Functional Programming

Data Wrangling in R

Functional Programming

“R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions. In particular, R has what’s known as first class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, **pass them as arguments to other functions**, create them inside functions, and even return them as the result of a function.” - Hadley Wickham

Don’t need to write for-loops! - check this video.

Allows you to flexibly iterate functions to multiple elements of a data object!

Useful when you want to apply a function to:
* lots of columns in a tibble
* multiple tibbles
* multiple data files
* or perform fancy functions with vectors (or tibble columns)

Working `across` multiple columns

Say we wanted to round multiple columns of the `mtcars` data. We could do so one column at a time, or we could use the `across` function from the `dplyr` package. Needs to be used **within other dplyr functions** such as `mutate`.

```
mutate(across(which_columns, which function or operation))
```

```
head(mtcars, 2)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4       21   6  160 110  3.9 2.620 16.46  0  1    4    4
## Mazda RX4 Wag   21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```
mtcars %>%
  mutate(across(.cols = c(disp, drat, wt, qsec), round)) %>%
  head(2)
```

```
##                mpg cyl disp  hp drat wt qsec vs am gear carb
## Mazda RX4       21   6  160 110    4  3   16  0  1    4    4
## Mazda RX4 Wag   21   6  160 110    4  3   17  0  1    4    4
```

functions in R

```r
my_function <- function(x){x + 1}
my_function
```

```
## function(x){x + 1}
```

```r
my_data  <- c(2,3,4)

my_function(x = my_data)
```

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

```r
my_function(my_data)
```

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

Special lamda use

If you see `~ .x` (or sometimes just `.` is used instead of `.x`) this means function(x) {x} - in other words we are passing x to a function.

For example - this is not necessary but you could use it here:

```
mtcars %>%
  mutate(across(.cols = c(disp, drat, wt, qsec), ~ round(.x))) %>%
  head(2)
```

```
##               mpg cyl disp  hp drat wt qsec vs am gear carb
## Mazda RX4      21   6  160 110    4  3   16  0  1    4    4
## Mazda RX4 Wag  21   6  160 110    4  3   17  0  1    4    4
```

```
mtcars %>%
  mutate(across(.cols = c(disp, drat, wt, qsec), round)) %>%
  head(2)
```

```
##               mpg cyl disp  hp drat wt qsec vs am gear carb
## Mazda RX4      21   6  160 110    4  3   16  0  1    4    4
## Mazda RX4 Wag  21   6  160 110    4  3   17  0  1    4    4
```

Using `across` with arguments

If you wish to also pass arguments to the function that you are applying to the various columns, then you need to use the `~` and `.x` (or `.`) as a place holder for what you the values you will be passing into the function.

```
round_x <- mtcars %>%
  mutate(across(.cols = c(disp, drat, wt, qsec), ~ round(.x, digits = 1)))

head(round_x, 2)
```

```
##               mpg cyl disp  hp drat  wt qsec vs am gear carb
## Mazda RX4      21   6  160 110  3.9 2.6 16.5  0  1    4    4
## Mazda RX4 Wag  21   6  160 110  3.9 2.9 17.0  0  1    4    4
```

```
round_per <- mtcars %>%
  mutate(across(.cols = c(disp, drat, wt, qsec), ~ round(., digits = 1)))

head(round_per, 2)
```

```
##               mpg cyl disp  hp drat  wt qsec vs am gear carb
## Mazda RX4      21   6  160 110  3.9 2.6 16.5  0  1    4    4
## Mazda RX4 Wag  21   6  160 110  3.9 2.9 17.0  0  1    4    4
```

```
identical(round_x, round_per)
```

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

Using across with helpers to apply function to multiple columns

https://tidyselect.r-lib.org/reference/select_helpers.html

```
mtcars %>%
  mutate(across(.cols = disp:qsec, round)) %>%
  head(2)
```

```
##                mpg cyl disp  hp drat wt qsec vs am gear carb
## Mazda RX4       21   6  160 110    4  3   16  0  1    4    4
## Mazda RX4 Wag   21   6  160 110    4  3   17  0  1    4    4
```

```
mtcars %>%
  mutate(across(.cols = everything(), round))%>%
  head(2)
```

```
##                mpg cyl disp  hp drat wt qsec vs am gear carb
## Mazda RX4       21   6  160 110    4  3   16  0  1    4    4
## Mazda RX4 Wag   21   6  160 110    4  3   17  0  1    4    4
```

`if_any()` and `if_all()` are also helpful!

if_any() filters for rows any columns in listed columns meet the condition if_all() filters for rows if all columns in listed columns meet the condition

```
diamonds %>%
   filter(if_any(c(x, y, z), ~. > 11))
```

```
## # A tibble: 3 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  2    Premium   H     SI2      58.9  57   12210  8.09 58.9   8.06
## 2  0.51 Very Good E     VS1      61.8  54.7  1970  5.12  5.15 31.8
## 3  0.51 Ideal     E     VS1      61.8  55    2075  5.15 31.8   5.12
```

```
diamonds %>%
   filter(if_all(c(x, y, z), ~. > 8))
```

```
## # A tibble: 1 x 10
##   carat cut     color clarity depth table price     x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1     2 Premium H     SI2      58.9    57 12210  8.09  58.9  8.06
```

Previously we filtered by many conditionals…2 general ways

```
library(stringr)
diamonds %>%
    filter(str_detect(cut, "Ideal|Premium")) %>% head(2)
```

```
## # A tibble: 2 x 10
##   carat cut     color clarity depth table price     x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium E     SI1      59.8    61   326  3.89  3.84  2.31
```

```
diamonds %>%
    filter(cut %in% c("Ideal", "Premium"), z > 4 , color == "E")
```

```
## # A tibble: 837 x 10
##    carat cut     color clarity depth table price     x     y     z
##    <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  1.22 Premium E     I1       60.9    57  2862  6.93  6.88  4.21
##  2  1.25 Ideal   E     I1       60.9    56  3276  6.95  6.91  4.22
##  3  1.05 Premium E     I1       62.2    61  3293  6.51  6.48  4.04
##  4  1.24 Premium E     I1       61.1    62  3528  6.91  6.86  4.21
##  5  1.19 Premium E     I1       60.2    61  3572  6.91  6.87  4.15
##  6  1.13 Ideal   E     I1       62      55  3729  6.66  6.7   4.14
##  7  1.13 Ideal   E     I1       62      55  3797  6.7   6.66  4.14
##  8  1.12 Ideal   E     SI2      60.9    57  3864  6.66  6.6   4.04
##  9  1.2  Premium E     I1       62.3    60  3871  6.78  6.71  4.2
## 10  1.1  Ideal   E     I1       61.9    56  3872  6.59  6.63  4.09
## # … with 827 more rows
```

Now we can filter multiple columns for multiple conditions simultaneously

```
diamonds %>%
   filter(if_all(c(x, y, z), ~.x > 4 & .x < 5.5))
```

```
## # A tibble: 3 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1   0.5 Very Good G     VVS1     63.7    58  2180  5.01  5.04  5.06
## 2   0.5 Fair      E     VS2      79      73  2579  5.21  5.18  4.09
## 3   0.5 Fair      E     VS2      79      73  2579  5.21  5.18  4.09
```

```
diamonds %>%
   filter(if_all(c(x, y, z), ~. == 0 | . > 8))
```

```
## # A tibble: 12 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  1    Very Good H     VS2      63.3    53  5139  0     0     0
##  2  1.14 Fair      G     VS1      57.5    67  6381  0     0     0
##  3  2    Premium   H     SI2      58.9    57 12210  8.09 58.9   8.06
##  4  2.18 Premium   H     SI2      59.4    61 12631  8.49  8.45  0
##  5  1.56 Ideal     G     VS2      62.2    54 12800  0     0     0
##  6  2.25 Premium   I     SI1      61.3    58 15397  8.52  8.42  0
##  7  1.2  Premium   D     VVS1     62.1    59 15686  0     0     0
##  8  2.2  Premium   H     SI1      61.2    59 17265  8.42  8.37  0
##  9  2.25 Premium   H     SI2      62.8    59 18034  0     0     0
```

`purrr` is also a super helpful package!

“Designed to make your functions purrr.”

`dplyr` is designed for data frames `purr` is designed for vectors

The `purrr` package can be very helpful!

- https://purrr.tidyverse.org/
- https://github.com/rstudio/cheatsheets/raw/master/purrr.pdf
- https://jennybc.github.io/purrr-tutorial/

`purrr` main functions

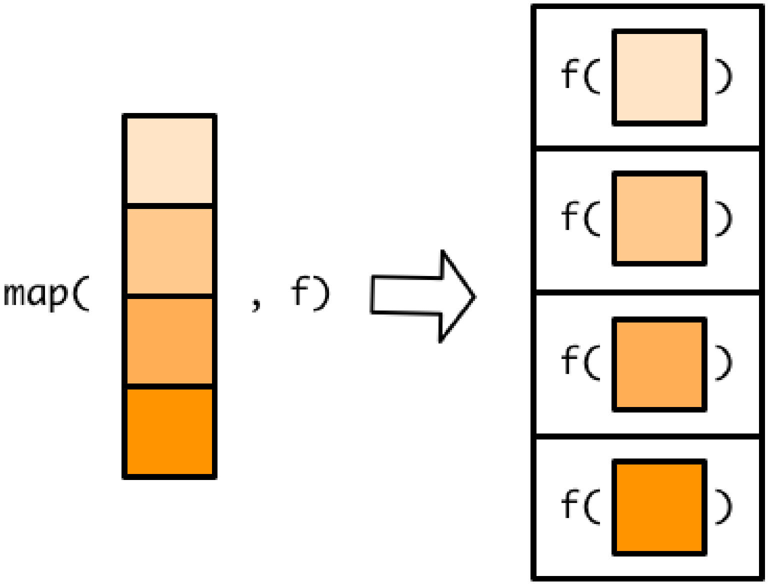
`map` and `map_*` and `modify`
- applies function to each element of vector or list

`map2` and `map2*`
- applies function to each element of two vectors or lists

`pmap` and `pmap_*` - applies function to each element of 3+ vectors or lists (requires a list for input)

the _* options specify the type of data output

map (and modify)



```
x <-c(1.2,2.3,3.5,4.6)
map(x, round) %>% unlist()
```

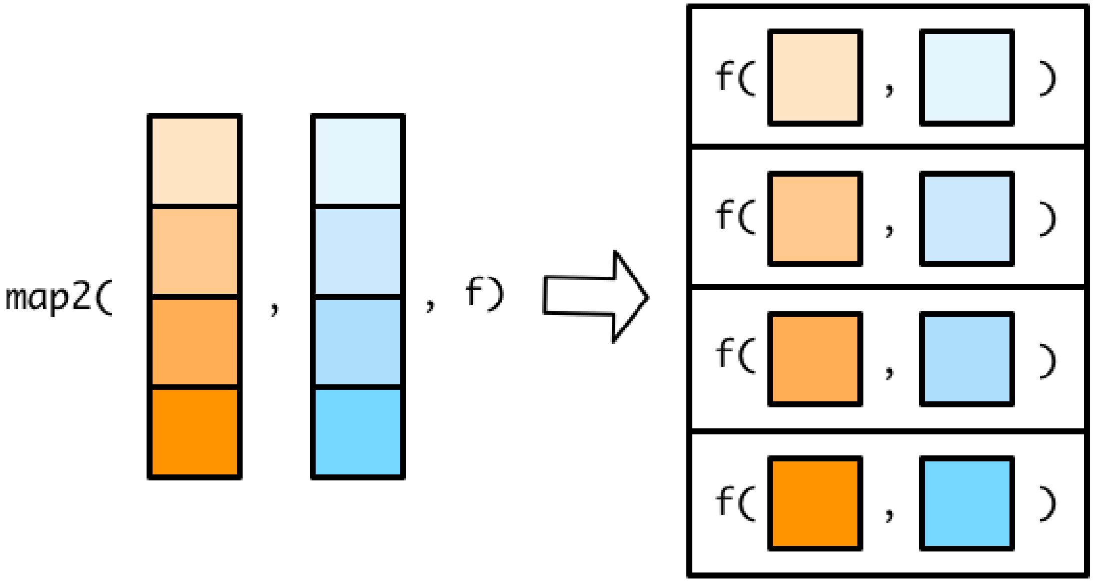
```
## [1] 1 2 4 5
```

```
x <-tibble(values = c(1.2,2.3,3.5,4.6))
map_df(x,  round)
```

```
## # A tibble: 4 x 1
##   values
##    <dbl>
## 1      1
## 2      2
## 3      4
## 4      5
```

map2

Good if you need to use multiple vectors in a function together.



```
x <-c(1.2,2.3,3.5,4.6)
y <-c(2.4,5.3, 6.4, 1.0)
map2(x, y, min) %>% unlist()
```

```
## [1] 1.2 2.3 3.5 1.0
```

map2 in practice

```
trees$new <- trees$Volume-3
head(trees)
```

```
##   Girth Height Volume  new
## 1   8.3     70   10.3  7.3
## 2   8.6     65   10.3  7.3
## 3   8.8     63   10.2  7.2
## 4  10.5     72   16.4 13.4
## 5  10.7     81   18.8 15.8
## 6  10.8     83   19.7 16.7
```

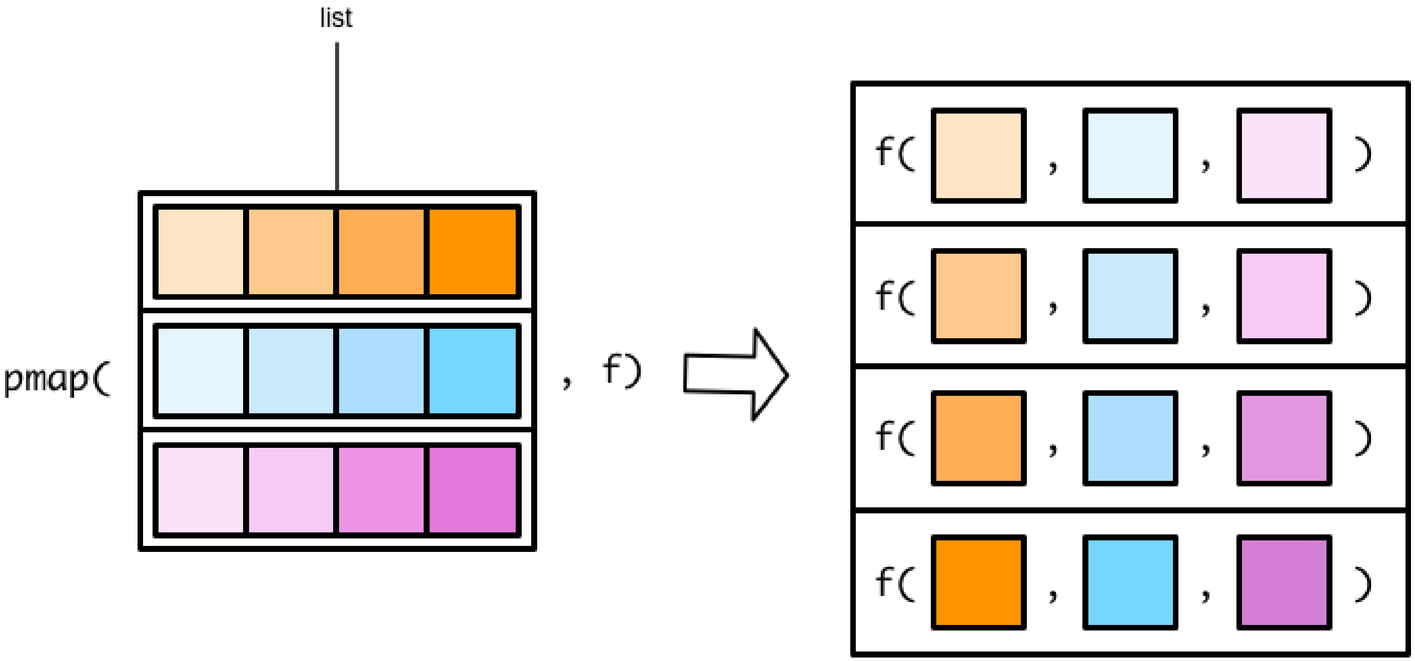
```
map2_dbl(trees$Girth, trees$new, min) %>% head()
```

```
## [1]  7.3  7.3  7.2 10.5 10.7 10.8
```

```
map2_dbl(trees$Girth, trees$Height, function(x,y){ pi * ((x/2)/12)^2 * y})
```

```
##  [1]  26.30157  26.22030  26.60929  43.29507  50.58013  52.80232  43.55687
##  [8]  49.49645  53.76050  51.31268  55.01883  53.87046  53.87046  51.51672
## [15]  58.90486  67.16431  77.14819  82.97153  72.68200  66.47610  83.38311
## [22]  87.98205  84.85845 100.53096 111.58179 132.22227 136.96744 139.80524
## [29] 141.37167 141.37167 201.36365
```

pmap



```
pmap_list <-
  list(x = c(1.2,2.3,3.5,4.6), y = c(2.4,5.3, 6.4, 1.0), z = c(2,9, 4, 11.0))

pmap(pmap_list, min) %>% unlist()
```

```
## [1] 1.2 2.3 3.5 1.0
```

`purrr` - apply function to all columns

two options `map_df` or `modify`

Lots of variations of `map` based on output

```
library(purrr)
head(mtcars, 2)
```

```
##               mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

```
 mtcars %>%
   map_df(round) %>% # will be a tibble now - will remove rownames
   head(2)
```

```
## # A tibble: 2 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    21     6   160   110     4     3    16     0     1     4     4
## 2    21     6   160   110     4     3    17     0     1     4     4
```

```
 mtcars %>%
   modify(round) %>% # modify keeps original data type
   head(2)
```

It’s a bit easier to pass arguments than across…

```
mtcars %>%
  map_df(round, digits = 1) %>%
  head()
```

```
## # A tibble: 6 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21       6   160   110   3.9   2.6  16.5     0     1     4     4
## 2  21       6   160   110   3.9   2.9  17       0     1     4     4
## 3  22.8     4   108    93   3.9   2.3  18.6     1     1     4     1
## 4  21.4     6   258   110   3.1   3.2  19.4     1     0     3     1
## 5  18.7     8   360   175   3.1   3.4  17       0     0     3     2
## 6  18.1     6   225   105   2.8   3.5  20.2     1     0     3     1
```

`purrr` apply function to some columns like accross

Using `modify_if()` or `map_if()`, we can specify what columns to modify

```
head(as_tibble(iris), 3)
```

```
## # A tibble: 3 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>  
## 1          5.1         3.5          1.4         0.2 setosa 
## 2          4.9         3            1.4         0.2 setosa 
## 3          4.7         3.2          1.3         0.2 setosa 
```

```
as_tibble(iris) %>%
  modify_if(is.numeric, as.character) %>%
  head(3)
```

```
## # A tibble: 3 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <chr>        <chr>       <chr>        <chr>       <fct>  
## 1 5.1          3.5         1.4          0.2         setosa 
## 2 4.9          3           1.4          0.2         setosa 
## 3 4.7          3.2         1.3          0.2         setosa 
```

```
as_tibble(iris) %>%
  map_if(is.numeric, as.character) %>%
  class()
```

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

great example with split()

```
mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

More on lists soon!!

–>

What is a 'list'?

- Lists are the most flexible/“generic” data class in R
- Can be created using list()
- Can hold vectors, strings, matrices, models, list of other list, lists upon lists!
- Can reference data using $ (if the elements are named), or using [], or [[]]

```
> mylist <- list(letters=c("A", "b", "c"),
+         numbers=1:3, matrix(1:25, ncol=5), matrix(1:25, ncol=5))
```

List Structure

```
> head(mylist)
```

```
$letters
[1] "A" "b" "c"

$numbers
[1] 1 2 3

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

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

List referencing

```
> mylist[1] # returns a list
```

```
$letters
[1] "A" "b" "c"
```

```
> mylist["letters"] # returns a list
```

```
$letters
[1] "A" "b" "c"
```

List referencing

```
> mylist[[1]] # returns the vector 'letters'
```

```
[1] "A" "b" "c"
```

```
> mylist$letters # returns vector
```

```
[1] "A" "b" "c"
```

```
> mylist[["letters"]] # returns the vector 'letters'
```

```
[1] "A" "b" "c"
```

List referencing

You can also select multiple lists with the single brackets.

```
> mylist[1:2] # returns a list
```

```
$letters
[1] "A" "b" "c"

$numbers
[1] 1 2 3
```

List referencing

You can also select down several levels of a list at once

```
> mylist$letters[1]
```

```
[1] "A"
```

```
> mylist[[2]][1]
```

```
[1] 1
```

```
> mylist[[3]][1:2,1:2]
```

```
     [,1] [,2]
[1,]    1    6
[2,]    2    7
```

How would I encounter lists?

This comes up a lot in data cleaning and also when reading in multiple files!

```
library(here)
library(readr)
list.files(here::here("data", "iris"), pattern = "*.csv")
```

```
## [1] "iris_q1.csv" "iris_q4.csv" "iris_q5.csv"
```

```
file_list <- paste0(here::here(), "/data/iris/", list.files(here::here("data", "iris"), pattern = "*.csv"))

file_list
```

```
## [1] "/Users/carriewright/Documents/GitHub/Teaching/Data-Wrangling/data/iris/iris_q1.csv"
## [2] "/Users/carriewright/Documents/GitHub/Teaching/Data-Wrangling/data/iris/iris_q4.csv"
## [3] "/Users/carriewright/Documents/GitHub/Teaching/Data-Wrangling/data/iris/iris_q5.csv"
```

```
multifile_data <- file_list %>%
  map(read_csv)

class(multifile_data)
```

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

Reading in multiple files

```
multifile_data[[1]]
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # … with 140 more rows
```

```
multifile_data[[2]]
```

```
## # A tibble: 150 x 1
##    `Sepal.Length:Sepal.Width:Petal.Length:Petal.Width:Species`
##    <chr>
##  1 5.1:3.5:1.4:0.2:setosa
##  2 4.9:3:1.4:0.2:setosa
##  3 4.7:3.2:1.3:0.2:setosa
##  4 4.6:3.1:1.5:0.2:setosa
##  5 5:3.6:1.4:0.2:setosa
##  6 5.4:3.9:1.7:0.4:setosa
##  7 4.6:3.4:1.4:0.3:setosa
##  8 5:3.4:1.5:0.2:setosa
##  9 4.4:2.9:1.4:0.2:setosa
## 10 4.9:3.1:1.5:0.1:setosa
## # … with 140 more rows
```

Reading in multiple files

```
multifile_data[[3]]
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1       -999           3.5          1.4         0.2 setosa
##  2       -999           3            1.4         0.2 setosa
##  3       -999           3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # … with 140 more rows
```

Fixing the second file

```
multifile_data[[2]] <-
  separate(multifile_data[[2]],
       col =`Sepal.Length:Sepal.Width:Petal.Length:Petal.Width:Species`,
              into = c("Sepal.Length", "Sepal.Width",
                      'Petal.Length', "Petal.Width", "Species"),
       sep = ":")

head(multifile_data[[2]], 3)
```

```
## # A tibble: 3 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <chr>        <chr>       <chr>        <chr>       <chr>
## 1 5.1          3.5         1.4          0.2         setosa
## 2 4.9          3           1.4          0.2         setosa
## 3 4.7          3.2         1.3          0.2         setosa
```

```
multifile_data[[2]] <-
  multifile_data[[2]] %>%
  mutate(across(!Species, as.numeric))
```

Reading in multiple files

The `bind_rows()` function can be great for simply combining data.

```
# bind_rows(multifile_data[[1]], multifile_data[[3]], multifile_data[[2]])
bindrows_data <- multifile_data %>%
    map_df(bind_rows, .id = "experiment") # recall that modify keeps the same data type
# so that will not do what we want here because we want a data frame instead of a list!
dim(bindrows_data)
```

```
## [1] 450   6
```

```
tail(bindrows_data, 2)
```

```
## # A tibble: 2 x 6
##   experiment Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <chr>             <dbl>       <dbl>        <dbl>       <dbl> <chr>
## 1 3                   6.2         3.4          5.4         2.3 virginica
## 2 3                   5.9         3            5.1         1.8 virginica
```

See https://www.opencasestudies.org/ocs-bp-vaping-case-study for more information!

Factors

First we will create some data about absences for different students. Each row is a different student. We have information about the number of days absent and the grade for the individual students. We will use the `tibble()` function to create the data. We will use the `sample()` function to create a random sequence of numbers from 0 to 7 with replacements for 32 hypothetical students. Since there are four grades and 8*4 is 32, we will repeat the grade values 8 times. We use the `set.seed()` function so that the random sample from 0 to 7 is the same each time the code is run.

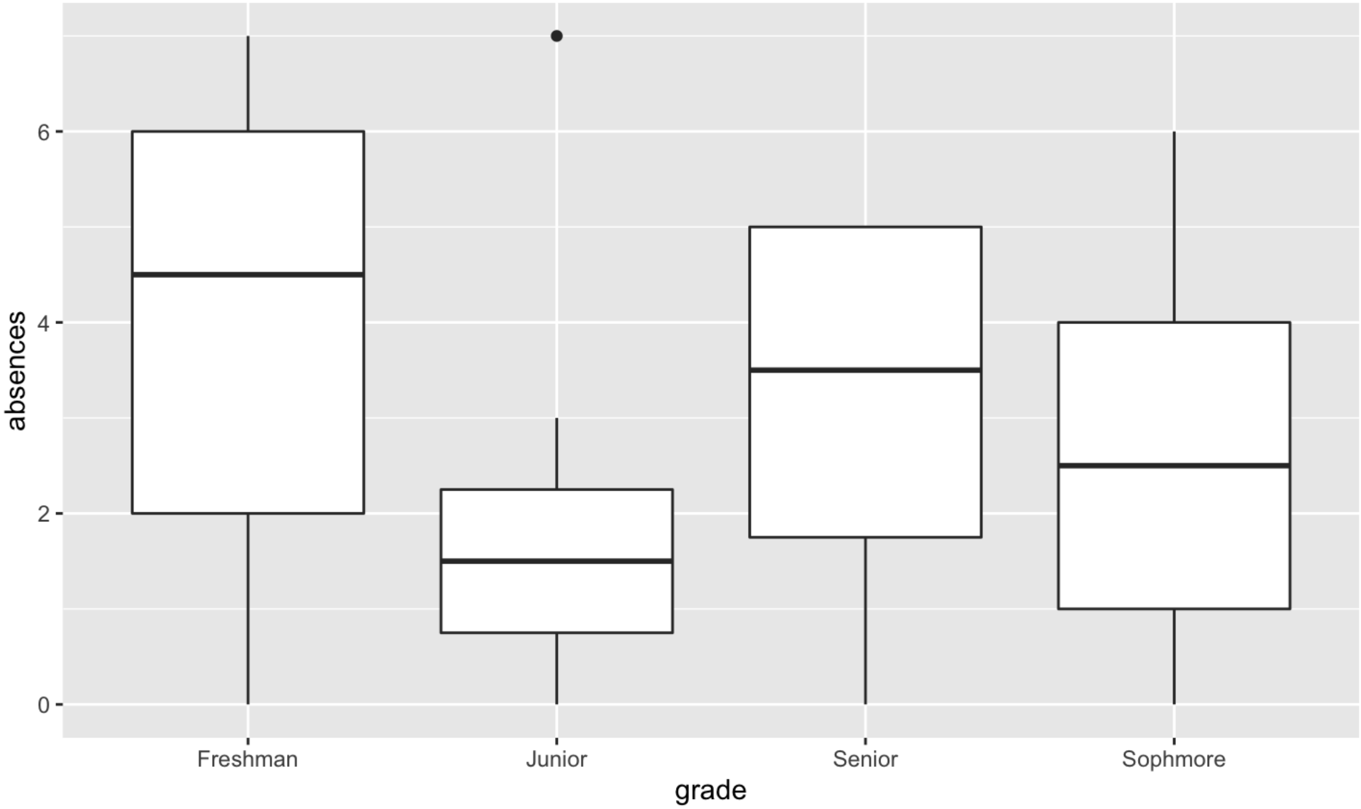
```
set.seed(123)
data_highschool <- tibble(absences = sample(0:7, size = 32, replace = TRUE),
                          grade = rep(c("Freshman","Sophmore",
                                        "Junior", "Senior"), 8))
head(data_highschool, 3)
```

```
## # A tibble: 3 x 2
##   absences grade
##      <int> <chr>
## 1        6 Freshman
## 2        6 Sophmore
## 3        2 Junior
```

Notice that `grade` is a `chr` variable. This indicates that the values are character strings. R does not realize that there is any order related to the `grade` values. However, we know that the order is: freshman, sophomore, junior, senior.

Let’s make a plot first:

```
#boxplot(data = data_highschool, absences ~ grade)
data_highschool %>%
  ggplot(mapping = aes(x = grade, y = absences)) +
  geom_boxplot()
```



Not quite what we want

OK this is very useful, but it is a bit difficult to read, because we expect the values to be plotted by the order that we know, not by alphabetical order. Currently `grade` is class `character` but let’s change that to class `factor` which allows us to specify the levels or order of the values.

As factor now

Using `as_factor()` from the `forcats` package the levels will be in the order in which they occur in the data!

https://forcats.tidyverse.org/

```
class(data_highschool$grade)
```

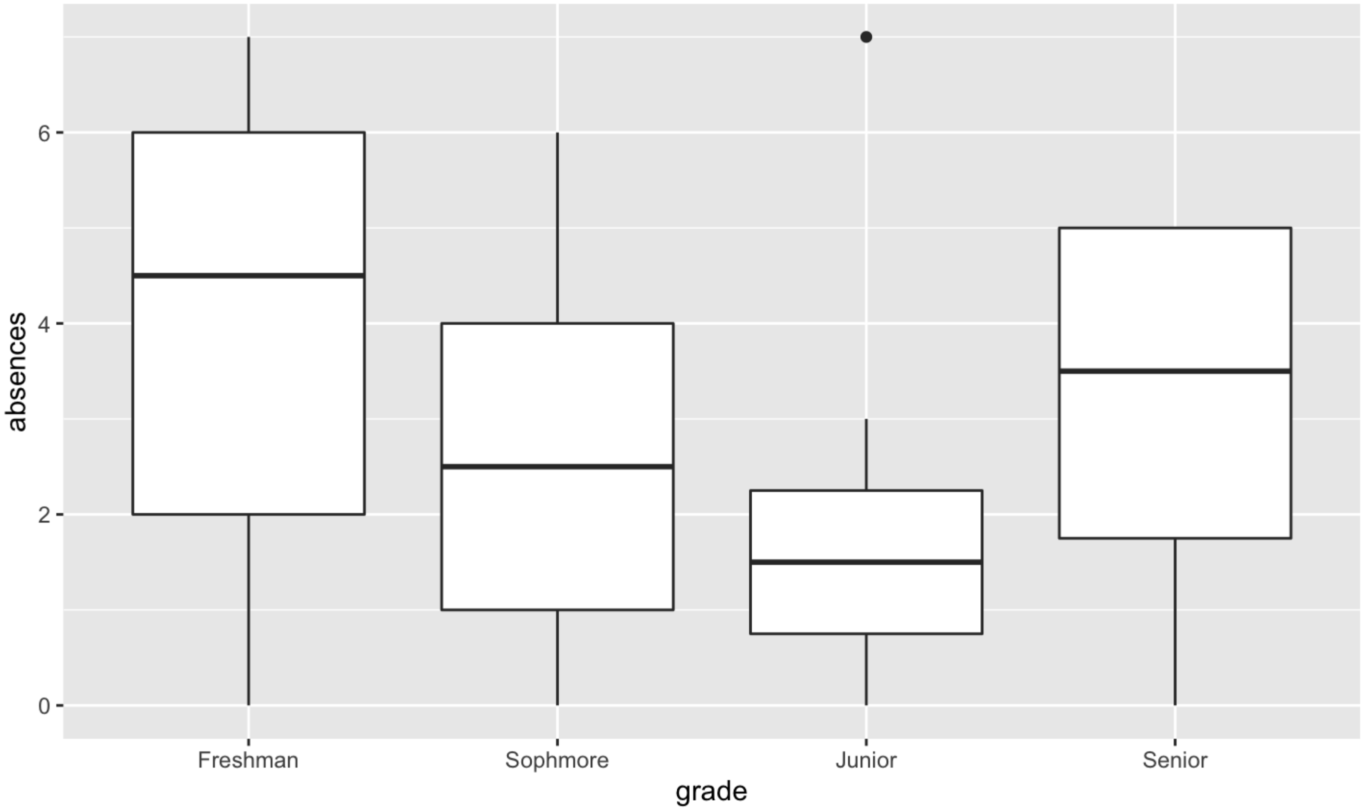
```
## [1] "character"
```

```
data_highschool_fct <- data_highschool %>%
  mutate(grade = as_factor(grade))
head(data_highschool_fct, 3)
```

```
## # A tibble: 3 x 2
##   absences grade
##      <int> <fct>
## 1        6 Freshman
## 2        6 Sophmore
## 3        2 Junior
```

Now let's make our plot again:

```
#boxplot(data = data_highschool_fct, absences ~ grade)
data_highschool_fct %>%
  ggplot(mapping = aes(x = grade, y = absences)) +
  geom_boxplot()
```



Calculatons with factors?

Now what about results from some calculations.

```
data_highschool %>% group_by(grade) %>% summarise(mean = mean(absences))
```

```
## # A tibble: 4 x 2
##   grade     mean
##   <chr>    <dbl>
## 1 Freshman  4
## 2 Junior    2
## 3 Senior    3.12
## 4 Sophmore  2.62
```

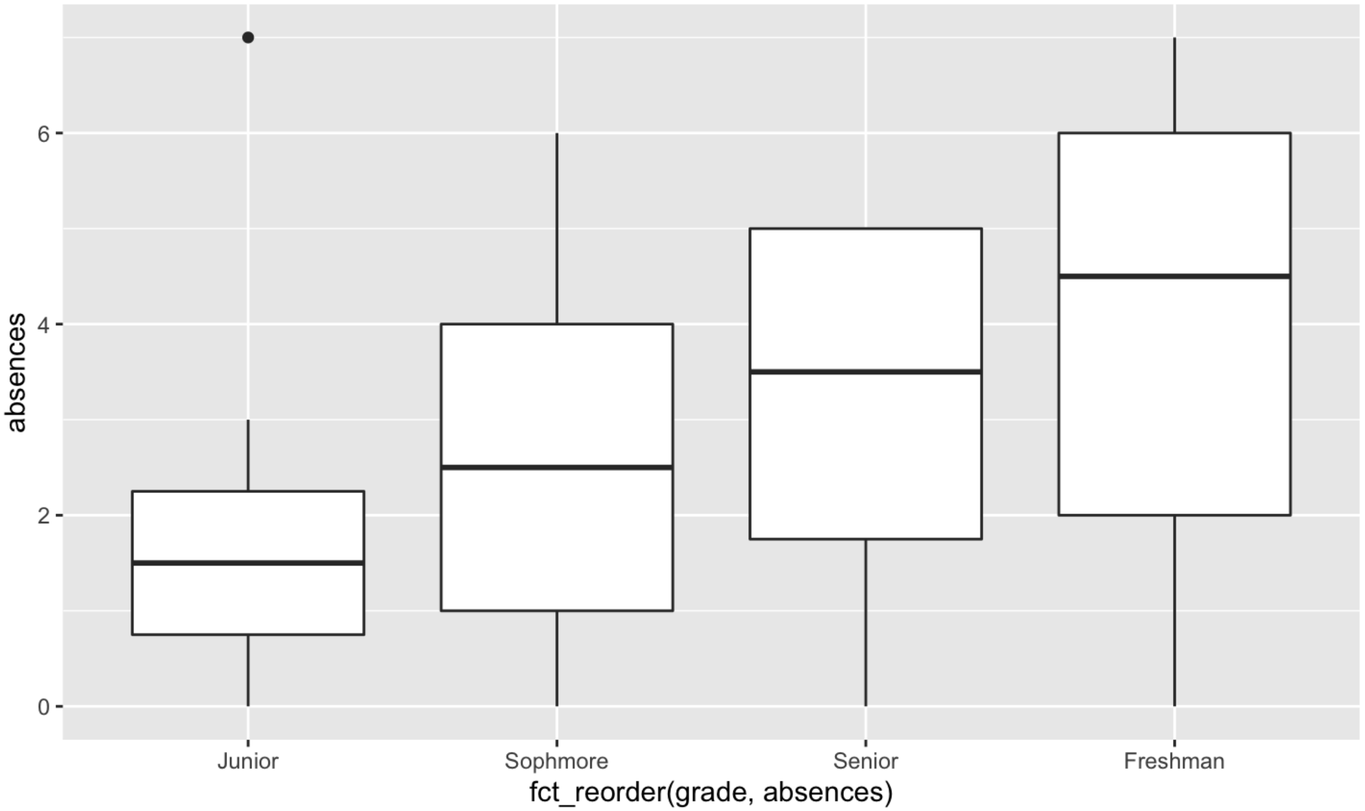
```
data_highschool_fct %>% group_by(grade) %>% summarise(mean = mean(absences))
```

```
## # A tibble: 4 x 2
##   grade     mean
##   <fct>    <dbl>
## 1 Freshman  4
## 2 Sophmore  2.62
## 3 Junior    2
## 4 Senior    3.12
```

Here we see that the mean is calculated in the order we would like only for the version of the data that has absences coded as a factor!

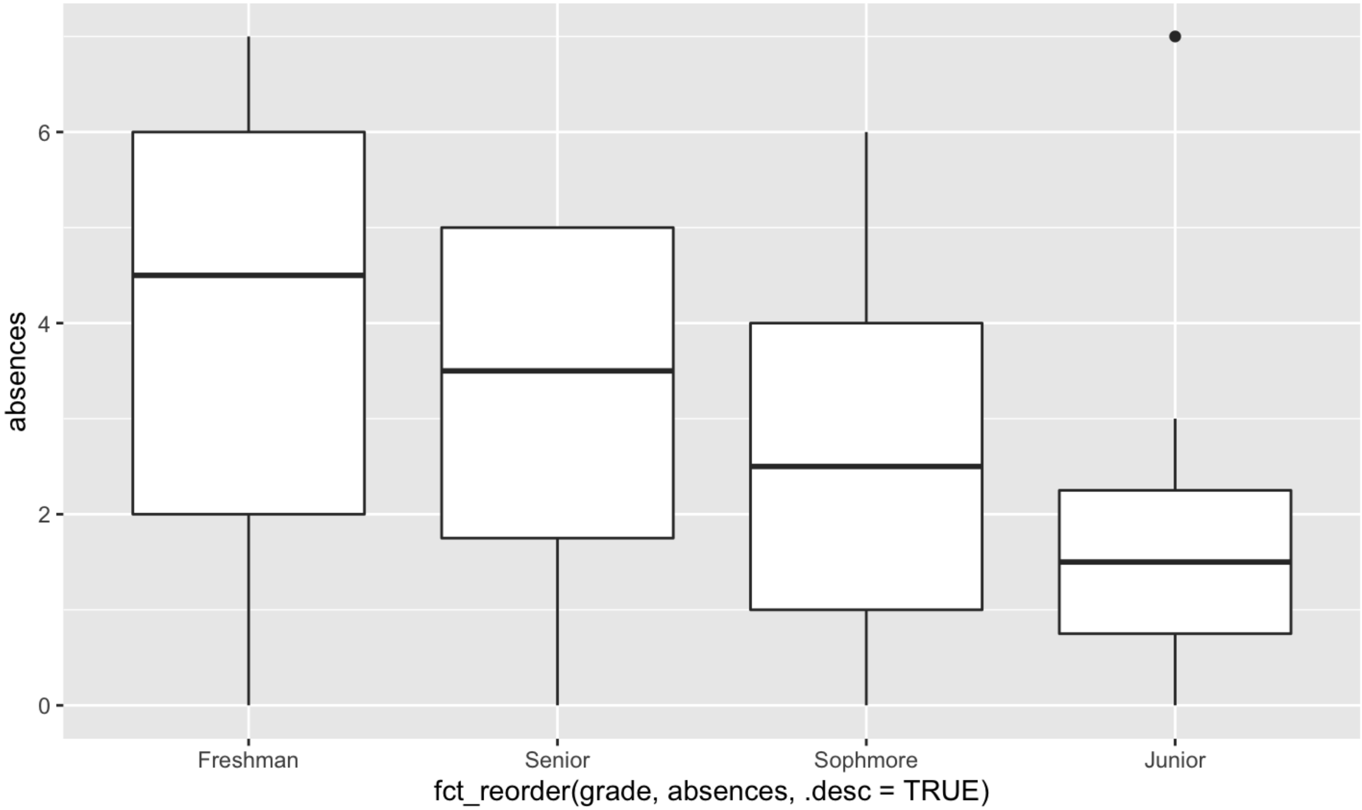
What if we want to change the factor level order?

```
#boxplot(data = data_highschool_fct, absences ~ fct_reorder(grade, absences))
data_highschool_fct %>%
  ggplot(mapping = aes(x = fct_reorder(grade, absences),
                       y = absences)) +
  geom_boxplot()
```



Descending factor order

```
#boxplot(data = data_highschool_fct, absences ~ fct_reorder(grade, absences, .desc = TRUE))
data_highschool_fct %>%
  ggplot(mapping = aes(x = fct_reorder(grade, absences, .desc = TRUE),
                       y = absences)) +
  geom_boxplot()
```



Claculations with reoder

```
data_highschool_fct %>% group_by(grade) %>% summarise(mean = mean(absences))
```

```
## # A tibble: 4 x 2
##   grade     mean
##   <fct>    <dbl>
## 1 Freshman  4
## 2 Sophmore  2.62
## 3 Junior    2
## 4 Senior    3.12
```

```
data_highschool_fct$grade <- fct_reorder(data_highschool_fct$grade,
                                         data_highschool_fct$absences,
                                         .desc = TRUE)
data_highschool_fct %>% group_by(grade) %>% summarise(mean = mean(absences))
```

```
## # A tibble: 4 x 2
##   grade     mean
##   <fct>    <dbl>
## 1 Freshman  4
## 2 Senior    3.12
## 3 Sophmore  2.62
## 4 Junior    2
```