

More Functions & Lab 3

Daniel Anderson

Week 7

Agenda

- What makes functions "good"
- Building up functions
- Non-standard evaluation

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"

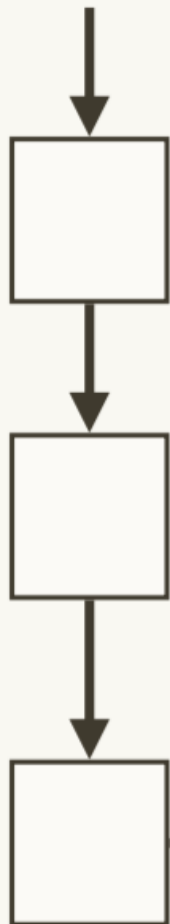
02:00

Two important tensions for understanding base R



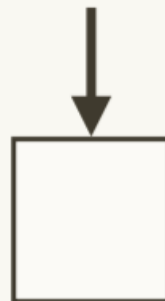
Slide from Hadley Master R training

Bad input



Uninformative error

Bad input



Useful error

Slide from Hadley Master R training

Best

Right answer

Useful error

Not useful error

Slide from

Hadley Master

R training

Worst

Wrong answer

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

02:00

- 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_lengths <- paste0("x = ", lx, ", y = ", ly)  
  
    if(lx %% ly == 0 | ly %% lx == 0) {  
      warning("Vectors were recycled (", v_lengths, ")")  
    }  
    else {  
      stop("Vectors are of different lengths and are not recycled (",  
           v_lengths)  
    }  
  }  
  sum(is.na(x) & is.na(y))  
}
```

Brain storm with your neighbor

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

Test it

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

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

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

```
## Warning: Vectors were recycled
```

```
## x = 7
```

```
## y = 14
```

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

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

```
## Error: Vectors are of different lengths and are not recyclable:  
## x = 14  
## y = 21
```

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

```
## Warning: 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\n",  
        "x = ", length(x), "\n",  
        "y = ", length(y),  
        call. = FALSE  
      )  
    }  
    else {  
      stop(  
        "Vectors are of different lengths and are not recyclable\n",  
        "x = ", length(x), "\n",  
        "y = ", length(y),  
        call. = FALSE  
      )  
    }  
  }  
}
```

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: Vectors were recycled
```

```
## x = 7
```

```
## y = 14
```

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

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

```
## Error: Vectors are of different lengths and are not recyclable:  
## x = 14  
## y = 21
```

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

```
## Warning: 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")  
traceback()
```

```
5: stop("`d` must be numeric", call. = FALSE) at #3  
4: i(c) at #1  
3: h(b) at #1  
2: g(a) at #1  
1: f("a")  
-
```

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`.

Example

Using the `percentile_df` function we created previously

- 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          4.481375         0.01  
## 2          4.518488         0.02  
## 3          4.592985         0.03  
## 4          4.598114         0.04  
## 5          4.703700         0.05  
## 6          4.729181         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.

Better

Use NSE for both arguments

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

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4  
## [9] 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4  
## [17] 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3  
## [25] 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

- The above is equivalent to `df$var` but where both `df` and `var` can be swapped programmatically.
- The `var` argument is being substituted in for whatever the user supplies, and is being evaluated within the `df` environment

Maybe more simply

```
extract_var <- function(df, var) {  
  df[, as.character(substitute(var))]  
}  
extract_var(mtcars, mpg)
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4  
## [9] 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4  
## [17] 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3  
## [25] 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

- Why `as.character`? Otherwise it is a symbol, which won't work.
- Note we could add `drop = FALSE` to this if we wanted to maintain the data frame structure

Taking this even further

```
extract_vars <- function(df, ...) {  
  vars <- substitute(alist(...))  
  df[, as.character(vars)[-1]]  
}  
head(extract_vars(mtcars, mpg, cyl, disp))
```

```
##           mpg  cyl  disp  
## Mazda RX4      21.0    6  160  
## Mazda RX4 Wag  21.0    6  160  
## Datsun 710      22.8    4  108  
## Hornet 4 Drive  21.4    6  258  
## Hornet Sportabout 18.7    8  360  
## Valiant        18.1    6  225
```

- We've now basically replicated `dplyr::select`
- Notice the use of `[-1]`, because calling `as.character` on `vars` always returns `alist` as the first element in the vector

Why is NSE used so frequently in the tidyverse?

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

```
## # A tibble: 234 × 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
```

Interactive work!

- It makes interactive work easier!
- But this make programming with these functions 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)) %>%  
  pivot_wider(names_from = cyl, values_from = mean)
```

```
## # A tibble: 3 × 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 the above code into a function

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)) %>%  
    pivot_wider(names_from = group_1, values_from = mean)  
}
```

But it doesn't...

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

```
## Error in `group_by()`:  
## ! Must group by  
##   variables found in `.data`.  
## Column `group_1` is not found.  
## Column `group_2` is not found.
```

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

```
## Error in `group_by()`:  
## ! Must group by  
##   variables found in `.data`.  
## Column `group_1` is not found.  
## Column `group_2` is not found.
```

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)) %>%  
    pivot_wider(names_from = !!g1, values_from = mean)  
}
```



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

```
## # A tibble: 3 × 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 × 6
##   clarity    Fair    Good `Very Good`
##   <ord>    <dbl>    <dbl>    <dbl>
## 1 I1      3703.533 3596.635  4078.226
## 2 SI2     5173.916 4580.261  4988.688
## 3 SI1     4208.279 3689.533  3932.391
## 4 VS2     4174.724 4262.236  4215.760
## 5 VS1     4165.141 3801.446  3805.353
## 6 VVS2     3349.768 3079.108  3037.765
## 7 VVS1     3871.353 2254.774  2459.441
## 8 IF      1912.333 4098.324  4396.216
## # ... with 2 more variables: Premium <dbl>,
## #   Ideal <dbl>
```

Alternative: Pass the dots!

- Note, I've made the function a bit simpler here by removing the call to `pivot_wider()`

```
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 × 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
```

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 × 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
```

Wait a minute!

- This isn't the same thing

Correct

So in this case passing the dots won't fully fix our problem, but there [is](#) another alternative

Alternative syntax

```
group_means3 <- function(data, outcome, group_1, group_2) {  
  data %>%  
    group_by({{group_1}}, {{group_2}}) %>%  
    summarize(mean = mean({{outcome}}, na.rm = TRUE)) %>%  
    pivot_wider(names_from = {{group_1}}, values_from = mean)  
}  
  
group_means3(mtcars, mpg, cyl, gear)
```

```
## # A tibble: 3 × 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
```

Pipe-centric

- Because the **data** argument comes first, this function works just like any other in the tidyverse, e.g.,

```
diamonds %>%  
  filter(color == "E") %>%  
  select(carat, cut, clarity) %>%  
  group_means3(carat, cut, clarity)
```

```
## # A tibble: 8 × 6  
##   clarity      Fair      Good `Very Good`  
##   <ord>      <dbl>      <dbl>      <dbl>  
## 1 I1         0.9688889  1.330870    1.069545  
## 2 SI2        1.015641   0.8825743   0.9304045  
## 3 SI1        0.8670769  0.7238592   0.7230831  
## 4 VS2        0.6902381  0.739375    0.6644135  
## 5 VS1        0.6328571  0.6806742   0.6097952  
## 6 VVS2       0.6007692  0.5601923   0.4267114  
## 7 VVS1       0.64         0.4181395   0.4000588  
## 8 IF         NA          0.3733333   0.5793023  
## # ... with 2 more variables: Premium <dbl>,  
## #   Ideal <dbl>
```

Syntax note

- The `{{}}` syntax is obviously *much* easier to apply
- It gets you out of tidyeval complications maybe 80% of the time
- If you start doing a lot of programming w/the tidyverse, you'll probably want to read up more on tidyeval and get comfy with quoting/unquoting

Challenge

07:00

If we have time

- Write a function that summarizes any numeric columns by returning the mean, standard deviation, min, and max values.
- Solve on an example case first, then generalize to a function

Example:

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

```
## # A tibble: 3 × 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
```


Pass the dots!

```
summarize_cols <- function(data, ...) {  
  data %>%  
    select(...) %>%  
    pivot_longer(everything(),  
                 names_to = "var",  
                 values_to = "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))  
}
```

Pipe-centric again

Just putting data as the first argument leads to a lot of benefits

```
library(palmerpenguins)
penguins %>%
  select_if(is.numeric) %>%
  summarize_cols(everything())
```

```
## # A tibble: 5 × 5
##   var                mean      sd
##   <chr>             <dbl>    <dbl>
## 1 bill_depth_mm    17.15117  1.974793
## 2 bill_length_mm   43.92193  5.459584
## 3 body_mass_g      4201.754  801.9545
## 4 flipper_length_mm 200.9152  14.06171
## 5 year             2008.029   0.8183559
## # ... with 2 more variables: min <dbl>,
## #   max <dbl>
```

Example with
plotting

Linearity

- We want to check if an x/y relation is linear
- Function should produce linear and non-linear relations
- Optionally show the data

Maybe this?

```
check_linear <- function(data, x, y, se = TRUE,
                          points = FALSE) {
  p <- ggplot(data, aes(x, y))

  if(isTRUE(points)) {
    p <- p + geom_point(color = "gray80")
  }
  if(isTRUE(se)) {
    p <- p +
      geom_smooth(method = "lm") +
      geom_smooth()
  } else {
    p <- p +
      geom_smooth(method = "lm", se = FALSE) +
      geom_smooth(se = FALSE)
  }
  p
}
```

Nope

```
check_linear(penguins, bill_length_mm, bill_depth_mm)
```

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

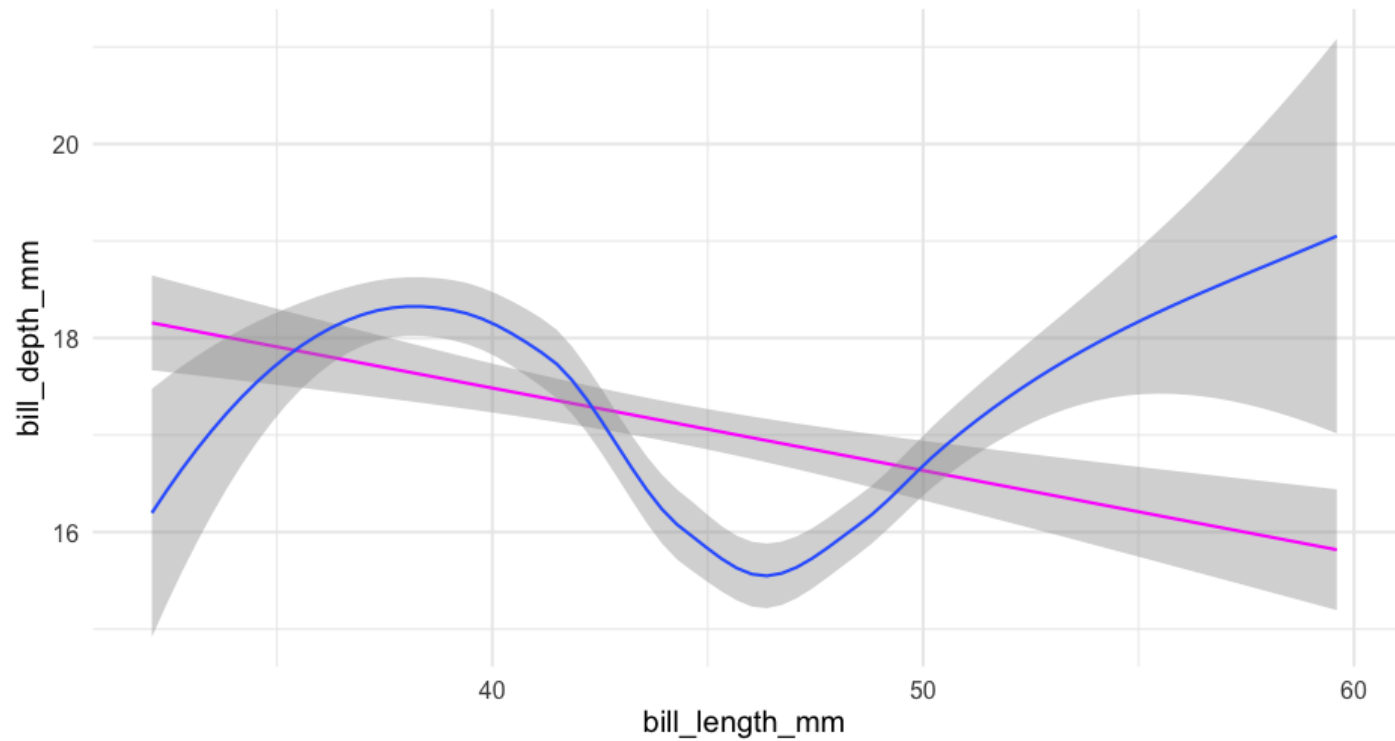
Use tidyeval

Note – there are other approaches too, but they are soft deprecated

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

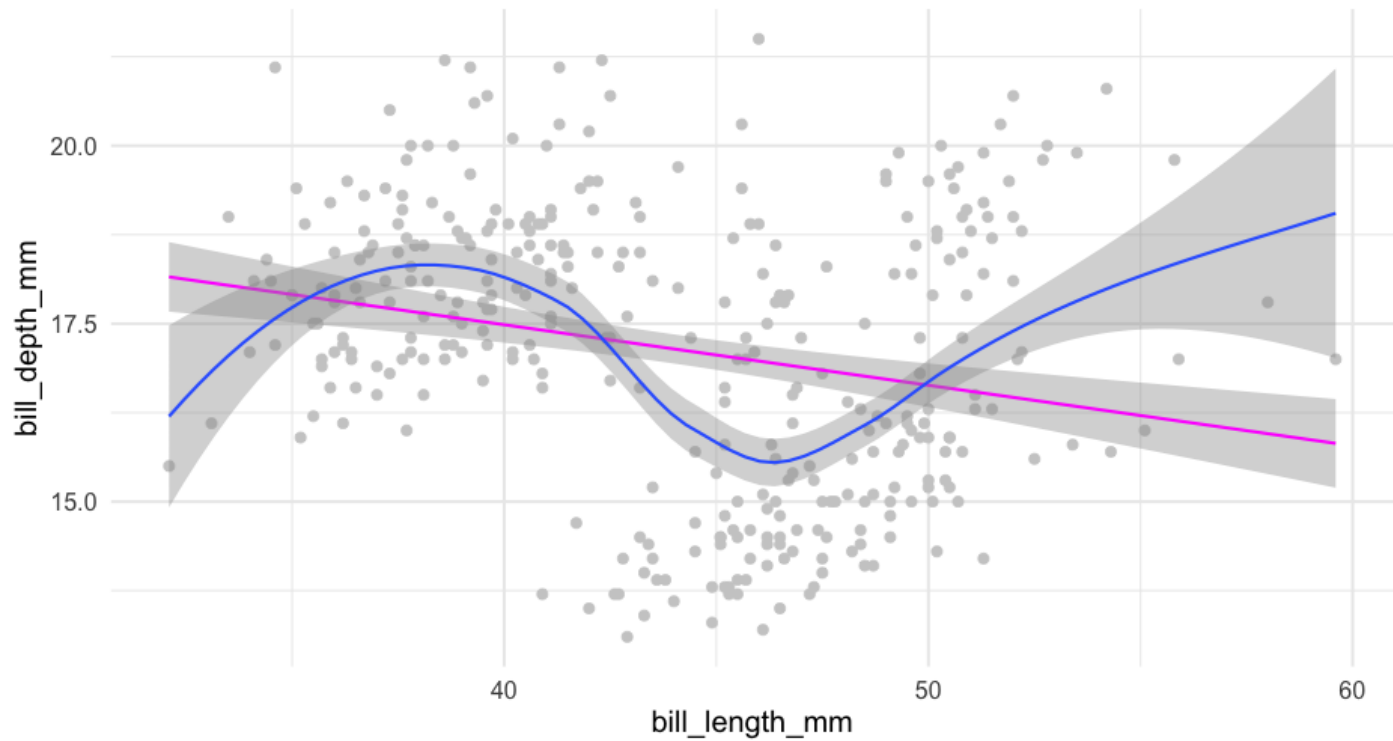
Notice I'm passing the dots here too to make it slightly more concise

```
check_linear(penguins, bill_length_mm, bill_depth_mm)
```



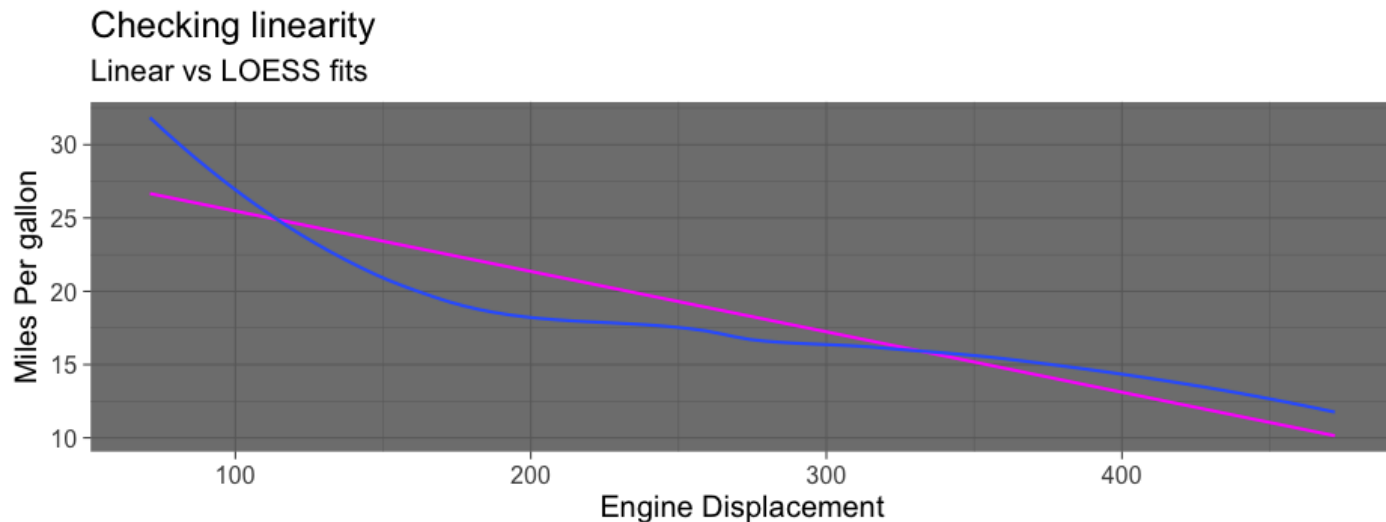
Add points

```
check_linear(penguins, bill_length_mm, bill_depth_mm,  
             points = TRUE)
```



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)
```



Building up plots

- Let's create a function that calculates means and standard errors for every numeric column in a data frame
- We'll then use this function to create a plotting function

Means/SE's

- First figure it out for a single example
- Note there's not built-in function for the standard error of the mean, which is $\frac{\sigma}{\sqrt{n}}$

```
# SE function
se <- function(x) {
  x <- x[!is.na(x)] # remove missing data
  sd(x) / sqrt(length(x))
}
```

Full means/SEs

```
penguins %>%
  select_if(is.numeric) %>%
  pivot_longer(everything(),
               names_to = "var",
               values_to = "val") %>%
  group_by(var) %>%
  summarize(mean = mean(val, na.rm = TRUE),
            se = se(val))
```

```
## # A tibble: 5 × 3
##   var          mean      se
##   <chr>      <dbl>    <dbl>
## 1 bill_depth_mm    17.15117  0.1067846
## 2 bill_length_mm   43.92193  0.2952205
## 3 body_mass_g     4201.754  43.36473
## 4 flipper_length_mm 200.9152  0.7603704
## 5 year           2008.029  0.04412279
```

Challenge

Try to generalize the code on the previous slide into a function. Test it on the `forcats::gss_cat` data.

03:00

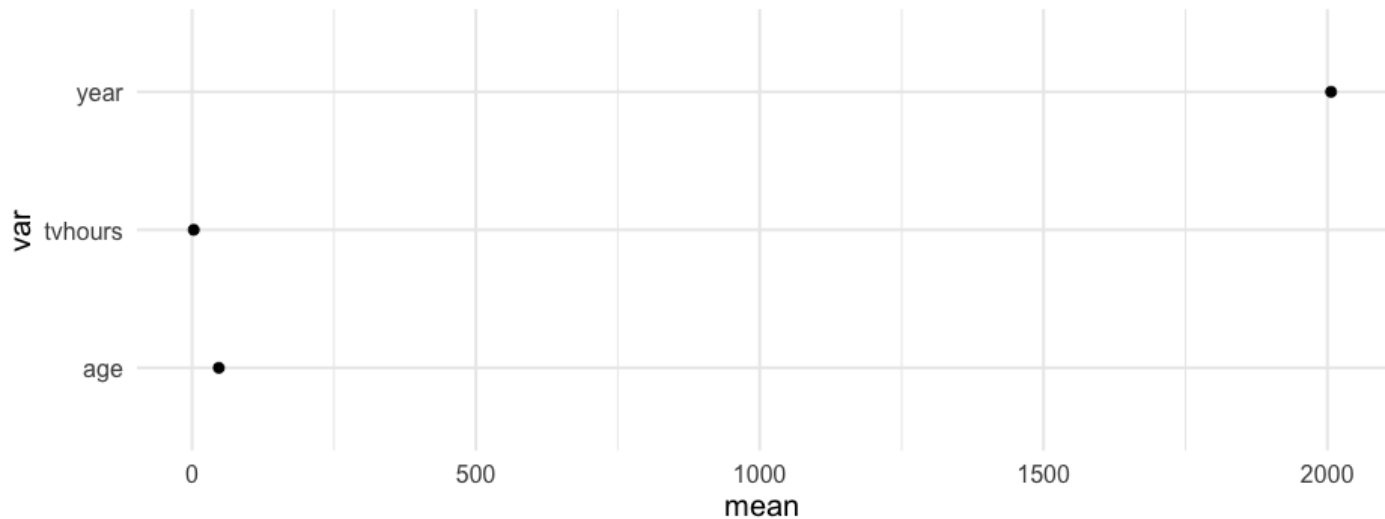
Translate to a function

```
estimate_means <- function(df) {  
  df %>%  
    select_if(is.numeric) %>%  
    pivot_longer(everything(),  
                 names_to = "var",  
                 values_to = "val") %>%  
    group_by(var) %>%  
    summarize(mean = mean(val, na.rm = TRUE),  
              se = se(val))  
}  
estimate_means(forcats::gss_cat)
```

```
## # A tibble: 3 × 3  
##   var      mean      se  
##   <chr>    <dbl>    <dbl>  
## 1 age      47.18008  0.1181556  
## 2 tvhours   2.980771  0.02429812  
## 3 year    2006.502   0.03037436
```

Plot

```
estimate_means(forcats::gss_cat) %>%  
  ggplot(aes(mean, var)) +  
    geom_point() +  
    geom_linerange(  
      aes(xmin = mean + (qnorm(0.025) * se),  
          xmax = mean + (qnorm(0.975) * se)),  
      color = "magenta",  
    )
```



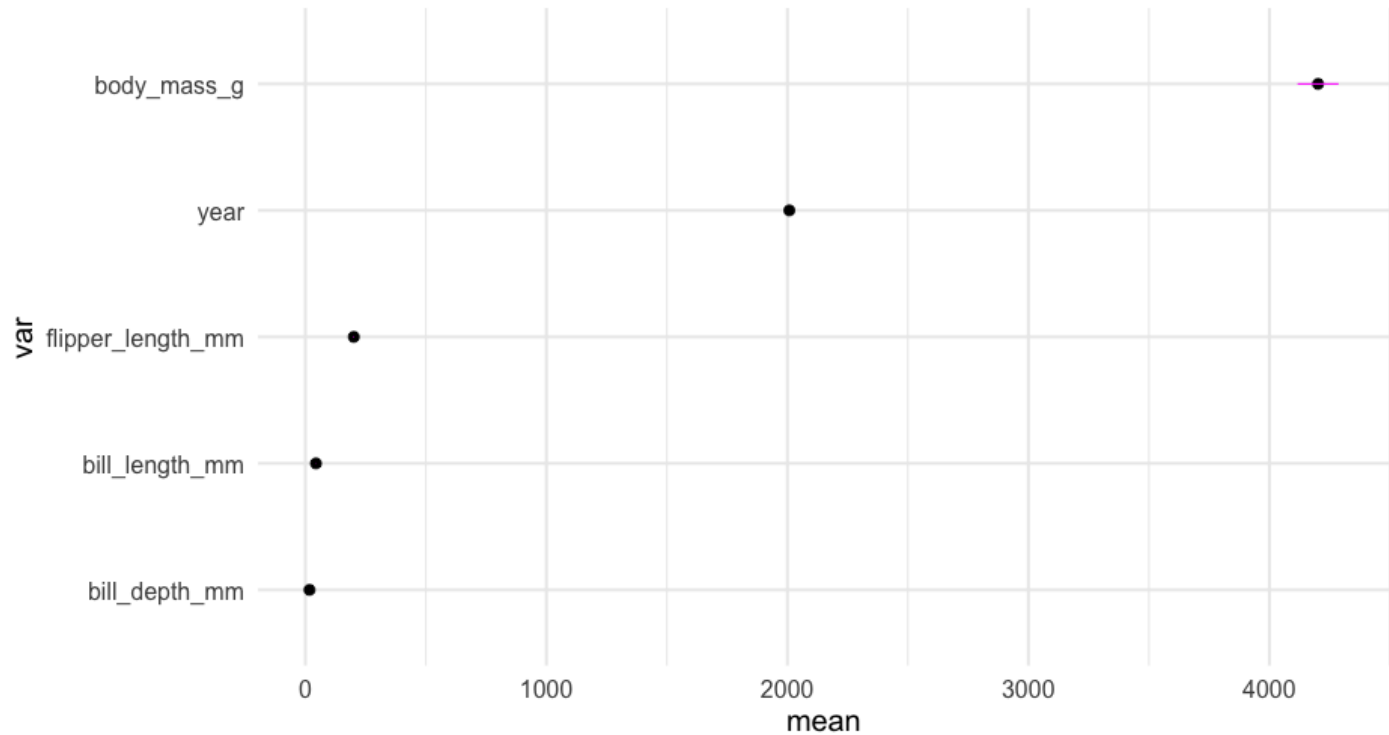
Create plot function

```
plot_means <- function(df) {  
  means <- estimate_means(df) %>%  
    mutate(var = reorder(factor(var), mean))  
  
  ggplot(means, aes(mean, var)) +  
    geom_point() +  
    geom_linerange(  
      aes(xmin = mean + (qnorm(0.025) * se),  
          xmax = mean + (qnorm(0.975) * se)),  
      color = "magenta",  
    )  
}
```

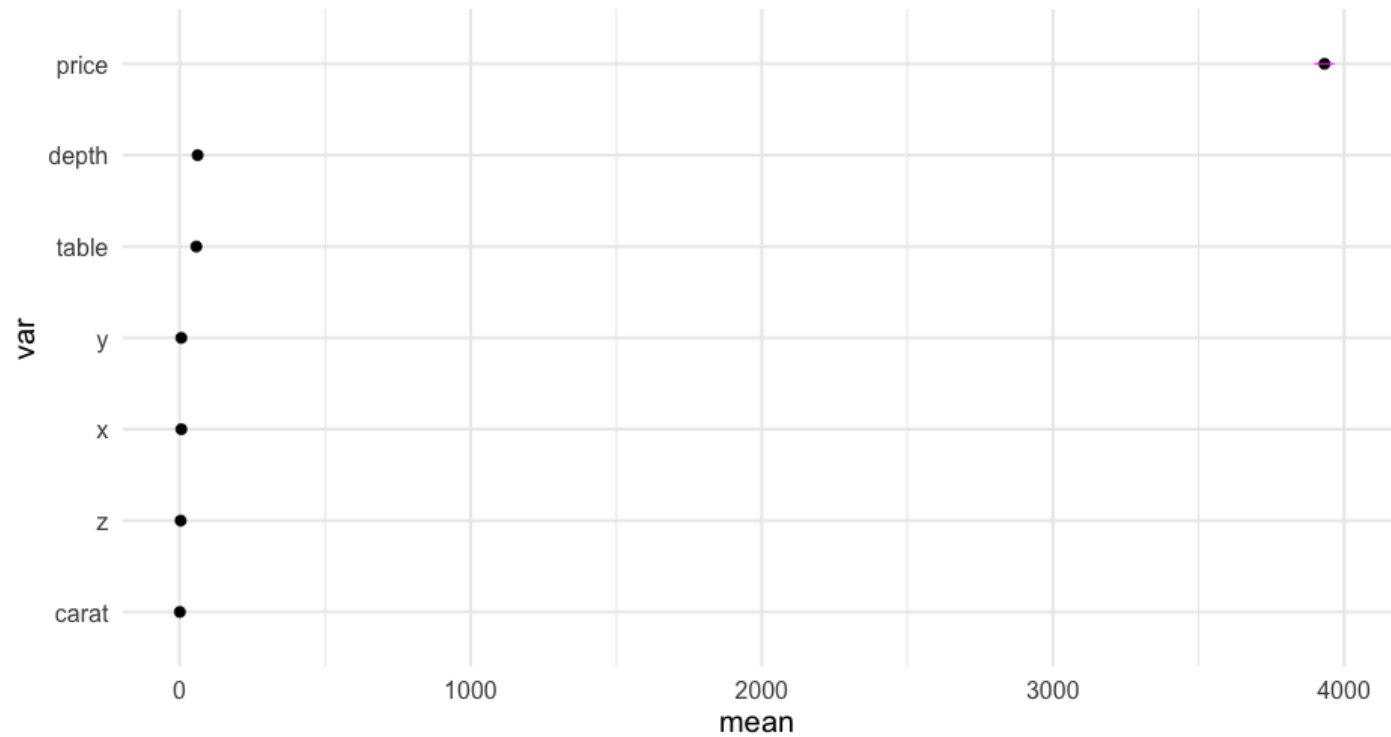
Notice we've successfully avoided `tideval` entirely in both examples!

Examples

```
plot_means(penguins)
```



```
plot_means(diamonds)
```



Overall takeaway

- Non-standard evaluation is confusing but kind of neat
- Programming with the tidyverse can be a bit more difficult, but not always
- Good to think about multiple ways to approach the same problem (and when programming, perhaps think outside the tidyverse at times)

Lab 3

Next time: Intro to shiny