic plots of the regression.

4.2 Weighted Least Squares

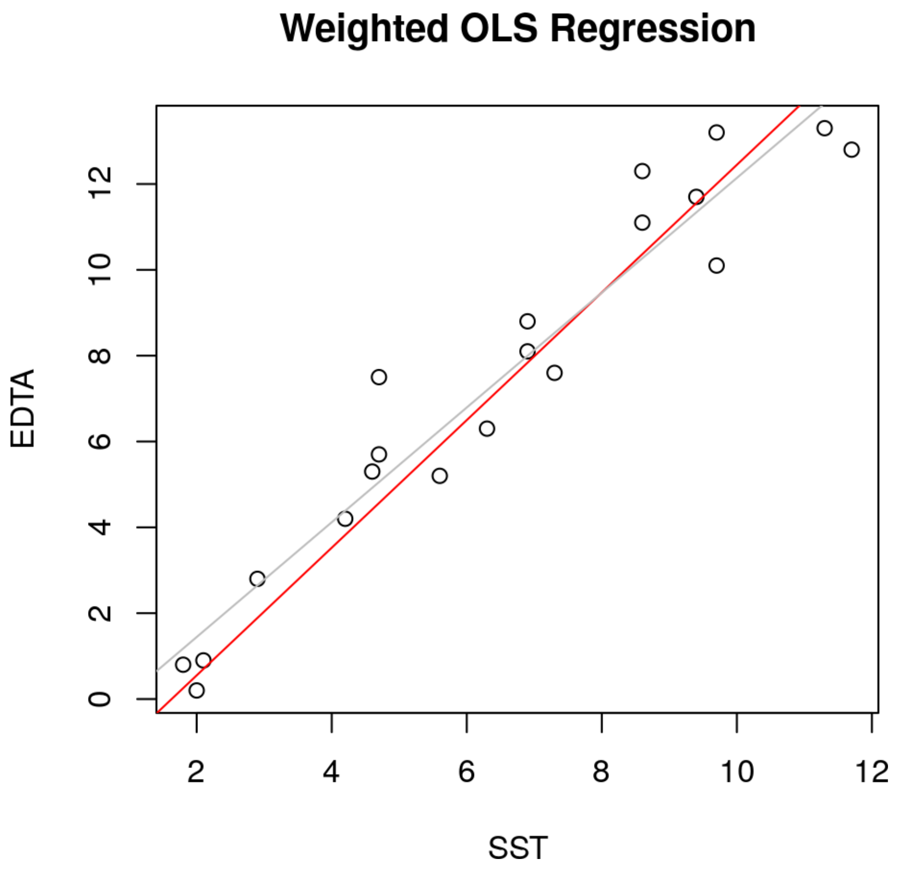
Now, sometimes you want to perform weighted regression. This is what we do on the mass spectrometer when we want to improve the recoveries of the low-level calibrators and the expense of the high level calibrators. In other words, when accuracy at the low end is clinically required, we weight the regression. How we weight is fairly arbitrary, but the bigger the weight, the more fitting effect a point has.

```
w.OLS.reg <-  lm(EDTA ~ SST, data = tube.data, weights = 1/EDTA)
summary(w.OLS.reg)
```

```
##
## Call:
## lm(formula = EDTA ~ SST, data = tube.data, weights = 1/EDTA)
##
## Weighted Residuals:
##     Min      1Q  Median      3Q     Max
## -0.7859 -0.2977  0.1973  0.4062  1.0712
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2.42248    0.27376  -8.849 5.66e-08 ***
## SST          1.48698    0.07372  20.170 8.32e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.5016 on 18 degrees of freedom
## Multiple R-squared:  0.9576, Adjusted R-squared:  0.9553
## F-statistic: 406.8 on 1 and 18 DF,  p-value: 8.319e-14
```

```
plot(EDTA ~ SST, data = tube.data, main = "Weighted OLS Regression")
#to add the regression line
abline(w.OLS.reg, col = "red")
#put the unweighted line in for comparison
abline(OLS.reg, col = "gray")
```

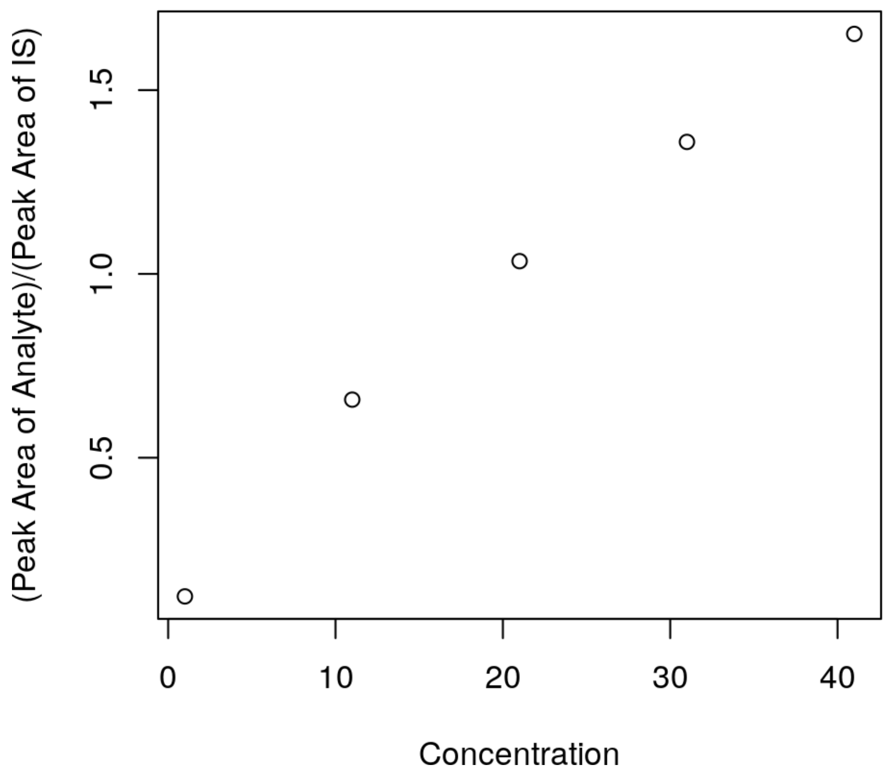


```
#What do you notice?
```

4.2.1 Exercise

- Here is some fake cal curve data that shows the characteristic flattening we see when we are trying to extend our linear range too far:

```
conc <- seq(from = 1, to = 50, by = 10)
response <- (conc/20)^(0.7)
plot(conc, response, xlab = "Concentration",
     ylab = "(Peak Area of Analyte)/(Peak Area of IS)")
```



- Find and plot the OLS regression line. Add it to the plot using abline() in green.
- Find and plot the $1/x^2$ weighted OLS regression line. Add it to the plot using `abline()` in purple.
- What is the effect of the weighting?

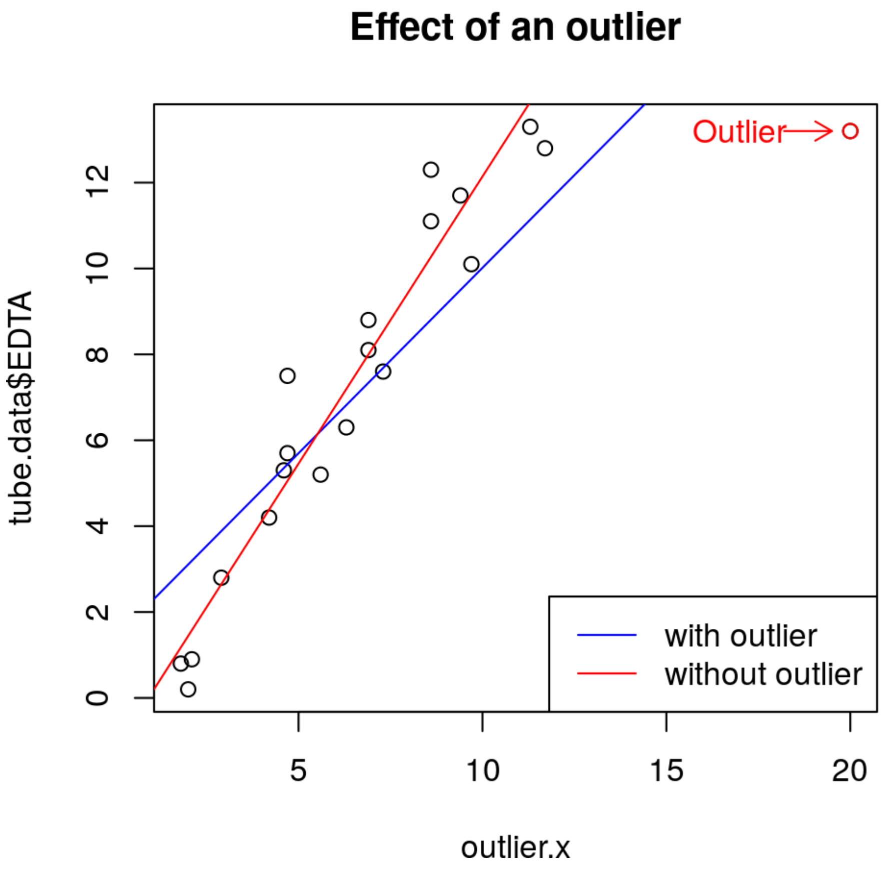
4.3 Outlier Effects in OLS

Outlier effects are significant with OLS regression. To illustrate, we can do the following (you don't need to understand the code, you just need to see the effect an outlier has):

```
outlier.x <- tube.data$SST
outlier.x[1] <- 20 #introduce a single outlier point
summary(lm(tube.data$EDTA ~ outlier.x))
```

```
##
## Call:
## lm(formula = tube.data$EDTA ~ outlier.x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.4511 -1.0334  0.1041  1.6111  3.4931
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.3805     0.9574   1.442    0.167
## outlier.x     0.8635     0.1180   7.320 8.47e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.198 on 18 degrees of freedom
## Multiple R-squared:  0.7486, Adjusted R-squared:  0.7346
## F-statistic: 53.59 on 1 and 18 DF,  p-value: 8.471e-07
```

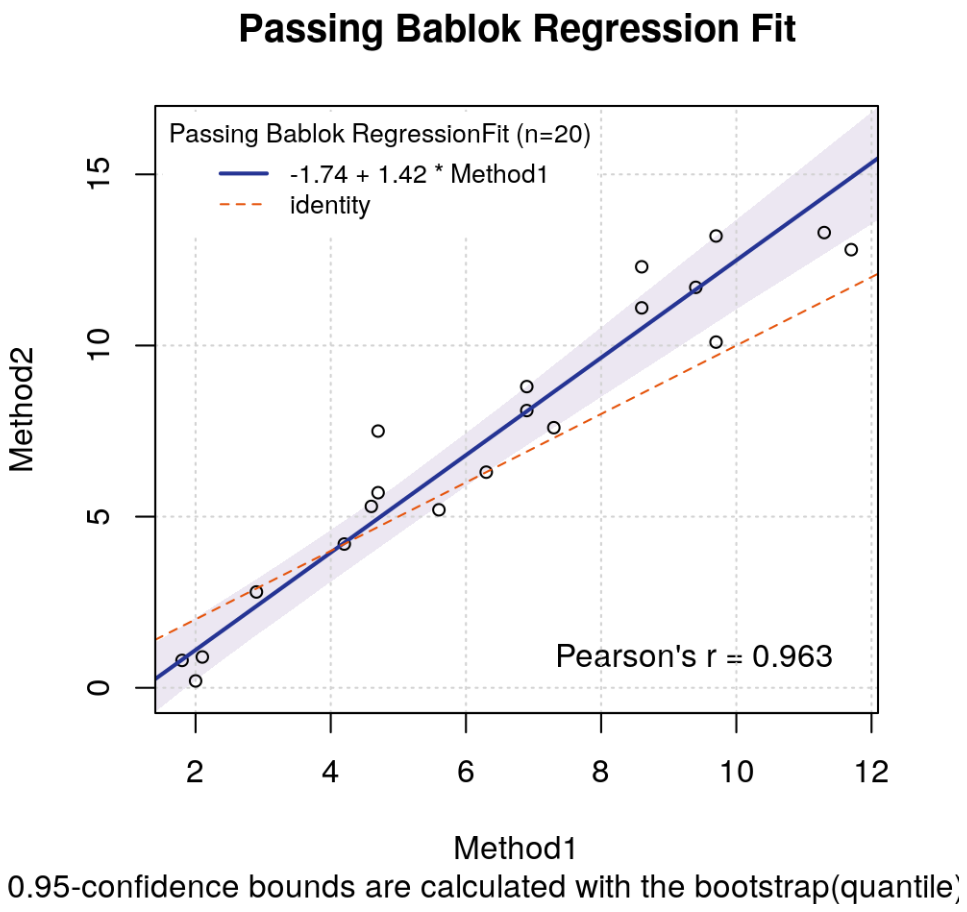
```
plot(outlier.x, tube.data$EDTA, main = "Effect of an outlier")
abline(lm(tube.data$EDTA ~ outlier.x), col = "blue") #regression with outlier
abline(OLS.reg, col = "red") #regression without outlier
legend("bottomright",
       c("with outlier","without outlier"),
       lty = c(1,1),
       col = c("blue","red"))
points(20, 13.2, col = "red")
text(17, 13.2,"Outlier", col = "red")
arrows(x0 = 18.2,
       y0 = 13.2,
       x1 = 19.5,
       y1 = 13.2,
       length = 0.1,
       col = "red")
```



4.4 Passing Bablok Regression

Now that you have seen the effect of an outlier, you can see that it would be great to have a regression method that more resistant to the effect of outliers. Passing Bablok regression is such a method. However, this function is not built-in to R. The good thing is that there are packages that include it and we can install them. Statisticians at Roche have contributed a package called "mcr" that contains PB regression (remember to run `install.packages("mcr")` before this next step).

```r
library("mcr")
PB.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "PaBa")
plot(PB.reg) # plots a nice generic plot automatically
```



4.4.1 Exercise

- Use the `mcreg()` function to show that PB regression is more resistant to an outlier than OLS regression. For your x data use outlier.x and for your y data use tube.data$EDTA. Plot the regression line. Use `abline()` to add the regression line from the PB.reg model shown immediately above.

Note that PB regression cannot be weighted by virtue of how the method works. There is no minimization of residuals, so there is nothing to weight. PB regression is very computationally intensive for larger data sets. For this reason, the code authors of the "mcr"" package have developed a method called `PaBaLarge` for large data sets. It's not exact, but it's very close. It would be called like this:

```r
PB.reg <- mcreg(tube.data$SST,tube.data$EDTA, method.reg = "PaBaLarge")
```

4.5 Deming Regression

OLS regression assumes that there is no error in the x-axis data. This is only true if the x-axis is mass spectrometry and the y-axis is an immunoassay (ba-dum-*ching*). OK–that was facetious. Deming regression also assumes that the ratios of the variances (i.e. CVs) is known for the two methods. This can only be meaningfully known if both x and y results are run in duplicate. Generally we don't do this because of the expense. For the most part, if the precision behavior of the two methods is approximately the same, then this value, called δ, is taken to be its default value of 1. The "mcr" package has both a Deming and a weighted Deming regression.

```r
Deming.reg <- mcreg(tube.data$SST,
                    tube.data$EDTA,
                    method.reg = "Deming")
WDeming.reg <- mcreg(tube.data$SST,
                     tube.data$EDTA,
                     method.reg = "WDeming")
```

If you do not like the syntax of the "mcr" package approach (because it uses a weird class for its results), you can also use the "MethComp" package (http://cran.r-project.org/web/packages/MethComp/MethComp.pdf) or "Deming" (http://cran.r-project.org/web/packages/deming/deming.pdf) package. Both have Deming and PB regression with output more similar to what you have seen before.

4.6 Preparing a Difference Plot

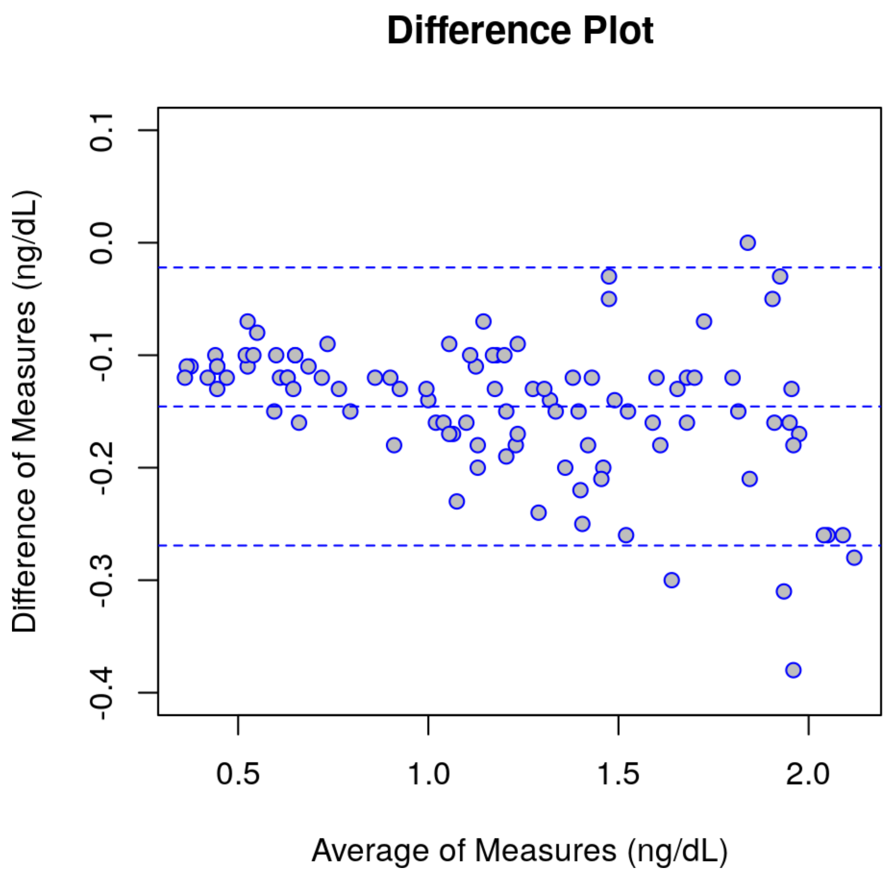
It is frequently necessary to prepare a difference plot of two methods as part of our work. Difference plots usually display the average of two methods on the x-axis and the difference of two methods on the y-axis–the difference can be expressed as an absolute or a percent. If one of the two methods happens to be a reference method, then the x-axis would display the values of the reference method rather than the average of the two methods.

Let's suppose we are comparing two creatinine methods. Let's read in the data from the prepared csv file "cre_data.csv". Remember that you have you use `setwd()` to get yourself to the right directory.

4.6.1 Exercise

1. Create a vector called `avg`, which stores the average of the creatinine values of Method 1 and Method 2. Use `cbind()` to append this vector to your dataframe.
2. Create a vector called `diff`, which stores the difference of the creatinine values of Method 1 and Method 2 (i.e. Method1 - Method2). Use `cbind()` to append the vector to your dataframe.
3. Use `plot()` to create a scatter plot of diff vs avg
4. What does the "fanning" of the difference as cre increases mean? What is the technical term for this phenomenon of inconstant scatter?

Now, we are not done. The plot does not look very nice. It also does not have a horizontal line at the mean difference and it does not have the upper and lower confidence intervals of the mean. We want it to look something like this ultimately:



We now need the standard deviation of the difference and the average difference. We will add these to our graph with `abline()`.

To add the mean difference we can type:

```
abline(h = mean(cre$diff))
```

4.6.2 Exercise

- The `abline()` function can be used to make both horizontal and vertical lines. The syntax is `abline(h = 5)` for a horizontal line at $y = 5$ and `abline(v = 3)` for a vertical line at $x = 3$. Whenever you are drawing a line, you can add the parameter lty = 1 through 6 to change the appearance of the line. For example, `abline(h = 4, col = "hotpink", lty = 2)` makes a horizontal dashed line in hot pink at $y = 4$. Groovy.

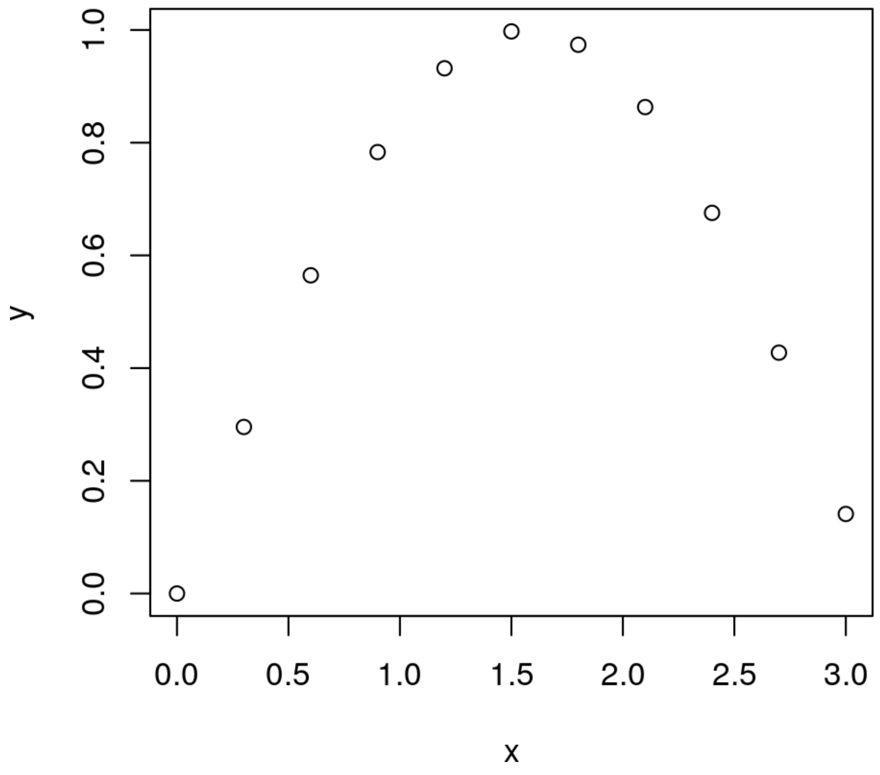
  Calculate the SD of the difference and add horizontal lines at the mean + 2SD and mean - 2SD. Use `abline()` to add the horizontal lines to the plot. Make the color blue and add the parameter `lty = 2` to make the horizontal lines dashed.

4.7 Non-Linear Regression

It is more than occasionally necessary to fit a series of points that lie on a curve. Now, you have to have some idea what shape of function you want to fit to your data…but if you do, based on some reasoned physical principle, you can determine the "best fit" using any function you want: polynomial, exponential, sinusoidal, etc.

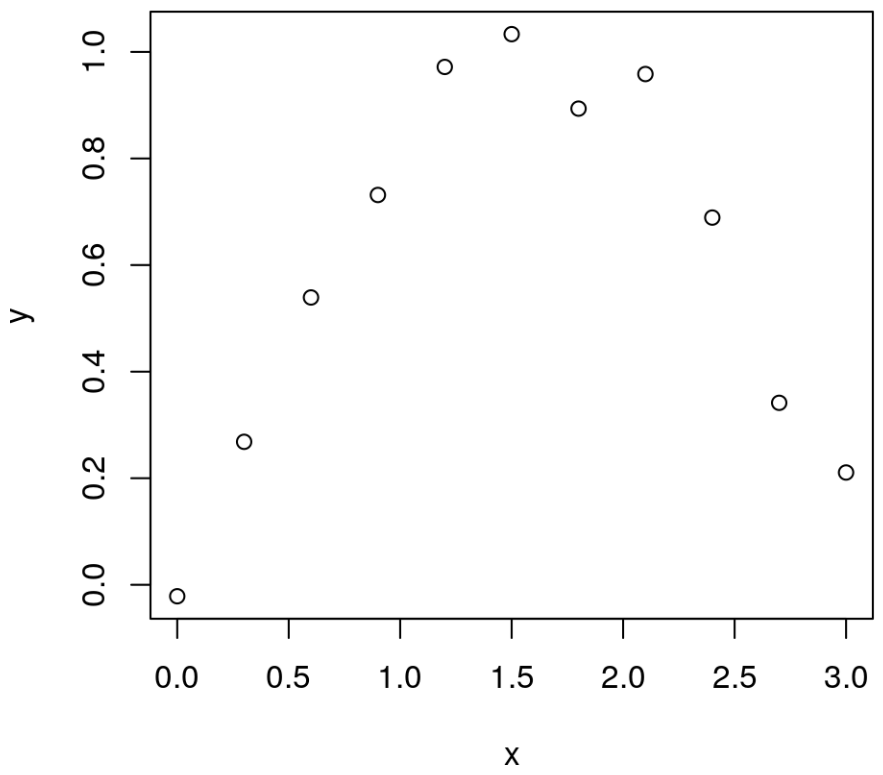
Just to illustrate the principle, let's take a few points that lie on the curve $y = sin(x)$. To do this we type:

```
x <- seq(from = 0, to = pi, by = 0.3)
y <- sin(x)
plot(x,y)
```



Now we want to add a little "noise" like we would have in real data sets... for example, if this data had come from the angle of a pendulum with a vertical line as a function of time or something.

```
y <- sin(x) + rnorm(11,0,0.05) #11 points, mean of 0, SD of 0.05
plot(x,y)
```



Now we can invoke the `nls()` function, which performs non-linear least squares. The syntax is `nls(formula, data, start)`. The `formula` is the functional form you want to fit to with unknowns included as variables, the `data` is the data you are fitting (in this case x and y), and `start` is a list of your best initial guesses for your unknowns.

In our case we are going to make no assumptions about our `sin` function. We will say that $y = Asin(kx + B) + C$, where A, B, C and k are unknowns. We will just guess that A, B, C and k are 1 for the simplicity. We could do better, but it illustrates the process.

```
sin.data <- data.frame(x,y)
nl.reg <- nls(y ~ A*sin(k*x+B)+C,
              data = sin.data,
              start = list(A = 1,B = 1,C = 1,k = 1),
              trace = TRUE)
```

```
## 11.19449 :  1 1 1 1
## 1.312533 :  0.50031019 0.10691156 0.02479001 1.03413469
## 0.09871427 :   0.4138918 -1.2086290  0.5925691  1.7272333
## 0.06646956 :   0.4979158 -0.8422831  0.5135715  1.5127455
## 0.04776411 :   0.6584595 -0.4445437  0.3591367  1.2676999
## 0.03048637 :   0.7065534 -0.4586217  0.3116418  1.2737894
## 0.03048485 :   0.7084927 -0.4534716  0.3095397  1.2708075
## 0.03048484 :   0.7083147 -0.4538873  0.3097338  1.2710509
## 0.03048484 :   0.7083285 -0.4538564  0.3097188  1.2710329
```

```
summary(nl.reg)
```

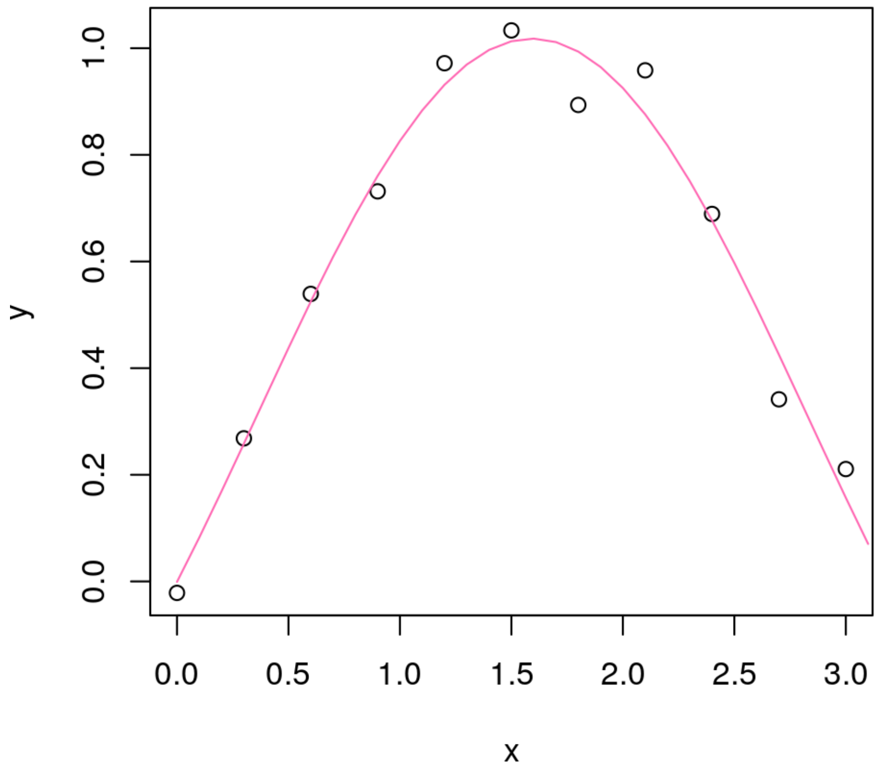
```
##
## Formula: y ~ A * sin(k * x + B) + C
##
## Parameters:
```

```
##   Estimate Std. Error t value Pr(>|t|)
## A   0.7083     0.2419   2.928  0.02210 *
## B  -0.4539     0.5126  -0.885  0.40534
## C   0.3097     0.2612   1.186  0.27435
## k   1.2710     0.3167   4.013  0.00511 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06599 on 7 degrees of freedom
##
## Number of iterations to convergence: 8
## Achieved convergence tolerance: 2.105e-06
```

The parameter trace = TRUE shows the work that `nls()` is doing as it tries to find a solution from the start values. If the start values really stink, `nls()` will choke and tell you so. You will need to find better guesses for the start values. There are ways to accomplish this which are outside the scope of the present discussion.

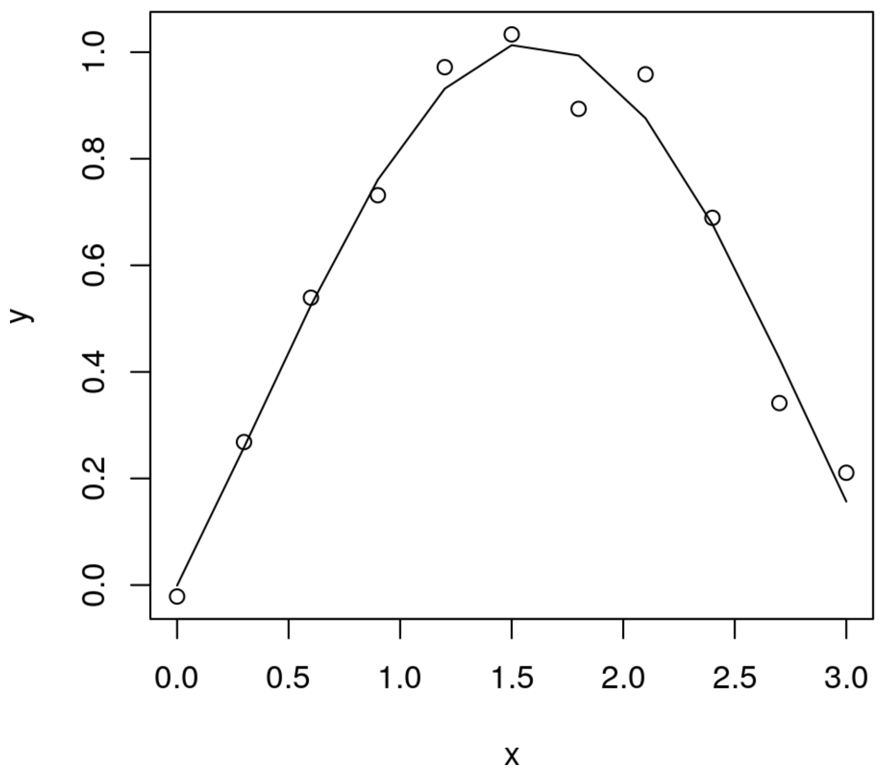
So now we can show our fit:

```
plot(x,y)
x.fit <- seq(from = 0, to = pi, by = 0.1) #make a bunch of tightly spaced points
y.fit <- 0.7083 * sin(1.2710 * x.fit - 0.4539) + 0.3097
lines(x.fit, y.fit, col = "hotpink")
```



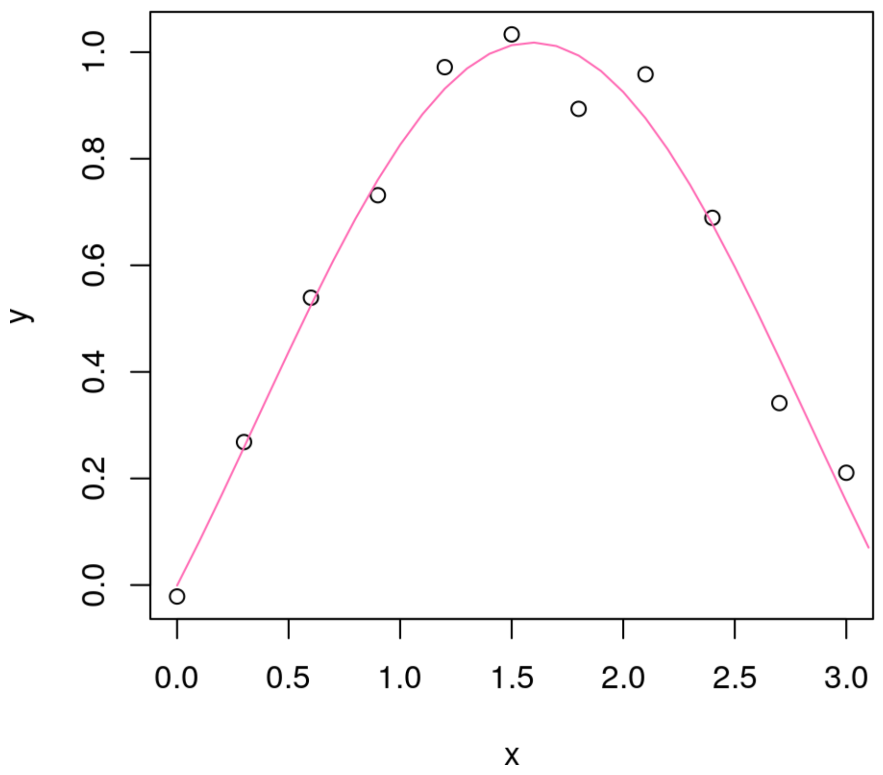
A less aesthetically pleasing but faster way to get this is to plot the fitted values from the `nl.reg()` output.

```
plot(x,y)
lines(x, predict(nl.reg))
```



. . . or to make a smooth curve:

```
plot(x,y)
x.fit <- seq(from = 0, to = pi, by = 0.1)
y.fit <- predict(nl.reg , list(x = x.fit)) #saves effort of transcribing A,B, C and k
lines(x.fit, y.fit, col = "hotpink")
```

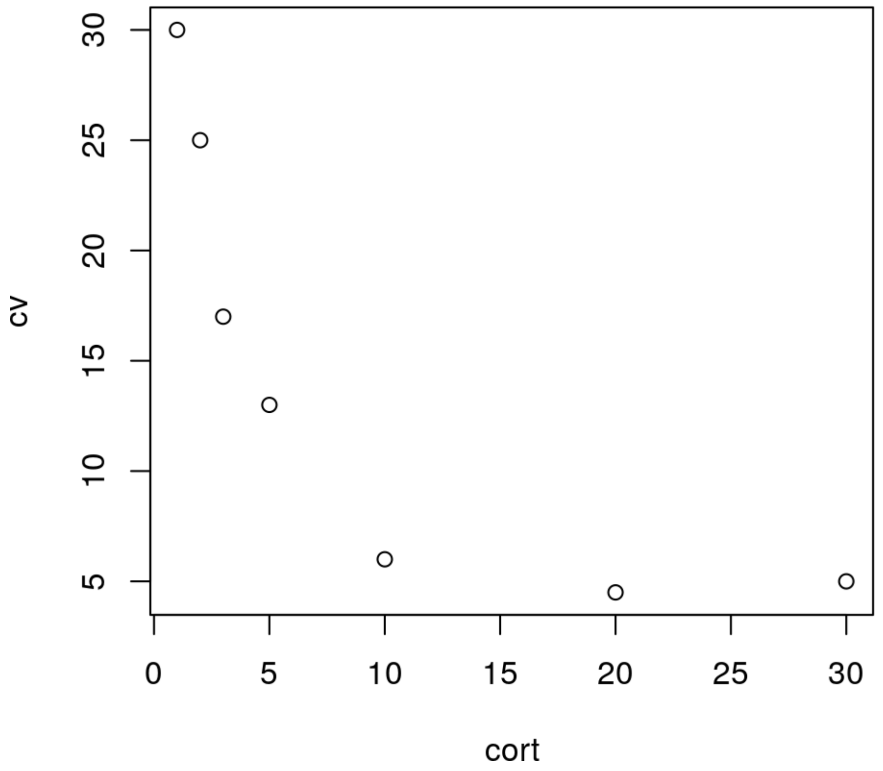


Helpful **NOTE**: Dan did not know about this technique. He just knew there should be a way, Googled it and found out how to do it.

4.7.1 Exercise

- Suppose that you have the following data for the precision of your cortisol assay at different concentrations. In Canada we live in a nmol/L world, so we'll use those units just to keep Steve as confused as possible. Enter the following initial data into R.

```
cort <- c(1,2,3,5,10,20,30)
cv <- c(30, 25, 17,13,6,4.5,5)
cv.data <- data.frame(cort,cv)
plot(cort,cv)
```



. . . giving this graph.

- Fit this data to a hyperbolic function of the form $y = A/(x - B) + C$. If your start list for A, B and C gives you errors, try fitting to $y = A/x$ without B and C and then use your result for A with B and $C = 0$.
- Calculate the fitted values of y from `x.fit <- seq(from = 0, to = 30, by = 1)`.
- Draw the fitted curve in the color "coral".
- Do your best to estimate the functional sensitivity – the concentration at which the CV = 20%.

5 Things with Strings and Tools for Data Cleansing

5.1 Working with Strings

In programming, the word 'string' refers to anything treated as text. Strings may be one word or several words, and can include other characters as well. For example, the word "computer" is a string, as is the sentence "I have 4 computers." There are an array of tools in R specifically for working with strings.

For example, suppose you have to generate 100 jpeg figures using a loop (we will be covering this kind of thing shortly) and you need to name them according to the index of the loop. In this circumstance you are going to need to string together some variables to make the names. The base-r function `paste()` does this for us as does `str_c()` from the stringr package. For example:

```r
number <- seq(from = 99, to = 95, by = -1)
for (i in number) {
  phrase <- str_c(i, "bottles of beer on the wall", i,
                "bottles of beer. \nTake one down, pass it around.",
                i-1, "bottles of beer on the wall \n", sep = " ")
  cat(phrase)
}
```

```
## 99 bottles of beer on the wall 99 bottles of beer.
## Take one down, pass it around. 98 bottles of beer on the wall
## 98 bottles of beer on the wall 98 bottles of beer.
## Take one down, pass it around. 97 bottles of beer on the wall
## 97 bottles of beer on the wall 97 bottles of beer.
## Take one down, pass it around. 96 bottles of beer on the wall
## 96 bottles of beer on the wall 96 bottles of beer.
## Take one down, pass it around. 95 bottles of beer on the wall
## 95 bottles of beer on the wall 95 bottles of beer.
## Take one down, pass it around. 94 bottles of beer on the wall
```

In other words, `str_c()` lets us create uninterrupted strings from other strings or variables. Here is a more practical example where we need to make a bunch of jpegs all at once. The script creates a directory so that you don't have to delete all the jpegs we are going to make. Paste automatically uses a space between all the things you are pasting together. This is not always what you want so you can specify with `sep = "blablabla"`.

```r
str_c("When my boss is talking my brain says, '", "'. It's like I am dying inside.",
      sep = "Bla Bla Bla")
```

```
## [1] "When my boss is talking my brain says, 'Bla Bla Bla'. It's like I am dying inside."
```

Here is a slightly more practical example:

```r
library(fs)

dir_create("my_figures") # create a junk folder in the current working directory.
#to create a directory elsewhere you would need the full path

for (i in 1:10) {
  random.data <- rnorm(1000,i,0.3) #create Gaussian random data
  #with a mean of i and an SD of 0.3
  file.name <- file.path("my_figures",
                         str_c("figure_number", i, ".jpg", sep = "" ))
  print(file.name) #for your benefit to see what is going on
  jpeg(file.name) #open a jpeg file to dump the image into
  hist(random.data, col = i) #if you want base-r style graphics
  dev.off() #close the jpeg file
}
```

Now go into the "My_Figures" directory and see what has happened. You will notice that R takes care of the fact that Mac/Linux and Windows platforms use different kinds of slashes in the file structures.

There are other cool things that you can do with strings. The `str_split()` function break strings apart based on separators within the string:

```r
line <- "Come on and rock me Amadeus."
split.line <- str_split(line, " ")
split.line
```

```
## [[1]]
## [1] "Come"     "on"       "and"      "rock"     "me"       "Amadeus."
```

At times it is necessary to reduce all letters to capitals or to smalls. The functions that can accomplish this are `str_to_upper()` and `str_to_lower()`. This is particularly useful to process data for further analysis. For example, if you had survey data containing "Yes", "YES", "yes", "Y", and "y" and wanted all of them to be turned into "Y" so that you could do statistics later, you might do something like this:

```r
responses <- c("Yes","no","y","y","n","N","NO","Y","YES","Y")
#clean up this mess
responses <- str_to_upper(responses)
responses
```

```
##  [1] "YES" "NO"  "Y"   "Y"   "N"   "N"   "NO"  "Y"   "YES" "Y"
```

```r
responses <- str_replace(responses, "YES", "Y")
responses <- str_replace(responses, "NO", "N")
responses
```

```
##  [1] "Y" "N" "Y" "Y" "N" "N" "N" "Y" "Y" "Y"
```

Thought: What if you had a lot more possible responses that all started with "Y" but all meant "Yes", like "Yo", "Yepper", "Yessiree"? Did you know that you can actually look for the pattern "Y" followed by any number of other characters? You can. It's called a "Regular Expression", and just to show you the magic:

```r
responses <- c("Yes","no","y","y","n","N","NO","Y","YES","Y","yes sir",
               "yep", "YO","Nay Laddie", "Nej", "Nein","Non", "Yayaya",
               "Nee","No way Jose","Nah", "Nei","não")
responses <- str_replace(responses, "(^[Yy].*)", "Y")
responses <- str_replace(responses, "(^[Nn].*)", "N")
responses #magic
```

```
##  [1] "Y" "N" "Y" "Y" "N" "N" "N" "Y" "Y" "Y" "Y" "Y" "Y" "N" "N" "N" "N" "Y" "N"
## [20] "N" "N" "N" "N"
```

```r
# ^ means "starts with"
# [Yy] means "Y" or "y" and [Nn] means "N" or "n"
# . means "followed by anything"
# * means "the preceding character can occur 0 or more times"
```

This is just meant to show you that stuff like this is possible. You can learn about it another time.

The `str_sub()` function has syntax of `substr(x, start, stop)`. This can be conveniently used to get part of a string when what you want is stuck in the middle. For example if you wanted the last 5 characters of an 18 character string–because it contained, say, the time–you could do something like this:

```r
str_sub("Feb,02,2015  07:34",14,18)
```

```
## [1] "07:34"
```

If you don't happen to know how long the string you are breaking up is, you can use the `str_length()` function. The `length()` function won't work; it tells you how many elements are in a vector. You want `str_length()`.

```r
str_length("Here kitty kitty kitty")
```

```
## [1] 22
```

```r
length("Here kitty kitty kitty")
```

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

5.1.1 Exercise

- Convert a full name (let's start with `name <- "William Osler"`) to an abbreviated name made up of the first three letters of the first name and the first the letters of the last name, all in lower case. For example “Stephen Master” would become “stemas”.

NOTE: the `str_split()` function produces a list as its output which we find kind of inconvenient generally. To turn it back into a vector just apply the command unlist to it. ie:

```
x <- str_split("Dan Holmes", " ") #creates a list
str(x)
```

```
## List of 1
##  $ : chr [1:2] "Dan" "Holmes"
```

```
x[[1]][1]
```

```
## [1] "Dan"
```

```
x[[1]][2] #inconvenient syntax
```

```
## [1] "Holmes"
```

```
x <- unlist(x)
x
```

```
## [1] "Dan"    "Holmes"
```

```
x[1]
```

```
## [1] "Dan"
```

```
x[2] #happy syntax
```

```
## [1] "Holmes"
```

```
#or, use the simplify = TRUE option to return a character matrix
#for easier indexing
x <- str_split("Dan Holmes", " ", simplify = TRUE)
x[1]
```

```
## [1] "Dan"
```

```
x[2]
```

```
## [1] "Holmes"
```

5.2 Oh `str_detect()`, how I love thee. Let me count the “wheys”

The function `str_detect()` is a very useful function that allows you to look for patterns rather than an exact match. This can be very helpful when the value of interest is buried in other text. For example, you might want to pull out anything that looks like a Personal Health Number or telephone number.

The syntax for `str_detect()` is

```
str_detect(string, pattern-to-find)
```

Note that `str_detect()` returns the *indices* of the pattern matches in the vector. For example, let’s pull the entire web of science publications list and then find all the journals that contain the word “ophthalmology”.

This code will download the Web of Science publication abbreviations database

```
if(!file.exists("Data_Files/wos_abbrev_table.txt")){
  download.file("https://ndownloader.figshare.com/files/5212423",
                "wos_abbrev_table.csv")
}
wos <- read_delim("Data_Files/wos_abbrev_table.txt",
                     delim = ";",
                     col_names = TRUE)
```

Now, let's find all the journals with the word ophthalmology in them.

```
ophtho.journals <- str_which(wos$full, "[Oo]phthalmology")
ophtho.journals[1:10] # show the first 10
```

```
##  [1]  5533  7277  7512  8419  8420  8421  9804 11081 11139 13309
```

```
length(ophtho.journals)
```

```
## [1] 58
```

If we want to find the *names* of these journals, it can be accomplished in two ways:

```
# wos[ophtho.journals,1]
# or
# wos[ophtho.journals,]$full
wos[ophtho.journals,]$full[1:10] # names of the first 10
```

```
##  [1] "Advances in Ophthalmology"
##  [2] "Ama Archives of Ophthalmology"
##  [3] "American Journal of Ophthalmology"
##  [4] "Annals of Ophthalmology"
##  [5] "Annals of Ophthalmology & Glaucoma"
##  [6] "Annals of Ophthalmology-glaucoma"
##  [7] "Archives of Ophthalmology"
##  [8] "Australian and New Zealand Journal of Ophthalmology"
##  [9] "Australian Journal of Ophthalmology"
## [10] "Bmc Ophthalmology"
```

5.2.1 Exercise

- Read in starwars_menu.csv. The data is the pre-selected menu for a banquet. Have a look at the data. The caterer needs to know how many individuals want "whey" but the "whey" is buried in things like "the whey, the shoots and the rice".
  - How many people want "whey"?
  - Produce a dataframe of all the rows containing the word whey.
  - Which individuals chose "whey"?
  - Which individuals did not choose "whey"?

Finding exact matches is just scratching the surface of what `str_which()` can do. It can also be used to match sophisticated patterns using the same regular expressions tools that `str_replace()` uses. For example, blood collection specimen numbers in lab start with X,M,T,W,H,F, or S and are then followed by 3–6 digits. So we can make a grep search that finds only this pattern so we can see what samples represent patient results (as opposed to quality control material).

```
x <- c("D123","F12414","X1324556","M432","H2141","S1")
```

Some of these will match the accession numbers

```
# ^ means "starts with"
# | means "OR"
# [0-9] means any digit
# {m,n} means previous item matches at least m and at most n times
# $ means ends with
pattern <- "^[X|M|T|W|H|F|S][0-9]{3,6}$"
str_which(x, pattern)
```

```
## [1] 2 4 5
```

Again, Regular Expressions could be an entire two day course to itself. For now, just know that this type of advanced pattern finding is possible.

6 Meet the 'tidyverse' - Pivot, Join, Filter, and Clean

There is a paradigm of R programming referred to as the "tidyverse" that is particularly useful for certain types of data summary and data visualization. It allows for rapid-high level commands accomplishing a great deal in few lines of highly readable code. The challenge of using tidyverse packages is that they are under very rapid development and this can mean that updates in the packages can cause your code to stop functioning. Additionally, many statistical packages use traditional "base R" and they may not cooperate with tidyverse output. However, in the context of our work in health care, we are usually doing fairly simple one-off reports for research or answering a specific question so using the tidyverse makes sense.

6.1 Packages

Before we can start using these functions we will have to install and load them. This is because the 'tidyverse' does not exist in 'base' R (that is, the pre-loaded set of function which are installed when you install the R program). Rather, the tidyverse is a set of packages. Packages are themselves sets of functions which have been written by whomever authoured the package. These user-created functions have been 'packaged' and made available for anyone to download and use.

Packages are what makes R great. Lots can be done in base R but often to acheive what you want, you end up having to write your own functions. This is fine if you like programming and find coding your own functions enjoyable. Sometimes, though, we just want to 'get it done' - and whatever 'it' is that you are trying to do, chances are that someone out there has done it before and written a package for you to make it easier.

A final word about packages - they are only as good as the person who wrote them. The tidyverse is a group of packages which have been written by Hadley Wickham (who created RStudio) and his team - which means that we can rely on them being functional, updated and usually bug-free. The same cannot always be said of smaller packages, so if you are ever poking around for a package to fill a niche need, read the documentation and do some google searches to see how others have been able to work with the package before spending too much time trying to code with it.

6.1.1 Exercise

- Hopefully you installed the tidyverse when you first installed R as per the pre-course instructions. Just in case, the following code will install the tidyverse package if it is not already there.

```
if("tidyverse" %in% rownames(installed.packages()) == FALSE) {install.packages("tidyverse")}
```

This command asks R to download the most current version of the tidyverse from CRAN, which is an online package repository. Note that you need to be connected to the internet for this to work.

Since quite a few packages are being downloaded, this may take a few minutes. Perhaps a good time for a coffee break.

Once tidyverse is installed, we need to load it by calling library():

```
library(tidyverse)
```

You only need to install a package once, but you will need to load any packages you need each time you restart R.

6.2 pivot_longer() and pivot_wider()

Now that we are in the tidyverse, the first thing we want to introduce is what is meant by "tidy data".

When humans prepare spreadsheet data, they typically have each row represent all the observations on single subject. This is usually pretty easy to look at but it happens to have a number of disadvantages from a statistical programming standpoint.

For example, in traditional "untidy" data, you might have each subject on a row and all of their lab tests represented as columns: Sodium, Potassium, Chloride, Bicarbonate, Creatinine, pH, Troponin etc. But in the tidy data paradigm, all of the blood tests should be factors under a single column labelled "Test" and then each row represents a single observation, not a single patient.

For example, this is a traditional view:

Patient	Sodium	Potassium	Chloride	Total CO2
1	143	4.2	110	25
2	129	4.4	100	18
3	142	4.6	105	27
4	148	5.0	111	25

And this is a tidy view of the same data:

Patient	Test	Result
1	Sodium	143.0
1	Potassium	4.2
1	Chloride	110.0
1	Total CO2	25.0
2	Sodium	129.0
2	Potassium	4.4
2	Chloride	100.0
2	Total CO2	18.0
3	Sodium	142.0
3	Potassium	4.6
3	Chloride	105.0
3	Total CO2	27.0
4	Sodium	148.0
4	Potassium	5.0
4	Chloride	111.0
4	Total CO2	25.0

It is frequently necessary to jump back and forth between the traditional view, which we call "data wide" and the tidy view which we call "data long". This is accomplished with the functions `pivot_longer()`

and `pivot_wider()`.

Let's starting exploring these funtions by reading in some data on "anthropometric" (for want of a better term) measurements on opossum.

```
possum <- read_csv("Data_Files/possum.csv")
# we used read_csv here (a tidyverse function) instead of read.csv (base R)
# the anthropomorphic measurements of the possums are wide
head(possum)
```

```
## # A tibble: 6 x 15
##   X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##   <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
## 1 C3        1     1 Vic   m         8    94.1   60.4     89    36       74.5
## 2 C5        2     1 Vic   f         6    92.5   57.6     91.5  36.5     72.5
## 3 C10       3     1 Vic   f         6    94     60       95.5  39       75.4
## 4 C15       4     1 Vic   f         6    93.2   57.1     92    38       76.1
## 5 C23       5     1 Vic   f         2    91.5   56.3     85.5  36       71
## 6 C24       6     1 Vic   f         1    93.1   54.8     90.5  35.5     73.2
## # ... with 4 more variables: earconch <dbl>, eye <dbl>, chest <dbl>,
## #   belly <dbl>
```

We can convert them to data long format with the `gather()` function. In the this function we must determine three things:

1. What columns are to be gathered together as factors of a similar type?–This parameter referred to as the "key" and in our output the column will be named "Possum_Metric".
2. What do you want to call the column that contains the associated values?–This parameter is referred to as the "value" and in our output the column will be named "Result".
3. Which columns (by name or number) are to be gathered?–In this case it is columns 7–15. You can also denote which columns you *don't* want gathered – so we could also write this as "-c(1:6)".

```
possum.long <- pivot_longer(possum, cols = 7:15,
                            names_to = "bodypart",
                            values_to = "measurement")
head(possum.long,10)
```

```
## # A tibble: 10 x 8
##    X1     case  site Pop   sex     age bodypart measurement
##    <chr> <dbl> <dbl> <chr> <chr> <dbl> <chr>          <dbl>
##  1 C3        1     1 Vic   m         8 hdlngth         94.1
##  2 C3        1     1 Vic   m         8 skullw          60.4
##  3 C3        1     1 Vic   m         8 totlngth        89
##  4 C3        1     1 Vic   m         8 taill           36
##  5 C3        1     1 Vic   m         8 footlgth        74.5
##  6 C3        1     1 Vic   m         8 earconch        54.5
##  7 C3        1     1 Vic   m         8 eye             15.2
##  8 C3        1     1 Vic   m         8 chest           28
##  9 C3        1     1 Vic   m         8 belly           36
## 10 C5        2     1 Vic   f         6 hdlngth         92.5
```

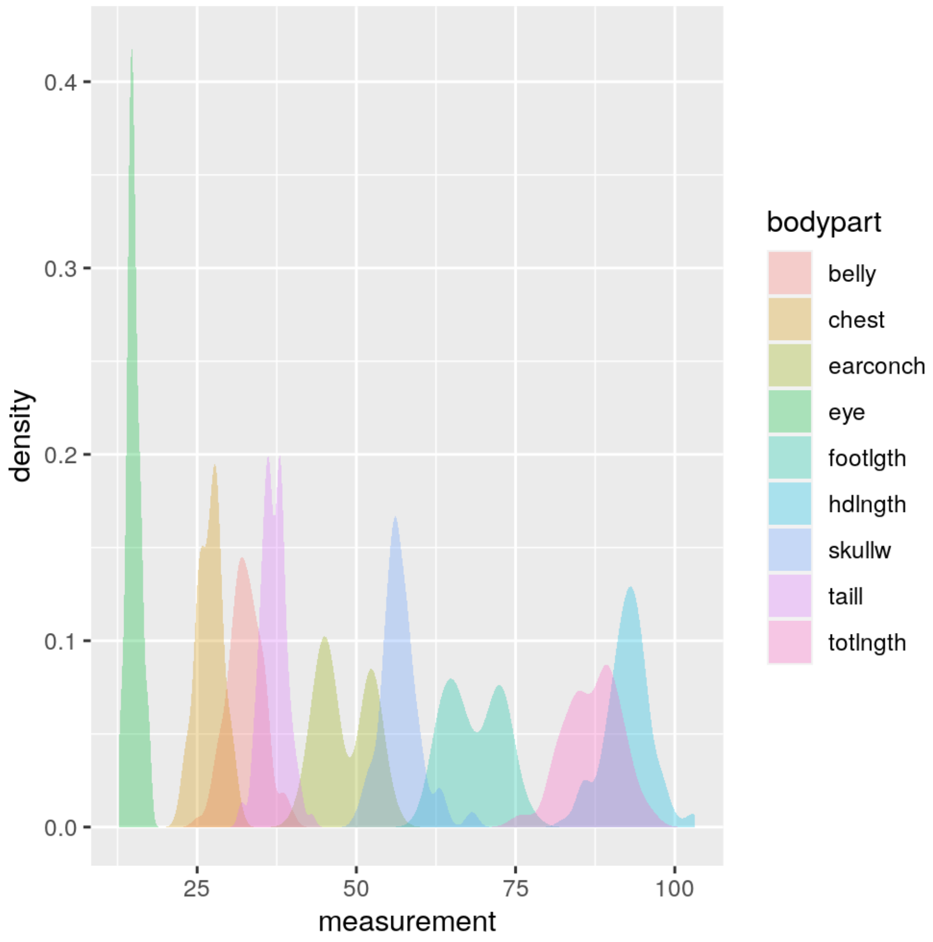
As it turns out (and we will explain more later), this permits very slick calculations:

```
#long data permits rapid calculations
group_by(possum.long, bodypart) %>%
  summarize(mean   = mean(measurement, na.rm = TRUE),
            median = median(measurement, na.rm = TRUE),
            sd = sd(measurement, na.rm = TRUE))
```

```
## # A tibble: 9 x 4
##   bodypart  mean median    sd
##   <chr>    <dbl>  <dbl> <dbl>
## 1 belly     32.6   32.5  2.76
## 2 chest     27     27    2.05
## 3 earconch  48.1   46.8  4.11
## 4 eye       15.0   14.9  1.05
## 5 footlgth  68.5   68    4.40
## 6 hdlngth   92.6   92.8  3.57
## 7 skullw    56.9   56.4  3.11
## 8 taill     37.0   37    1.96
## 9 totlngth  87.1   88    4.31
```

and plots:

```
#long data permits rapid visualizations which we will cover later
library(ggplot2)
p <- ggplot(possum.long, aes(x = measurement, fill = bodypart)) +
  geom_density(alpha = 0.3, color = NA)
p
```



If we want to convert back to ‘untidy’ data, this is accomplished by `spread()`. For this function again we need to determine our “key” and “value” columns:

1. The “key” column will be “spread” across the top of the table (this column becomes the column names).
2. The “value” column contains the values we want to populate these new columns.

```
possum.wide <- pivot_wider(possum.long,
                           names_from = bodypart,
                           values_from = measurement)
head(possum.wide, 10)
```

```
## # A tibble: 10 x 15
##    X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
##  1 C3        1     1 Vic   m         8    94.1   60.4     89    36       74.5
##  2 C5        2     1 Vic   f         6    92.5   57.6     91.5  36.5     72.5
##  3 C10       3     1 Vic   f         6    94     60       95.5  39       75.4
##  4 C15       4     1 Vic   f         6    93.2   57.1     92    38       76.1
##  5 C23       5     1 Vic   f         2    91.5   56.3     85.5  36       71  
##  6 C24       6     1 Vic   f         1    93.1   54.8     90.5  35.5     73.2
##  7 C26       7     1 Vic   m         2    95.3   58.2     89.5  36       71.5
##  8 C27       8     1 Vic   f         6    94.8   57.6     91    37       72.7
##  9 C28       9     1 Vic   f         9    93.4   56.3     91.5  37       72.4
## 10 C31      10     1 Vic   f         6    91.8   58       89.5  37.5     70.9
## # ... with 4 more variables: earconch <dbl>, eye <dbl>, chest <dbl>,
## #   belly <dbl>
```

6.3 arrange(), filter(), and select()

There are some more very useful and simple functions in the tidyverse which we will need before we get too much further.

The first is `arrange()` and it does pretty much what you think it would. Let’s try it out on the possum data:

```
arrange(possum, age)
```

```
## # A tibble: 104 x 15
##    X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
##  1 C24       6     1 Vic   f         1    93.1   54.8     90.5  35.5     73.2
##  2 C45      17     1 Vic   f         1    94.7   67.7     89.5  36.5     73.2
##  3 BB31     39     2 Vic   f         1    84.7   51.5     75    34       68.7
##  4 CD12     72     5 other m         1    85.9   52.4     80.5  35       62  
##  5 CD13     73     5 other m         1    82.5   52.3     82    36.5     65.7
##  6 BSF5     78     6 other m         1    86.5   51       81    36.5     63  
##  7 BSF6     79     6 other m         1    85.8   50       81    36.5     62.8
##  8 BSF7     80     6 other m         1    86.7   52.6     84    38       62.3
##  9 BTP16   100     7 other m         1    89.5   56       81.5  36.5     66  
## 10 BTP17   101     7 other m         1    88.6   54.7     82.5  39       64.4
## # ... with 94 more rows, and 4 more variables: earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

If we want to arrange from largest to smallest, we would ask for the ages in descending `desc()` order:

```
arrange(possum, desc(age))
```

```
## # A tibble: 104 x 15
##    X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
##  1 C28       9     1 Vic   f         9    93.4   56.3     91.5  37       72.4
##  2 C32      11     1 Vic   f         9    93.3   57.2     89.5  39       77.2
##  3 C3        1     1 Vic   m         8    94.1   60.4     89    36       74.5
##  4 C61      26     1 Vic   m         7    96     59       90    36       73.6
##  5 BB15     36     2 Vic   m         7    93.3   59.3     88    35       74.3
##  6 BB25     38     2 Vic   m         7    92.4   56       80.5  35.5     68.4
##  7 BR1      54     4 other m         7    96.9   63       91.5  43       71.3
##  8 CD1      61     5 other m         7    95.7   59       86    38       63.1
##  9 CD10     70     5 other f         7    91.9   56.4     87    38       65.4
## 10 BTP9     94     7 other m         7    91.8   57.6     84    35.5     64.2
## # ... with 94 more rows, and 4 more variables: earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

What happens if we `arrange()` something that is not a number?

```
arrange(possum, X1)
```

```
## # A tibble: 104 x 15
##    X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
##  1 A1       29     1 Vic   f         3    92.8   56       88    35       74.9
##  2 A2       30     1 Vic   f         2    92.1   54.4     84    33.5     70.6
##  3 A3       31     1 Vic   m         3    92.8   54.1     93    37       68
##  4 A4       32     1 Vic   f         4    94.3   56.7     94    39       74.8
##  5 AD1      33     1 Vic   m         3    91.4   54.6     89    37       70.8
##  6 BB13     35     2 Vic   m         4    94.4   57.9     85    35.5     71.2
##  7 BB15     36     2 Vic   m         7    93.3   59.3     88    35       74.3
##  8 BB17     37     2 Vic   f         2    89.3   54.8     82.5  35       71.2
##  9 BB25     38     2 Vic   m         7    92.4   56       80.5  35.5     68.4
## 10 BB31     39     2 Vic   f         1    84.7   51.5     75    34       68.7
## # ... with 94 more rows, and 4 more variables: earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

`arrange()` can alphabetize for us too. This is useful in combination with `head()` when surveying our data.

Next up, `filter()` and `select()`. Filtering means choosing *rows* while selecting means choosing *columns*. You can filter rows based on what the rows contain, but you can only select columns by name or position.

```
filter(possum, sex == "m")
```

```
## # A tibble: 61 x 15
##    X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
##  1 C3        1     1 Vic   m         8    94.1   60.4     89    36       74.5
##  2 C26       7     1 Vic   m         2    95.3   58.2     89.5  36       71.5
##  3 C36      13     1 Vic   m         5    95.1   59.9     89.5  36       71
##  4 C37      14     1 Vic   m         3    95.4   57.6     91.5  36       74.3
##  5 C39      15     1 Vic   m         5    92.9   57.6     85.5  34       69.7
##  6 C40      16     1 Vic   m         4    91.6   56       86    34.5     73
##  7 C47      18     1 Vic   m         2    93.5   55.7     90    36       73.7
##  8 C55      22     1 Vic   m         3    96.3   58.5     91    39.5     73.5
##  9 C59      24     1 Vic   m         2    94.4   54.9     84    34       75
## 10 C60      25     1 Vic   m         3    95.8   58.5     91.5  35.5     72.3
## # ... with 51 more rows, and 4 more variables: earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

```
filter(possum, taill>38)
```

```
## # A tibble: 22 x 15
##    X1     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth
##    <chr> <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>
##  1 C10       3     1 Vic   f         6    94     60       95.5  39       75.4
##  2 C32      11     1 Vic   f         9    93.3   57.2     89.5  39       77.2
##  3 C54      21     1 Vic   f         3    95.9   58.1     96.5  39.5     77.9
##  4 C55      22     1 Vic   m         3    96.3   58.5     91    39.5     73.5
##  5 A4       32     1 Vic   f         4    94.3   56.7     94    39       74.8
##  6 WW6      52     3 other m         6    97.6   61       93.5  40       67.9
##  7 BR1      54     4 other m         7    96.9   63       91.5  43       71.3
##  8 BR4      57     4 other f         4    95.1   59.4     93    41       67.2
##  9 BR5      58     4 other m         3    94.5   64.2     91    39       66.5
## 10 BR6      59     4 other m         2   102.    62.8     96    40       73.2
## # ... with 12 more rows, and 4 more variables: earconch <dbl>, eye <dbl>,
## #   chest <dbl>, belly <dbl>
```

```
select(possum, case, site, age)
```

```
## # A tibble: 104 x 3
##     case  site   age
##    <dbl> <dbl> <dbl>
##  1     1     1     8
##  2     2     1     6
##  3     3     1     6
##  4     4     1     6
##  5     5     1     2
##  6     6     1     1
##  7     7     1     2
##  8     8     1     6
##  9     9     1     9
## 10    10     1     6
## # ... with 94 more rows
```

```
select(possum, 1:5)
```

```
## # A tibble: 104 x 5
##    X1     case  site Pop   sex
##    <chr> <dbl> <dbl> <chr> <chr>
##  1 C3        1     1 Vic   m
##  2 C5        2     1 Vic   f
##  3 C10       3     1 Vic   f
##  4 C15       4     1 Vic   f
##  5 C23       5     1 Vic   f
##  6 C24       6     1 Vic   f
##  7 C26       7     1 Vic   m
##  8 C27       8     1 Vic   f
##  9 C28       9     1 Vic   f
## 10 C31      10     1 Vic   f
## # ... with 94 more rows
```

```r
select(possum, -X1)
```

```
## # A tibble: 104 x 14
##     case  site Pop   sex     age hdlngth skullw totlngth taill footlgth earconch
##    <dbl> <dbl> <chr> <chr> <dbl>   <dbl>  <dbl>    <dbl> <dbl>    <dbl>    <dbl>
##  1     1     1 Vic   m         8    94.1   60.4     89    36       74.5     54.5
##  2     2     1 Vic   f         6    92.5   57.6     91.5  36.5     72.5     51.2
##  3     3     1 Vic   f         6    94     60       95.5  39       75.4     51.9
##  4     4     1 Vic   f         6    93.2   57.1     92    38       76.1     52.2
##  5     5     1 Vic   f         2    91.5   56.3     85.5  36       71       53.2
##  6     6     1 Vic   f         1    93.1   54.8     90.5  35.5     73.2     53.6
##  7     7     1 Vic   m         2    95.3   58.2     89.5  36       71.5     52
##  8     8     1 Vic   f         6    94.8   57.6     91    37       72.7     53.9
##  9     9     1 Vic   f         9    93.4   56.3     91.5  37       72.4     52.9
## 10    10     1 Vic   f         6    91.8   58       89.5  37.5     70.9     53.4
## # ... with 94 more rows, and 3 more variables: eye <dbl>, chest <dbl>,
## #   belly <dbl>
```

6.3.1 Exercise

- From the possum data, create a table which only has the site, sex, and age data for female possums under the age of 5. Arrange this table from youngest to oldest.

6.4 join() ME!

The tidyverse comes with a number of very useful functions to join related data frames (called relational data). These functions all work with two data frames at a time and fall into two main categories:

Mutating Joins

- left_join(), right_join()
- inner_join()
- full_join()

Filtering Joins

- semi_join()
- anti_join()

To demonstrate how joins work, we will need some relational data sets. Relational data is data which shares some overlapping 'key' variable (or variables) but each data set has some data which the other does not. Joining is a way of combining these data sets - but there are a surprising number of ways to do this.

Here are two very small data sets we can start with:

```
## # A tibble: 7 x 5
##   name     smoker gender diagnosis  dose
##   <chr>    <chr>  <chr>  <chr>     <dbl>
## 1 Magneto  yes    male   diabetes    250
## 2 Storm    yes    female diabetes    500
## 3 Mystique no     female IBD         400
## 4 Batman   yes    male   arthritis   325
## 5 Joker    no     male   arthritis   650
## 6 Catwoman yes    female IBD         400
## 7 Hellboy  no     male   lupus        50
```

```
## # A tibble: 4 x 2
##   diagnosis    medication
##   <chr>        <chr>
## 1 diabetes     metformin
## 2 arthritis    tylenol
## 3 IBD          5-ASA
## 4 Hypertension ramipril
```

We have doses for medications in the first table, but we don't know which medications each patient is taking!

Let's join the two tables using `left_join()`:

```r
left_join(patients, medications)
```

```
## # A tibble: 7 x 6
##   name     smoker gender diagnosis  dose medication
##   <chr>    <chr>  <chr>  <chr>     <dbl> <chr>
## 1 Magneto  yes    male   diabetes    250 metformin
## 2 Storm    yes    female diabetes    500 metformin
## 3 Mystique no     female IBD         400 5-ASA
## 4 Batman   yes    male   arthritis   325 tylenol
## 5 Joker    no     male   arthritis   650 tylenol
## 6 Catwoman yes    female IBD         400 5-ASA
## 7 Hellboy  no     male   lupus        50 <NA>
```

Notice that we get a message - the function has figured out which column is the 'key' column - the column which is present in both data sets. Sometimes, though, we don't want to leave this to chance. Better to specify this using the 'by' argument.

```r
left_join(patients, medications, by = "diagnosis")
```

We get `NA` in our output because we don't have a full set of medication data for each diagnosis.

`left_join()` keeps all the rows in the first (left hand) table and adds columns from the second. `right_join()` does the opposite:

```
right_join(patients, medications, by = "diagnosis")
```

```
## # A tibble: 7 x 6
##   name     smoker gender diagnosis     dose medication
##   <chr>    <chr>  <chr>  <chr>        <dbl> <chr>
## 1 Magneto  yes    male   diabetes       250 metformin
## 2 Storm    yes    female diabetes       500 metformin
## 3 Batman   yes    male   arthritis      325 tylenol
## 4 Joker    no     male   arthritis      650 tylenol
## 5 Mystique no     female IBD            400 5-ASA
## 6 Catwoman yes    female IBD            400 5-ASA
## 7 <NA>     <NA>   <NA>   Hypertension    NA ramipril
```

Notice now that we lost some information about Hellboy because his diagnosis wasn't in the medications table.

Usually we use `left_join()`, but sometimes when we are piping (we'll get to that soon), `right_join()` comes in handy.

`full_join()` keeps all the rows from both data sets, again introducing NA as needed:

```
full_join(patients, medications, by = "diagnosis")
```

```
## # A tibble: 8 x 6
##   name     smoker gender diagnosis     dose medication
##   <chr>    <chr>  <chr>  <chr>        <dbl> <chr>
## 1 Magneto  yes    male   diabetes       250 metformin
## 2 Storm    yes    female diabetes       500 metformin
## 3 Mystique no     female IBD            400 5-ASA
## 4 Batman   yes    male   arthritis      325 tylenol
## 5 Joker    no     male   arthritis      650 tylenol
## 6 Catwoman yes    female IBD            400 5-ASA
## 7 Hellboy  no     male   lupus           50 <NA>
## 8 <NA>     <NA>   <NA>   Hypertension    NA ramipril
```

And finally, `inner_join()` keeps only rows which are common to both data sets:

```
inner_join(patients, medications, by = "diagnosis")
```

```
## # A tibble: 6 x 6
##   name     smoker gender diagnosis  dose medication
##   <chr>    <chr>  <chr>  <chr>     <dbl> <chr>
## 1 Magneto  yes    male   diabetes    250 metformin
## 2 Storm    yes    female diabetes    500 metformin
## 3 Mystique no     female IBD         400 5-ASA
## 4 Batman   yes    male   arthritis   325 tylenol
## 5 Joker    no     male   arthritis   650 tylenol
## 6 Catwoman yes    female IBD         400 5-ASA
```

These four functions are called mutating joins because they add columns. We'll meet `mutate()` soon and you will see why that makes sense.

The other main type of join are filtering joins. We just met `filter()` so we remember that filtering is a way of choosing rows. Filtering joins only ever remove rows. No new columns get introduced.

`semi_join()` keeps all the rows in the first table which have a match in the second table:

```
semi_join(patients, medications, by = "diagnosis")
```

```
## # A tibble: 6 x 5
##   name     smoker gender diagnosis  dose
##   <chr>    <chr>  <chr>  <chr>     <dbl>
## 1 Magneto  yes    male   diabetes    250
## 2 Storm    yes    female diabetes    500
## 3 Mystique no     female IBD         400
## 4 Batman   yes    male   arthritis   325
## 5 Joker    no     male   arthritis   650
## 6 Catwoman yes    female IBD         400
```

Whereas `anti_join()` *drops* all the rows in the first table which have a match in the second table:

```r
anti_join(patients, medications, by = "diagnosis")
```

```
## # A tibble: 1 x 5
##   name    smoker gender diagnosis  dose
##   <chr>   <chr>  <chr>  <chr>     <dbl>
## 1 Hellboy no     male   lupus        50
```

Remember, these two functions only ever *remove* data.

There is a third category of two table functions. These are the set operations, and you might find them useful for the exercises. These functions assume you have two data frames with all the same variables (columns). They are pretty self-explanatory:

- intersect(x, y): return only observations in both x and y
- union(x, y): return unique observations in x and y
- setdiff(x, y): return observations in x, but not in y.

6.4.1 Exercises

1. Is the output from `right_join(patients, medications)` the same as `left_join(patients, medications)`? Decide what you think the answer is before trying it out in your R session.

2. Here's some more data:

```
## # A tibble: 6 x 3
##   patient  script_ID license
##   <chr>        <dbl>   <dbl>
## 1 Magneto         15   23546
## 2 Storm           22   23546
## 3 Mystique         5    6536
## 4 Catwoman         4    6536
## 5 Hellboy          5   23546
## 6 Hellboy         12   67589
```

```
## # A tibble: 10 x 3
##    patient  script_ID license
##    <chr>        <dbl>   <dbl>
##  1 Magneto         17   32567
##  2 Storm           22   23546
##  3 Mystique        55    6536
##  4 Catwoman        24    6536
##  5 Hellboy         51   23546
##  6 Hellboy         12   67589
##  7 Storm           22   23546
##  8 Mystique         5    6536
##  9 Catwoman         4    6536
## 10 Hellboy          5   23546
```

```
## # A tibble: 4 x 3
##   physician license speciality
##   <chr>       <dbl> <chr>
## 1 Prof.X      23546 endocrinology
## 2 Grey         6536 gastroenterology
## 3 Frost       67589 family.practice
## 4 Agent       32567 family.practice
```

Use mutating and filtering joins to piece all the data together and figure out:

- What meds are over the counter? (No prescriptions)
- Which specialities prescribe metformin?

Note that the two scripts table contain some overlapping data - make sure to only look at each script_ID once!

You'll also need to know that if you are trying to join by columns which have different names, you can use `by = c("x" = "a")`.

6.5 A tidy way to handle strings

In lesson four we learned some useful tricks for handling strings using `str_replace()`, `str_detect()`, `oupper()`, `str_to_lower()`, and `str_split()`. These are all in the `stringr` package (included in the tidyverse) and you may find some of these come in handy when working to clean your data:

- `str_length()`
- `str_pad()`, `str_trim()`
- `str_to_upper()`, `str_to_lower()`, `str_to_title()`
- `str_order()`, `str_sort()`
- `str_detect()`, `str_sub()`, `str_replace()`
- `str_count()`
- `str_split()`

These are fairly self explanatory based on the name. We can jump right into trying them and seeing how they work.

6.5.1 Exercises

Take another crack at the exercises from lesson four and see if you can perform the same task using tidyverse functions. Notice any differences in behaviour or ease of use?

1. Convert a full name (let's start with `name <- "William Osler"`) to an abbreviated name made up of the first three letters of the first name and the first the letters of the last name, all in lower case. For example "Stephen Master" would become "stemas".

2. Read in starwars_menu.csv. The data is the pre-selected menu for a banquet. Have a look at the data. The caterer needs to know how many individuals want "whey" but the "whey" is buried in things like "the whey, the shoots and the rice".

- How many people want "whey"?
- Produce a data frame of all the rows containing the word whey.
- Which individuals chose "whey"?
- Which individuals did not choose "whey"?

3. Use this character vector to experiment with the functions `str_order()`, `str_sort()`, `str_count()`. Can you figure out what each of these functions does? What do `str_pad()` and `str_trim()` do?

```
letters <- c("e", "r", "q", "a", "a", "f", "c", "x", "z")
```

7 Piping, Mutating, and Summarizing

7.1 Pipes

Before we go further, we are going to employ a function called the pipe which is denoted `%>%`. This is analogous to the OSX and Linux command line pipe `|` used (constantly) in shell programming. What is does is "push" the output of one function to another function which avoids repeatedly nested functions.

Here is an example. This can be hard to read when many functions are nested:

```
# traditional nested functions
summary(select(filter(possum, sex == "m"), matches("hdlngth")))
```

```
##     hdlngth
##  Min.   : 82.50
##  1st Qu.: 90.70
##  Median : 93.20
##  Mean   : 92.92
##  3rd Qu.: 95.40
##  Max.   :103.10
```

But it's easier to read with pipes:

```
# compare with piped functions
possum %>% filter(sex == "m") %>%
  select(hdlngth) %>%
  summary()
```

```
##     hdlngth
##  Min.   : 82.50
##  1st Qu.: 90.70
##  Median : 93.20
##  Mean   : 92.92
##  3rd Qu.: 95.40
##  Max.   :103.10
```

When we pipe, we start with the data we want to work with. All the functions which come next in the 'pipeline' inherit the data as it is fed to them. So we don't need to specify what data we want each function to use, as the default is to use whatever has come 'down the pipe' to that function.

If this is confusing, imagine the `%>%` operator as being the adverb 'then', with the functions being the verbs, and the data being the noun (or subject). So the above pipe can be 'read' as the sentence "Start with the possum data, then filter rows with sex equal to 'm', then select the column 'hdlength', then provide a summary."

7.2 `mutate()` and `transmute()`

The `mutate()` function permits the calculation and addition of a new column. First let's use the electrolyte results from earlier and add a column with the AG the traditional way:

```
#traditional way
lytes.wide$Anion.Gap <- lytes.wide$Sodium + lytes.wide$Potassium -
  lytes.wide$Chloride - lytes.wide$`Total CO2`
lytes.wide
```

```
## # A tibble: 4 x 6
##   Patient Sodium Potassium Chloride `Total CO2` Anion.Gap
##     <dbl>  <dbl>     <dbl>    <dbl>       <dbl>     <dbl>
## 1       1    143       4.2      110          25      12.2
## 2       2    129       4.4      100          18      15.4
## 3       3    142       4.6      105          27      14.6
## 4       4    148       5        111          25      17
```

Now try with pipes and `mutate()`

```
Anion.Gaps <- lytes.wide %>%
  mutate(Anion.Gap = Sodium+Potassium - Chloride - `Total CO2`)
Anion.Gaps
```

```
## # A tibble: 4 x 6
##   Patient Sodium Potassium Chloride `Total CO2` Anion.Gap
##     <dbl>  <dbl>     <dbl>    <dbl>       <dbl>     <dbl>
## 1       1    143       4.2      110          25      12.2
## 2       2    129       4.4      100          18      15.4
## 3       3    142       4.6      105          27      14.6
## 4       4    148       5        111          25      17
```

See how piping keeps our code cleaner and easier to type?

The `transmute()` function does the same as `mutate()` but discards all the columns in the original data frame and only returns the newly calculated column:

```
Anion.Gaps.2 <- lytes.wide %>%
  transmute(Anion.Gap = Sodium + Potassium - Chloride -`Total CO2`)
Anion.Gaps.2
```

```
## # A tibble: 4 x 1
##   Anion.Gap
##       <dbl>
## 1      12.2
## 2      15.4
## 3      14.6
## 4      17
```

If you want to retain columns with transmute, you can just name them and they will stay:

```
#keep the patient IDs
Anion.Gaps.3 <- lytes.wide %>%
  transmute(Patient,
            Anion.Gap = Sodium + Potassium - Chloride - `Total CO2`)
Anion.Gaps.3
```

```
## # A tibble: 4 x 2
##   Patient Anion.Gap
##     <dbl>     <dbl>
## 1       1      12.2
## 2       2      15.4
## 3       3      14.6
## 4        4       17
```

7.2.1 Exercise

- Load the starwars dataset and use `mutate()` to add a column to the starwars dataframe with the BMI of the character. You can simply type `data(starwars)` to get the starwars data to load.
- What is Jabba's BMI?

$$BMI = \frac{mass(kg)}{[height(cm)]^2} \times 10000$$

7.3 `separate()` and `unite()`

These are pretty easy functions to understand. The `separate()` function breaks columns up based on a separator.

```r
names <- c("Dan", "Janet","Andre", "Angela","Mari") %>% factor()
startdate <- c("04-Jul-2006", "01-Jul-2016", "01-Sep-2009", "01-Jul-2016","01-Jul-2012")
staff <- tibble(names,startdate)
staff %>%
  separate(startdate,
           into = c("StartDay", "StartMonth", "StartYear"),
           sep = "-") -> staff.sep
staff.sep
```

```
## # A tibble: 5 x 4
##   names  StartDay StartMonth StartYear
##   <fct>  <chr>    <chr>      <chr>
## 1 Dan    04       Jul        2006
## 2 Janet  01       Jul        2016
## 3 Andre  01       Sep        2009
## 4 Angela 01       Jul        2016
## 5 Mari   01       Jul        2012
```

And `unite()` does the opposite.

```r
staff.sep %>% unite(StartDate, StartDay, StartMonth, StartYear, sep = "|")
```

```
## # A tibble: 5 x 2
##   names  StartDate
##   <fct>  <chr>
## 1 Dan    04|Jul|2006
## 2 Janet  01|Jul|2016
## 3 Andre  01|Sep|2009
## 4 Angela 01|Jul|2016
## 5 Mari   01|Jul|2012
```

The first argument to unite is the name of the new column you are creating, and then list the columns you want to unite.

###Exercise Using the 'staff' data frame we created above, can you separate the 'startdate' column into colums for day, month and year while keeping the original column?

Desired output:

```
## # A tibble: 5 x 5
##   names  startdate   StartDay StartMonth StartYear
##   <fct>  <chr>       <chr>    <chr>      <chr>
## 1 Dan    04-Jul-2006 04       Jul        2006
## 2 Janet  01-Jul-2016 01       Jul        2016
## 3 Andre  01-Sep-2009 01       Sep        2009
## 4 Angela 01-Jul-2016 01       Jul        2016
## 5 Mari   01-Jul-2012 01       Jul        2012
```

Hint - to see what else a function can do, type '?' and then the function name to bring up the documentation. Often functions have hidden arguments which can do useful things.

7.4 group_by() %>% summarize()

The `summarize()` function allows you to rapidly produce summary statistics and it is usually (but not always) used with another function called `group_by()`

```
summarize(possum, m.skullw = mean(skullw), sd.skullw = sd(skullw), n = n())
```

```
## # A tibble: 1 x 3
##   m.skullw sd.skullw     n
##      <dbl>     <dbl> <int>
## 1     56.9      3.11   104
```

Used by itself, this is not all that useful or interesting but when paired with `group_by()` it is very useful.

7.4.1 Using group_by() %>% summarize()

When `summarize()` is combined with grouping by factors, we get very rapid summaries that would be traditionally more cumbersome.

They can be used to produce counts:

```
#group_by summarize
iris %>%
  group_by(Species) %>%
  summarize(count = n())
```

```
## # A tibble: 3 x 2
##   Species    count
##   <fct>      <int>
## 1 setosa        50
## 2 versicolor    50
## 3 virginica     50
```

They can be used to calculate summary statistics:

```
#demonstrate group_by summarize with iris in data wide
iris %>%
  group_by(Species) %>%
  summarize(m.Sepal.Length = mean(Sepal.Length),
            m.Sepal.Width = mean(Sepal.Width),
            m.Petal.Length = mean(Petal.Length),
            m.Petal.Width = mean(Petal.Width))
```

```
## # A tibble: 3 x 5
##   Species    m.Sepal.Length m.Sepal.Width m.Petal.Length m.Petal.Width
##   <fct>                <dbl>         <dbl>          <dbl>         <dbl>
## 1 setosa                5.01          3.43           1.46         0.246
## 2 versicolor            5.94          2.77           4.26         1.33
## 3 virginica             6.59          2.97           5.55         2.03
```

There are special summarize functions to do the same thing in one line:

```
#demonstrate the compactness of summarize_all()
iris %>%
  group_by( Species) %>%
  summarize_all(funs(mean)) -> iris.res1
iris.res1
```

```
## # A tibble: 3 x 5
##   Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>             <dbl>       <dbl>        <dbl>       <dbl>
## 1 setosa             5.01        3.43         1.46       0.246
## 2 versicolor         5.94        2.77         4.26       1.33
## 3 virginica          6.59        2.97         5.55       2.03
```

```
#multiple functions
iris %>%
  group_by(Species) %>%
  summarize_all(funs(mean, median, sd)) -> iris.res2
iris.res2
```

```
## # A tibble: 3 x 13
##   Species Sepal.Length_me~ Sepal.Width_mean Petal.Length_me~ Petal.Width_mean
##   <fct>              <dbl>            <dbl>            <dbl>            <dbl>
## 1 setosa              5.01             3.43             1.46            0.246
## 2 versic~             5.94             2.77             4.26            1.33
## 3 virgin~             6.59             2.97             5.55            2.03
## # ... with 8 more variables: Sepal.Length_median <dbl>,
## #   Sepal.Width_median <dbl>, Petal.Length_median <dbl>,
## #   Petal.Width_median <dbl>, Sepal.Length_sd <dbl>, Sepal.Width_sd <dbl>,
## #   Petal.Length_sd <dbl>, Petal.Width_sd <dbl>
```

The summary can be applied to specific columns:

```
#specific columns
iris %>%
  group_by(Species) %>%
  summarize_at(c("Petal.Length","Petal.Width"),
               funs(mean, median, sd)) -> iris.res3
iris.res3
```

```
## # A tibble: 3 x 7
##   Species Petal.Length_me~ Petal.Width_mean Petal.Length_me~ Petal.Width_med~
##   <fct>              <dbl>            <dbl>            <dbl>            <dbl>
## 1 setosa              1.46            0.246             1.5               0.2
## 2 versic~             4.26            1.33              4.35              1.3
## 3 virgin~             5.55            2.03              5.55              2
## # ... with 2 more variables: Petal.Length_sd <dbl>, Petal.Width_sd <dbl>
```

7.4.2 Using group_by() %>% summarize() with data in long format

But if we put the data in a datalong format, we get real horsepower and this can also be used to generate quick visualizations by factors as we will show later.

```
iris %>%
  pivot_longer(1:4,
               names_to = "Flower.Part",
               values_to = "Length") -> iris.long

#find the means
iris.long %>%
  group_by(Species, Flower.Part) %>%
  summarize(mean = mean(Length)) -> iris.res4
iris.res4
```

```
## # A tibble: 12 x 3
## # Groups:   Species [3]
##    Species    Flower.Part   mean
##    <fct>      <chr>        <dbl>
##  1 setosa     Petal.Length 1.46
##  2 setosa     Petal.Width  0.246
##  3 setosa     Sepal.Length 5.01
##  4 setosa     Sepal.Width  3.43
##  5 versicolor Petal.Length 4.26
##  6 versicolor Petal.Width  1.33
##  7 versicolor Sepal.Length 5.94
##  8 versicolor Sepal.Width  2.77
##  9 virginica  Petal.Length 5.55
## 10 virginica  Petal.Width  2.03
## 11 virginica  Sepal.Length 6.59
## 12 virginica  Sepal.Width  2.97
```

```
#find a bunch of summary stats
iris.long %>%
  group_by(Species, Flower.Part) %>%
  summarize_all(list(min, mean, sd, median, IQR, max)) -> iris.res5
iris.res5
```

```
## # A tibble: 12 x 8
## # Groups:   Species [3]
##    Species    Flower.Part    fn1   fn2   fn3   fn4   fn5   fn6
##    <fct>      <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 setosa     Petal.Length   1    1.46 0.174 1.5   0.175   1.9
##  2 setosa     Petal.Width    0.1 0.246 0.105 0.2   0.100   0.6
##  3 setosa     Sepal.Length   4.3 5.01  0.352 5     0.4     5.8
##  4 setosa     Sepal.Width    2.3 3.43  0.379 3.4   0.475   4.4
##  5 versicolor Petal.Length   3   4.26  0.470 4.35  0.600   5.1
##  6 versicolor Petal.Width    1   1.33  0.198 1.3   0.3     1.8
##  7 versicolor Sepal.Length   4.9 5.94  0.516 5.9   0.7     7
##  8 versicolor Sepal.Width    2   2.77  0.314 2.8   0.475   3.4
##  9 virginica  Petal.Length   4.5 5.55  0.552 5.55  0.775   6.9
## 10 virginica  Petal.Width    1.4 2.03  0.275 2     0.500   2.5
## 11 virginica  Sepal.Length   4.9 6.59  0.636 6.5   0.675   7.9
## 12 virginica  Sepal.Width    2.2 2.97  0.322 3     0.375   3.8
```

Note that all the summarise/summarize verbs work with either the US or UK/Canadian spelling.

7.5 Putting all the pipes together

7.5.1 Exercise

Using pipes and the various tidyverse 'verbs' (aka functions) you've learned in the last two lessons, write a pipe which starts with the dataframe 'possum' from earlier and returns a dataframe which gives us the number of female possums and the mean age and median foot to tail length ratio for just the female possums, grouped by research site and population? Can you then make the resulting table 'tidy'?

Start with this:

```
possum.f.sum<- possum %>%
  filter() %>%
  mutate() %>%
```

And add what you need.

Desired output:

```
## # A tibble: 21 x 4
## # Groups:   site [7]
##     site Pop   Metric               Value
##    <dbl> <chr> <chr>                <dbl>
##  1     1 Vic   mean_age              4.42
##  2     2 Vic   mean_age              2.6
##  3     3 other mean_age              4.33
##  4     4 other mean_age              3
##  5     5 other mean_age              4.5
##  6     6 other mean_age              3.5
##  7     7 other mean_age              3.5
##  8     1 Vic   median_foot_to_tail   1.99
##  9     2 Vic   median_foot_to_tail   2.03
## 10     3 other median_foot_to_tail   1.77
## # ... with 11 more rows
```

8 Lubridate

8.1 A sign of the times:lubridate()

The lubridate() package allows us to deal with dates and times and do algebra on them as we would with other vectors. This represents a major advantage over the handling of dates using base R packages.

Lubridate makes logical assumptions about what you probably mean based on typical date formats.

The functions that lubridate uses are mdy(), ymd() and dmy(). They have very predictable behavior.

```
library(lubridate)
# May need to change locale to English US if errors are seen
# Sys.setlocale("LC_TIME", "en_US.UTF-8") #OSX
# Sys.setlocale("LC_TIME", "English") #Windows
mdy("Aug-20,1755")
```

```
## [1] "1755-08-20"
```

```r
mdy("Aug/20/1755")
```

```
## [1] "1755-08-20"
```

```r
mdy("08-20-1755")
```

```
## [1] "1755-08-20"
```

```r
mdy("08201755")
```

```
## [1] "1755-08-20"
```

```r
mdy("August 20, 1755")
```

```
## [1] "1755-08-20"
```

```r
mdy("August 20 1755")
```

```
## [1] "1755-08-20"
```

```r
#all work perfectly without any explicit statements about format.
```

You can also change the language to suit your preferences. For example, if we want to work in French dates.

```r
dmy("Lundi le 12 Septembre, 2016") #gives us an error
```

```
## [1] NA
```

```r
Sys.setlocale("LC_TIME", "fr_CA.UTF-8") #change settings to French Canadian
```

```
## [1] ""
```

```r
dmy("Lundi le 12 Septembre, 2016")
```

```
## [1] NA
```

```r
Sys.setlocale("LC_TIME", "en_US.UTF-8") #change back to English USA
```

```
## [1] "en_US.UTF-8"
```

The real gold with lubridate is how it can handle times too!

```r
#calculating the number of days since the Declaration of Independence
then <- mdy_hms("July 04,  1776 14:32:45")
then
```

```
## [1] "1776-07-04 14:32:45 UTC"
```

```r
now <- ymd_hms(Sys.time())
now
```

```
## [1] "2020-10-13 19:48:24 UTC"
```

```r
delta <- difftime(now, then, units = "days")
delta #wow
```

```
## Time difference of 89220.22 days
```

```r
#calculating the number of days Canada has been a country
then <- mdy_hms("July 01,  1867 11:17:21")
then
```

```
## [1] "1867-07-01 11:17:21 UTC"
```

```r
now <- ymd_hms(Sys.time())
now
```

```
## [1] "2020-10-13 19:48:24 UTC"
```

```r
delta <- difftime(now, then, units = "days")
delta
```

```
## Time difference of 55987.35 days
```

8.1.1 Exercise

1. Read the file Potassium.csv into a variable called `K.data`. This file contains real K^+ data extracted from SunQuest from Dan's lab for one month from the two ER bays. It contains order time, collection time, receive time and result time. Use the `head()` function to get an idea of how the dates and times are formatted. All the K^+ results presented were run on an ABL800 whole blood analyzer directly from an unspun PST tube.

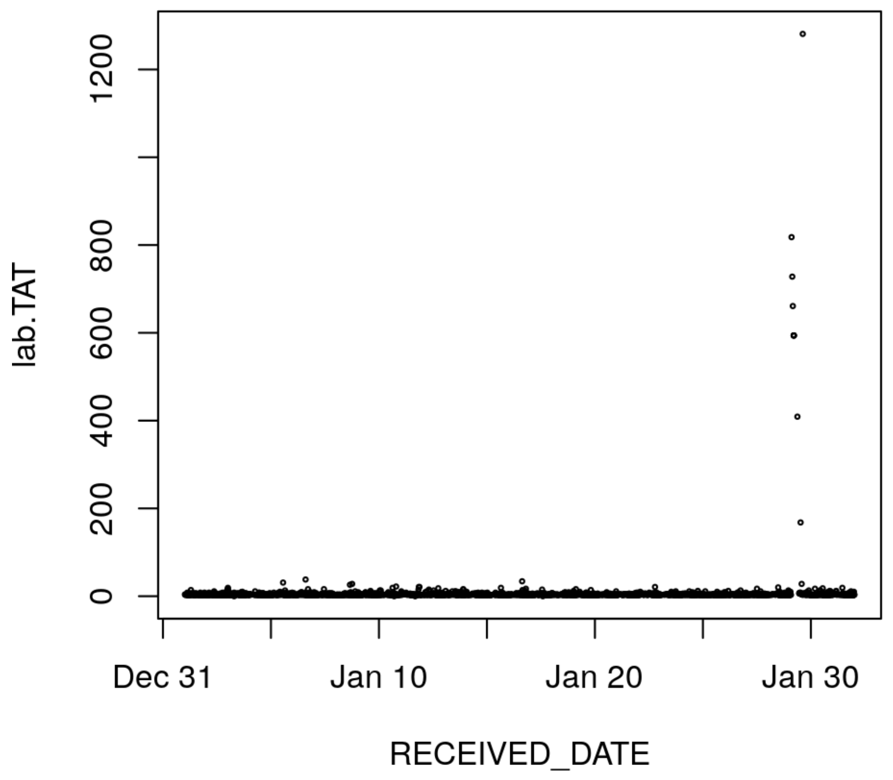
   - Convert the order, receive, and result times into dates
   - Calculate the order-to-result times and store it a column called TAT ("turn around time"). Use the function `difftime()` to calculate the time difference.
     - What is the median and IQR of the TAT?
     - What is the 90th and 99th percentiles of the TAT?
     - What is the maximum value of TAT?
   - Calculate the receive-to-result times and store it a column called lab.TAT (the analysis time).
     - What is the median and IQR of the lab.TAT?
     - What is the 90th and 99th percentiles of the lab.TAT?
     - What is the maximum value of lab.TAT? What day did it occur?
     - What strange finding have you discovered?

2. Produce a histogram of TAT and lab.TAT on the same histogram. Recall: `hist(my.vector, breaks  = 20, add = TRUE)`

   Start with your hist of TAT so that the x axis is long and can display both TAT and lab.TAT. In this particular, `breaks = 20` works nicely for TAT and `breaks = 200` works nicely for lab.TAT. This was just trial and error.

From this kind of data we can plot the daily turn around time, which can be really helpful. We won't do it now but you have the code here so you can see what is possible.

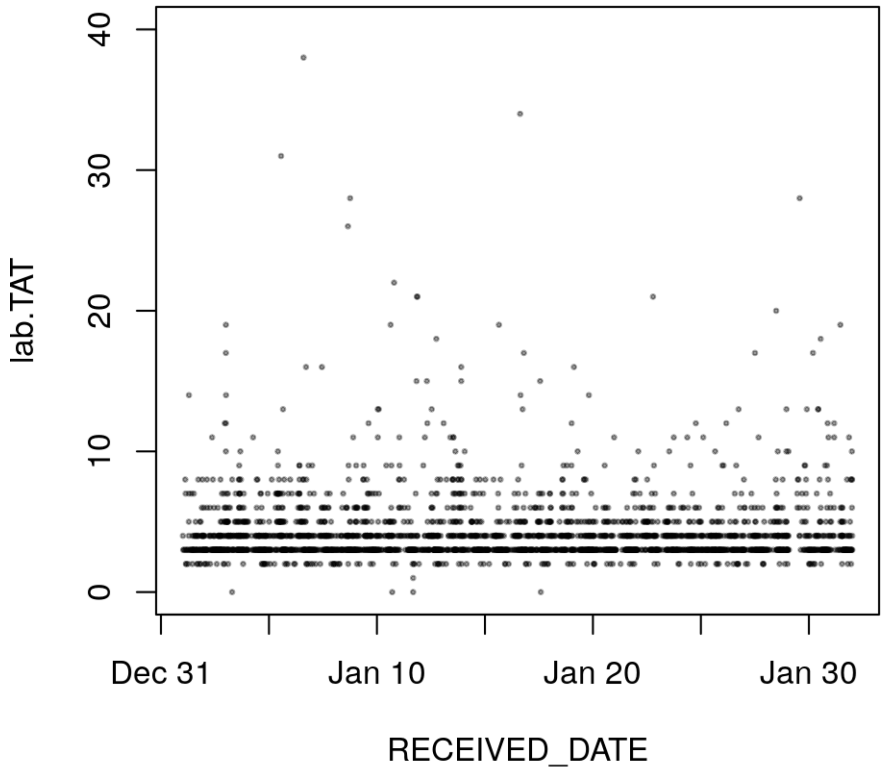
```r
plot(lab.TAT ~ RECEIVED_DATE, data = K.data, cex = 0.3)
```



```
#the LIS downtime on Jan 29 is evident
```

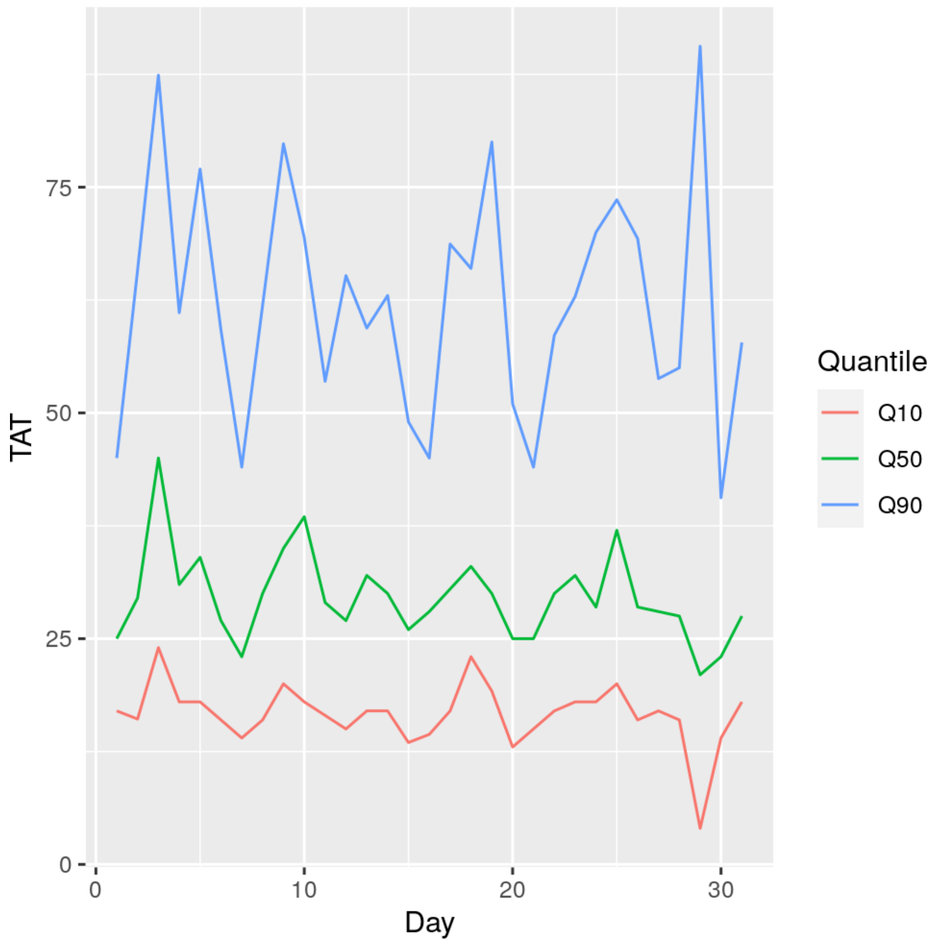
Making it a little prettier:

```
plot(lab.TAT ~ RECEIVED_DATE,
     data = K.data,
     ylim = c(0,40),
     col = "#00000060",
     pch = 19,
     cex = 0.3) #as it happens individual TATs.
```



And you can do lots of other things with this data. For demonstration purposes only, in future you will be able to plot daily statistics with some very compact syntax. Plunk this in and see what you get.

```
library(ggplot2)
library(scales)
library(tidyr)
K.data <- read_csv("Data_Files/Potassium.csv")
K.data %>%
  mutate_at(.funs = funs(dmy_hm), .vars = vars(COLLECTION_DATE:RECEIVED_DATE)) %>%
  mutate(TAT = difftime(RESULT_DATE, ORDERED_DATE, units = "min")) %>%
  group_by(Day = day(COLLECTION_DATE)) %>%
  summarize(Q10 = quantile(TAT,probs = 0.1),
            Q50 = median(TAT),
            Q90 = quantile(TAT,probs = 0.9)) %>%
  gather(key = "Quantile", value = "TAT", c("Q10","Q50","Q90")) %>%
  ggplot(aes(x = Day, y = TAT, col = Quantile)) +
  geom_line()
```



9 Functions, Conditional Responses and Loops

9.1 Making your own functions

It is frequently necessary to create a function that does a specific calculation so you do not need to repeatedly write out the same code. We are going to create some functions, save them, and reload them for future convenient use. This way, we are going to save ourselves *so much* work.

The syntax of the custom function looks like this:

```
function.name <- function(arg1,arg2,arg3 - ie input variables) {
  statements to generate an R object
  return(object)
}
```

A simple function might look like this:

```
square <- function(x) { # x is just a variable name. Any allowable variable name is OK
    result <- x^2      # calculate the square and assign to a variable "result"
    return(result)     # return the value of the result variable
}
square(5)
```

```
## [1] 25
```

Note that variables used in functions are local to the function and cannot be called outside the function. In other words, you can use the variable `x` anywhere else in your script and it will not interfere with your `square()` function. Note that if you define a variable inside your function, you will not be able to inquire as to its value after the function has been executed.

Let's look at a more complex example. We will go back to handling strings, first by using `substr()` to take a word and remove the first and last letters.

```
chopper <- function(word) {
  return(substr(word, 2, nchar(word) - 1))
}
# now try it out on some words
# will it work on a sentence?
```

9.1.1 Exercise

1. Write a function called cv that takes a vector of numbers and calculates the CV of those numbers in %. Recall that $\%CV = sd/mean \times 100$.
2. Write a function called `clean` that removes any less than or greater than signs from your data but preserves the number that follows and returns a numeric vector of your data.

Things you should be aware of when making your own functions:

- You are not limited to having variables as the thing a function does. The function can generate graphs, read or write files, and do anything that R can do.
- You can have as many input variables and output variables as you want. If you are returning multiple output variables, you can return them as a dataframe or a list.
- You can save your function as a text file and then read it back in at another time using the `source()` function.

9.2 Conditional Responses

It is very common to have to provide responses based on conditions that are not known *a priori*. For example, you might have to do something different if you have an even number of components to a vector than if you had an odd number of components. Or you might have to apply a different calculation for eGFR for men than for women. In these cases, we need to use "if" statements to make the right decision. The structure goes like this:

```
if (condition 1) {
    do something
} else if (condition 2) {
    do something else
} else {
    do something with everything that fails conditions 1 and 2
}
```

Here's a useless function that shows how `if` statements work. It takes a name and tells you if the name corresponds to either of the instructors.

```
is.teaching.now <- function(name) {
    if (name == "Dan" | name == "Daniel") {
        response <- TRUE
    } else if (name == "William" | name == "Will") {
        response <- TRUE
    } else {
        response <- FALSE
    }
    return(response)
}
#try it
is.teaching.now("Wolfgang")
```

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

```
is.teaching.now("Will")
```

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

Here is another function that tells you if a number is even.

```
iseven <- function(x) {
   if (floor(x/2) == (x/2)) {
        return(TRUE)
    } else {
        return(FALSE)
    }
}
#try it
iseven(8)
```

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

```
iseven(19)
```

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

```
iseven(pi)
```

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

```
iseven(round(2*pi,0))
```

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

9.2.1 Exercise

1. Write a function that tells you whether a number is an integer or not by using the `floor()` function. If it's an integer, have the function return the string "This number has integrity". Otherwise have it return "This number lacks integrity".
2. Write an MDRD eGFR calculator that accounts for female sex and African race. Recall:

$$eGFR = 186 \times (SCr)^{-1.154} \times Age^{0.203}$$

   which must be scaled by a factor of 0.742 for females and scaled by 1.210 for African Americans.
3. Write a function that takes a sentence and determines if the word "schnitzel" is in the sentence. It should return "This sentence contains schnitzel" or "This sentence is schnitzel-free". You may find a function called `%in%` helpful - this tells you if quantity is in a vector. For example:

```
sentence <- c("It's","a","beautiful","day","in","Salzburg")
"Paris" %in% sentence
```

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

```
"Salzburg" %in% sentence
```

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

9.3 Loops–When you need to say the same thing over and over and over

Very often we need to do the same thing over and over and perform a calculation or inquiry over an entire vector. (See also the `apply()` family of functions or the purrr package). Loops help us to accomplish things like this, and they are an essential tool in many things we will want to do.

There are two kinds of loops: the "for" loop and the "while" loop. In general we'll use "for" more often but we will cover both.

The syntax of the `for` loop goes like this:

```
for (counter in sequence) {
    do something
}
```

So, in reality, that might look like this:

```
for (i in 1:5) {
    print(c("Hi number",i))
}
```

```
## [1] "Hi number" "1"
## [1] "Hi number" "2"
## [1] "Hi number" "3"
## [1] "Hi number" "4"
## [1] "Hi number" "5"
```

The `iseven()` function that we created earlier in this chapter did not function on a vector. It only worked for single numbers. Let's make a loop to apply it to a vector of numbers.

```
x <- c(1,3,4,2,5,6,7,3,10) # defines a vector to operate on
result <- vector() # creates an empty vector to store our result in
for (i in 1:length(x)) {
    result[i] <- iseven(x[i])
}
result
```

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

```
# Seems to work.  Now let's use it to pull out the numbers.
x[result]
```

```
## [1]  4  2  6 10
```

You can also embed loops inside of loops and, where necessary, call the indices of the outer loop from within the inner loop.

```
# Examine features of irises
# Loop over the different species and find the mean of the
# 4 different petal measurements.
iris <- read_csv("Data_Files/iris.csv")
species <- unique(iris$Species)
features <- names(iris)[1:4]
mean.results <- data.frame(species = rep(NA,3),
                           m.Sepal.Length = rep(NA,3),
                           m.Sepal.Width = rep(NA,3),
                           m.Petal.Length = rep(NA,3),
                           m.Petal.Width = rep(NA,3))
for (i in 1:length(species)){
  flower.data <- subset(iris, Species == species[i])
  mean.results[i,1]<- as.character(species[i])
  for (j in 1:length(features)){
    mean.results[i,j + 1] <- mean(flower.data[,j], na.rm = TRUE)
  }
}
mean.results
```

```
##      species m.Sepal.Length m.Sepal.Width m.Petal.Length m.Petal.Width
## 1     setosa             NA            NA             NA            NA
## 2 versicolor             NA            NA             NA            NA
## 3  virginica             NA            NA             NA            NA
```

Sometimes you don't want to loop to perform a specific (pre-defined) number of tasks. Sometimes you want to loop until an event occurs. For example, you might stop when the data point of interest is identified, or when a specific time is reached, or when until the user clicks "quit" (yes–you can make a GUI in R). In situations like this when the stopping point is not well-defined, you might want to use a while loop in R.

It's pretty straight-forward syntax:

```
while (condition) {
  do something
}
```

Here's a simple example that is essentially identical to a `for` loop:

```
i <- 1
while (i <=  5) {
    print(c("This is loop number", i))
    i <- i + 1
}
```

```
## [1] "This is loop number" "1"
## [1] "This is loop number" "2"
## [1] "This is loop number" "3"
## [1] "This is loop number" "4"
## [1] "This is loop number" "5"
```

Now let's try an example where a `while` loop makes more sense. The loop generates Gaussian random numbers with a mean of 0 and a standard deviation of 1. The loop quits when a number is generated that is more than 4 standard deviations from the mean. On average, this will occur about 1 in 10,000 attempts.

```
i <- 1
z <- 0
while (abs(z) < 4) { #abs means "absolute value" & takes care of neg or pos numbers
    z <- rnorm(1, 0, 1) #generates a single random number of mean = 0 and sd = 1
    i <- i + 1
}
print(paste("It took",i,"replicates to find a Gaussian random number more than",
            "4 standard deviations from the mean"))
```

```
## [1] "It took 1259 replicates to find a Gaussian random number more than 4 standard deviations fr
```

9.3.1 Exercise

- List all possible 3 card hands from a standard card deck with no jokers. Cards can be annotated with character vectors. For example the hearts would be `hearts <- c(paste0(c(2:10),"H"),"JH","QH","KH","AH")`.

10 ggplot

So far, we've seen how to do some 'quick and dirty' plots with plotting functions like `hist()` and `plot()` which are built in to base R. The tidyverse has its own paradigm for creating graphics called `ggplot`. The advantage to using `ggplot` over base R functions is that the gpplot paradigm comes with many built in defaults to make your plots look nice without having to code too much customization. Overall, the degree to which you can customize ggplot graphics is less than in base R, but it's soo much easier. As we go through some examples, note how ggplot has automatically chosen colour schemes, scales, and axis labels for us, without us specifying any of this. Of course, we can override these, but having some usable defaults built in makes it very fast to produce nice plots.

10.1 What is ggplot?

The "gg" in "ggplot" stands for "grammar of graphics", and the basic idea is this: when you plot data, you are creating a visual representation of numeric or categorical information. The most basic example is a scatterplot; the position of a point on the x axis reflects one variable, and the position on the y axis reflects another variable. That works well for simple examples, but often we have a large number of parameters that we'd like to display. Ideally, we want a clear, flexible framework that maps arbitrary variables to arbitrary visual elements such as x position, y position, size, color, shape, transparency, etc. This would let us rapidly explore different ways of looking at our data to see what is the most helpful. `ggplot` provides this sort of framework, with a clean mapping of variables to output.

Let's illustrate this using one of R's built in data sets, `mpg` which has a variety of data about different models of cars:

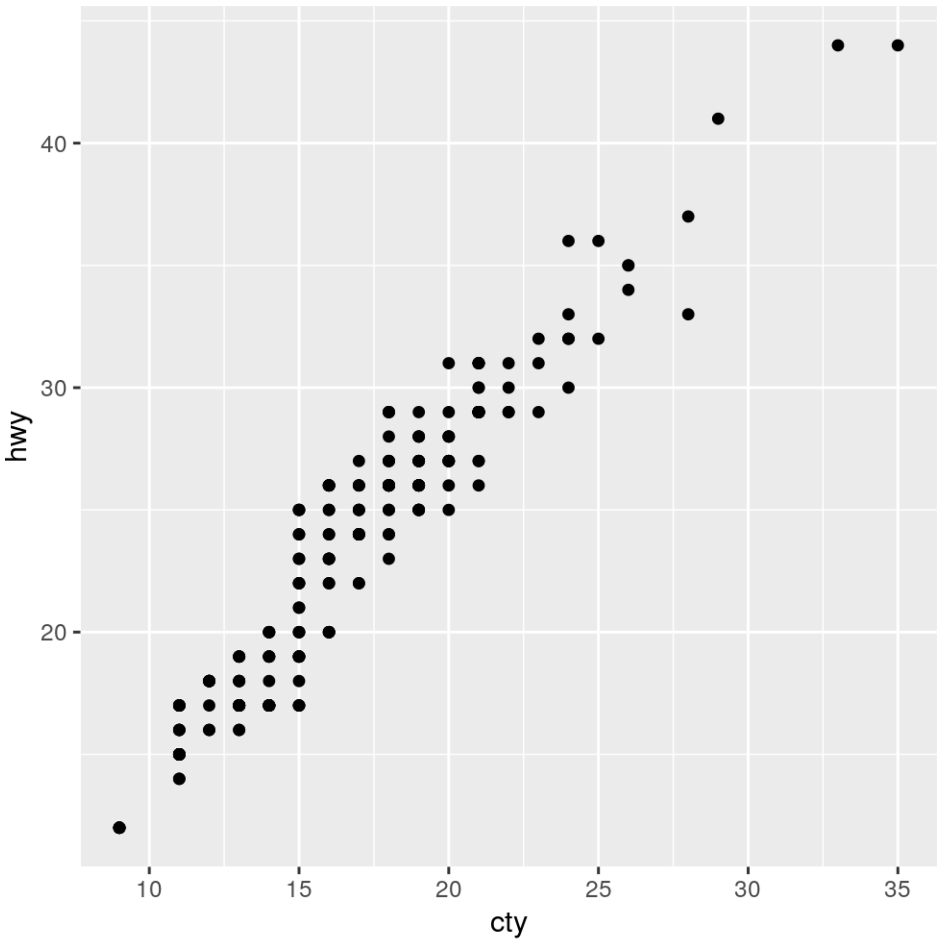
```r
glimpse(mpg)
```

```
## Observations: 234
## Variables: 11
## $ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "audi"...
## $ model        <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro"...
## $ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0,...
## $ year         <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, ...
## $ cyl          <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 8, ...
## $ trans        <chr> "auto(l5)", "manual(m5)", "manual(m6)", "auto(av)", "a...
## $ drv          <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4", "4",...
## $ cty          <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17...
## $ hwy          <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25...
## $ fl           <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p",...
## $ class        <chr> "compact", "compact", "compact", "compact", "compact",...
```

10.2 Scatterplots

We can start by plotting city verus highway fuel economy. `ggplot()` wants us to provide some data, a mapping of the data onto parameters, and a geometry with which to render that data.

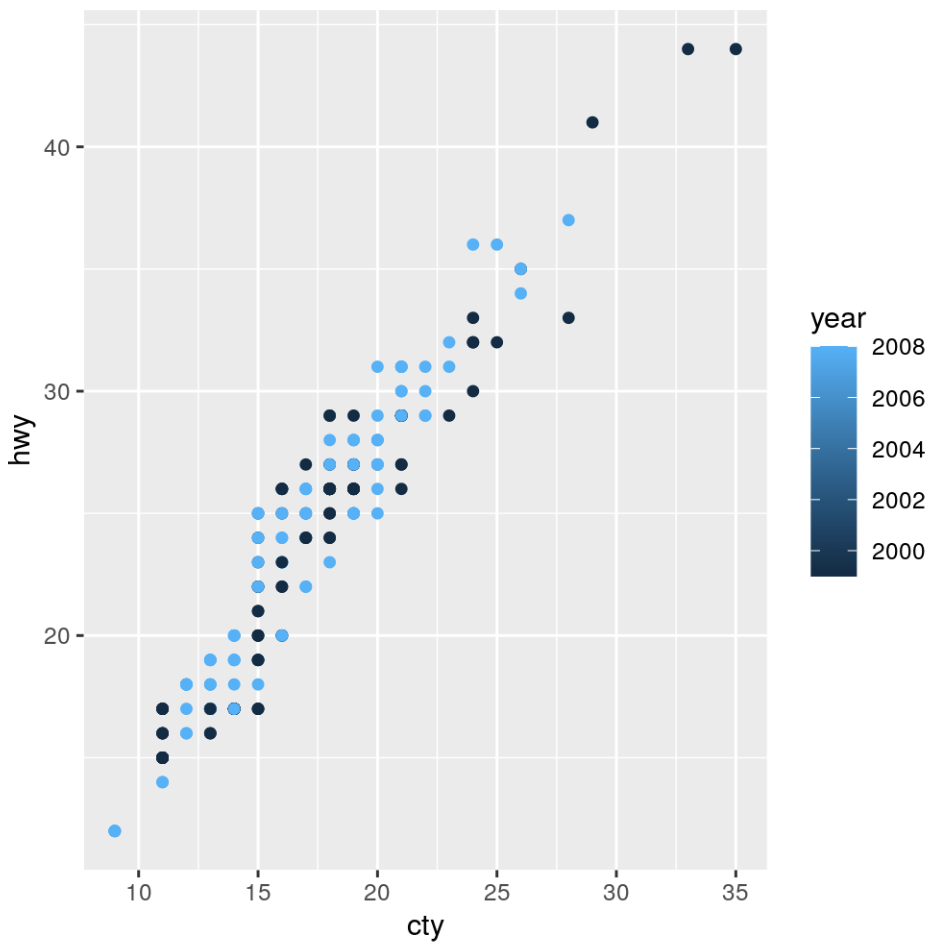
```r
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point()
```



Notice that our `ggplot()` command had three parts: data, a set of "mapping" aesthetics (x and y in this case), and a way of rendering the geometry (`geom_point`). This doesn't look like our ordinary function calls, but you can think of the `+` as saying "OK, add this geometry rendering to the plot that I just made".

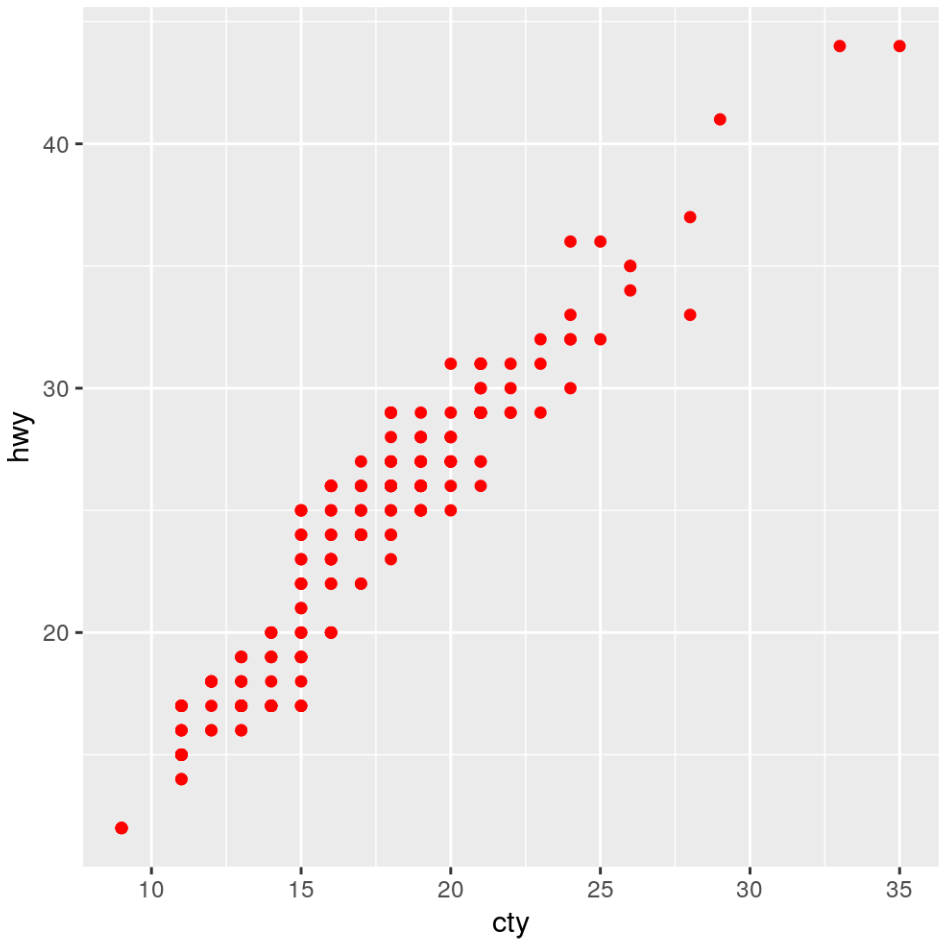
The "aesthetics" that are specified within the `aes()` call are where the real fun starts. The aesthetics for x and y can be specified in a global `aes()` call in the main `ggplot()` call, and further customization for colour, size, shape etc can be specified inside the geometry call. For example, let's look at city vs highway fuel economy again, and map the year of the model to color.

```r
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point(aes(colour = year))
```



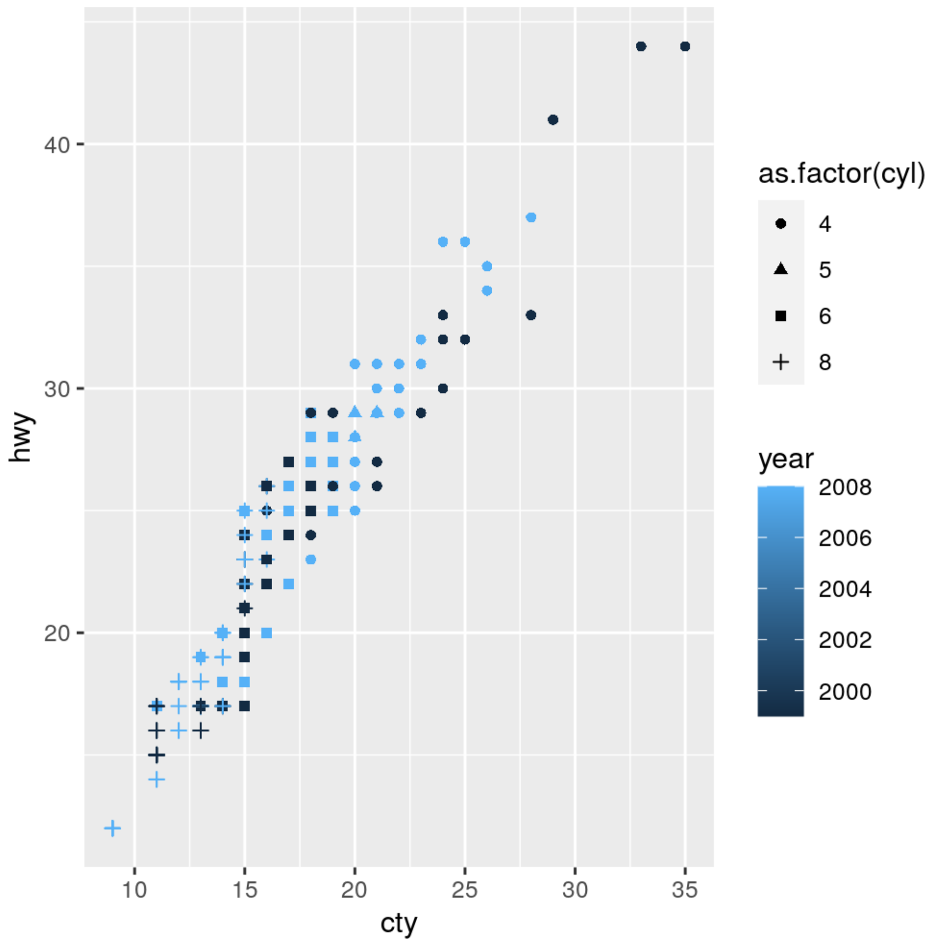
Notice how that is different than specifing colour *outside* of the `aes()` call:

```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point(colour = "red")
```



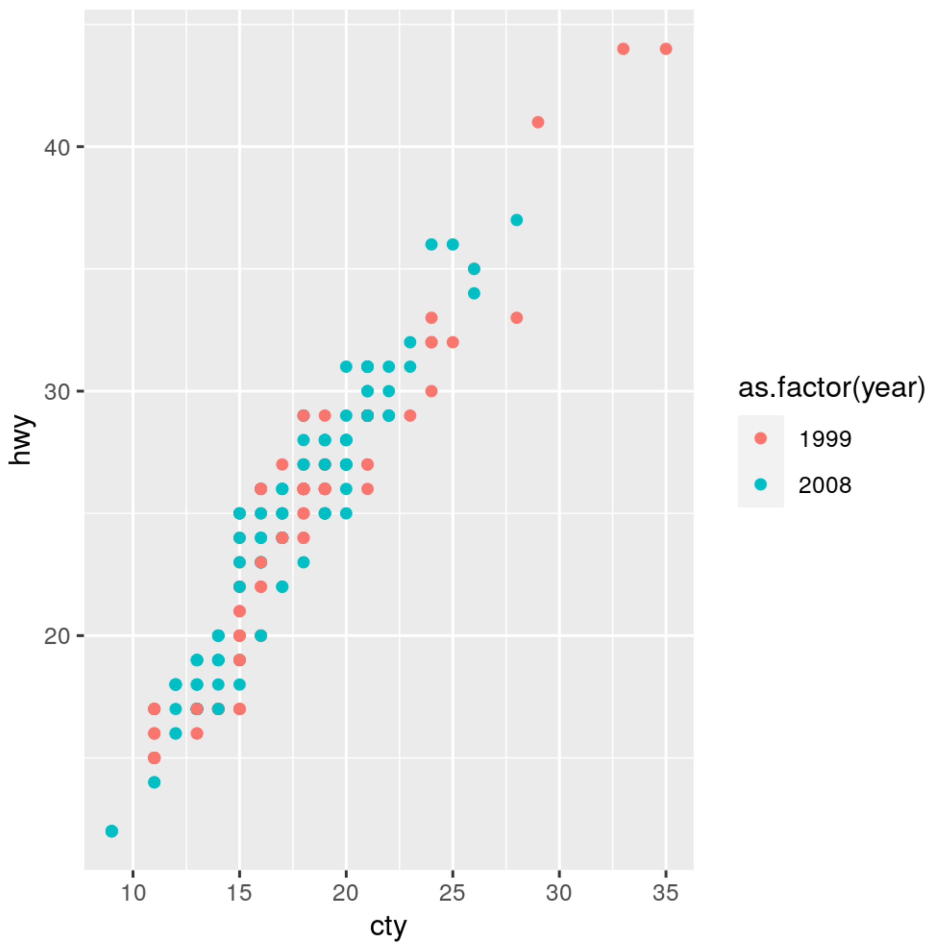
You can combine aesthetics:

```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point(aes(colour = year, shape=as.factor(cyl)))
```



Notice how ggplot handles numeric values differently from factors when it comes to colour:

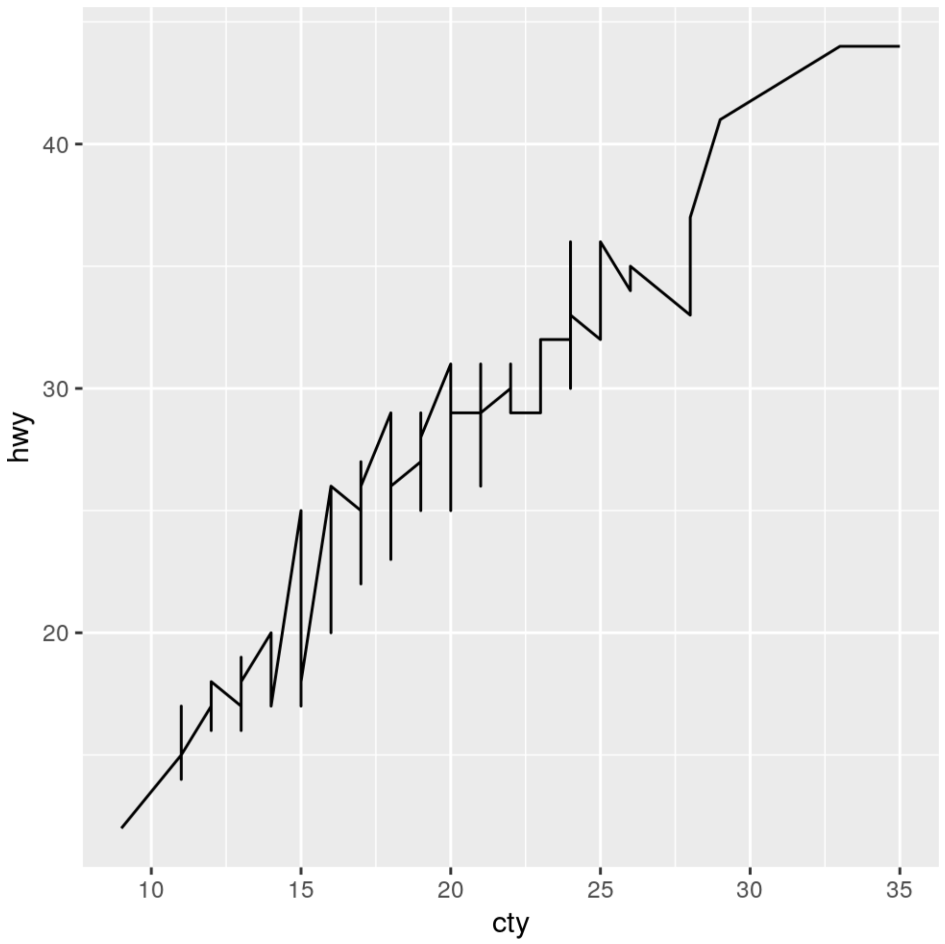
```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point(aes(colour = as.factor(year)))
```



10.3 Line Plots

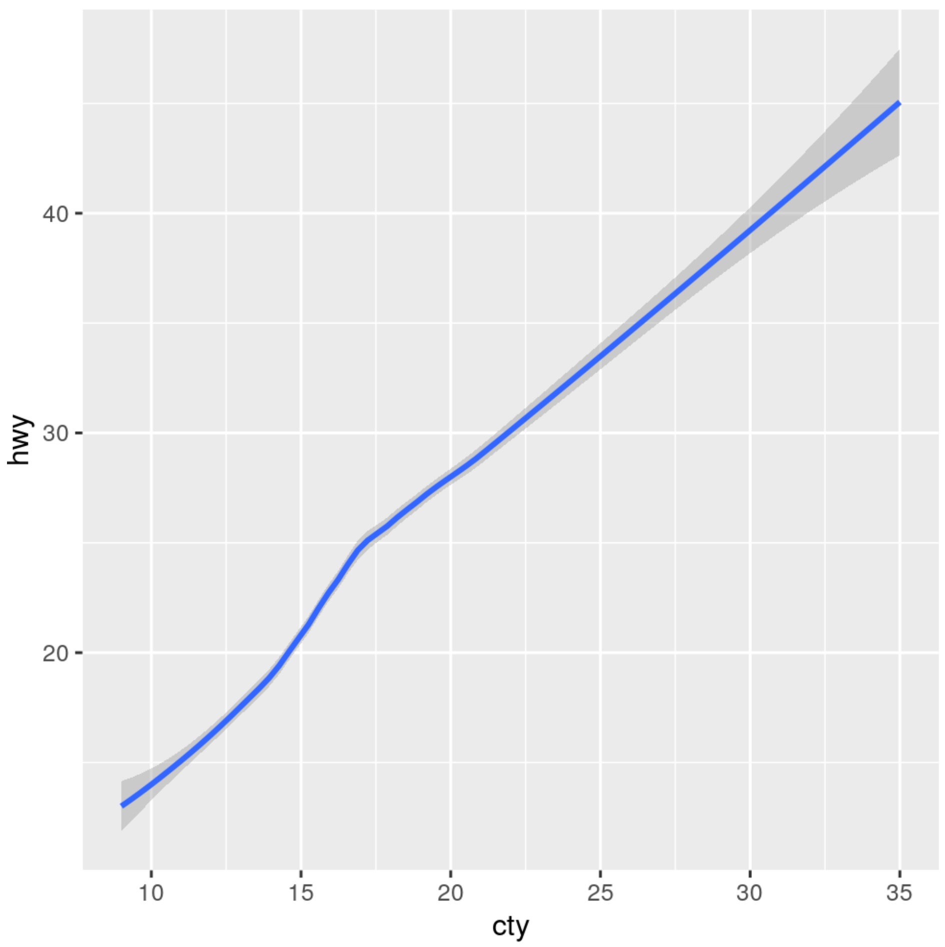
There are dozens of geometries at your disposal. Some other common useful ones are `geom_line` and `geom_smooth`.

```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_line()
```



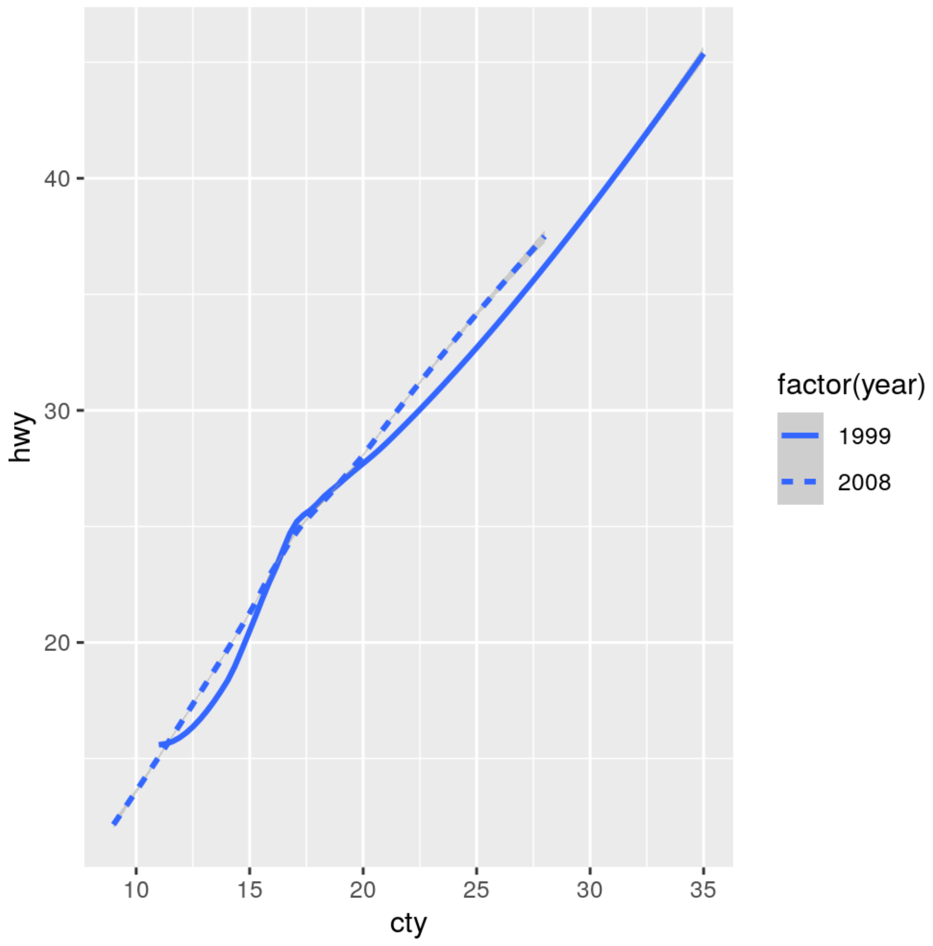
We can geom_smooth that line if it looks too choppy:

```
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_smooth()
```



And of course we can map more aesthetics:

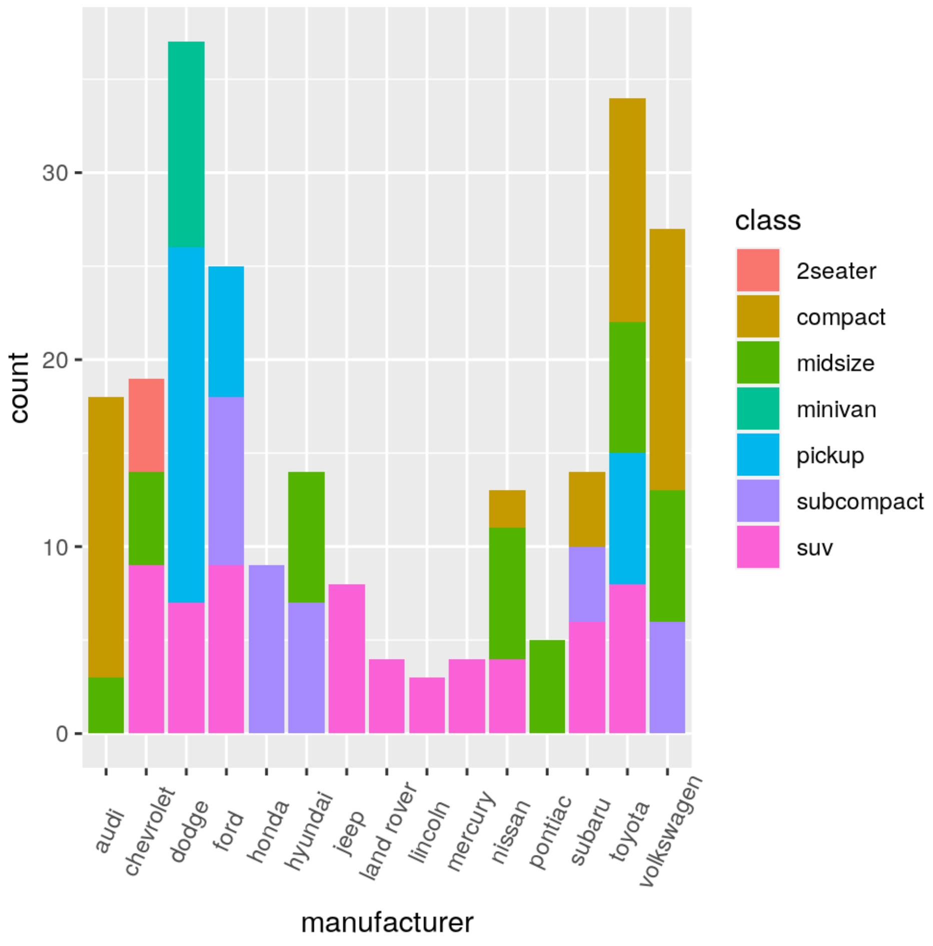
```r
ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_smooth(aes(linetype = factor(year)), level = 0.2)
```



10.4 Bar Plots

`geom_bar` has built in stats, with the default being ‘count’, for which no y aesthetic needs to be specified, as the y axis will display the number of observations for each x axis value:

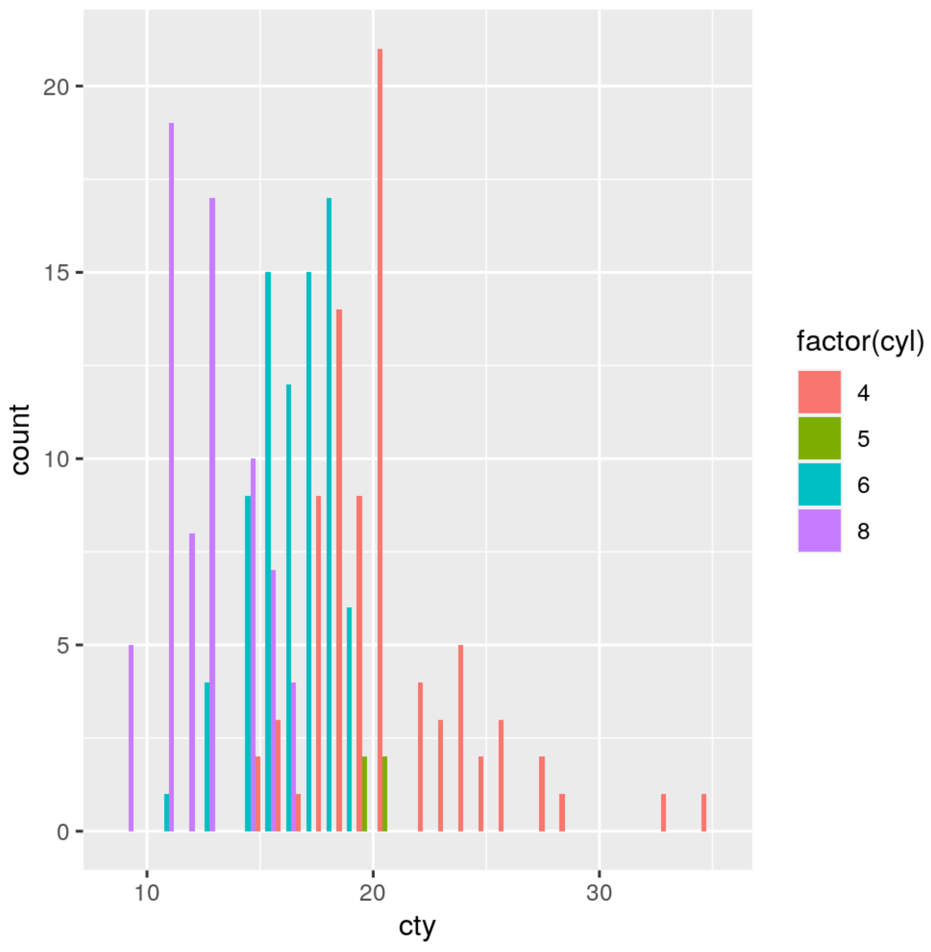
```
ggplot(mpg, aes(x = manufacturer)) +
  geom_bar(aes(fill = class)) +
  theme(axis.text.x = element_text(angle = 65, vjust = 0.6))
```



Note we used 'fill' here to colour the bars rather than 'colour' - what happens if you use 'colour' instead?

And of course, good old histograms get their own geom:

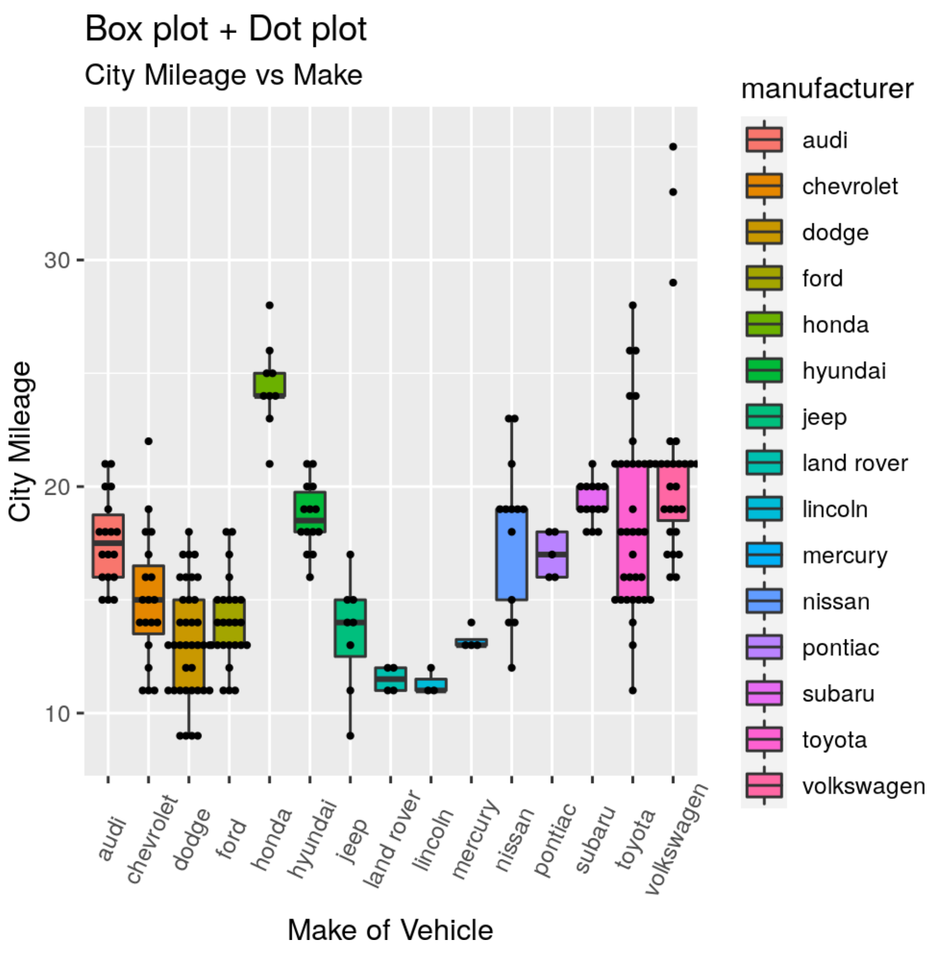
```
ggplot(mpg, aes(x = cty)) +
  geom_histogram(aes(fill = factor(cyl)), position = "dodge")
```



10.5 And more!

`ggplot()` objects can be layered and layered upon, including multiple geoms, labels, custom scales, and more. Here's just one example:

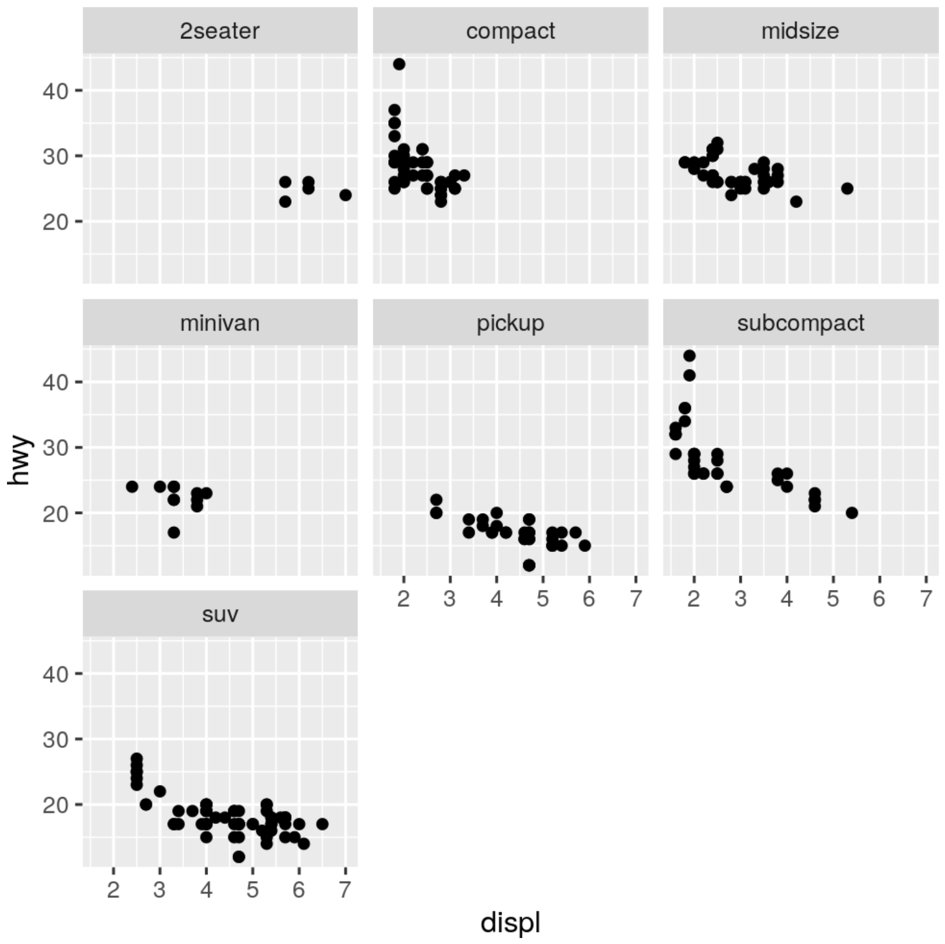
```
ggplot(mpg, aes(manufacturer, cty))+
  geom_boxplot(aes(fill = manufacturer),
               outlier.shape = NA)  +
  geom_dotplot(binaxis = 'y',
               stackdir = 'center',
               dotsize = .3,
               fill = "black",
               col = "black") +
  theme(axis.text.x = element_text(angle = 65, vjust = 0.6)) +
  labs(title="Box plot + Dot plot",
       subtitle="City Mileage vs Make",
       x = "Make of Vehicle",
       y = "City Mileage",
       legend = "")
```



10.6 Multipanel Plots

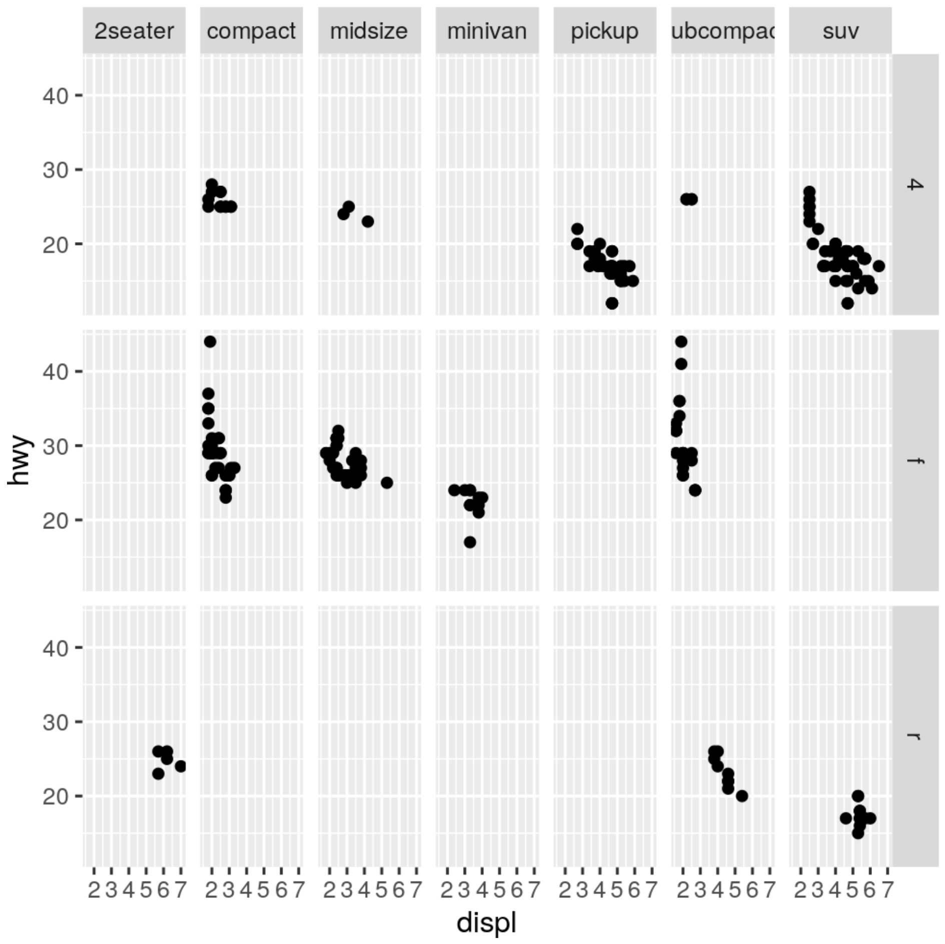
If you want to produce multi-panel plots that are based on a factor within your data set, this can be accomplished using the `facet_wrap()` function:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_wrap(~ class)
```



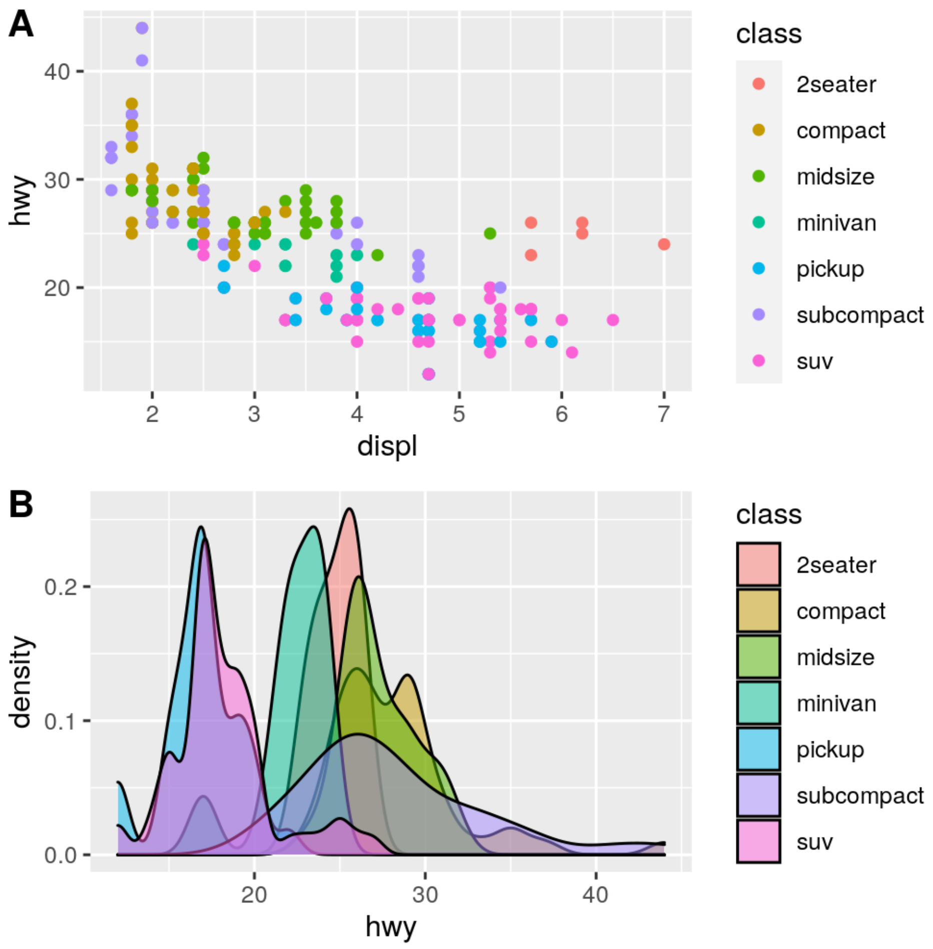
You can use `facet_grid()` if you want to produce all possible combinations of two factors in your data:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(drv~class)
```



And if you just want to glue images together that may not originate from a single `ggplot()` command, this can be achieved with the `plot_grid()` function from the cowplot package.

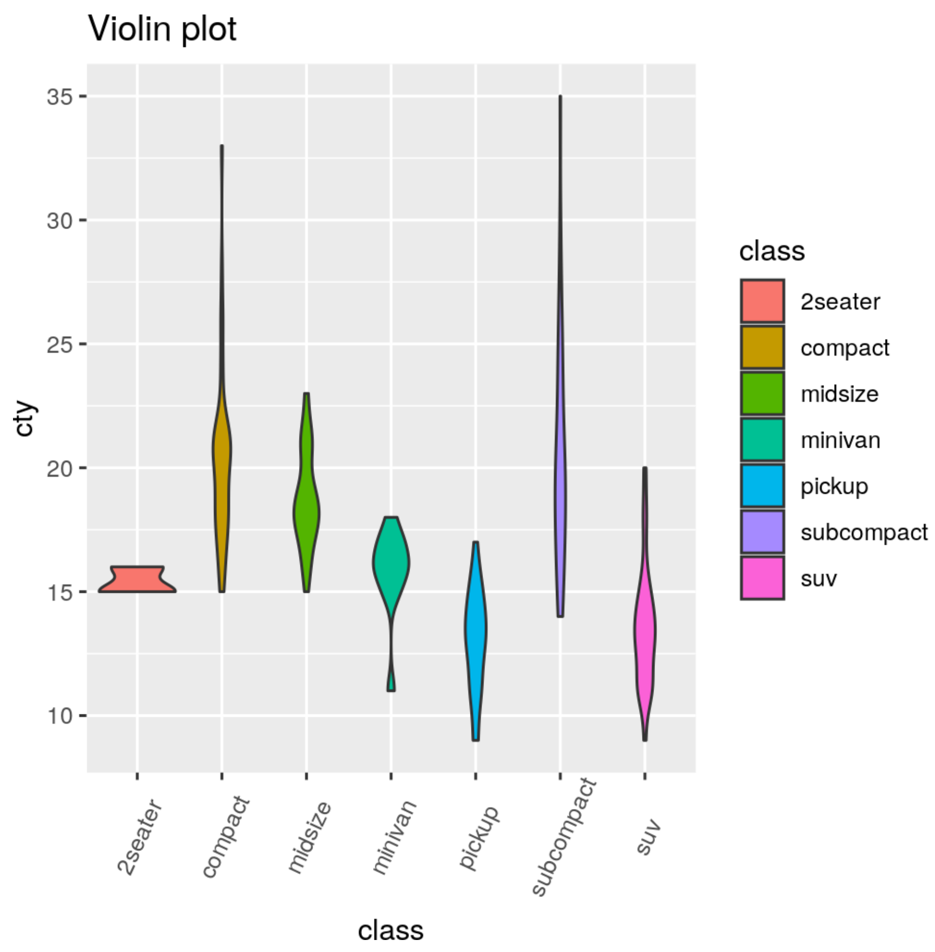
```r
library(cowplot)
p1 <- ggplot(mpg, aes(x = displ, y = hwy, colour = class)) +
  geom_point()
p2 <- ggplot(mpg, aes(x = hwy, fill = class)) +
  geom_density(alpha = 0.5)
plot_grid(p1, p2, nrow=2, labels=c('A', 'B'))
```



There is so much more you can do with `ggplot()` that we won't have time to cover. If you are looking for more ways to display data and customize your plots, check out one of the many ggplot 'cheat sheets' available online. You can also start by going through the data visialization chapter of Hadley Wickham's book R for Data Science or see the R Graphics Cookbook by Winston Chan.

10.6.1 Exercise

- Using the mpg dataset or any of the other datasets we've used so far, spend some time playing around with various geoms and aesthetics. Remember you can type `?geom_smooth` or similar to bring up the documentation in RStudio. Can you figure out how to make a "violin" plot?



10.7 Saving a plot

We have a lot of flexibility about saving our plots. We can save in bmp, jpg, png, tiff and in vector formats of svg, ps and pdf. This is very good.

Unfortunately, the syntax between Mac and Windows is a little different.

10.7.1 Saving as a jpeg , png, tiff or bmp.

Execute the following:

```
jpeg(filename = "violin_plot.jpg",  width = 10, height = 10, units = "in", res = 200)
ggplot(mpg, aes(class, cty))+
  geom_violin(aes(fill=class)) +
  labs(title="Violin plot") +
  theme(axis.text.x = element_text(angle=65, vjust=0.6))
dev.off()
```

On Macs this translates to:

```
quartz(file = "violin_plot.jpg", width = 10, height = 10, type = "jpg", dpi = 200)
ggplot(mpg, aes(class, cty))+
  geom_violin(aes(fill=class)) +
  labs(title="Violin plot") +
  theme(axis.text.x = element_text(angle=65, vjust=0.6))
dev.off()
```

If you want output that is "vector" in nature:

```
postscript(file = "violin_plot.ps", width = 10, height = 10)
ggplot(mpg, aes(class, cty))+
  geom_violin(aes(fill=class)) +
  labs(title="Violin plot") +
  theme(axis.text.x = element_text(angle=65, vjust=0.6))
dev.off()
```

You will notice that the aspect ratio needs adjustment for the ps output. You can do pdf also for example:

```
pdf(file = "violin_plot.pdf", width = 10, height = 10)
ggplot(mpg, aes(class, cty))+
  geom_violin(aes(fill=class)) +
  labs(title="Violin plot") +
  theme(axis.text.x = element_text(angle=65, vjust=0.6))
dev.off()
```

11 File Operations

You are going to need to read files in, as you have done with `read_csv()`. You are going to need to write files–whether it is the statistical output or perhaps the content of a dataframe that you have produced through your efforts. Sometimes you need to remove files or move files too. All of these file operations are possible from within R. You can also both read and write xls or xlsx files, which we will touch on also. This stuff is dry but it is necessary.

What files are in the current working directory? This does the trick, as we have discussed.

```
dir_ls()
```

If you want the figures in a specific directory, you can put it in the parentheses as in `dir_ls(path = "Name_of_Subdirectory")`. You can also provide the "full path" of a directory too.

```
dir_ls(path = "/")
# should give you all the files in the C: directory in windows
#   and the root directory in Linux and Mac OSX
```

Let's suppose we want to see if a file is present–so that we don't try to perform an operation on a non-existent file, or to ensure that some other event is complete. We can ask as follows:

```
file_exists("Data_Files/Potassium.csv")
```

```
## Data_Files/Potassium.csv
##                     TRUE
```

```
file_exists("Data_Files/Data_Set_3.1459.csv")
```

```
## Data_Files/Data_Set_3.1459.csv
##                          FALSE
```

We might want to create a directory and then move or copy some files into it.

```r
dir_create("junk_here")

if (file_exists("junk_here")) {
  file_copy("Data_Files/Potassium.csv", "junk_here/bla.csv")
} else {
  print("Hey -- That directory did not get created")
}
```

Note that if you provide `file_copy()` a vector of file names, `file_copy()` will copy all the files in the vector.

11.0.1 Exercise

1. You have a number of .csv files used in this course in the directory "Data_Files". Create a new directory called "Data_Files_Copy" and use `file_copy()` to copy all of the .csv files into the new "Data_Files_Copy" directory. Make sure you copy *only* the .csv files.
2. Rename the files created in (1) appending today's date to the file name so that the filenames look like this name-yyyy-mm-dd.csv. Make sure you do this in a generic fashion so that it will work regardless of how many files there are and what their initial names are. Use `Sys.Date()` to get the date. Use the `gsub()` command to replace ".jpg" with "yyyy-mm-dd.jpg".}

11.1 Writing a data frame to a csv file

Suppose you have done a bunch of work and you want to write it to a file so that you can share it or open it in a spreadsheet program.

For the sake of showing you what we mean, let's create a fake-o dataframe.

```r
subjects <- 1:10
secret_identity <- c("Clark", "Peter", "Bruce",
                     "David", "Orin", "Dick",
                     "Diana", "Steve", "Tony", "Barry")
hero <- c("Superman","Spiderman","Batman",
          "Hulk","Aquaman","Robin the Boy Wonder",
          "Wonder Woman","Captain America","Iron Man","Flash")
hero_data <- tibble(subjects, secret_identity, hero)
write_csv(hero_data, "hero_data.csv")
```

Now go confirm that the file is there and open it in a spreadsheet program.

But what if you forgot one superhero and you needed to append? You can append the individual record to the existing file. Note that `col.names` is turned off and `append` is turned on.

```r
subjects <- 1:10
secret_identity <- c("Clark", "Peter", "Bruce",
                     "David", "Orin", "Dick",
                     "Diana", "Steve", "Tony", "Barry")
hero <- c("Superman","Spiderman","Batman",
          "Hulk","Aquaman","Robin the Boy Wonder",
          "Wonder Woman","Captain America","Iron Man","Flash")
hero_data <- tibble(subjects, secret_identity, hero)
write_csv(hero_data, "hero_data.csv")

forgotten_hero <- tibble(subjects = 11,
                          secret_identity = "Kara",
                          hero = "Supergirl")
#you have to create it in an object of 1 column and 3 rows
write_csv(forgotten_hero, "hero_data.csv", append = TRUE)
```

Using this technique, you can add data to a file from a loop as the data is found to meet specific predefined criteria. For example, a resident and DH pulled all the hyponatremia occurrences from the LIS in 2014 and then looked for samples showing correction of Na$^+$ that is clinically too rapid (> 8 mmol/L in less than 24h). As these records were found, they were written to a file.

11.2 Writing xlsx files

Writing xlsx files with multiple sheets is quite easy also using the package xlsx. Note that there are multiple packages that can do this. You can also create fancy Excel files with formatting, bold, colored fonts and fill colors. This would only be of interest if you were writing automated reports for those folks for whom the appearance matters more than the content.

```r
install.packages("openxlsx")
```

```r
library(openxlsx)
hero.list <- list(Main_Heros = hero_data,
                  Forgotten_Heros = forgotten_hero)
write.xlsx(hero.list, file = "hero_data.xlsx", row.names = FALSE)
```

The same package allows you to read in a specific sheet from an xlsx file starting and finishing at a specific row.

12 Projects

12.1 Option 1: Lab Utilization Investigation

This file contains some very nice clean data about lab tests at a *totally fictional* hospital:

```r
lab.data <- read_csv("Data_Files/lab_data.csv")
glimpse(lab.data)
```

```
## Observations: 54,406
## Variables: 14
## $ ID                  <chr> "00003945", "00003945", "00003222", "00002489", ...
## $ SAMPLE              <chr> "00015416", "00015416", "00008687", "00010888", ...
## $ COLLECTION_DATE     <dttm> 2018-10-12 08:05:00, 2018-10-12 08:05:00, 2018-...
## $ ORDERED_DATE        <dttm> 2018-10-12 07:53:00, 2018-10-12 07:53:00, 2018-...
## $ RESULT_DATE         <dttm> 2018-10-12 08:25:58, 2018-10-12 08:33:38, 2018-...
## $ RECEIVED_DATE       <dttm> 2018-10-12 08:13:00, 2018-10-12 08:13:00, 2018-...
## $ SCHEDULED_DATE      <dttm> 2018-10-12 07:53:00, 2018-10-12 07:53:00, 2018-...
## $ BATTERY             <chr> "Complete Blood Count(CBC)", "INR PTT Battery(CO...
## $ TEST                <chr> "Hemoglobin(HB)", "INR(INR)", "Hemoglobin(HB)", ...
## $ ABNORMAL_FLAG       <chr> "L", NA, NA, NA, NA, "L", NA, NA, NA, NA, "L", "...
## $ RESULT              <dbl> 111.93, 1.04, 205.99, 126.01, 126.95, 0.82, 53.9...
## $ LOCATION_TYPE       <chr> "Inpatient", "Inpatient", "Inpatient", "Inpatien...
## $ ORDERING_PHYSICIAN  <chr> "Leach, Mario", "Leach, Mario", "al-Rehman, Muth...
## $ PRIORITY            <chr> "S", "S", "TD", "S", "S", "S", "S", "S", "S", "S...
```

Load it into your R session and see if you can produce a report on some KPIs (key performance indicators) for the lab:

- Start by surveying your data - what time period does it cover? How many tests do you have information on?
- What is the median and 90th percentile for Scheduled to Result turn around time for each test?
- Break it up by location type and create a nice plot (perhaps boxplot or violin plot or time-series plots) of TAT for the Emergency room versus Inpatients.
- Are there certain hours of the day when the turn around time suffers? Days of the week?

Then see if you can find some utilization insights:

- From what location type do the bulk of orders for each test originate?
- Which doctors order the most tests overall? Which doctors order the most tests per patient seen?
- How many tests are ordered on each patient?

12.2 Option 2: Clean Up Some Research Data – Bad Medicine

This is raw data from a study published on a familial hypercholesterolemia cohort from 2005. The birthdates are not real. The data were hand-transcribed by a resident into Excel.

12.2.1 Birthday Blues

If you would rather jump straight to analyzing cleaned data, run the solution code in the .Rmd file and then jump down to the next section.

- Read in the xlxs file entitled "badly_entered_data.xlxs".
- Look at the DOB format using `head()` or `glimpse()`
- Parse the birthdays using the appropriate lubridate function and append them as a new column.
- What does the error message mean?
- Have a look at what lubridate has done. Why did this happen? How could this be avoided? Can you suggest a fix? Consider `str_c()` or `paste()`.
- Perform the parsing again. What has happened this time?
- Determine what is still wrong with the problematic birthdates.
- Can you come up with a programmatic way of salvaging the data other than fixing it by hand?

12.2.2 Descriptive Analysis

- In this data set 1 codes as Male and 2 codes as Female. How many males and females are there?
- If the study analyis was performed on 30-Aug-2003, calculate the ages of the patients in years on that date and append it as a new column.
- Determine the median and IQR of the patients in this study by sex.
- Is there a significant difference in the distribution of the ages of males and females? You can look at the functions `t.test()` and `wilcox.test()` as appropriate.
- Compare the distribution of cholesterol results between males and females and determine if there is a significant difference.
- Calculate the LDL-cholesterol and append as a new column.

12.2.3 A Physiology Question

There is a body of literature that says that lipoprotein(a) is unrelated to LDL cholesterol concentration because it is regulated through an independent pathway.

- Produce a correlation plot of Lp(a) versus LDL-C.
- Is there a statistically significant correlation?
- What is the regression equation of Lp(a) as a function of LDL-C? Look at `lm()`
- Does the data support or refute the traditional claim?

12.3 Option 3: Flat File Formatting

We are going to take flat file export data from an SCIEX API 5000 mass spectrometer and reformat the flat file for upload to the SunQuest lab information system. The process will be nearly identical for any instrument and LIS combination. The file you want to import is called API_Flat_File.csv. Before you do, open the file in a text editor or Excel. You will notice that the first 26 lines contain useless information, which you want to exclude when you read the file in. In fact, if you don't exclude them, R won't interpret the shape of the dataframe properly and this will cause confusion and delay, as Sir Topham Hatt says. The other thing to note is that this is a tab-delimited text file, not a comma-delimited file. So, sep = "\t" is a required parameter for `read_delim()`, not sep = ",". Have a look at `help(read_delim)` to learn how to exclude the initial lines of a file–you want to look at the example of the skip option.

You want to pull out three relevant columns, which are named "Sample Name", "Analyte Peak Name" and "Calculated Concentration (pmol/L)." It is better to address them by their name rather than their number because changes in AB SCIEX software could change which column number you are pulling…but not likely the name. Build a new dataframe containing these three columns.

This new dataframe contains extraneous information. You only need to up load the results of the quantifier ion ("Aldo 1"), not the qualifier ion ("Aldo 2"). Use `str_which()` to identify the rows containing "Aldo 1" data. Filter out the rows with "Aldo 1" data and assign this to a new dataframe.

Your dataframe still contains stuff that the LIS cannot digest. The patient samples have "Sample Name" in the format E followed by 10 integer digits. We only want these rows. Use `str_which()` to identify the rows, filter them out, and assign them to a new variable. In this simple example, it is sufficient to search based on the pattern "E", but in general this would not be a good idea. In general you would use a regular expression to look for the pattern of E followed by 10 repetitions of the pattern 0-9. The search term, in this case, would be ^E[0-9]{10}' (Starts with E followed by 0-9 exactly 10 times). Filter out the patient data and assign to a new dataframe.

Use `str()` to examine the "Calculated Concentration" column. You will see that we have a problem. It is not numeric because to the undetectable results which come across as "No Peak". We need to force this column to be numeric while assigning a concentration of 0 to the "No Peak" results. Sometimes the instrument sends other non-numeric terms over for undetectable results (like "< 0"). So, the safest thing to do is to convert the whole column to numeric and then replace the NA results with 0.

If you do not convert to character first, terrible things will befall you … you will create results corresponding to the integer of the level R assigned to the column when it was read it. Oh, the horror. Apply `as.numeric(as.character())` to the Calculated.Concentration column to generate numeric data, assigning it to a temporary vector. This vector will contain NA results. Replace the NAs with 0 in this manner:

```r
temp <- replace_na(temp,0)
```

Now overwrite the `Calculated Concentration (pmol/L)` field of your dataframe with the numeric data in your temp vector. Now we are almost done.

SunQuest wants its upload format like this:

```
API,E2660178393,ALD,230
```

```
API,E2660172438,ALD,95
```

```
API,E2660174731,ALD,634
```

```
API,E2660174643,ALD,217
```

API is the instrument name, the E-numbers are the patient IDs, ALD is the testcode and the numbers afterward are the results. Two of these columns are built already in your dataframe. Build the other two with `rep("API",n)` and `rep("ALD",n)` where `n` is the number of rows. Build a final dataframe corresponding to the required format (immediately above) and write it to a file with a comma as separator. Name the file "processed_flat_file.txt" so that when you click on it, it does not open in Excel, only in a text editor.

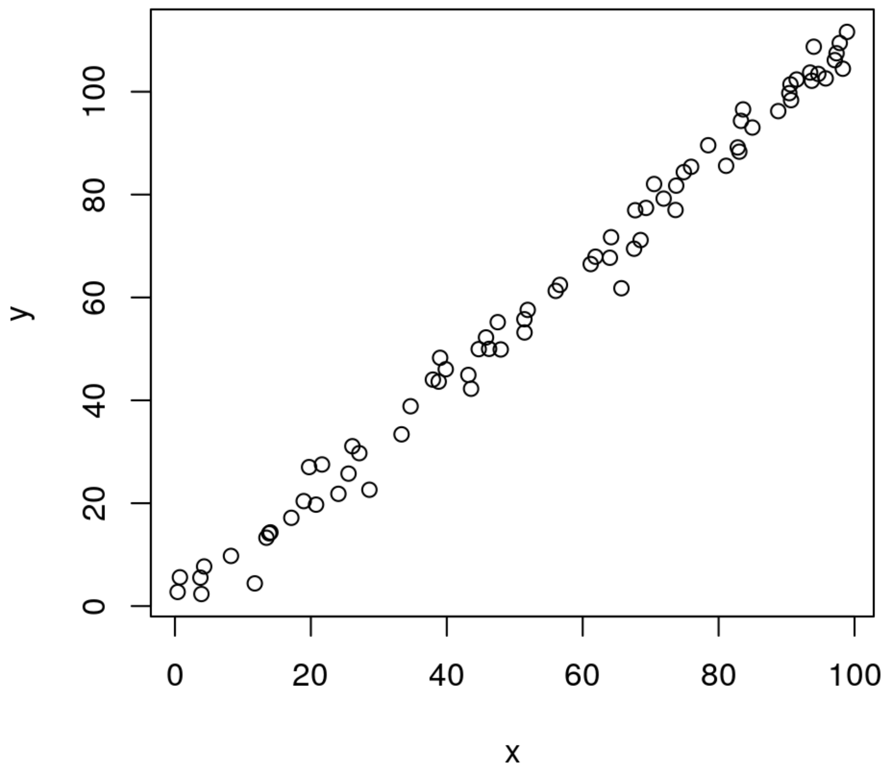
13 Appendix: Old Skool Base R Plotting

We are going to learn how to make a very high quality publication-read figures that are completely customizable and in any image format. While the techniques described in this lesson are not as powerful in and of themselves than those of the tidyverse, they are far more customizable.

```r
x <- runif(75, 0, 100) # produces 75 random numbers between 0 and 100.
y <- 1.1 * x + rnorm(75,0,3) #add some bias and some scatter
```

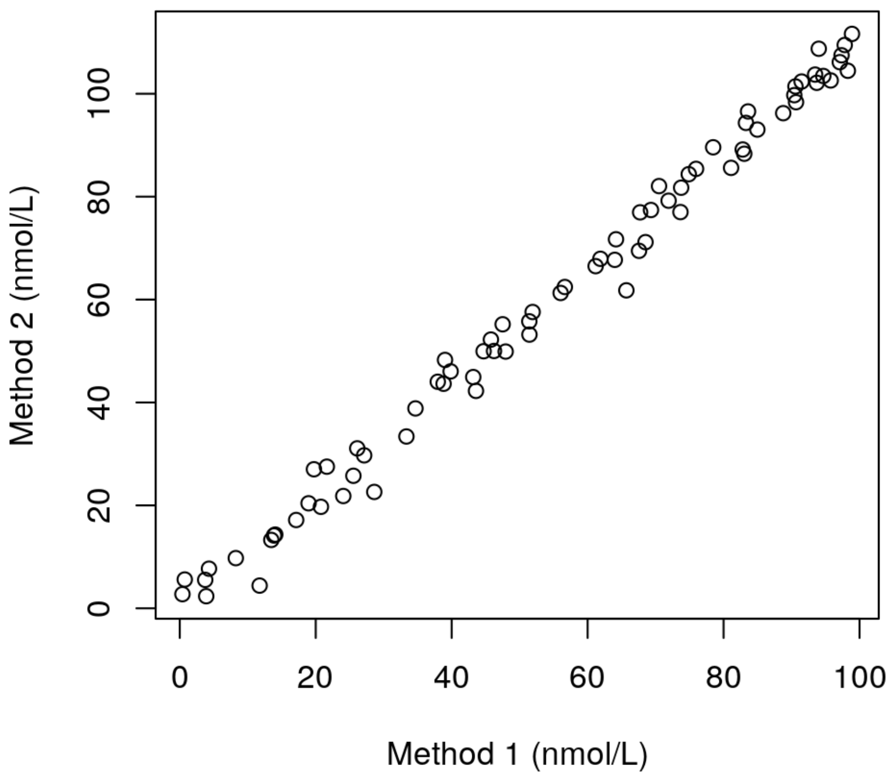
The most rudimentary default plot is created by

```r
plot(x,y)
```



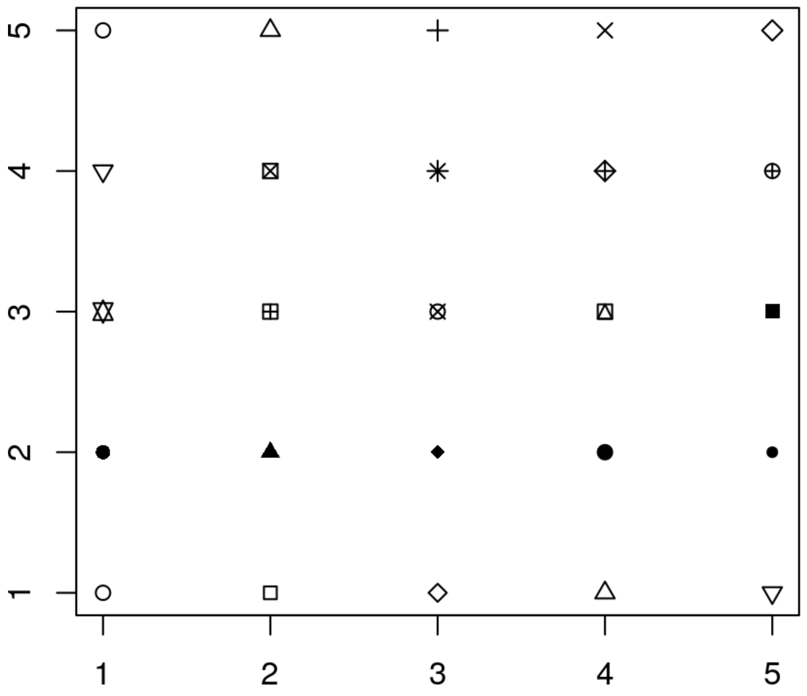
First, we will want to make better axis labels. This is easy. It is accomplished, for example, by adding xlab = "Method 1 (nmol/L)" to the x-axis and corresponding text to the y-axis.

```r
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)")
```



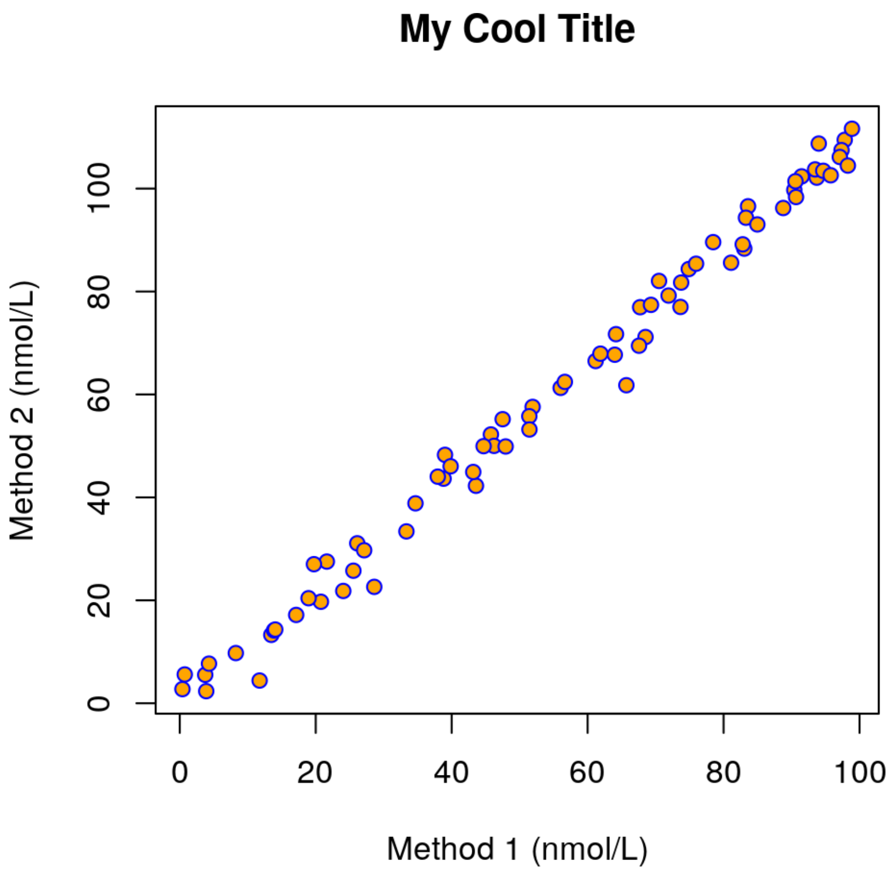
The title can be altered with the parameter main = “My Cool Title”.

```r
plot(rep((1:5),5),c(rep(5,5),rep(4,5),rep(3,5),rep(2,5),rep(1,5)),pch = 1:25,
     xlab = "",ylab = "")
```



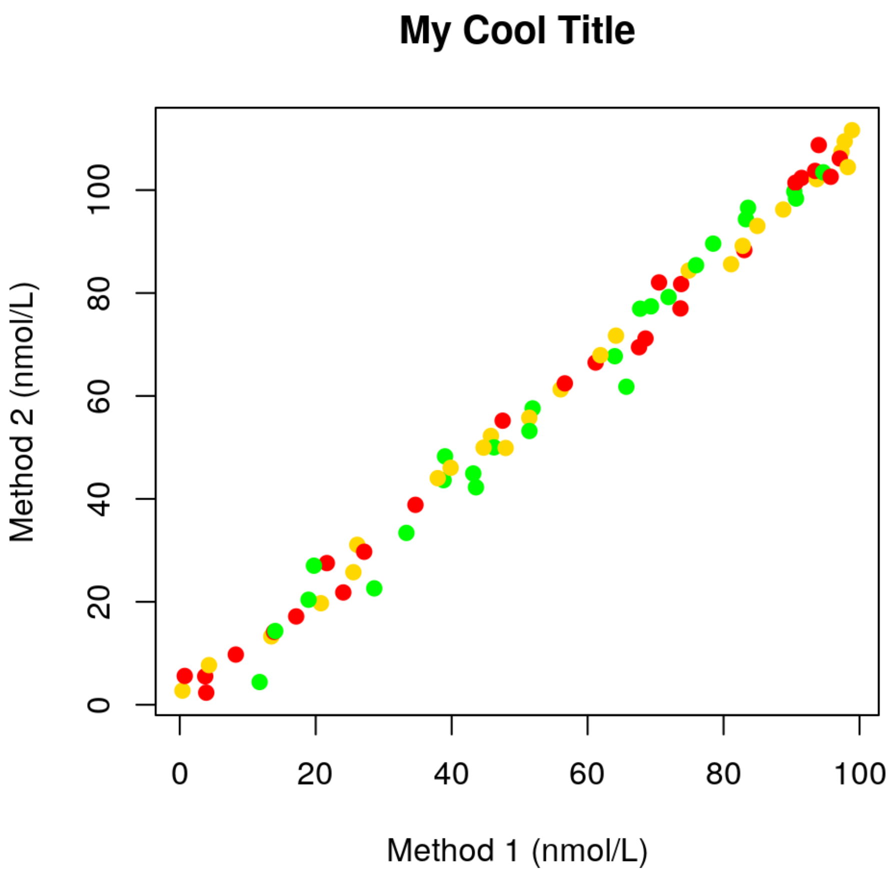
The parameter that alters the point character is called pch and you can set its value between 1 and 25. The colour of the points is set with col = “the color I want”. Characters 21-25 have fill colors in addition to outline colours and these can be set with the parameter bg.

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 21, col = "blue", bg = "orange")
```



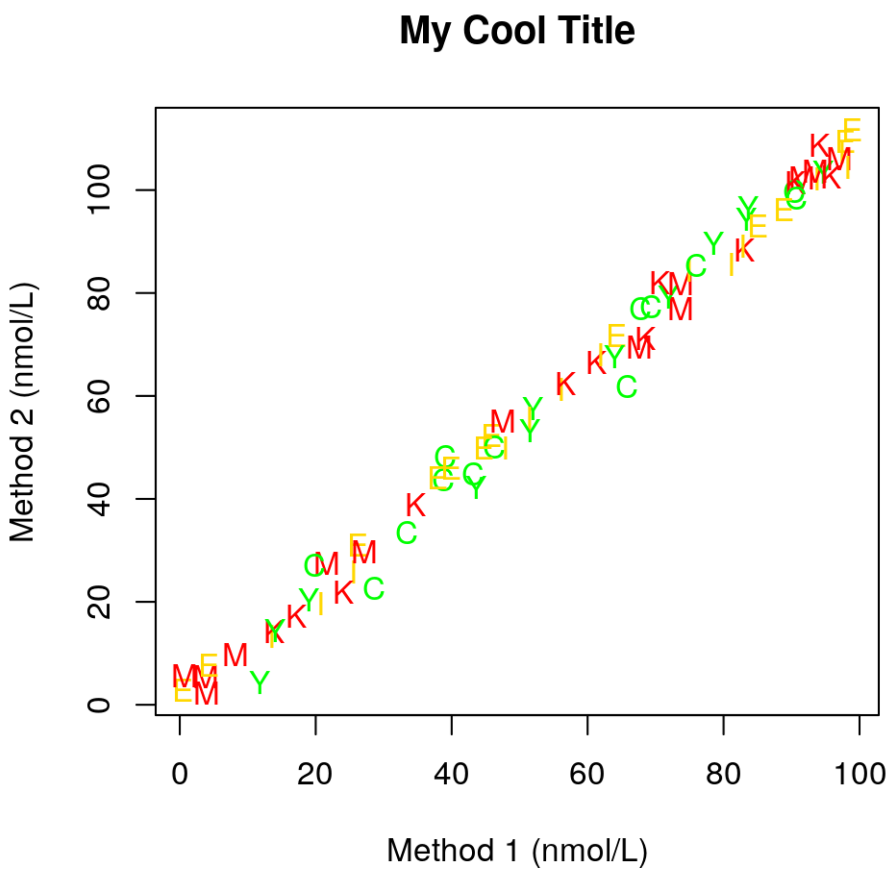
Your point colors can be in a vector if you want and the colours will cycle through the vector:

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 19, col = c("red", "gold","green") )
```



In the event that you want the pch to be a letter…feel free!

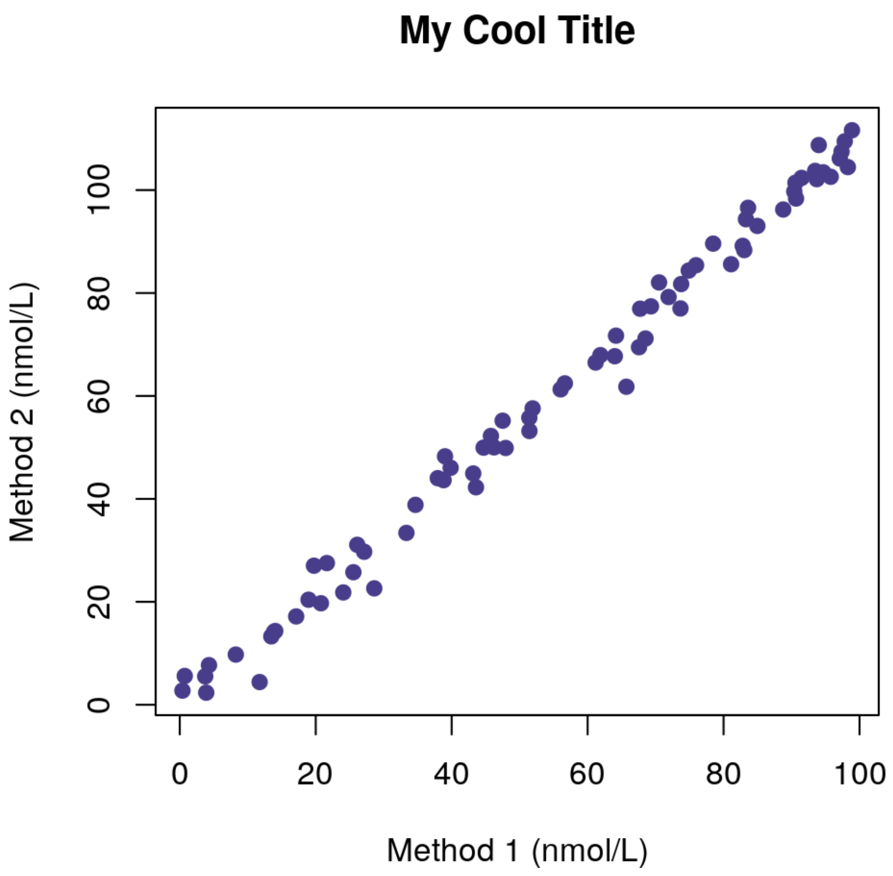
```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = c("M","I","C","K","E","Y"), col = c("red", "gold","green") )
```



You can see a full chart of R's colours here: <research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf>

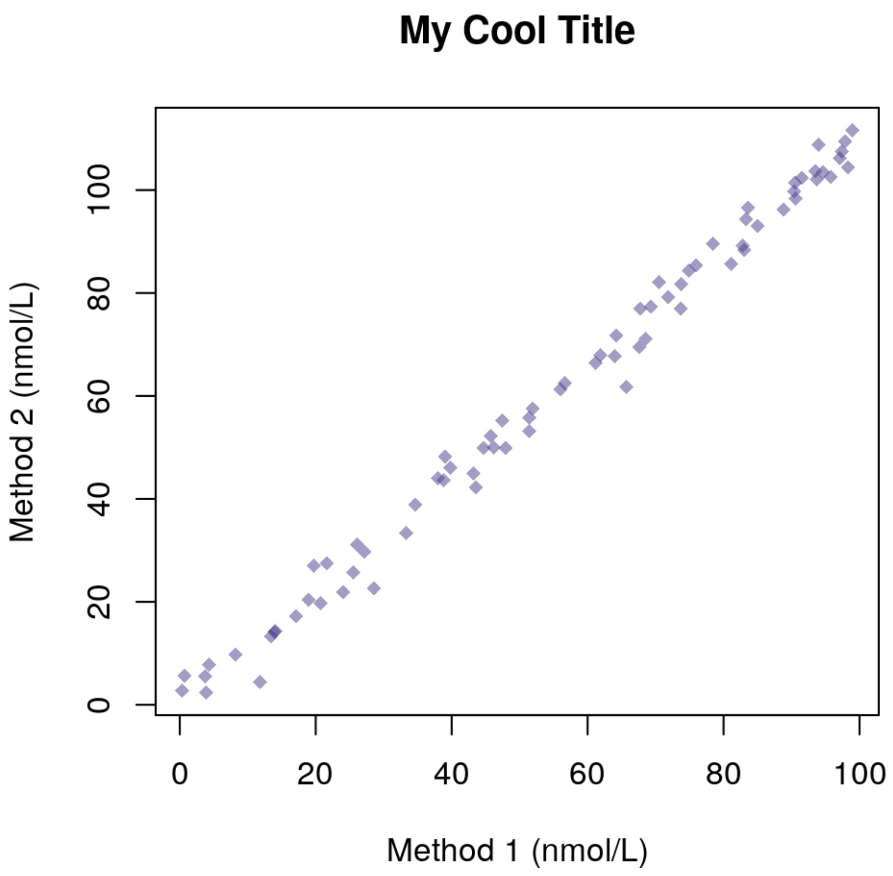
You can address colours using their hexadecimal code if you want.

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 19, col = "#483D8B") #slateblue
```



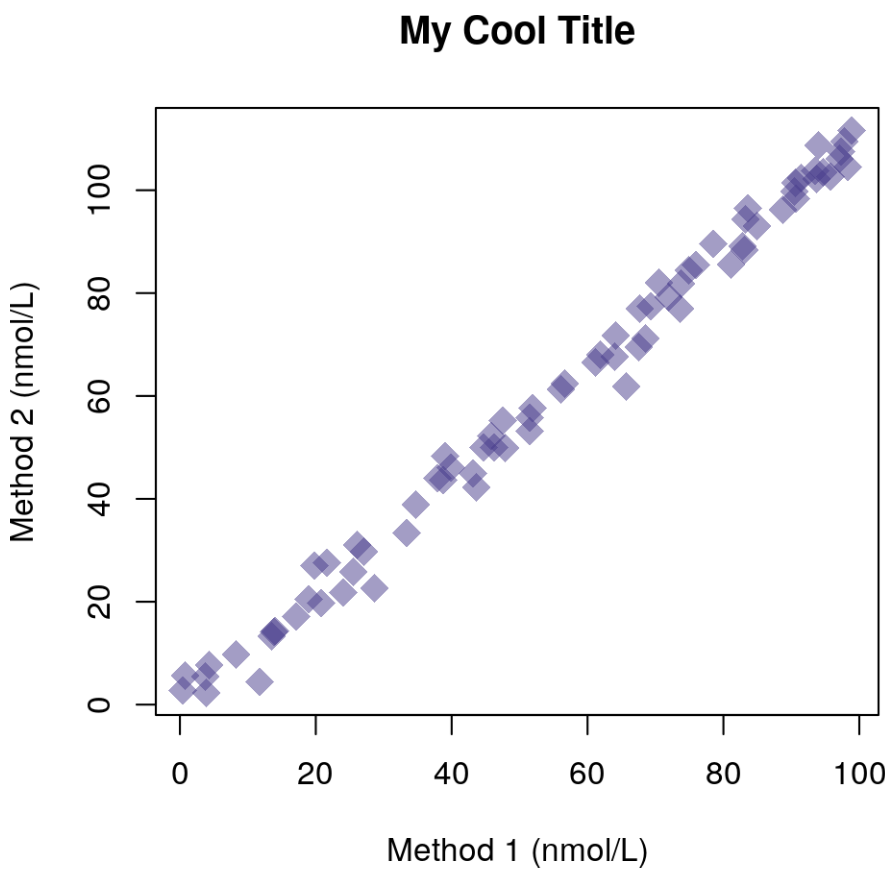
Another cool thing you can do is make the points semitransparent by appending a number in hexadecimal after the hex colour code from 00 to FF (256). A 00 makes the point invisible and FF makes it full transparent. A value of 80 makes a nice semitransparent look for high density plots.

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 18, col = "#483D8B80") #slateblue
```



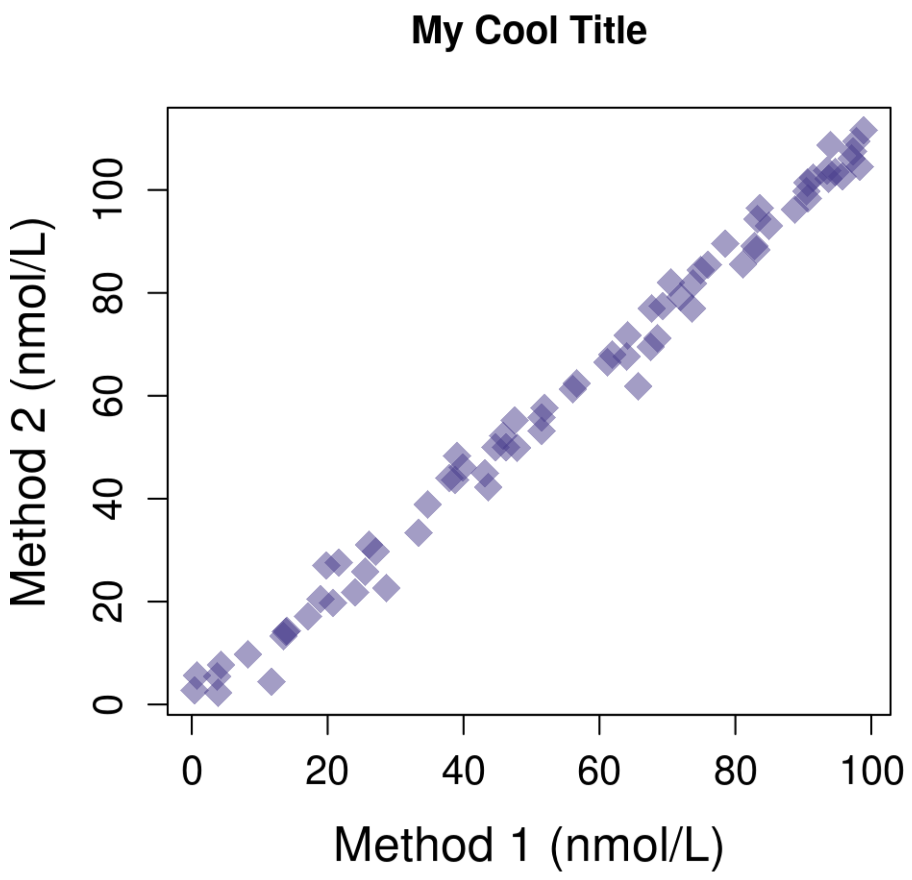
You can make your points bigger with the parameter cex. A value of 1 is default. 2 is double sized and 0.5 is half-sized etc.

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 18, col = "#483D8B80", cex = 2) #slateblue
```



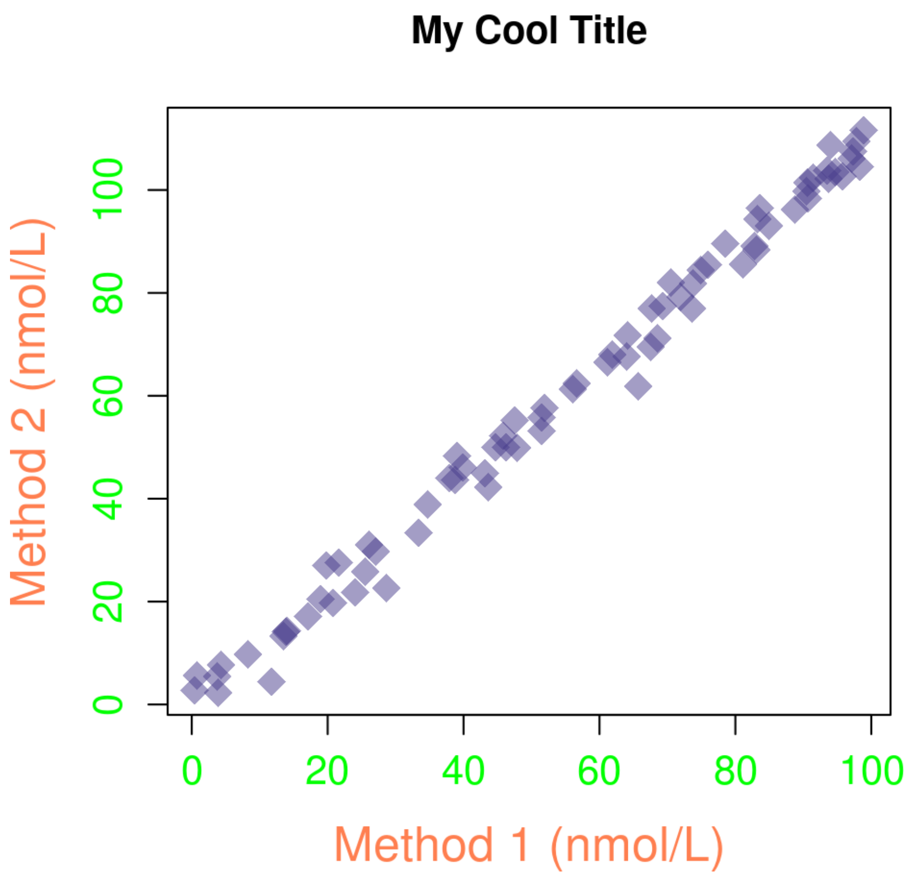
The size of the axis annotation is adjusted by cex.axis and the axis labels with cex.lab

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 18, col = "#483D8B80", cex = 2, cex.axis = 1.25, cex.lab = 1.5) #slateblue
```



Not surprisingly, there are axis.col and lab.col parameters to.

```
plot(x,y,xlab = "Method 1 (nmol/L)", ylab = "Method 2 (nmol/L)", main = "My Cool Title",
     pch = 18, col = "#483D8B80", cex = 2, cex.axis = 1.25, cex.lab = 1.5, col.axis = "green",
     col.lab = "coral") #slateblue
```



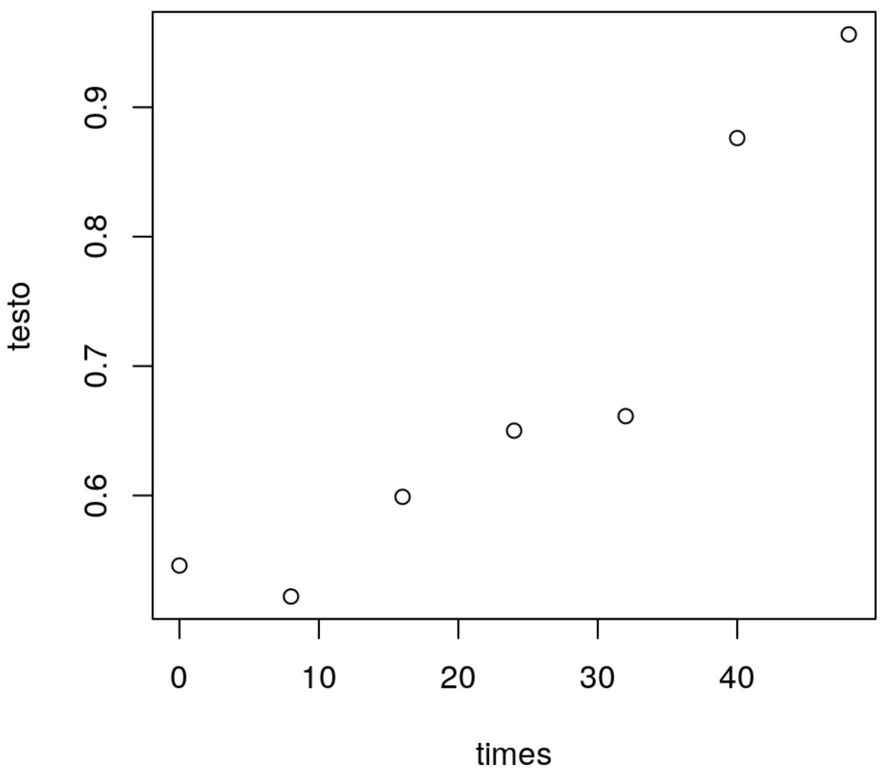
For now we will stop, but you get the picture that all plot parameters are adjustable.

13.1 Connecting the dots

In other kinds of plots, we might want our dots connected–like in a stability study perhaps.

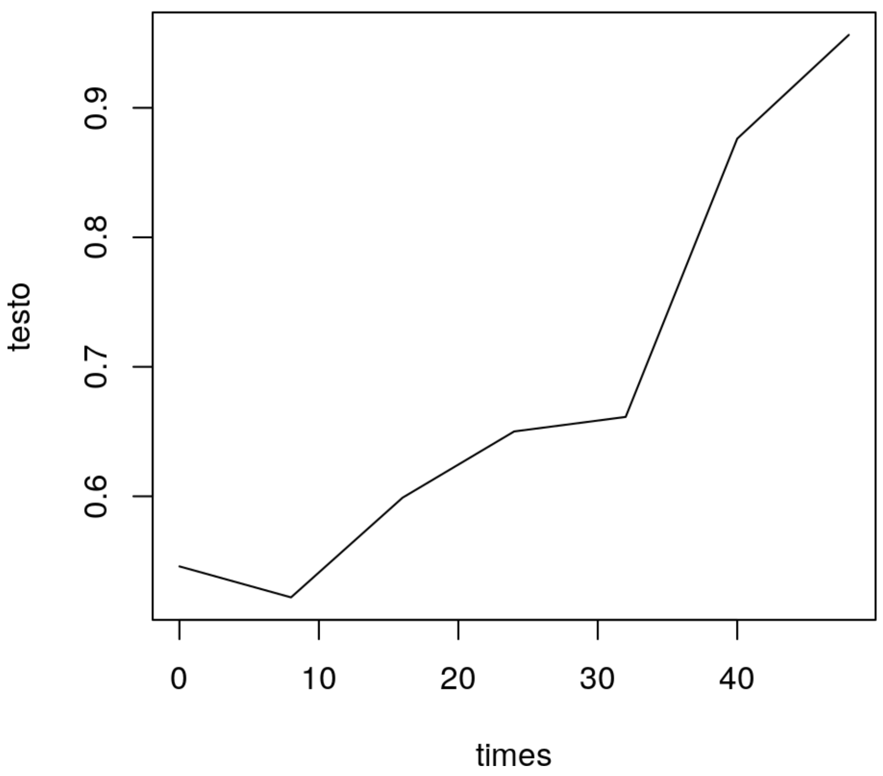
Suppose you are looking at how testosterone increases with time exposed to the gel in a gel separator tube. This is mock data, but the phenomenon is real.

```
times <- seq(from = 0, to = 48, by = 8)
testo <- seq(from = 0.5, to = 0.9, length.out = 7) + rnorm(7,0,0.05)
plot(times,testo)
```



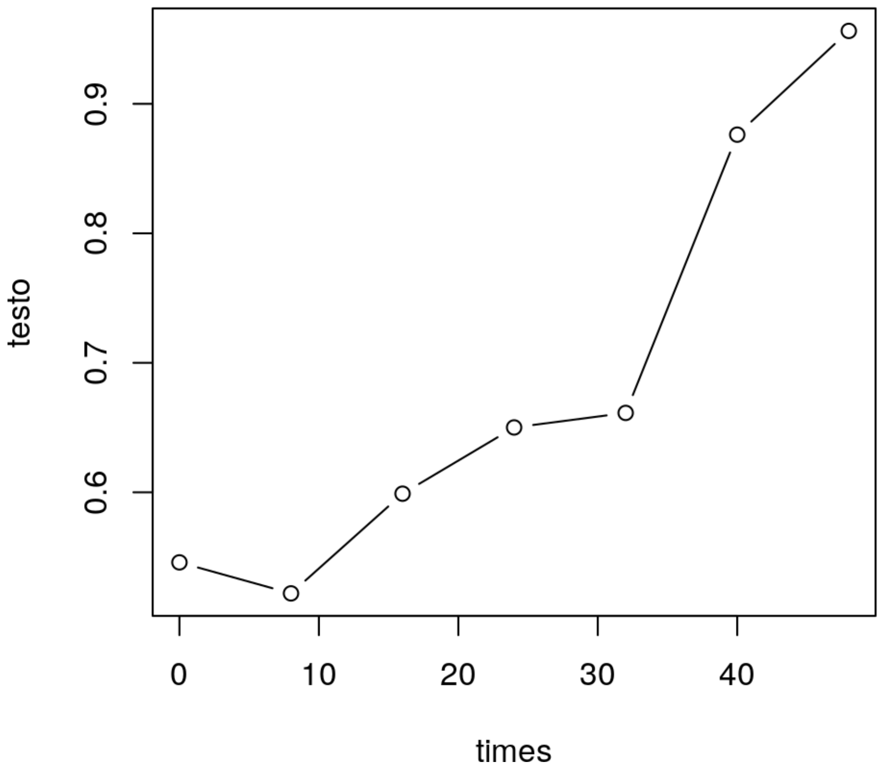
But to make a line use "l"

```
plot(times,testo, type = "l")
```



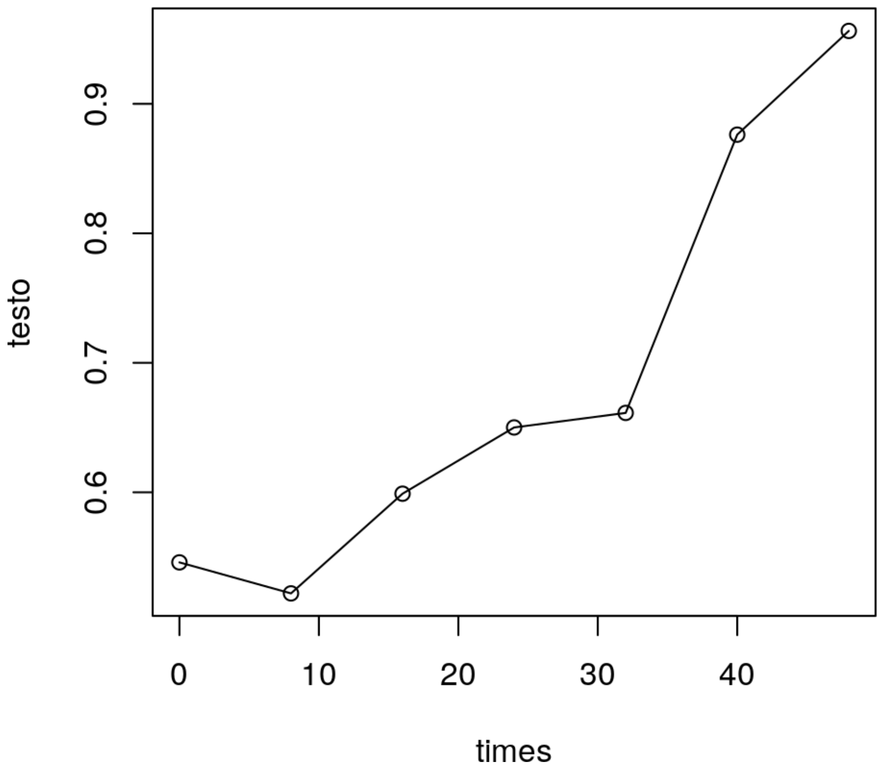
or both "b"

```r
plot(times,testo, type = "b")
```



or overwritten "o"

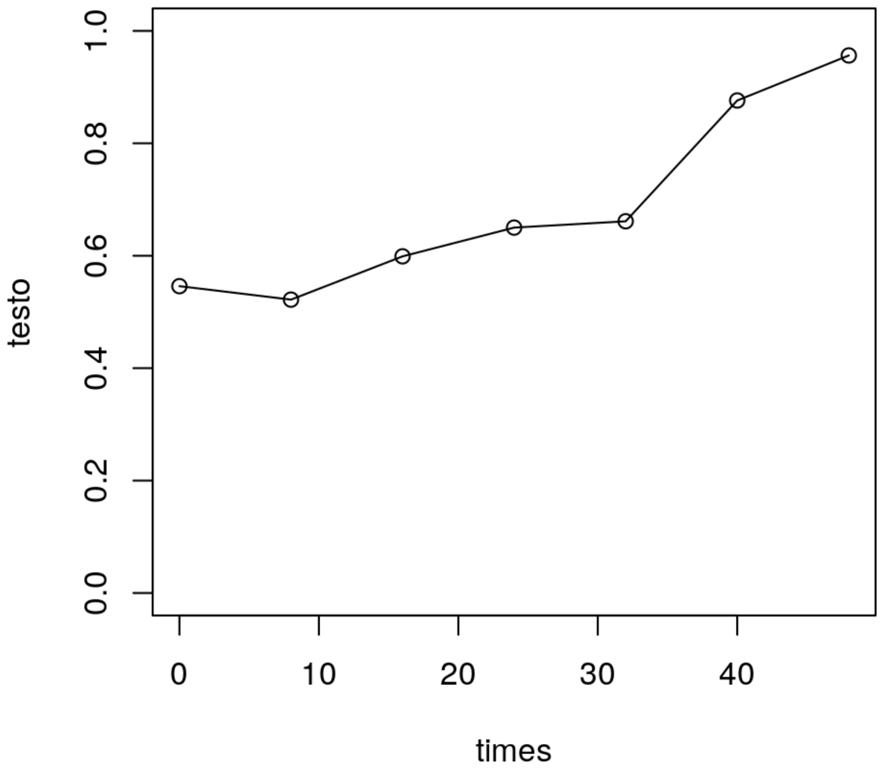
```r
plot(times,testo, type = "o")
```



Try these out on your own.

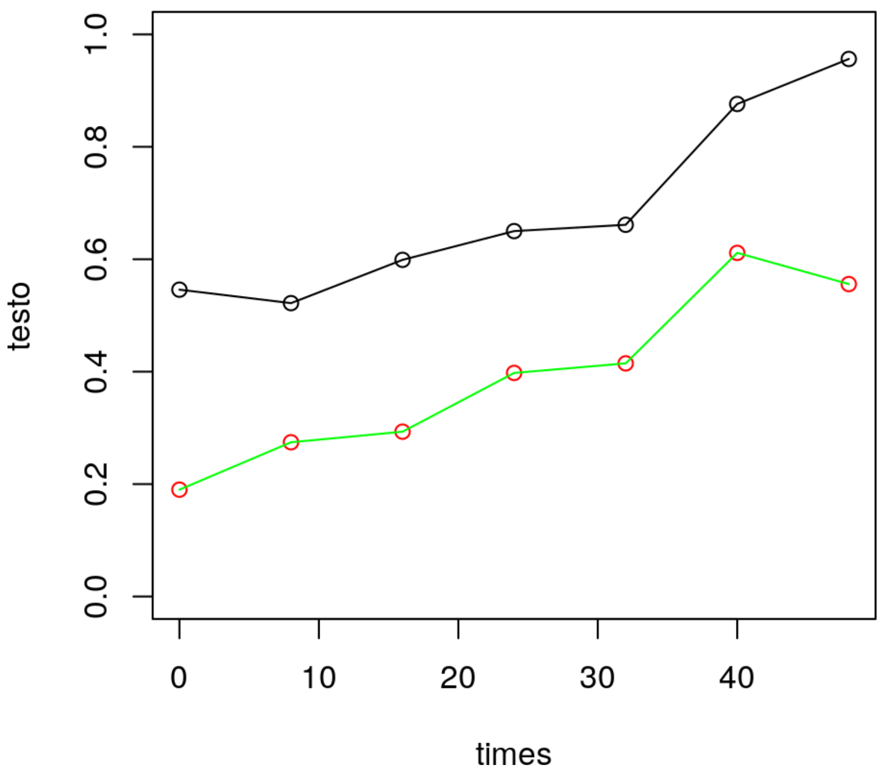
Next, to adjust the y and x axis, use xlim and ylim

```
plot(times, testo, type = "o", xlim = c(0,48), ylim = c(0,1) )
```



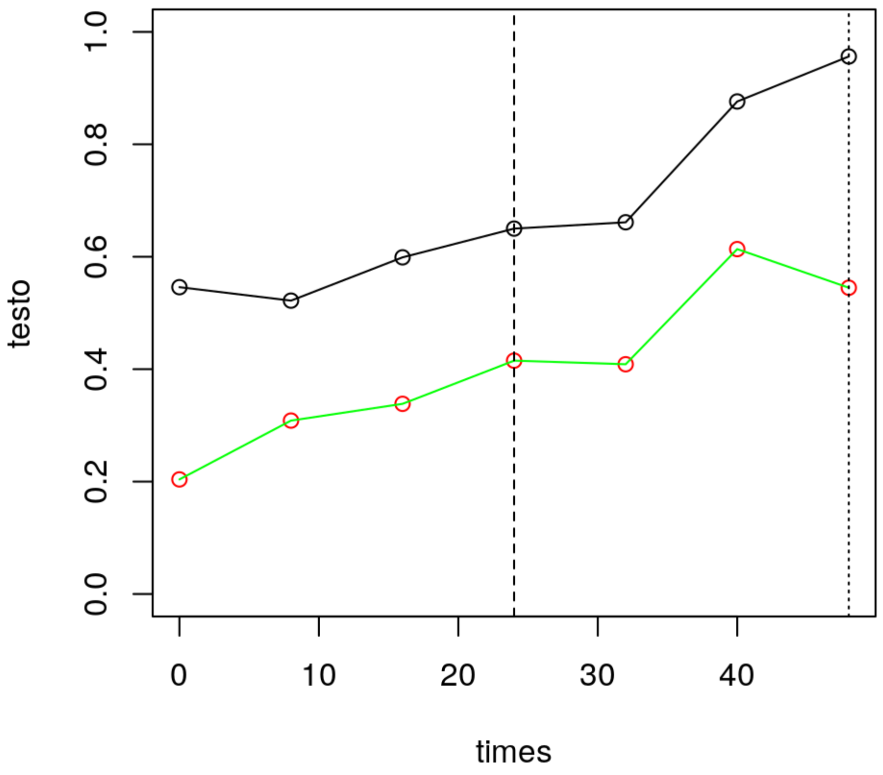
When a plot already exists (and is active), you can add other points or lines to the graph with the commands `points()` and `lines()`

```
testo2 <- seq(from = 0.2, to = 0.6, length.out = 7) + rnorm(7,0,0.05)
points(times, testo2, col = "red")
lines(times, testo2, col = "green")
```



You can add lines to your plot with `abline()` as we had done earlier, and there are 6 lines styles to adjust with the parameter `lty`.

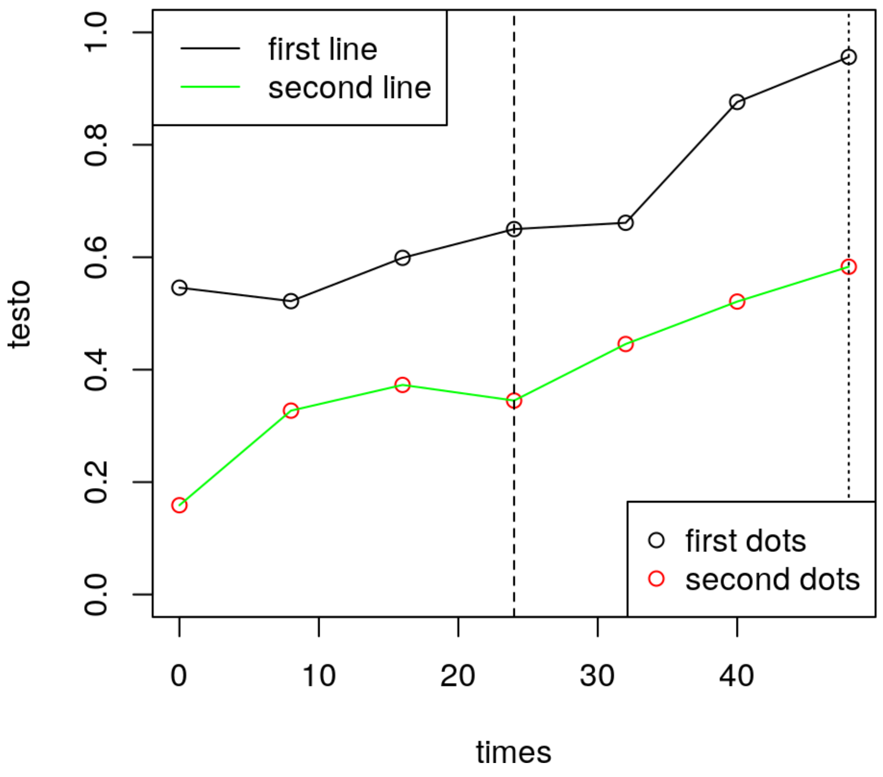
```
abline(v = 24, lty = 2) #vertical line at 24 hours, dashed
abline(v = 48, lty = 3) #vertical line at 48 hours, fine dashed
```



13.2 A legend from your own mind

You can add a legend to the plot very easily, and it is highly customizable also.

```
legend("topleft", c("first line", "second line"), lty = c(1,1), col = c("black","green"))
legend("bottomright", c("first dots", "second dots"), pch = c(1,1), col = c("black","red"))
```



13.3 Opening a new plot window

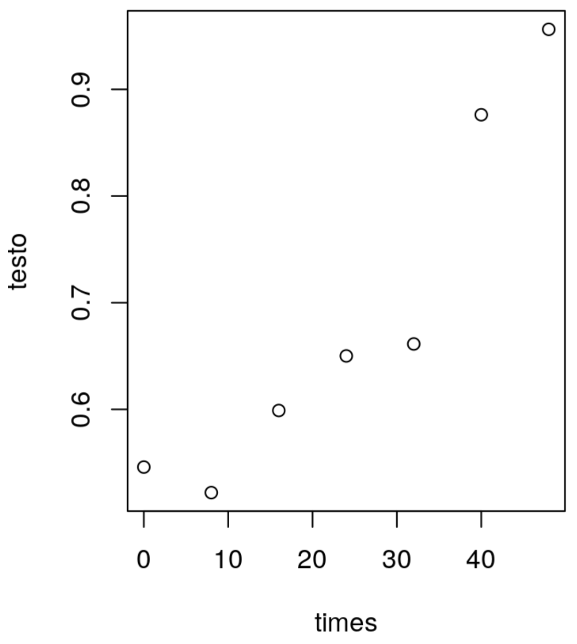
If you want to open a new plot window to work on, type `quartz()` on Macs and type `x11()` on Windows and Linux X11 is the name of the Unix windowing system, which is where the command's name comes from. To close an active plot window, type `dev.off()`.

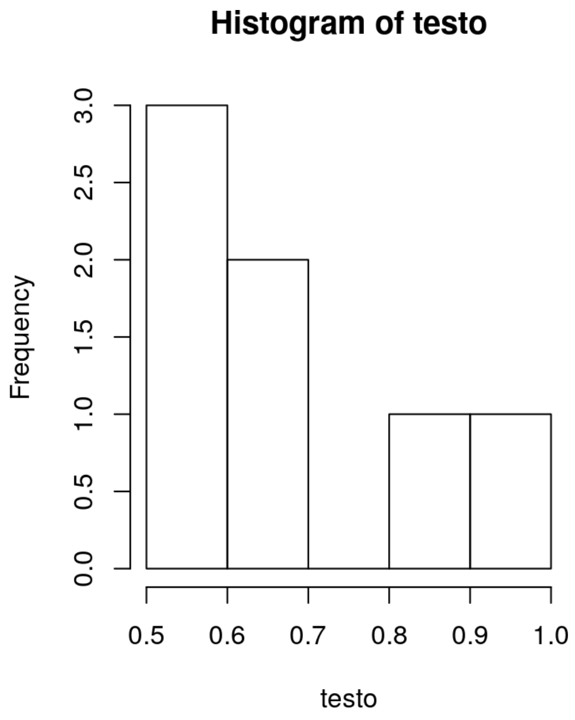
13.4 Putting more than one plot in a single figure.

To create multiple plotting environments in a single window, you can use the `par` command.

For example `par(mfrow = c(1,2))` creates a 1 row, 2 column plotting environment. Then if you make two consecutive plots, they will fill the two environments.

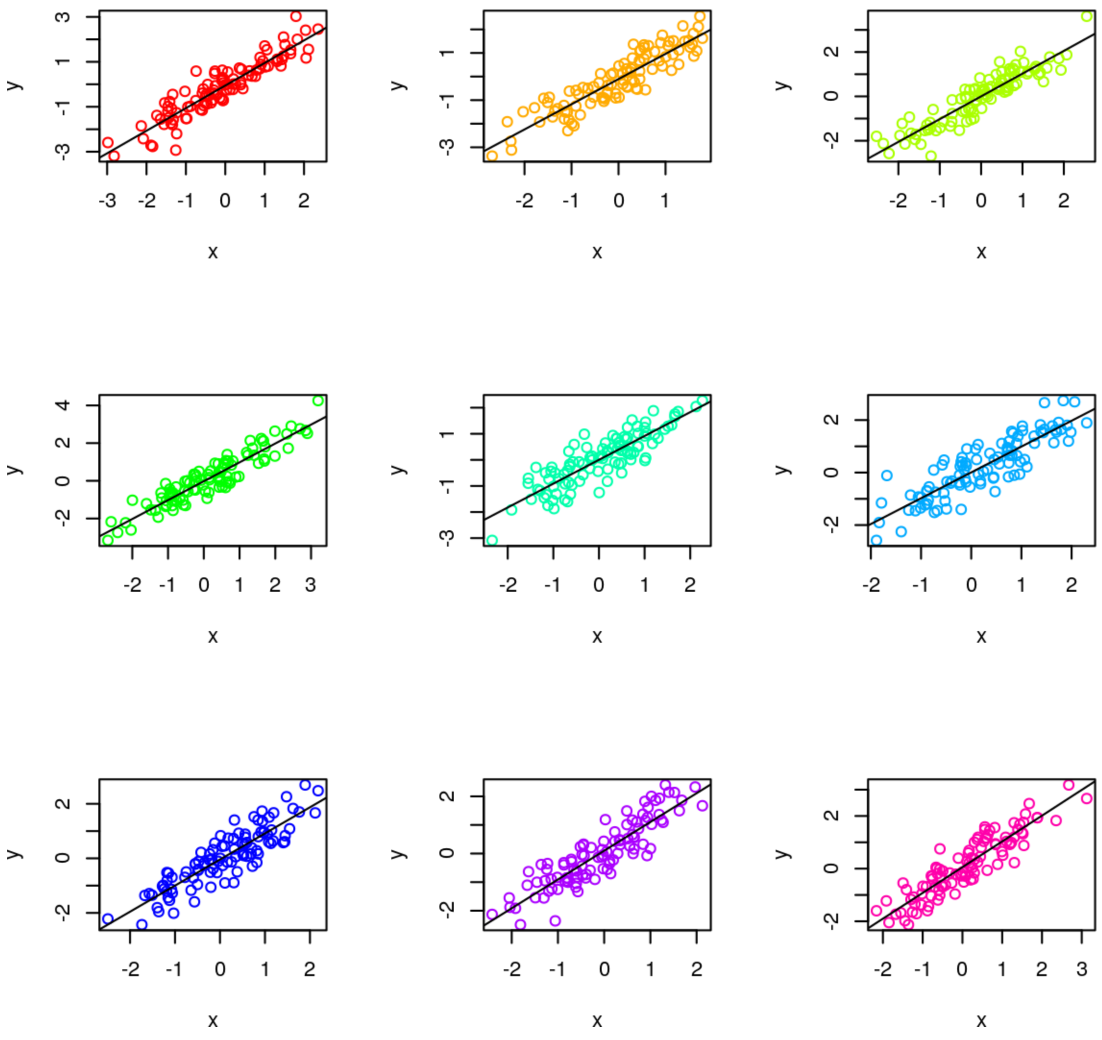
```
par(mfrow = c(1,2))
plot(times, testo)
hist(testo)
```





Use the mouse to adjust the plot window and see what will happen. You can put as many plots are you like.

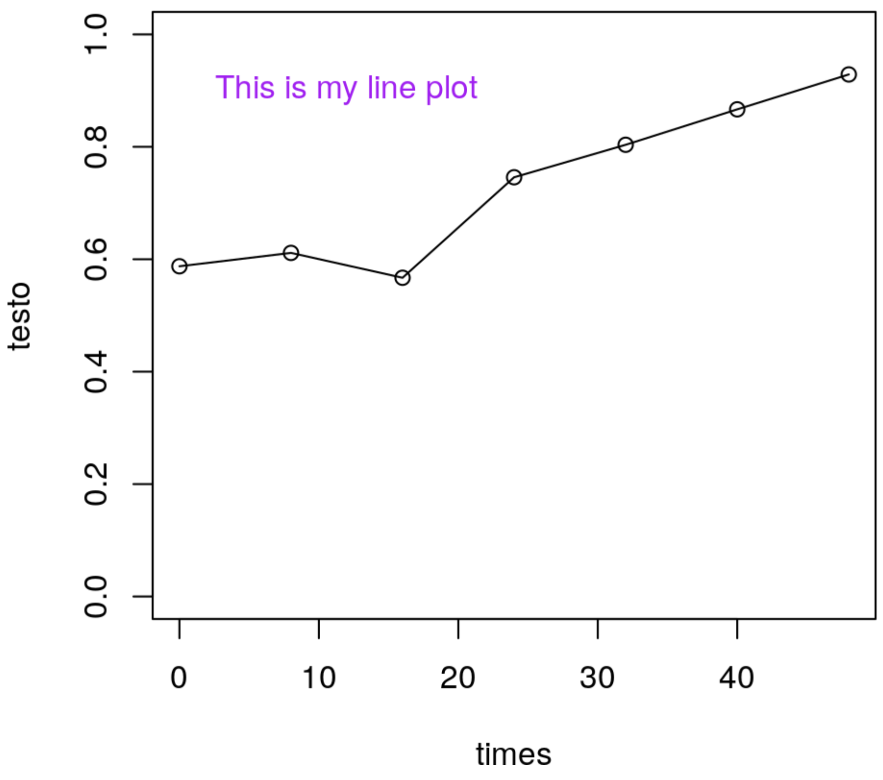
```r
par(mfrow = c(3,3))
colors <- rainbow(9)
for (i in 1:9){
    x <-  rnorm(100,0,1)
    y <- x + rnorm(100,0,0.5)
    plot(x,y,col = colors[i])
    abline(lm(y ~ x))
}
```



13.5 Adding text to a graph

You can easily add text to a graph with the `text()` command. The syntax is `text(x,y,"what you want to say")`. You can adjust whether the text is left or right justified to the starting point, its size, its color, its rotation and its font.

```r
times <- seq(from = 0, to = 48, by = 8)
testo <- seq(from = 0.5, to = 0.9, length.out = 7) + rnorm(7,0,0.05)
plot(times,testo, xlim = c(0,48), ylim = c(0,1), type = "o")
text(12, 0.9, "This is my line plot", col = "purple", cex = 1)
```



13.5.1 Exercise

- test_data.csv is method comparison data between the Siemens Advia Centaur and Deborah French's LC-MS/MS testosterone method. Read the data in test_data.csv.}
  - Examine the data with `head()`.
  - Produce a scatter plot and a difference plot side by side using the nmol/L data or the ng/dL data as you prefer.
  - Make the color of the points green and semitransparent in pch = 16.
  - Label the x and y-axes appropriately showing the units you have chosen using `xlab` and `ylab`.
  - Add an appropriate title using the main parameter.
  - Add a regression line–any type you like and the color of your choosing. Use `abline()` and `lm()` or `mcreg()`.
  - Add the line of identity in a dashed red line using `abline()` and the `lty` parameter.
  - Put a legend in the top left hand corner identifying the regression line and the line of identity specifying its color and type of dashing. Use `legend()`.
  - Below the regression equation put a statement about the correlation coefficient.
  - Write the regression relationship on the graph with the `text()` function.}
  - Produce an appropriately proportioned difference plot with difference expressed as %.
  - Add the mean difference line.
  - Add the ± 2SD lines
  - Put the two figures on the same image with `par(mfrow = c(1,2))`.
  - Save them as a png, 300 dpi, 12 inches across, 6 inches deep.
  - Save also as a pdf.