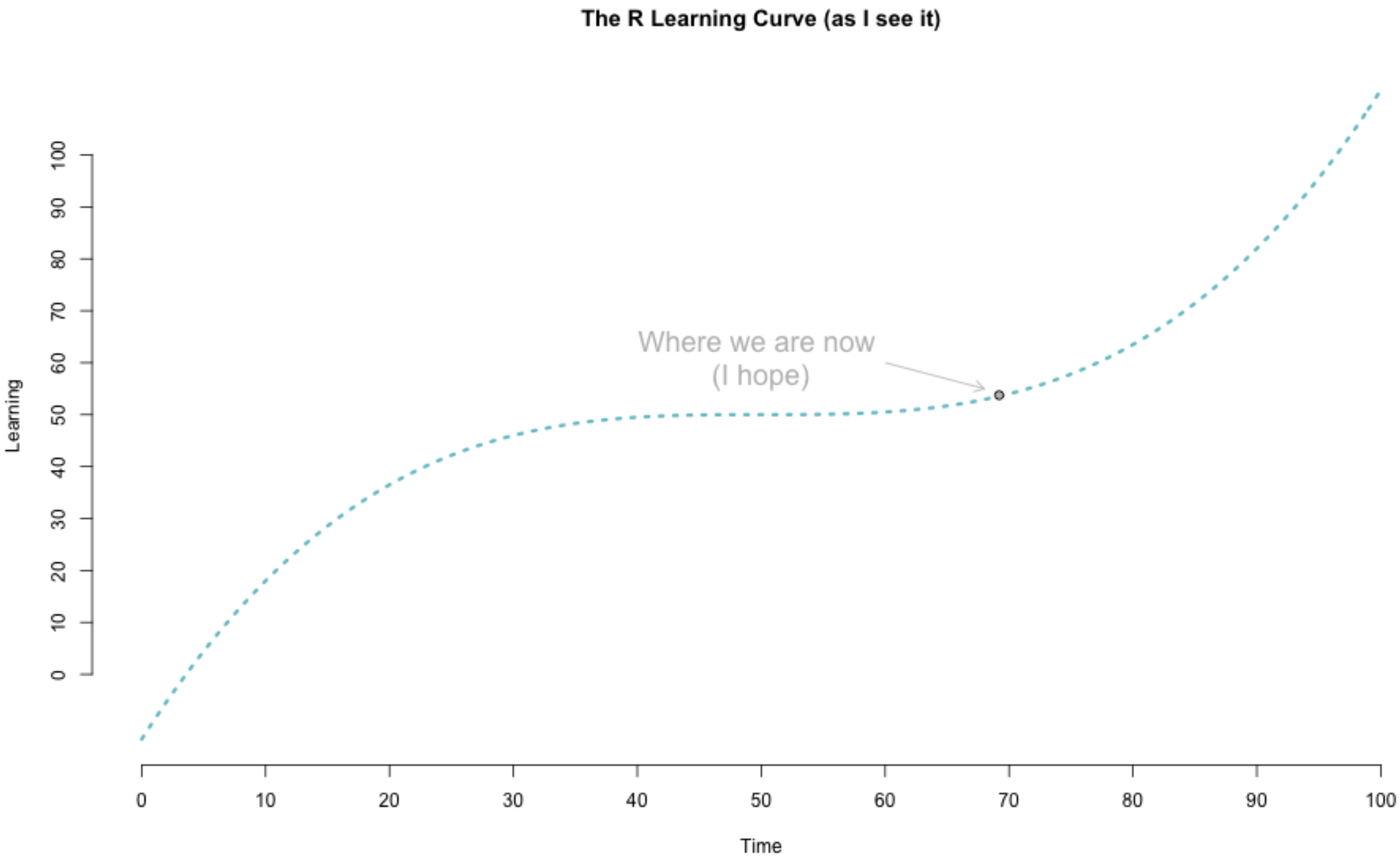
Beyond the course

Daniel Anderson

A reminder: The R learning curve



Agenda

- Introduction to functions
- Introduction to Lists
- Introduction to the *purrr* package for working with lists
- Artificial example
- Walk through a real script (at least some of it)

Before we get started...

- There's SO MUCH we still haven't talked about
    - Haven't even discussed modeling!
    - Version control (git/github)
    - Functional programming (get a bit today)
    - Object oriented programming
- We also briefly discussed a few important topics
    - Text manipulations/factors/dates
    - Data structures besides data frames

Todays's lecture is on the next really big topic I'd recommend you dive into, once you feel comfortable with most of what we've covered in this class: Functions and loops

functions

Writing functions: A basic example

```
pow <- function(x, power) {
    x^power
}
```

This function takes two arguments: `x` and `power`, with `x` being a generic numeric vector (could be a scalar), and power being the power to which the number or vector will be raised (could also be a vector).

While the function appears (and is) simple, it is actually quite flexible.

```
pow(x = 2, power = 3)
```

```
## [1] 8
```

```
pow(c(3, 5, 7), 9)
```

```
## [1]    19683  1953125 40353607
```

```
pow(c(7, 1, -5, 8), c(2, 3))
```

```
## [1]  49   1  25 512
```

```
pow(c(8, 5), c(3, 5))
```

```
## [1]  512 3125
```

```
pow(8, c(5, 3, 2))
```

```
## [1] 32768   512    64
```

Why is this so powerful?

- This is what makes R a programming language, rather than a GUI
- You can program R to do anything
  - Scrape data from the web (*rvest*, *twitteR*, *rtweet*, *tuber*)
  - Complex visualizations (*ggplot*, *lattice*, *visreg*)
  - Any analysis (*lme4*, *randomForest*, *gamm4*)

Calculate Cohen's *d*

- Suppose we wanted to write a function: What would be the arguments we'd need?

$$d = \frac{\bar{X}_{foc} - \bar{X}_{ref}}{\sqrt{\frac{(n_{foc}-1)Var_{foc}+(n_{ref}-1)Var_{ref}}{n_{foc}+n_{ref}-2}}}$$

- Start simple: Let's write a function to compute mean differences
- Think about what format you'd expect the data to come in

- For simplicity, let's assume separate columns for the focal and reference group

```r
mean_diff <- function(foc, ref) {
    mean(foc, na.rm = TRUE) - mean(ref, na.rm = TRUE)
}
```

Test it out

```
test_df <- data.frame(group1 = rnorm(n = 100, mean = 10, sd = 1),
                      group2 = rnorm(n = 100, mean = 8, sd = 1))
head(test_df)
```

```
##      group1   group2
## 1 10.079855 7.775533
## 2  9.848619 7.269473
## 3 10.865970 8.058229
## 4 10.663259 9.267586
## 5 10.181277 7.761954
## 6  9.030272 8.705604
```

```r
mean_diff(test_df$group1, test_df$group2)
```

```
## [1] 1.981619
```

```r
mean_diff(rnorm(100, 10), rnorm(100, 12))
```

```
## [1] -2.129773
```

```r
mean_diff(rnorm(100, 9), rnorm(100, 90))
```

```
## [1] -80.95977
```

Now let's build the denominator

$$\sqrt{\frac{(n_{foc}-1)Var_{foc}+(n_{ref}-1)Var_{ref}}{n_{foc}+n_{ref}-2}}$$

```
pooled_sd <- function(foc, ref) {
    n_foc <- length(na.omit(foc))
    n_ref <- length(na.omit(ref))

    v_foc <- var(foc, na.rm = TRUE)
    v_ref <- var(ref, na.rm = TRUE)

    sqrt( ((n_foc - 1)*v_foc + (n_ref - 1)*v_ref) / ( (n_foc + n_ref) - 2) )

}
```

Put them together

```
coh_d <- function(foc, ref) {
    mean_diff(foc, ref) / pooled_sd(foc, ref)
}
coh_d(test_df$group1, test_df$group2)
```

```
## [1] 2.054554
```

```
coh_d(rnorm(100, 10), rnorm(180, 12, 20))
```

```
## [1] -0.03242877
```

Testing out functions demo

When do you write a function?

- General rule of thumb - if you copy and paste more than three times, might want to write a function
- Functions written for interactive purposes can depend on objects in your global environment

Quick contrived example

Calculate one standard deviation below the mean for each column

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
```

One method - avoid writing a function at all

```
library(tidyverse)
mtcars %>%
    gather(var, val) %>%
    group_by(var) %>%
    summarize(low = min(val) - sd(val))
```

```
## # A tibble: 11 x 2
##      var        low
##    <chr>      <dbl>
##  1    am  -0.4989909
##  2  carb  -0.6152000
##  3   cyl   2.2140784
##  4  disp -52.8386938
##  5  drat   2.2253213
##  6  gear   2.2621959
##  7    hp -16.5628685
##  8   mpg   4.3730519
##  9  qsec  12.7130568
## 10    vs  -0.5040161
## 11    wt   0.5345426
```

Another method - copy and paste

```
summarize(mtcars, low = min(mpg) - sd(mpg))
```

```
##        low
## 1 4.373052
```

```
summarize(mtcars, low = min(cyl) - sd(cyl))
```

```
##        low
## 1 2.214078
```

```
summarize(mtcars, low = min(disp) - sd(disp))
```

```
##         low
## 1 -52.83869
```

```
summarize(mtcars, low = min(hp) - sd(hp))
```

Write a function

```
low <- function(var) {
    min(mtcars[[var]]) - sd(mtcars[[var]])
}
low("mpg")
```

```
## [1] 4.373052
```

Loop the function through

```
for(i in seq_along(names(mtcars))) {
    print( low(names(mtcars)[i]) )
}
```

```
## [1] 4.373052
## [1] 2.214078
## [1] -52.83869
## [1] -16.56287
## [1] 2.225321
## [1] 0.5345426
## [1] 12.71306
## [1] -0.5040161
## [1] -0.4989909
## [1] 2.262196
## [1] -0.6152
```

When writing functions

- You may run into some problems with tidyverse syntax (non-standard evaluation)
- If you're writing functions for others to use, it is generally best to avoid external dependencies anyway, when you can
    - That said, if relying on an external dependency substantially speeds up your code, makes it more clear, etc., it is probably be worth it.
- If you really want to jump into the programming side of things, probably best to learn some base R (I'd recommend Hadley's books: Advanced R and R Packages)

If you want to use tidyverse in programming...

You have to use *tidyeval* concepts. For example you would probably think the following would work, but it fails:

```
library(tidyverse)

grouped_means <- function(group) {
  mtcars %>%
    group_by(group) %>%
    summarize(mpg = mean(mpg))
}
grouped_means(cyl)
```

```
## Error in grouped_df_impl(data, unname(vars), drop): Column `group` is unknown
```

Maybe quote it?

```
grouped_means("cyl")
```

```
## Error in grouped_df_impl(data, unname(vars), drop): Column `group` is unknown
```

- We're still getting errors because of *Non-standard evaluation*.
- I'll briefly discuss this, but I would recommend avoiding it until you feel pretty comfortable with functions.

Problem

- You can't pass a bare variable name to a function without quoting it (not `""`, but capturing it with `quote`).
- BUT, even this won't work because tidyverse functions already do the quoting for us in the background.

Process

- Quote the input to capture the argument
- Tell the tidyverse functions it has already been quoted so it doesn't try to quote it again.

Back to previous example

```r
grouped_means <- function(group) {
  group_var <- enquo(group)

  mtcars %>%
    group_by(!!group_var) %>%
    summarize(mpg = mean(mpg))
}
grouped_means(cyl)
```

```
## # A tibble: 3 x 2
##     cyl      mpg
##   <dbl>    <dbl>
## 1     4 26.66364
## 2     6 19.74286
## 3     8 15.10000
```

```
grouped_means(gear)
```

```
## # A tibble: 3 x 2
##    gear      mpg
##   <dbl>    <dbl>
## 1     3 16.10667
## 2     4 24.53333
## 3     5 21.38000
```

```
grouped_means(vs)
```

```
## # A tibble: 2 x 2
##      vs      mpg
##   <dbl>    <dbl>
## 1     0 16.61667
## 2     1 24.55714
```

Want to learn more? Look here

Quick-ish

The formula for calculating the mean is

$$\bar{x} = \frac{\Sigma x_i}{n}$$

- Write a function that calculates the sample mean, $\bar{x}$, but removes missing data by default
- Only restriction: You can't use the `mean` function in your function (and you also probably want to call it something other than `mean`)
- Compare the results of your function with `mean()`. Do the results match?

- Write the function

```
xbar <- function(x) {
    x <- na.omit(x)
    sum(x) / length(x)
}
```

- Test it out: No missing data

```
xbar(mtcars$mpg); mean(mtcars$mpg)
```

```
## [1] 20.09062
```

```
## [1] 20.09062
```

```
xbar(mtcars$drat); mean(mtcars$drat)
```

```
## [1] 3.596563
```

```
## [1] 3.596563
```

- Test it out: missing data

```r
v1 <- c(1:5, NA, NA, 7:10)
v2 <- c(NA, seq(32:80), NA, 12)
```

```r
xbar(v1); mean(v1, na.rm = TRUE)
```

```
## [1] 5.444444
```

```
## [1] 5.444444
```

```r
xbar(v2); mean(v2, na.rm = TRUE)
```

```
## [1] 24.74
```

```
## [1] 24.74
```

Another alternative

```r
xbar <- function(x) {
    sum(x, na.rm = TRUE) / length(na.omit(x))
}
```

Writing packages

- It's not that hard!
- Once you start writing more generic functions, and start to get fluent with them, you can write a package
  - This is the hard part - much harder than writing packages
- Write your own personal R package to start
- Good resource for getting started: https://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/

You can do it! I promise!

lists

Lists versus other data structures

- To date, we have mostly worked with data frames
    - type of list
- Each column of a data frame is (almost always) an atomic vector of a specific type
    - double
    - integer
    - character
    - logical
- All elements within an atomic vector must be of the same type (implicit coercion).
- Lists are vectors (not atomic) where every element can be of a different type, including other lists.

Contrasting lists and atomic vectors

Lists

```
list("a", 4.35, TRUE, 7L)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 4.35
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 7
```

Vectors

(implicit coercion)

```
c("a", 4.35, TRUE, 7L)
```

```
## [1] "a"    "4.35" "TRUE" "7"
```

```
c(4.35, TRUE, 7L)
```

```
## [1] 4.35 1.00 7.00
```

Lists

Note that the length of list elements can all be different.

```
l <- list(
    c("a", "b", "c"),
    1:5,
    rep(c(T,F), 7),
    rnorm(3, 100, 25)
          )
```

```
l
```

```
## [[1]]
## [1] "a" "b" "c"
##
## [[2]]
## [1] 1 2 3 4 5
##
## [[3]]
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRU
## [12] FALSE  TRUE FALSE
##
## [[4]]
## [1] 69.01684 63.50577 98.50681
```

Lists returned by functions

- Many functions return a list of objects. This is because lists are a great way to store a lot of varied information. For example: `lm`.

```
mod <- lm(hp ~ mpg, data = mtcars)
str(mod)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 324.08 -8.83
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "mpg"
##  $ residuals    : Named num [1:32] -28.7 -28.7 -29.8 -25.1 16 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive
##  $ effects      : Named num [1:32] -829.8 296.3 -23.6 -20 19.3 ...
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "mpg" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 139 139 123 135 159 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
```

You can access the elements through the list

```
mod$coefficients
```

```
## (Intercept)         mpg
##  324.082314   -8.829731
```

```
mod$residuals
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710
##         -28.6579634         -28.6579634         -29.7644476
##      Hornet 4 Drive   Hornet Sportabout             Valiant
##         -25.1260710          16.0336553         -59.2641833
##          Duster 360           Merc 240D            Merc 230
##          47.1828390         -46.6368780         -27.7644476
##            Merc 280           Merc 280C          Merc 450SE
##         -31.5514792         -43.9131026           0.7252741
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood
##           8.6720320          -9.8704031         -27.2531119
## Lincoln Continental   Chrysler Imperial            Fiat 128
##         -17.2531119          35.7147314          28.0009699
##         Honda Civic      Toyota Corolla       Toyota Corona
##          -3.6584921          40.2455664         -37.2430979
##    Dodge Challenger         AMC Javelin          Camaro Z28
```

Other functions will transform data into lists

```
cyls <- split(mtcars, mtcars$cyl)
str(cyls)
```

```
## List of 3
##  $ 4:'data.frame':   11 obs. of  11 variables:
##   ..$ mpg : num [1:11] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26 30.4 ...
##   ..$ cyl : num [1:11] 4 4 4 4 4 4 4 4 4 4 ...
##   ..$ disp: num [1:11] 108 146.7 140.8 78.7 75.7 ...
##   ..$ hp  : num [1:11] 93 62 95 66 52 65 97 66 91 113 ...
##   ..$ drat: num [1:11] 3.85 3.69 3.92 4.08 4.93 4.22 3.7 4.08 4.43 3.77 ...
##   ..$ wt  : num [1:11] 2.32 3.19 3.15 2.2 1.61 ...
##   ..$ qsec: num [1:11] 18.6 20 22.9 19.5 18.5 ...
##   ..$ vs  : num [1:11] 1 1 1 1 1 1 1 1 0 1 ...
##   ..$ am  : num [1:11] 1 0 0 1 1 1 0 1 1 1 ...
##   ..$ gear: num [1:11] 4 4 4 4 4 4 3 4 5 5 ...
##   ..$ carb: num [1:11] 1 2 2 1 2 1 1 1 2 2 ...
##  $ 6:'data.frame':   7 obs. of  11 variables:
##   ..$ mpg : num [1:7] 21 21 21.4 18.1 19.2 17.8 19.7
##   ..$ cyl : num [1:7] 6 6 6 6 6 6 6
##   ..$ disp: num [1:7] 160 160 258 225 168 ...
##   ..$ hp  : num [1:7] 110 110 110 105 123 123 175
```

More on lists

- Note that the previous slide looked like a nested list (list inside a list). This is because data frames are lists, where each element of the list is a vector of the same length.
- lists are tremendously useful and flexible, and can lead to massive jumps in efficiency when combined with loops
  - Often want to loop through a list and apply a function to each element of the list.

Lists and data frames

```
l <- list(
    lets = letters[1:5],
    ints = 9:5,
    dbl = rnorm(5, 12, 0.75)
    )
l
```

```
## $lets
## [1] "a" "b" "c" "d" "e"
##
## $ints
## [1] 9 8 7 6 5
##
## $dbl
## [1] 11.11093 11.67222 11.55736 11.03530 11.02976
```

```
as.data.frame(l)
```

```
##   lets ints      dbl
## 1    a    9 11.11093
## 2    b    8 11.67222
## 3    c    7 11.55736
## 4    d    6 11.03530
## 5    e    5 11.02976
```

Alternative

```
dframe <- data.frame(
    lets = letters[1:5],
    ints = 9:5,
    dbl = rnorm(5, 12, 0.75)
    )
dframe
```

```
##   lets ints      dbl
## 1    a    9 12.27340
## 2    b    8 12.02279
## 3    c    7 11.72385
## 4    d    6 12.88239
## 5    e    5 12.46470
```

```
as.list(dframe)
```

```
## $lets
## [1] a b c d e
## Levels: a b c d e
##
## $ints
## [1] 9 8 7 6 5
##
## $dbl
## [1] 12.27340 12.02279 11.72385 12.88239 12.4
```

Brief introduction to *purrr*

purrr

- As everything with the tidyverse, base equivalents exist
- *purrr* (note three r's) is pipe (`%>%`) friendly
- Has nice parallelization features.
- We'll focus today on `map` and friends, which is the primary function from the package.
- Generally used with lists (for me at least), but can work with any type of vector.

Data

mtcars dataset split by cylinder

```
cyls <- split(mtcars, mtcars$cyl)
str(cyls)
```

```
## List of 3
##  $ 4:'data.frame':   11 obs. of  11 variables:
##   ..$ mpg : num [1:11] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26 30.4 ...
##   ..$ cyl : num [1:11] 4 4 4 4 4 4 4 4 4 4 ...
##   ..$ disp: num [1:11] 108 146.7 140.8 78.7 75.7 ...
##   ..$ hp  : num [1:11] 93 62 95 66 52 65 97 66 91 113 ...
##   ..$ drat: num [1:11] 3.85 3.69 3.92 4.08 4.93 4.22 3.7 4.08 4.43 3.77 ...
##   ..$ wt  : num [1:11] 2.32 3.19 3.15 2.2 1.61 ...
##   ..$ qsec: num [1:11] 18.6 20 22.9 19.5 18.5 ...
##   ..$ vs  : num [1:11] 1 1 1 1 1 1 1 1 0 1 ...
##   ..$ am  : num [1:11] 1 0 0 1 1 1 0 1 1 1 ...
##   ..$ gear: num [1:11] 4 4 4 4 4 4 3 4 5 5 ...
##   ..$ carb: num [1:11] 1 2 2 1 2 1 1 1 2 2 ...
##  $ 6:'data.frame':   7 obs. of  11 variables:
##   ..$ mpg : num [1:7] 21 21 21.4 18.1 19.2 17.8 19.7
##   ..$ cyl : num [1:7] 6 6 6 6 6 6 6
```

map basic usage

```r
library(purrr)
map(cyls, ~lm(hp ~ mpg, data = .))
```

```
## $`4`
##
## Call:
## lm(formula = hp ~ mpg, data = .)
##
## Coefficients:
## (Intercept)          mpg
##      147.43        -2.43
##
##
## $`6`
##
## Call:
## lm(formula = hp ~ mpg, data = .)
##
## Coefficients:
## (Intercept)          mpg
##     164.156       -2.121
```

Basic usage

```
map(LIST, FUN, ...)
```

- LIST = list to loop through
- FUN = Function to loop through the list
- Other arguments passed to the function

Different ways to specify functions

The below are equivalent

```
map(cyls, function(x) lm(hp ~ mpg, data = x))
map(cyls, ~lm(hp ~ mpg, data = .))
```

If we want to extract something from each element of the list, and the that something is named, we can also just supply that name as a string.

```
models <- map(cyls, ~lm(hp ~ mpg, data = .))
map(models, "coefficients")
```

```
## $`4`
## (Intercept)         mpg
##  147.431465   -2.430092
##
## $`6`
## (Intercept)         mpg
##  164.156412   -2.120802
##
## $`8`
```

Different versions of map

If we call a single function, just list it.

```
lst <- list(first = 1:3,
            second = 80:100,
            third = -2:4)

map(lst, mean)
```

```
## $first
## [1] 2
##
## $second
## [1] 90
##
## $third
## [1] 1
```

`map` will alway return a list. Other variants will return other output.

```
map_df(lst, mean)
```

```
## # A tibble: 1 x 3
##   first second third
##   <dbl>  <dbl> <dbl>
## 1     2     90     1
```

```
map_dbl(lst, mean)
```

```
##  first second  third
##      2     90      1
```

```
map_chr(lst, mean)
```

```
##       first      second       third
##  "2.000000" "90.000000"  "1.000000"
```

Remember, a data frame is a list

- Loop through all the columns and apply a specific function

```
map_dbl(mtcars, mean)
```

```
##        mpg        cyl       disp         hp       drat         wt
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##       qsec         vs         am       gear       carb
##  17.848750   0.437500   0.406250   3.687500   2.812500
```

Putting them together

```
library(tidyverse)
map(cyls, ~lm(hp ~ mpg, data = .)) %>%
    map_df(coef) %>%
    mutate(param = c("intercept", "slope")) %>%
    gather(cyl, val, -4) %>%
    spread(param, val)
```

```
## # A tibble: 3 x 3
##     cyl intercept     slope
## * <chr>     <dbl>     <dbl>
## 1     4  147.4315 -2.430092
## 2     6  164.1564 -2.120802
## 3     8  294.4974 -5.647887
```

Alternatively: *broom*

```r
library(broom)
map(cyls, ~lm(hp ~ mpg, data = .)) %>%
    map_df(tidy, .id = "cyl")
```

```
##   cyl        term   estimate  std.error  statistic     p.value
## 1   4 (Intercept) 147.431465  35.606406  4.1405882 0.002519431
## 2   4         mpg  -2.430092   1.318359 -1.8432709 0.098398581
## 3   6 (Intercept) 164.156412 146.508014  1.1204603 0.313430721
## 4   6         mpg  -2.120802   7.403631 -0.2864543 0.786020206
## 5   8 (Intercept) 294.497384  84.337195  3.4919040 0.004447715
## 6   8         mpg  -5.647887   5.512168 -1.0246218 0.325753780
```

More complex versions of `map`

- `map2` iterates over two lists/vectors in parallel
- `pmap` iterates over *p* lists/vectors in parallel

Calculate differences in means

(spacing added just for clarity)

```
set.seed(222)
map2(list(rnorm(100),
          rnorm(100),
          rnorm(100)),

     list(rnorm(100, 0.5),
          rnorm(100, 1),
          rnorm(100, 0.2)),

    ~mean(.x) - mean(.y))
```

```
## [[1]]
## [1] -0.5997455
##
## [[2]]
## [1] -1.083101
##
## [[3]]
## [1] -0.2517111
```

Calculate effect sizes

```
set.seed(222)
map2_dbl(list(rnorm(100),
              rnorm(100),
              rnorm(100)),

         list(rnorm(100, 0.5),
              rnorm(100, 1),
              rnorm(100, 0.2)),

    ~ (mean(.x) - mean(.y)) / sqrt(
        ((length(.x) - 1) * var(.x) +  (length(.y) - 1) * var(.y)) /
                     ((length(.x) + length(.y)) - 2)))
```

```
## [1] -0.6125145 -1.0812928 -0.2446314
```

pmap

- We won't focus on `pmap` today, but it's worth noting that the syntax is slightly different. You supply one (possibly nested) list with all the arguments to the function, and then supply the function.

For example, setup a simulation with different sample sizes, means, and standard deviations. (In this particular example we could)

```
n <- list(50, 100, 250, 500)
mu <- list(10, 15, 10, 15)
stdev <- list(1, 1, 2, 2)

sim_data <- pmap(list(n, mu, stdev), rnorm)
str(sim_data)
```

```
## List of 4
##  $ : num [1:50] 10.06 10.2 10.39 9.67 10.75 ...
##  $ : num [1:100] 15.1 16.5 17.2 14.4 15.2 ...
##  $ : num [1:250] 8.04 7.37 8.17 10.49 11.96 ...
##  $ : num [1:500] 15.4 15.7 15 18.4 17.8 ...
```

Use `map` to check simulation

- *sim_data* is a list, so we can loop through it
- We saw on the previous slide that the sample sizes were correct. What about the means and standard deviations?

```
map_dbl(sim_data, mean)
```

```
## [1] 10.045945 15.062124  9.768937 14.980899
```

```
map_dbl(sim_data, sd)
```

```
## [1] 0.7623378 0.9640901 2.0434633 2.0012167
```

Another `pmap` example

(could do this while avoiding the loop altogether)

```
pmap_chr(list(rep(1:10, 2),
          letters[1:20],
          rep(c("control", "treatment"), each = 10)),
    paste, sep = "-")
```

```
##  [1] "1-a-control"    "2-b-control"    "3-c-control"    "4-d-control"
##  [5] "5-e-control"    "6-f-control"    "7-g-control"    "8-h-control"
##  [9] "9-i-control"    "10-j-control"   "1-k-treatment"  "2-l-treatment"
## [13] "3-m-treatment"  "4-n-treatment"  "5-o-treatment"  "6-p-treatment"
## [17] "7-q-treatment"  "8-r-treatment"  "9-s-treatment"  "10-t-treatment"
```

Nesting data frames

Rather than splitting data frames (as we did before), it can often be helpful to `nest()` them instead. The reason you would want to nest a data frame is for similar reasons to wanting to split it.

For example,

```
nested <- mtcars %>%
    group_by(cyl) %>%
    nest()
nested
```

```
## # A tibble: 3 x 2
##     cyl               data
##   <dbl>             <list>
## 1     6  <tibble [7 x 10]>
## 2     4 <tibble [11 x 10]>
## 3     8 <tibble [14 x 10]>
```

List columns

In the previous example:

- Data are split by cylinder, just as before
- The list of data are then stored into a list column in a data frame
- Each "cell" in the list column contains all the data for that corresponding row in the data frame (cylinder).
- In some ways this is a bit odd, but it can help us stay organized.

```
nested$data
```

```
## [[1]]
## # A tibble: 7 x 10
##     mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21.0 160.0   110  3.90 2.620 16.46     0     1     4     4
## 2  21.0 160.0   110  3.90 2.875 17.02     0     1     4     4
## 3  21.4 258.0   110  3.08 3.215 19.44     1     0     3     1
## 4  18.1 225.0   105  2.76 3.460 20.22     1     0     3     1
## 5  19.2 167.6   123  3.92 3.440 18.30     1     0     4     4
## 6  17.8 167.6   123  3.92 3.440 18.90     1     0     4     4
## 7  19.7 145.0   175  3.62 2.770 15.50     0     1     5     6
##
## [[2]]
## # A tibble: 11 x 10
##      mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  22.8 108.0    93  3.85 2.320 18.61     1     1     4     1
##  2  24.4 146.7    62  3.69 3.190 20.00     1     0     4     2
##  3  22.8 140.8    95  3.92 3.150 22.90     1     0     4     2
##  4  32.4  78.7    66  4.08 2.200 19.47     1     1     4     1
##  5  30.4  75.7    52  4.93 1.615 18.52     1     1     4     2
```

Fit multiple models

```
nested <- nested %>%
    mutate(m1_mpg = map(data, ~lm(hp ~ mpg, data = .)),
           m2_mpg_disp = map(data, ~lm(hp ~ mpg + disp, data = .)))
nested
```

```
## # A tibble: 3 x 4
##     cyl               data   m1_mpg m2_mpg_disp
##   <dbl>             <list>   <list>      <list>
## 1     6  <tibble [7 x 10]> <S3: lm>    <S3: lm>
## 2     4 <tibble [11 x 10]> <S3: lm>    <S3: lm>
## 3     8 <tibble [14 x 10]> <S3: lm>    <S3: lm>
```

See models for *mpg* and *disp*

```
nested$m2_mpg_disp
```

```
## [[1]]
##
## Call:
## lm(formula = hp ~ mpg + disp, data = .)
##
## Coefficients:
## (Intercept)          mpg         disp
##    201.1055      -1.2504      -0.2953
##
##
## [[2]]
##
## Call:
## lm(formula = hp ~ mpg + disp, data = .)
##
## Coefficients:
## (Intercept)          mpg         disp
##   140.68630     -2.29124      0.02894
##
```

Summary of models for *mpg* and *disp*

```
map(nested$m2_mpg_disp, summary)
```

```
## [[1]]
##
## Call:
## lm(formula = hp ~ mpg + disp, data = .)
##
## Residuals:
##       1       2       3       4       5       6       7
## -17.599 -17.599  11.841  -7.030  -4.605  -6.356  41.346
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 201.1055   144.6027   1.391    0.237
## mpg          -1.2504     7.1714  -0.174    0.870
## disp         -0.2953     0.2508  -1.177    0.304
##
## Residual standard error: 25.4 on 4 degrees of freedom
## Multiple R-squared:  0.2694, Adjusted R-squared:  -0.09595
## F-statistic: 0.7374 on 2 and 4 DF,  p-value: 0.5338
##
```

For the $ averse

```
nested %>%
    transmute(smry = map(m2_mpg_disp, summary)) %>%
    flatten()
```

```
## [[1]]
##
## Call:
## lm(formula = hp ~ mpg + disp, data = .)
##
## Residuals:
##       1       2       3       4       5       6       7
## -17.599 -17.599  11.841  -7.030  -4.605  -6.356  41.346
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 201.1055   144.6027   1.391    0.237
## mpg          -1.2504     7.1714  -0.174    0.870
## disp         -0.2953     0.2508  -1.177    0.304
##
## Residual standard error: 25.4 on 4 degrees of freedom
## Multiple R-squared:  0.2694, Adjusted R-squared:  -0.09595
```

Compare Models

```
nested <- nested %>%
    mutate(m12_comp = map2(m1_mpg, m2_mpg_disp, anova),
           p = map(m12_comp, "Pr(>F)"),
           p = map_dbl(p, `[`, 2))
nested
```

```
## # A tibble: 3 x 6
##     cyl               data    m1_mpg m2_mpg_disp         m12_comp          p
##   <dbl>             <list>    <list>      <list>           <list>      <dbl>
## 1     6  <tibble [7 x 10]> <S3: lm>    <S3: lm> <anova [2 x 6]> 0.3043020
## 2     4 <tibble [11 x 10]> <S3: lm>    <S3: lm> <anova [2 x 6]> 0.9434844
## 3     8 <tibble [14 x 10]> <S3: lm>    <S3: lm> <anova [2 x 6]> 0.9080672
```

Extract all coefficients

```
coefs <- nested %>%
    select(3:4) %>%
    map(map_df, tidy) %>%
    map_df(bind_rows, .id = "model")
coefs
```

```
##          model        term    estimate   std.error   statistic
## 1       m1_mpg (Intercept) 164.15641199 146.5080145  1.12046029
## 2       m1_mpg         mpg  -2.12080234   7.4036315 -0.28645434
## 3       m1_mpg (Intercept) 147.43146466  35.6064062  4.14058818
## 4       m1_mpg         mpg  -2.43009244   1.3183588 -1.84327091
## 5       m1_mpg (Intercept) 294.49738431  84.3371946  3.49190397
## 6       m1_mpg         mpg  -5.64788732   5.5121677 -1.02462183
## 7  m2_mpg_disp (Intercept) 201.10545950 144.6026821  1.39074502
## 8  m2_mpg_disp         mpg  -1.25040042   7.1713987 -0.17435935
## 9  m2_mpg_disp        disp  -0.29530305   0.2508059 -1.17741686
## 10 m2_mpg_disp (Intercept) 140.68630194  99.6421350  1.41191577
## 11 m2_mpg_disp         mpg  -2.29123552   2.3574552 -0.97191054
## 12 m2_mpg_disp        disp   0.02894082   0.3956489  0.07314773
## 13 m2_mpg_disp (Intercept) 311.35824930 167.6596917  1.85708471
## 14 m2_mpg_disp         mpg  -6.06152869   6.7348316 -0.90002676
```

Next steps

From here we could go on to plotting, etc., instead, let's look at a full, applied example that uses some of these topics.

Context

- Evaluating intervention response through "checkpoints"
- Only concerning if students do not receive full credit at each checkpoint
- Evaluate patterns of checkpoint response to see if we can identify different types of non-responders

Analysis

- Examine means at pre- and post-test on various measures of mathematics for student groups (according to their patterns of response)
- Examine residual gains by groups