Assignment 9

Veronika Palotai

11/20/2019

Exercise 1.

Importing the necessary libraries.

```
library(modelr)
library(ggplot2)
library(tidyverse)
library(dplyr)
```

Adding mod1 and mod2 results and residulas to sim3

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)

(sim3 <- sim3 %>%
  add_predictions(mod1, var = "mod1") %>%
  add_predictions(mod2, var = "mod2") %>%
  add_residuals(mod1, var = "resid1") %>%
  add_residuals(mod2, var = "resid2"))
```

Summary of the two models

```
summary(mod1)
```

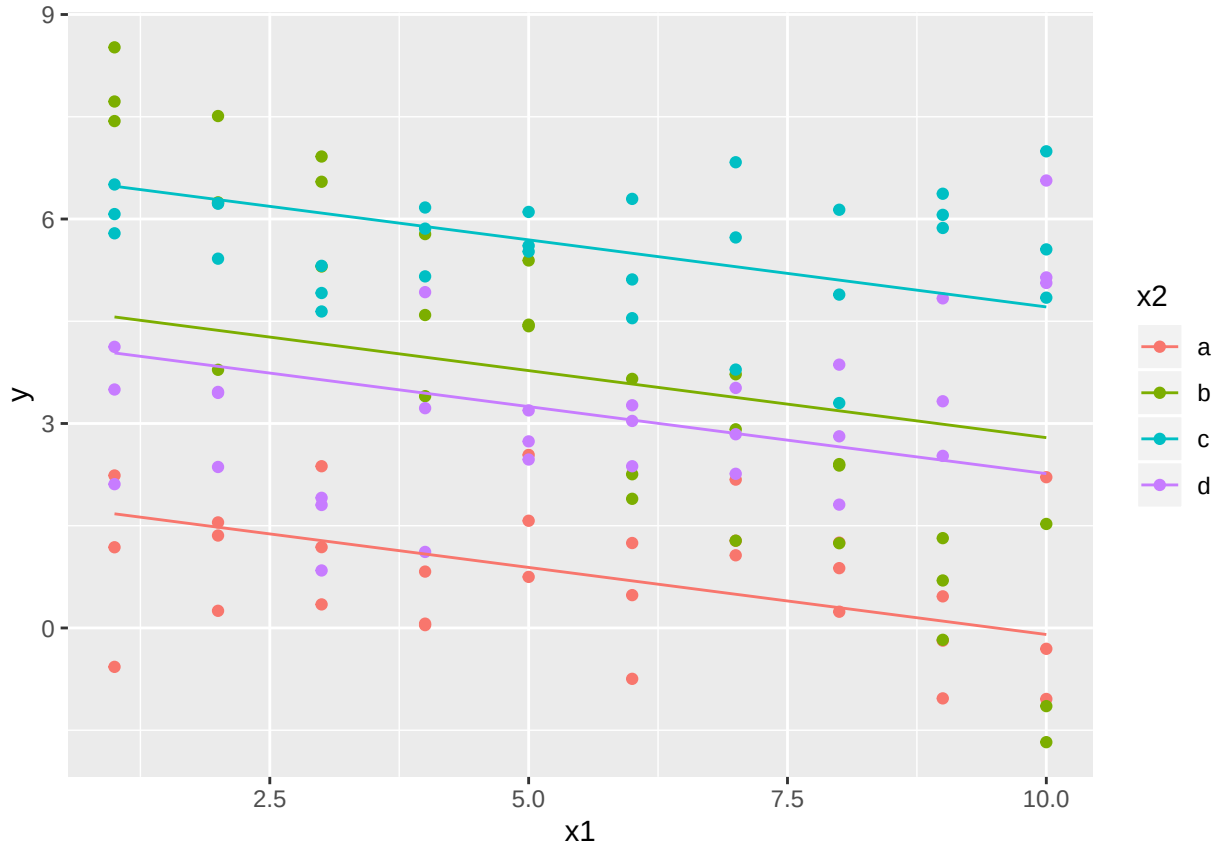
```
##
## Call:
## lm(formula = y ~ x1 + x2, data = sim3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.4674 -0.8524 -0.0729  0.7886  4.3005
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.87167    0.38738   4.832 4.22e-06 ***
## x1          -0.19674    0.04871  -4.039 9.72e-05 ***
## x2b          2.88781    0.39571   7.298 4.07e-11 ***
## x2c          4.80574    0.39571  12.145  < 2e-16 ***
## x2d          2.35959    0.39571   5.963 2.79e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.533 on 115 degrees of freedom
## Multiple R-squared:  0.5911, Adjusted R-squared:  0.5768
## F-statistic: 41.55 on 4 and 115 DF,  p-value: < 2.2e-16
```

```r
summary(mod2)
```

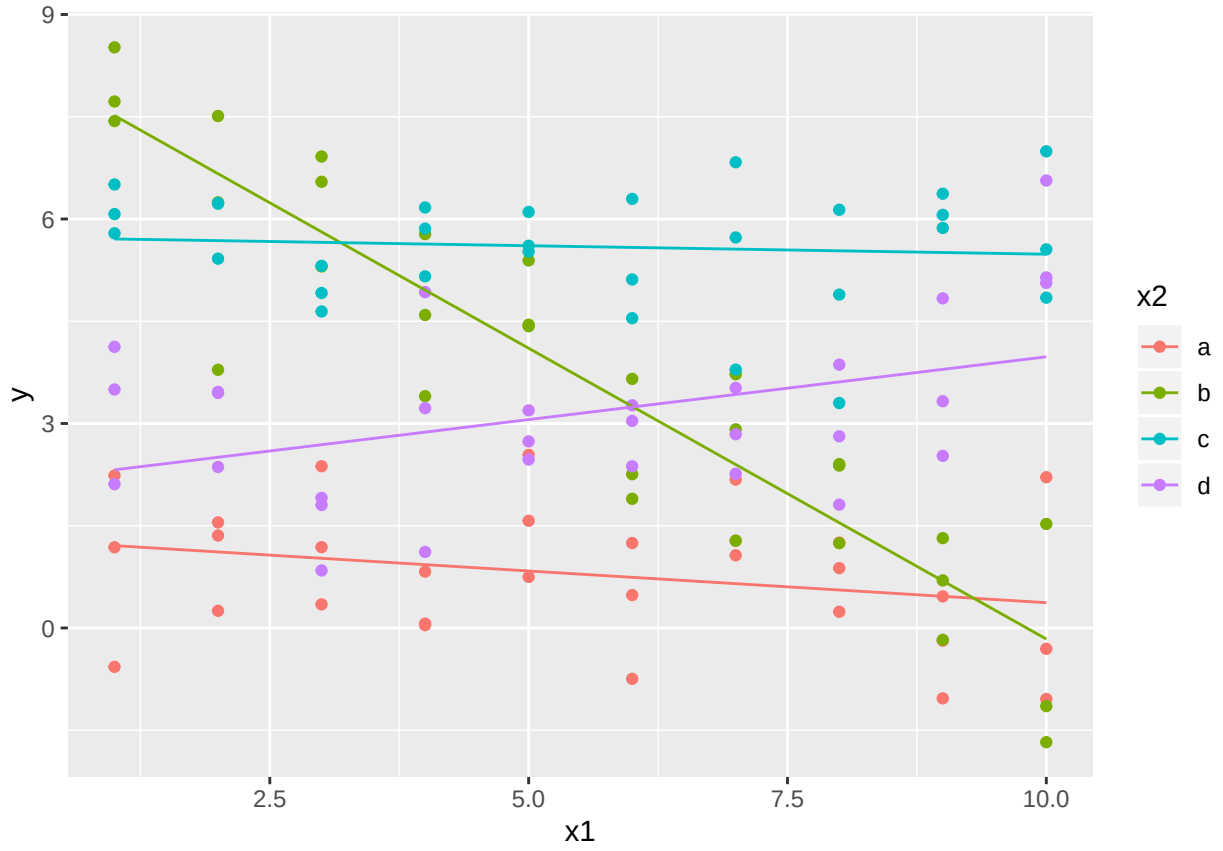
```
##
## Call:
## lm(formula = y ~ x1 * x2, data = sim3)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.87634 -0.67655  0.04837  0.69963  2.58607
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  1.30124    0.40400   3.221  0.00167 **
## x1          -0.09302    0.06511  -1.429  0.15587
## x2b          7.06938    0.57134  12.373  < 2e-16 ***
## x2c          4.43090    0.57134   7.755 4.41e-12 ***
## x2d          0.83455    0.57134   1.461  0.14690
## x1:x2b      -0.76029    0.09208  -8.257 3.30e-13 ***
## x1:x2c       0.06815    0.09208   0.740  0.46076
## x1:x2d       0.27728    0.09208   3.011  0.00322 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.024 on 112 degrees of freedom
## Multiple R-squared:  0.8221, Adjusted R-squared:  0.811
## F-statistic: 73.93 on 7 and 112 DF,  p-value: < 2.2e-16
```

Mapping coefficients

```r
# Plot model 1 lines
ggplot(sim3, aes(x = x1, y = y, color = x2)) +
  geom_point() +
  geom_line(aes(y = mod1))
```

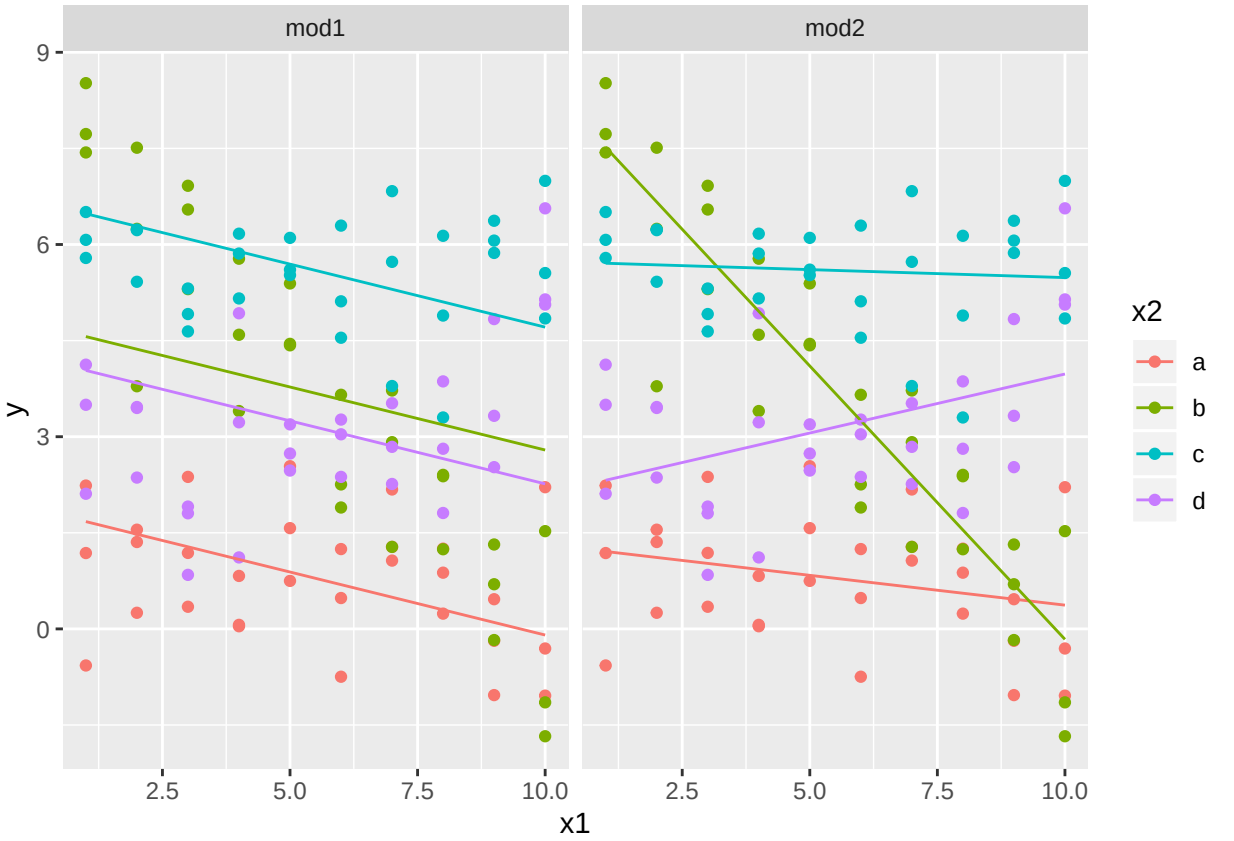



```
# Plot model 2 lines
ggplot(sim3, aes(x = x1, y = y, color = x2)) +
  geom_point() +
  geom_line(aes(y = mod2))
```




```
sim3 <- sim3 %>%
    gather(key = model, value = pred, mod1, mod2)

ggplot(sim3, aes(x = x1, y = y, color = x2)) +
  geom_point() +
  geom_line(aes(y = pred)) +
  facet_wrap(vars(model))
```



Based on the coefficients we can construct the linear functions fitted to the dataset

Mod1:

- Linear function fitted to datapoints where x2=a: `y = 1.8716659 - 0.1967378*x1`
- Linear function fitted to datapoints where x2=b: `y = 1.8716659 - 0.1967378*x1 + 2.8878108*x2b`
- Linear function fitted to datapoints where x2=c: `y = 1.8716659 - 0.1967378*x1 + 4.8057359*x2c`
- Linear function fitted to datapoints where x2=d: `y = 1.8716659 - 0.1967378*x1 + 2.3595867*x2d`

Explanation:

- Intercept is the expected y value when x1 and x2 are both equal to 0
- *-0.1967378* is the slope coefficient of x1 which shows that on average, y is *0.1967378* units smaller in the data for observations with one unit larger x1 but with the same x2.
- *2.8878108* is the slope coefficient of x2 where x2=b. It shows that on average, y is *2.8878108* units larger in the data for observations with one unit larger x2 but with the same x1.
- *4.8057359* is the slope coefficient of x2 where x2=c. It shows that on average, y is *4.8057359* units larger in the data for observations with one unit larger x2 but with the same x1.
- *2.3595867* is the slope coefficient of x2 where x2=d. It shows that on average, y is *2.3595867* units larger in the data for observations with one unit larger x2 but with the same x1.

Mod2:

- Linear function fitted to datapoints where x2=a: `y = 1.30124266 - 0.09302444*x1`
- Linear function fitted to datapoints where x2=b: `y = 1.30124266 - 0.09302444*x1 + 7.06937991*x2b - 0.76028528*(x1:x2b) = 1.30124266 - 0.85330972*x1 + 7.06937991*x2b`
- Linear function fitted to datapoints where x2=c: `y = 1.30124266 - 0.09302444*x1 + 4.43089525*x2c + 0.06815284*(x1:x2c) = 1.30124266 - 0.0248716*x1 + 4.43089525*x2c`
- Linear function fitted to datapoints where x2=d: `y = 1.30124266 - 0.09302444*x1 + 0.83455115*x2d + 0.27727920*(x1:x2d) = 1.30124266 + 0.18425476*x1 + 0.83455115*x2d`

Explanation:

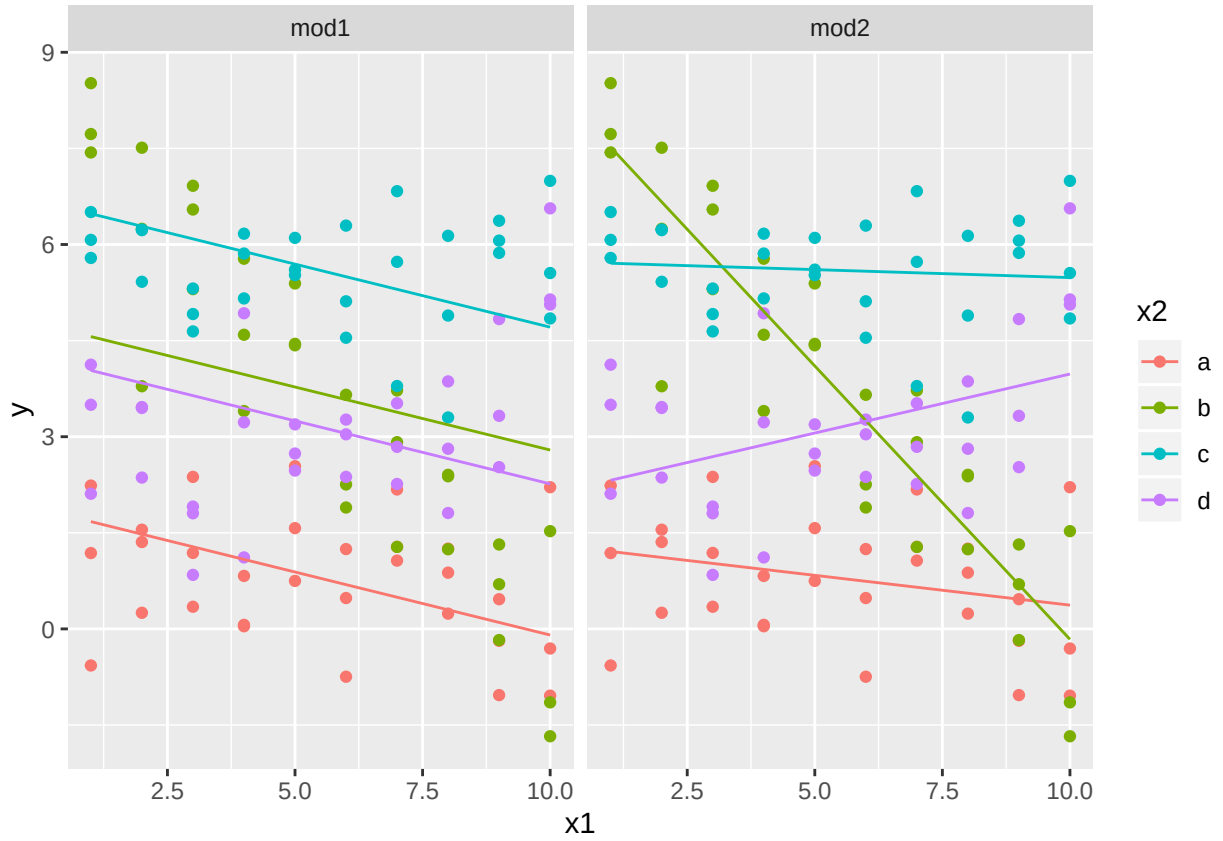
- Intercept is the expected y value when x1 and x2 are both equal to 0.
- *-0.09302444* is the slope coefficient of x1 which shows that on average, y is *0.09302444* units smaller in the data for observations with one unit larger x1 but with the same x2. In this case this is only true when x2=a.
- *7.06937991* is the slope coefficient of x2 where x2=b. It shows that on average, y is *7.06937991* units larger in the data for observations with one unit larger x2 but with the same x1.
- *4.43089525* is the slope coefficient of x2 where x2=c. It shows that on average, y is *4.43089525* units larger in the data for observations with one unit larger x2 but with the same x1.
- *0.83455115* is the slope coefficient of x2 where x2=d. It shows that on average, y is *0.83455115* units larger in the data for observations with one unit larger x2 but with the same x1.
- *-0.76028528* is the difference between slope coefficients of x1 when x2=a and when x2=b. So when x2=b, the slope coefficient of x1 is *-0.85330972* which means that on average, y is *0.85330972* units smaller in the data for observations with one unit larger x1 but with the same x2, in this case it must be x2=b.
- *0.06815284* is the difference between slope coefficients of x1 when x2=a and when x2=c. So when x2=c, the slope coefficient of x1 is *-0.0248716* which means that on average, y is *0.0248716* units smaller in the data for observations with one unit larger x1 but with the same x2, in this case it must be x2=c.
- *0.27727920* is the difference between slope coefficients of x1 when x2=a and when x2=d. So when x2=d, the slope coefficient of x1 is *0.18425476* which means that on average, y is *0.18425476* units larger in the data for observations with one unit larger x1 but with the same x2, in this case it must be x2=d.

Exercise 2.

Faceting with `gather_predictions()`

```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)

ggplot(sim3, aes(x1, y, colour = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```



Exercise 3.

21.2 For loops

Let's take a simple tibble

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

If we want to compute the mean for each column, instead of copy pasting we could use a for loop:

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
```

```
## [1]  0.47986414  0.06407867 -0.17757217  0.45606430
```

Every for loop has three components:

- The output: output <- vector("double", length(x)).

- The sequence: i in seq_along(df). This determines what to loop over: each run of the for loop will assign i to a different value from seq_along(df). It's useful to think of i as a pronoun, like "it".
- The body: output[[i]] <- median(df[[i]]). This is the code that does the work. It's run repeatedly, each time with a different value for i. The first iteration will run output[[1]] <- median(df[[1]]), the second will run output[[2]] <- median(df[[2]]), and so on.

21.3 For loop variations

There are four variations on the basic theme of the for loop:

1. Modifying an existing object, instead of creating a new object.

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

2. Looping over names or values, instead of indices.

- Loop over the elements: `for (x in xs)`. This is most useful if we only care about side-effects, like plotting or saving a file, because it's difficult to save the output efficiently.
- Loop over the names: `for (nm in names(xs))`. This gives us name, which we can use to access the value with `x[[nm]]`. This is useful if we want to use the name in a plot title or a file name. If we're creating named output, we have to make sure to name the results vector like so:

```
results <- vector("list", length(x))
names(results) <- names(x)
```

3. Handling outputs of unknown length.

Sometimes we might not know how long the output will be so we might be tempted to solve this problem by progressively growing the vector. A better solution to save the results in a list, and then combine into a single vector after the loop is done:

```
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
str(out)

str(unlist(out))
```

4. Handling sequences of unknown length.

Sometimes we don't even know how long the input sequence should run for. This is common when doing simulations. We can't do that sort of iteration with the for loop. Instead, we can use a while loop. A while loop is simpler than for loop because it only has two components, a condition and a body:

```
while (condition) {
  # body
}
```

A while loop is also more general than a for loop, because you can rewrite any for loop as a while loop, but you can't rewrite every while loop as a for loop:

```
for (i in seq_along(x)) {
  # body
}

# Equivalent to
i <- 1
while (i <= length(x)) {
  # body
  i <- i + 1
}
```

Here's how we could use a while loop to find how many tries it takes to get three heads in a row:

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
```

```
## [1] 12
```

21.4 For loops vs. functionals

For loops are not as important in R as they are in other languages because R is a functional programming language. This means that it's possible to wrap up for loops in a function, and call that function instead of using the for loop directly.

To see why this is important, consider (again) this simple data frame:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Imagine we want to compute the mean of every column. We could do that with a for loop:

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
```

```
## [1]  0.268129785  0.091721298 -0.006490634 -0.412725528
```

We realise that we're going to want to compute the means of every column pretty frequently, so extract it out into a function:

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

But then we think it'd also be helpful to be able to compute the median, and the standard deviation, so we copy and paste your col_mean() function and replace the mean() with median() and sd():

```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

Uh oh! We've copied-and-pasted this code twice, so it's time to think about how to generalise it. Notice that most of this code is for-loop boilerplate and it's hard to see the one thing (mean(), median(), sd()) that is different between the functions.

What would you do if you saw a set of functions like this:

```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

Hopefully, you'd notice that there's a lot of duplication, and extract it out into an additional argument:

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

You've reduced the chance of bugs (because you now have 1/3 of the original code), and made it easy to generalise to new situations.

We can do exactly the same thing with col_mean(), col_median() and col_sd() by adding an argument that supplies the function to apply to each column:

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
```

```
## [1]  0.05561572 -0.23074106  0.16326567 -0.64843006
```

```
col_summary(df, mean)
```

```
## [1]  0.268129785  0.091721298 -0.006490634 -0.412725528
```

Short example

There were plenty of examples for for loops so I'm trying to see how the while function works by adding up prices of diamonds in the diamonds dataset:

```
i <- 1

sum_prices = 0

while (i <= length(diamonds$price)) {
  sum_prices <- sum_prices + diamonds$price[i]
  i <- i + 1
}

sum_prices
```

```
## [1] 212135217
```