Functions: Part 3

Daniel Anderson
Week 7, Class 2

Agenda

- What makes functions "good"
- Return values
- Thinking more about function names
- Non-standard evaluation
- An example: Cohen's d
- Practice (if there's time)

Learning objectives

- Understand how functions build on top of each other and why "only do one thing" is a good mantra
- Understand non-standard evaluation is, even if you aren't able to fully work with it

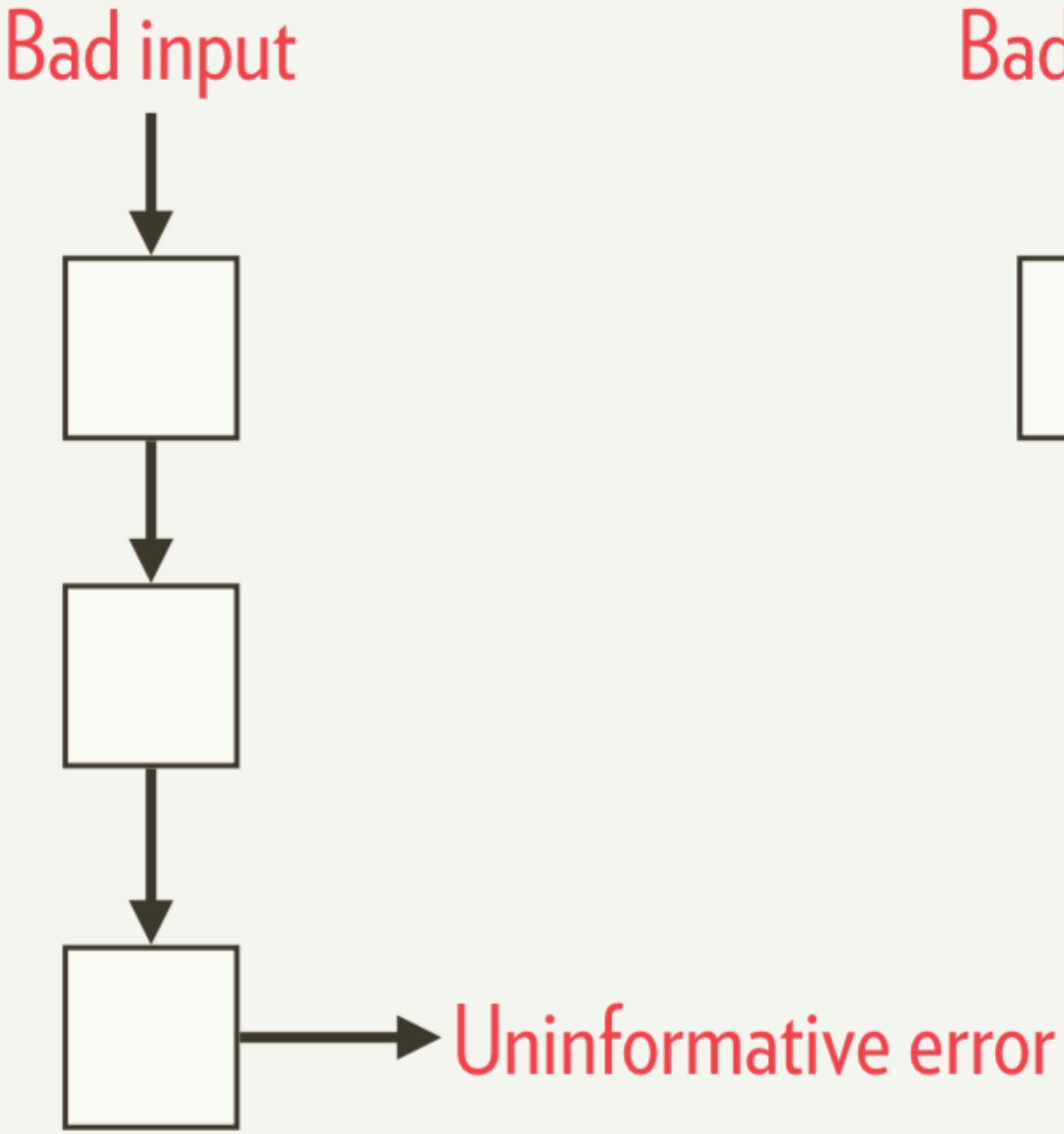
Brainstorm

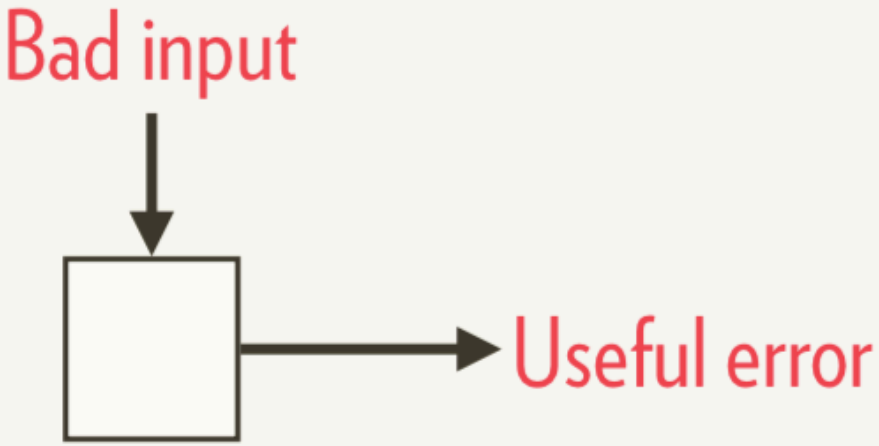
What makes a function "good" or "bad"

02:00

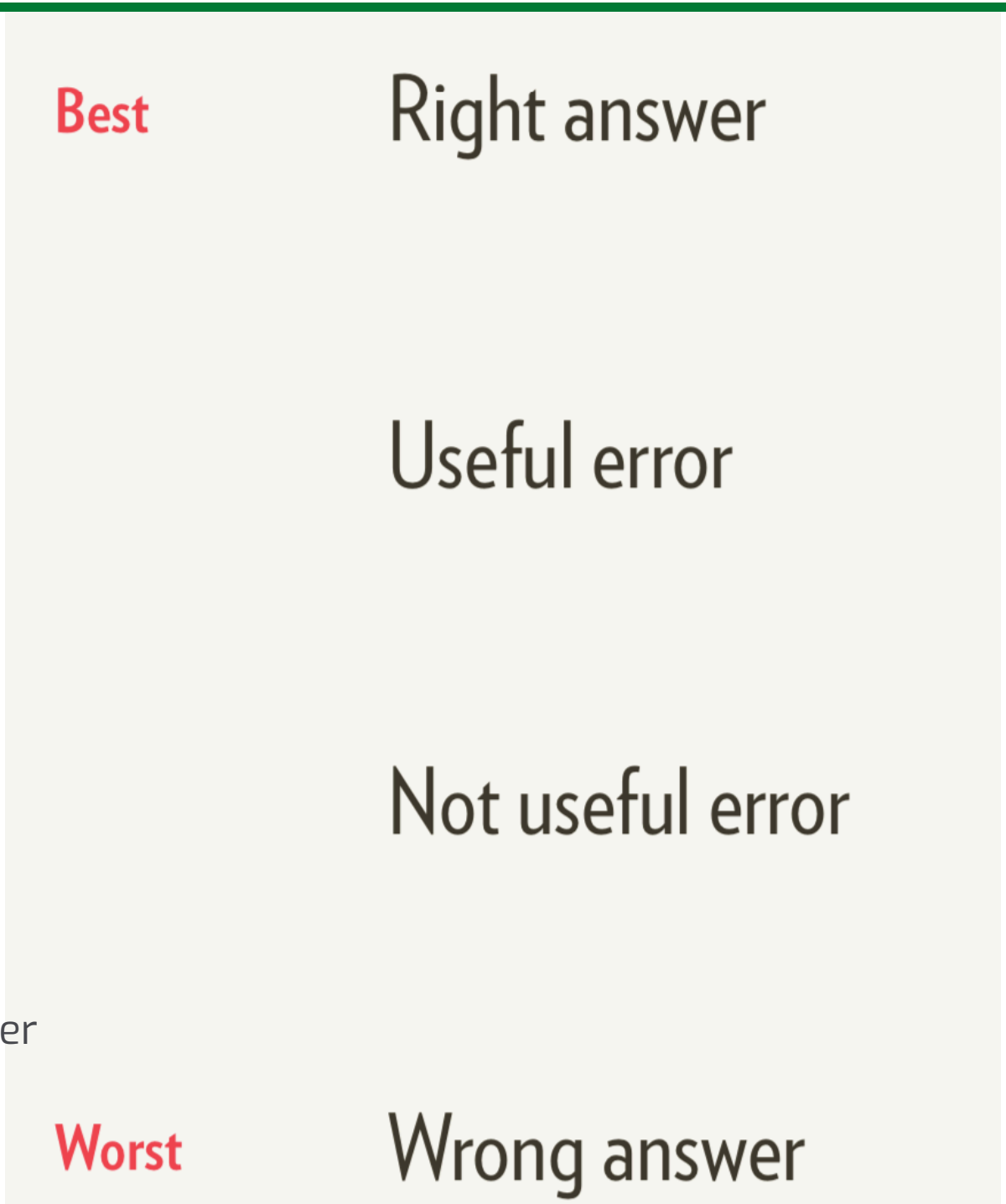
Two important tensions for understanding base R

Slide from Hadley Master R training





Slide from Hadley Master R training



Slide from

Hadley Master

R training

What does this mean operationally?

- Your function should do ONE thing (and do it well)
- Careful when naming functions - be as clear as possible
- Embed useful error messages and warnings
    - Particularly if you're working on a package or set of functions or others are using your functions
- Refactor your code to be more clear after initial drafts (it's okay to be messy on a first draft)

Example 1

- Anything we can do to clean this up?

```
both_na <- function(x, y) {
    if(length(x) != length(y)) {
        lx <- length(x)
        ly <- length(y)

        v_lngths <- paste0("x = ", lx, ", y = ", ly)

        if(lx %% ly == 0 | ly %% lx == 0) {
            warning("Vectors were recycled (", v_lngths, ")")
        }
        else {
            stop("Vectors are of different lengths and are not recyclable:",
                 v_lngths)
        }
    }
    sum(is.na(x) & is.na(y))
}
```

Calculate if recyclable

```
recyclable <- function(x, y) {
    test1 <- length(x) %% length(y)
    test2 <- length(y) %% length(x)

    any(c(test1, test2) == 0)
}
```

Test it

```
a <- c(1, NA, NA, 3, 3, 9, NA)
b <- c(NA, 3, NA, 4, NA, NA, NA)

recyclable(a, b)
```

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

```
recyclable(a, c(b, b))
```

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

```
recyclable(a, c(b, b, b))
```

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

```
recyclable(c(a, a), c(b, b, b))
```

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

Revision

```
both_na <- function(x, y) {

    if(!recyclable(x, y)) {
        stop("Vectors are of different lengths and are not recyclable: ",
             "(x = ", length(x),
             ", y = ", length(y), ")")
    }

    if(length(x) == length(y)) {
        return(sum(is.na(x) & is.na(y)))
    }

    if(recyclable(x, y)) {
        warning("Vectors were recycled (",
                "x = ", length(x),
                ", y = ", length(y), ")")
        return(sum(is.na(x) & is.na(y)))
    }
}
```

Test it

```
both_na(a, b)
```

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

```
both_na(a, c(b, b))
```

```
## Warning in both_na(a, c(b, b)): Vectors were recycled (x = 7, y = 14)
```

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

```
both_na(c(a, b), c(b, b, b))
```

```
## Error in both_na(c(a, b), c(b, b, b)): Vectors are of different lengths and are
```

```
both_na(c(a, a), b)
```

```
## Warning in both_na(c(a, a), b): Vectors were recycled (x = 14, y = 7)
```

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

Anything else?

Make errors/warnings a function

```
check_lengths <- function(x, y) {
    if(!length(x) == length(y)) {
        if(recyclable(x, y)) {
            warning("Vectors were recycled (",
                    "x = ", length(x),
                    ", y = ", length(y), ")")
        }
        else {
            stop("Vectors are of different lengths and are not recyclable: '
             "(x = ", length(x),
             ", y = ", length(y), ")")
        }
    }
}
```

Revision 2

```
both_na <- function(x, y) {
    check_lengths(x, y)
    sum(is.na(x) & is.na(y))
}
```

Test it

```
both_na(a, b)
```

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

```
both_na(a, c(b, b))
```

```
## Warning in check_lengths(x, y): Vectors were recycled (x = 7, y = 14)
```

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

```
both_na(c(a, b), c(b, b, b))
```

```
## Error in check_lengths(x, y): Vectors are of different lengths and are not recy
```

```
both_na(c(a, a), b)
```

```
## Warning in check_lengths(x, y): Vectors were recycled (x = 14, y = 7)
```

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

Why would we do this?

- In this case - more readable code
- We might re-use the `recyclable` or `check_lengths` functions in other/new functions
- Helps make de-bugging easier

Quick de-bugging example

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) {
    stop("`d` must be numeric", call. = FALSE)
  }
  d + 10
}
```

traceback

```
f("a")
```

```
## Error: `d` must be numeric
```

```
traceback()
```

```
## No traceback available
```

Thinking more about return values

- Our first revision was the first instance we've seen where there is a possible return value before the end of the function.
- By default the function will return the last thing that is evaluated
- Override this behavior with `return`

Pop quiz

- What will the following return?

```
add_two <- function(x) {
    result <- x + 2
}
```

Answer: Nothing! Why?

```
add_two(7)
add_two(5)
```

Specify the return value

The below are all equivalent, and all result in the same function behavior

```
add_two.1 <- function(x) {
    result <- x + 2
    result
}
add_two.2 <- function(x) {
    x + 2
}
```

```
add_two.3 <- function(x) {
    result <- x + 2
    return(result)
}
```

When to use `return`?

Generally reserve `return` for you're returning a value prior to the full evaluation of the function. Otherwise, use `.1` or `.2` methods from prior slide.

Thinking about function names

Which of these is most intuitive?

```
f <- function(x) {
    x <- sort(x)
    data.frame(value = x,
               p = ecdf(x)(x))
}

ptile <- function(x) {
    x <- sort(x)
    data.frame(value = x,
               ptile = ecdf(x)(x))
}
percentile_df <- function(x) {
    x <- sort(x)
    data.frame(value = x,
               percentile = ecdf(x)(x))
}
```

Output

- The descriptive nature of the output can also help
- Maybe a little too tricky but...

```
percentile_df <- function(x) {
    arg <- as.list(match.call())
    x <- sort(x)
    d <- data.frame(value = x,
                    percentile = ecdf(x)(x))

    names(d)[1] <- paste0(as.character(arg$x), collapse = "_")
    d
}
```

```
random_vector <- rnorm(100)
tail(percentile_df(random_vector))
```

```
##     random_vector percentile
## 95       1.630973       0.95
## 96       1.665885       0.96
## 97       1.685320       0.97
## 98       1.918553       0.98
## 99       2.540644       0.99
## 100      2.698724       1.00
```

```
head(percentile_df(rnorm(50)))
```

```
##    rnorm_50 percentile
## 1 -2.413570       0.02
## 2 -1.780276       0.04
## 3 -1.429753       0.06
## 4 -1.334554       0.08
## 5 -1.252887       0.10
## 6 -1.176174       0.12
```

How do we figure these things out?

- Change the return value to whatever it is you want, and run it over and over.

(demo)

Thinking about dependencies

- What's the purpose of the function?
  - Just your use? Never needed again? Don't worry about it at all.
  - Mass scale? Worry a fair bit, but make informed decisions.
- What's the likelihood of needing to reproduce the results in the future?
  - If high, worry more.
- Consider using name spacing (`::`)

Non-standard evaluation (NSE)

A high-level look

Note

- Were it not for the tidyverse, I would not even mention NSE
- Generally, it's not an incredibly important topic
- But, NSE is ubiquitous in the tidyverse - literally just about everything uses NSE, which makes programming with tidyverse functions more difficult

What is NSE

- Implementation of different scoping rules
- In dplyr and many others, arguments are evaluated inside the specified data frames, rather than the current or global environment.

How?

(a) Capture an expression (quote it) (b) Use the expression within the correct context (evaluate it)

So, `x` is evaluated as, e.g., `df$x` rather than `globalenv()$x`.

Could be applied with our previous example!

- Here `base::substitute`

```
percentile_df <- function(x) {
    sorted <- sort(x)
    d <- data.frame(sorted, percentile = ecdf(sorted)(sorted))
    names(d)[1] <- paste0(substitute(x), collapse = "_")
    d
}
percentile_df(rnorm(100, 5, 0.2)) %>%
    head()
```

```
##   rnorm_100_5_0.2 percentile
## 1        4.629187       0.01
## 2        4.681535       0.02
## 3        4.693287       0.03
## 4        4.720016       0.04
## 5        4.727363       0.05
## 6        4.729581       0.06
```

Confusing

- Outside of a function, `substitute` operates just like `quote` - it quotes the input.
- Inside of a function, `substitute` does as its name implies - it substitutes the input for the name.

Example

```
quote(subset(df, select = var))
```

```
## subset(df, select = var)
```

```
substitute(subset(df, select = var))
```

```
## subset(df, select = var)
```

```
extract_var <- function(df, var) {
    substitute(df)
}
extract_var(mtcars)
```

```
## mtcars
```

Actually getting this thing to work

```
extract_var <- function(df, var) {
    subset(eval(substitute(df)),
           select = var)
}
extract_var(mtcars, "mpg")
```

```
##                      mpg
## Mazda RX4           21.0
## Mazda RX4 Wag       21.0
## Datsun 710          22.8
## Hornet 4 Drive      21.4
## Hornet Sportabout   18.7
## Valiant             18.1
## Duster 360          14.3
## Merc 240D           24.4
## Merc 230            22.8
## Merc 280            19.2
## Merc 280C           17.8
## Merc 450SE          16.4
## Merc 450SL          17.3
## Merc 450SLC         15.2
```

Why `eval`

- `substitute` is quoting the input, but we then need to evaluate it.
- All of this is rather confusing
- The tidyverse uses it so frequently, they've decided to implement their own version, called `tidyeval`, which we'll get to in a minute.

Why is NSE used so frequently in the tidyverse?

```
select(mpg,
       manufacturer, model, hwy)
```

```
## # A tibble: 234 x 3
##    manufacturer model          hwy
##    <chr>        <chr>        <int>
##  1 audi         a4              29
##  2 audi         a4              29
##  3 audi         a4              31
##  4 audi         a4              30
##  5 audi         a4              26
##  6 audi         a4              26
##  7 audi         a4              27
##  8 audi         a4 quattro      26
##  9 audi         a4 quattro      25
## 10 audi         a4 quattro      28
## # … with 224 more rows
```

- It makes interactive work easier!
- But programming is a harder...
- Without NSE, `select` and similar functions would not know where `manufacturer`, `model`, or `hwy` "live". It would be looking for objects in the global environment with these names.

dplyr programming fail

- Let's say we wanted a function that returned means in a nice table-y format for a variable by two groups (e.g., cross-tab sort of format)
- Typically, we would start by solving this problem for a single situation, then we'd generalize it to a function.
- Let's do it!

```
mtcars %>%
    group_by(cyl, gear) %>%
    summarize(mean = mean(mpg, na.rm = TRUE)) %>%
    spread(cyl, mean)
```

```
## # A tibble: 3 x 4
##    gear    `4`    `6`    `8`
##   <dbl>  <dbl> <dbl> <dbl>
## 1     3 21.5   19.75 15.05
## 2     4 26.925 19.75 NA
## 3     5 28.2   19.7  15.4
```

- Try generalizing this to a function writing a fun

04:00

Generalize to a function

Typically, we would expect something like this to work

```
group_means <- function(data, outcome, group_1, group_2) {
    data %>%
        group_by(group_1, group_2) %>%
        summarize(mean = mean(outcome, na.rm = TRUE)) %>%
        spread(group_1, mean)
}
```

But it doesn't...

```
group_means(mtcars, mpg, cyl, gear)
```

```
## Error: Column `group_1` is unknown
```

```
group_means(diamonds, price, cut, clarity)
```

```
## Error: Column `group_1` is unknown
```

Why?

- It's looking for an object called `group_1` that doesn't exist inside the function or in the global workspace!

Solution

- Quote it, and evaluate it in the correct place

The {rlang} version

```
group_means <- function(data, outcome, group_1, group_2) {
    out <- enquo(outcome) # Quote the inputs
    g1 <- enquo(group_1)
    g2 <- enquo(group_2)

    data %>%
        group_by(!!g1, !!g2) %>% # !! to evaluate (bang-bang)
        summarize(mean = mean(!!out, na.rm = TRUE)) %>%
        spread(!!g1, mean)
}
```

```
group_means(mtcars, mpg, cyl, gear)
```

```
## # A tibble: 3 x 4
##    gear    `4`   `6`   `8`
##   <dbl>  <dbl> <dbl> <dbl>
## 1     3 21.5   19.75 15.05
## 2     4 26.925 19.75 NA
## 3     5 28.2   19.7  15.4
```

```
group_means(diamonds, price, cut, clarity)
```

```
## # A tibble: 8 x 6
##   clarity     Fair     Good `Very Good`  Premium    Ideal
##   <ord>      <dbl>    <dbl>       <dbl>    <dbl>    <dbl>
## 1 I1      3703.533 3596.635    4078.226 3947.332 4335.726
## 2 SI2     5173.916 4580.261    4988.688 5545.937 4755.953
## 3 SI1     4208.279 3689.533    3932.391 4455.269 3752.118
## 4 VS2     4174.724 4262.236    4215.760 4550.331 3284.550
## 5 VS1     4165.141 3801.446    3805.353 4485.462 3489.744
## 6 VVS2    3349.768 3079.108    3037.765 3795.123 3250.290
## 7 VVS1    3871.353 2254.774    2459.441 2831.206 2468.129
## 8 IF      1912.333 4098.324    4396.216 3856.143 2272.913
```

Alternative: Pass the dots!

- Note, I've made the function a bit simpler here by removing the spread

```
group_means2 <- function(data, outcome, ...) {
    out <- enquo(outcome) # Still have to quote the outcome

    data %>%
        group_by(...) %>%
        summarize(mean = mean(!!out, na.rm = TRUE))
}

group_means2(mtcars, mpg, cyl, gear)
```

```
## # A tibble: 8 x 3
## # Groups:   cyl [3]
##     cyl  gear   mean
##   <dbl> <dbl>  <dbl>
## 1     4     3 21.5
## 2     4     4 26.925
## 3     4     5 28.2
## 4     6     3 19.75
## 5     6     4 19.75
## 6     6     5 19.7
## 7     8     3 15.05
```

Added benefit

I can now also pass as many columns as I want, and it will still work!

```
group_means2(diamonds, price, cut, clarity, color)
```

```
## # A tibble: 276 x 4
## # Groups:   cut, clarity [40]
##    cut   clarity color     mean
##    <ord> <ord>   <ord>    <dbl>
##  1 Fair  I1      D     7383
##  2 Fair  I1      E     2095.222
##  3 Fair  I1      F     2543.514
##  4 Fair  I1      G     3187.472
##  5 Fair  I1      H     4212.962
##  6 Fair  I1      I     3501
##  7 Fair  I1      J     5795.043
##  8 Fair  SI2     D     4355.143
##  9 Fair  SI2     E     4172.385
## 10 Fair  SI2     F     4520.112
## # … with 266 more rows
```

Challenge

- Write a function that summarizes any numeric columns by returning the mean, standard deviation, min, and max values.
- For bonus points, embed a meaningful error message if the columns supplied are not numeric.

Example

```
summarize_cols(diamonds, depth, table, price)
```

```
## # A tibble: 3 x 5
##   var          mean          sd   min   max
##   <chr>       <dbl>       <dbl> <dbl> <dbl>
## 1 depth   61.74940    1.432621    43    79
## 2 price 3932.800   3989.440      326 18823
## 3 table   57.45718    2.234491    43    95
```

07:00

Pass the dots!

```
summarize_cols <- function(data, ...) {
    data %>%
        select(...) %>%
        gather(var, val) %>%
        group_by(var) %>%
        summarize(mean = mean(val, na.rm = TRUE),
                  sd = sd(val, na.rm = TRUE),
                  min = min(val, na.rm = TRUE),
                  max = max(val, na.rm = TRUE))
}
```

Example with plotting

- ggplot, for now, has a few tools to make programming with it easier
- These have all been soft deprecated, in favor of {rlang}
- But overall we have the same basic problem

Does not work

```
check_linear <- function(data, x, y, se = TRUE) {
    p <- ggplot(data, aes(x, y)) +
              geom_point(color = "gray80")
  if(se) {
      p <- p +
                  geom_smooth(method = "lm") +
                  geom_smooth()
  }
  else {
      p <- p +
                  geom_smooth(method = "lm", se = FALSE) +
                  geom_smooth(se = FALSE)
  }
  p
}
check_linear(mtcars, disp, mpg)
```

```
## Error in FUN(X[[i]], ...): object 'disp' not found
```

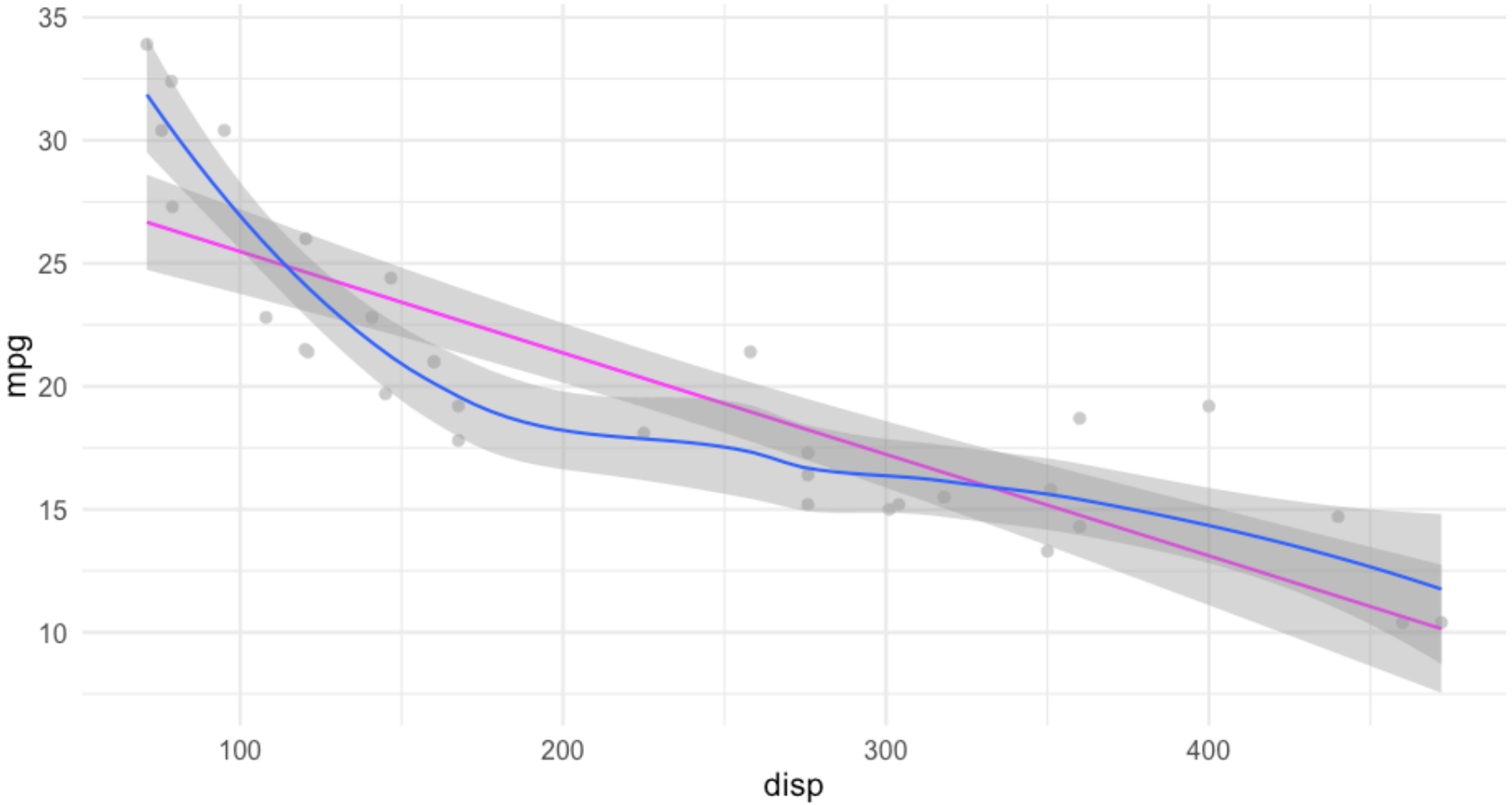
Use `aes_string`

Soft deprecated

(notice dots being passed now too)

```
check_linear <- function(data, x, y, ...) {
    ggplot(data, aes_string(x, y)) +
      geom_point(color = "gray80") +
        geom_smooth(method = "lm",
                    color = "magenta",
                      ...) +
        geom_smooth(...)
}
```

```
check_linear(mtcars, "disp", "mpg")
```

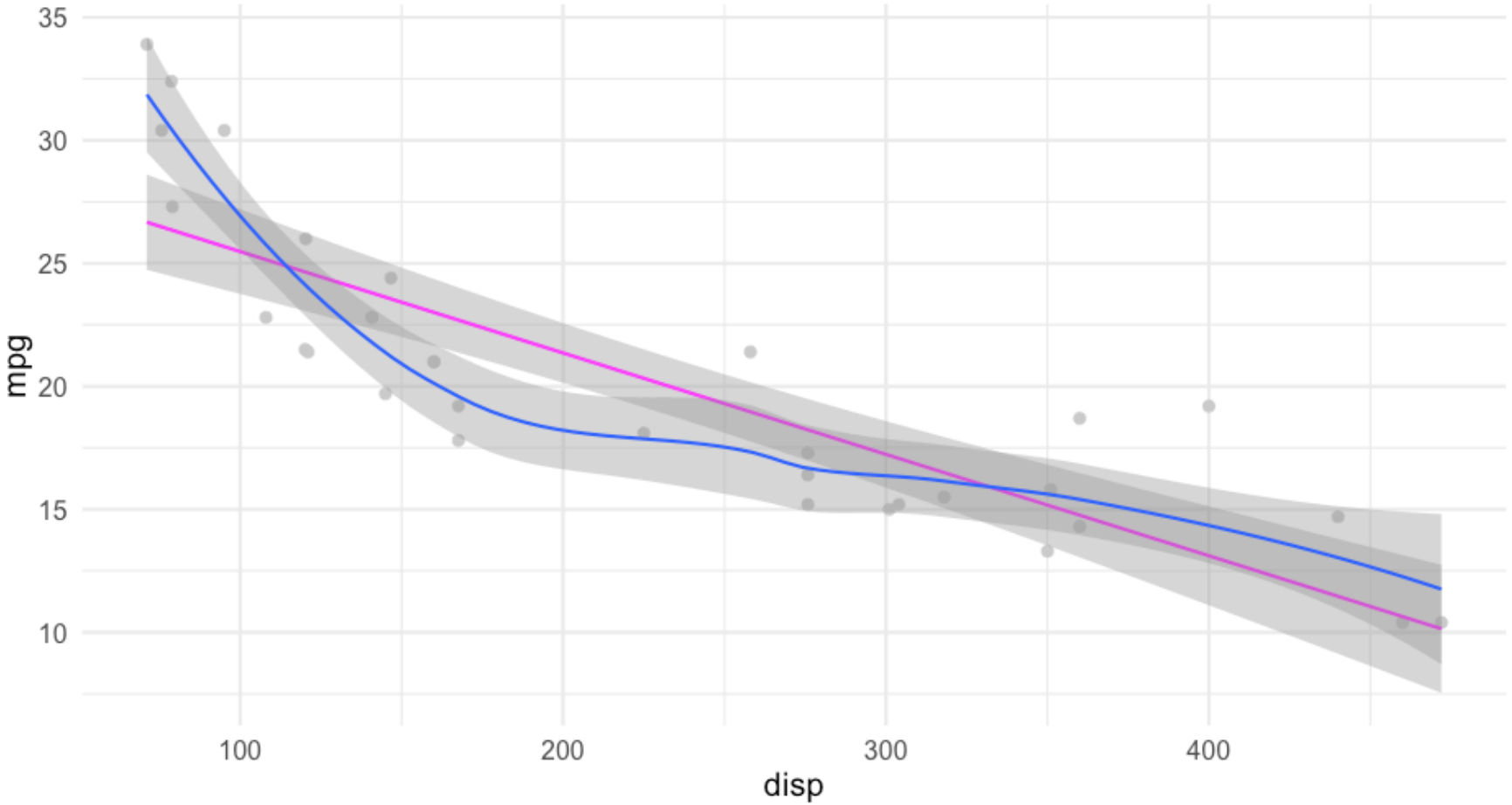


Use aes_ with as.name

Soft deprecated

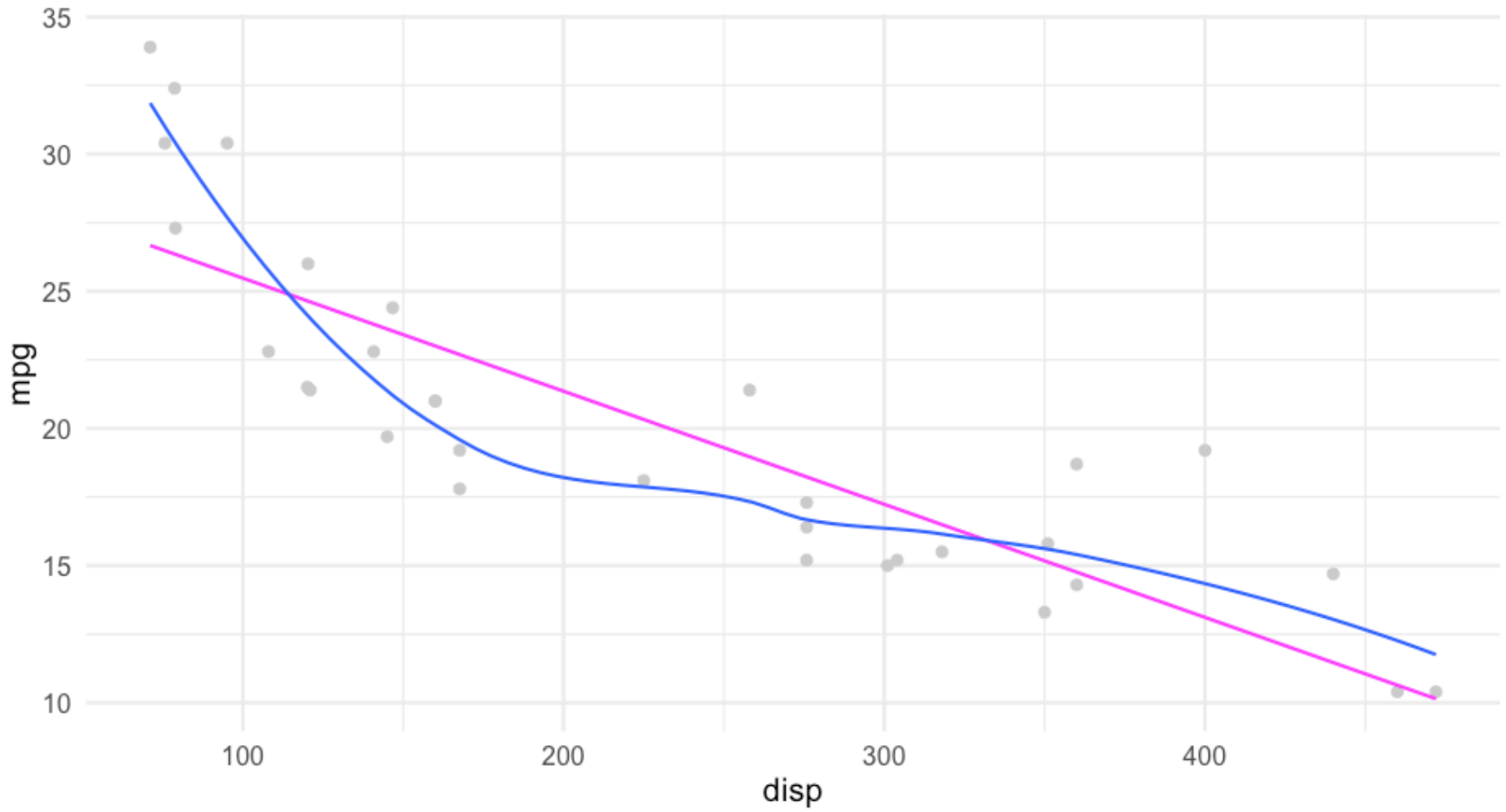
```
check_linear <- function(data, x, y, ...) {
    ggplot(data, aes_(as.name(x), as.name(y))) +
      geom_point(color = "gray80") +
        geom_smooth(method = "lm",
                    color = "magenta",
                      ...) +
        geom_smooth(...)
}
```

```
check_linear(mtcars, "disp", "mpg")
```



Passing dots

```
check_linear(mtcars, "disp", "mpg", se = FALSE)
```

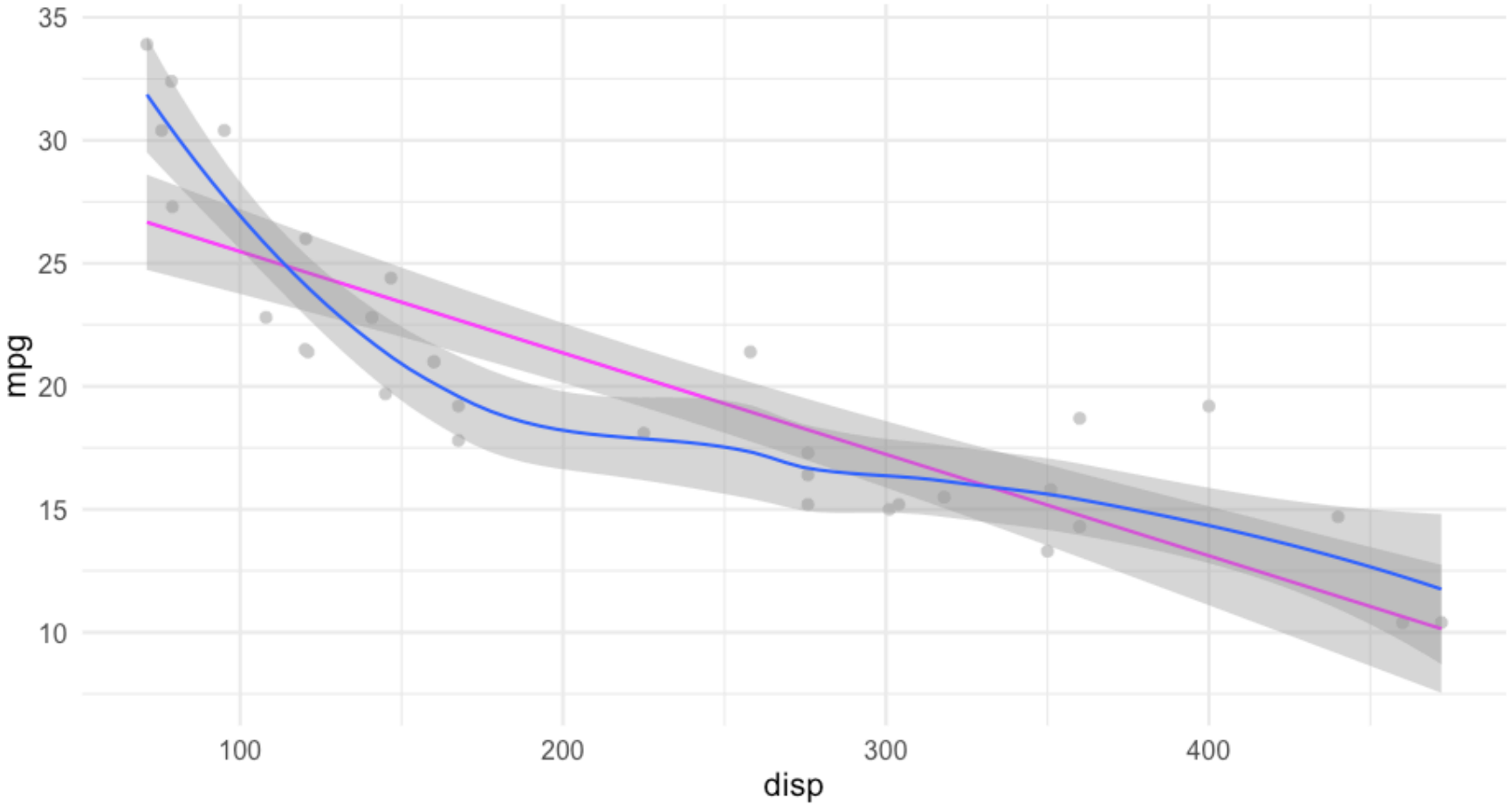


Use tidyeval

Method to use going forward

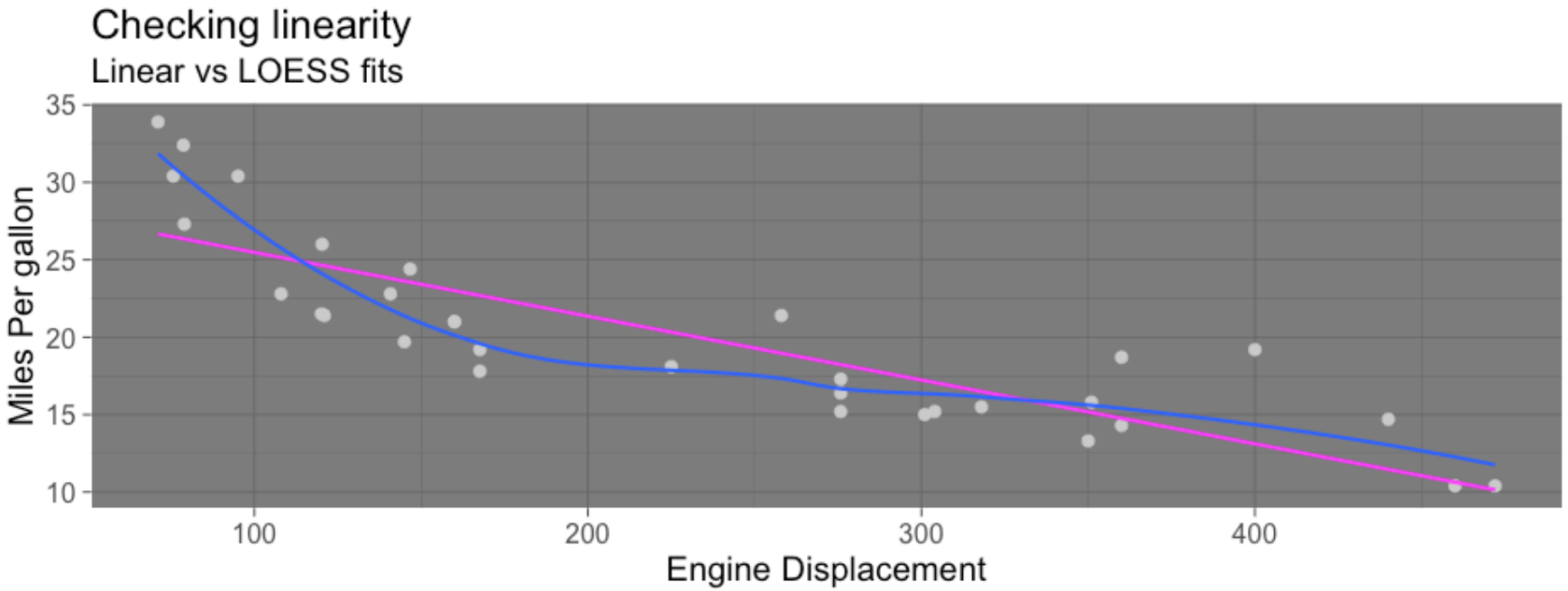
```
check_linear <- function(data, x, y, ...) {
    xvar <- enquo(x)
    yvar <- enquo(y)

    ggplot(data, aes(!!xvar, !!yvar)) +
      geom_point(color = "gray80") +
        geom_smooth(method = "lm",
                    color = "magenta",
                      ...) +
        geom_smooth(...)
}
```

```
check_linear(mtcars, disp, mpg)
```



Add ggplot functions

```
check_linear(mtcars, disp, mpg, se = FALSE) +
    labs(title = "Checking linearity",
         subtitle = "Linear vs LOESS fits",
         x = "Engine Displacement",
         y = "Miles Per gallon") +
    theme_dark(20)
```



Overall takeaway

- Programming with the tidyverse is a bit more difficult
- Also introduces dependencies
- Doesn't mean it's not worth it, if the context fits.
    - If you're plotting, I think `ggplot` is worth it