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}
## <environment: 0x10bcc5518>
```

```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 tilda use

If you see `~ .x` or `.` … this means `function(x){x}`. In other words, we are passing `x` to a function. See purrr-shortcuts

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.

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

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

Using across with helpers to apply function to multiple columns

```
?tidyr_tidy_select
```

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

```
##                mpg cyl disp  hp drat wt  qsec vs am gear carb
## Mazda RX4      21   6  160 110    4  3 16.46  0  1    4    4
## Mazda RX4 Wag  21   6  160 110    4  3 17.02  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!

Work inside other tidyverse functions, like `across()`

```
iris %>% filter(Sepal.Length > 2.4 & Sepal.Width > 2.4 &
                Petal.Length > 2.4 & Petal.Width > 2.4)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          6.3         3.3          6.0         2.5 virginica
## 2          7.2         3.6          6.1         2.5 virginica
## 3          6.7         3.3          5.7         2.5 virginica
```

```
iris %>% filter(if_all(Sepal.Length:Petal.Width, ~ . > 2.4))
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          6.3         3.3          6.0         2.5 virginica
## 2          7.2         3.6          6.1         2.5 virginica
## 3          6.7         3.3          5.7         2.5 virginica
```

```
iris %>% filter(if_all(where(is.numeric), ~ . > 2.4))
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          6.3         3.3          6.0         2.5 virginica
## 2          7.2         3.6          6.1         2.5 virginica
## 3          6.7         3.3          5.7         2.5 virginica
```

Previously we filtered for patterns or conditions…2 general ways

This can be done on multiple patterns like so:

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

```
## # A tibble: 2 × 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") %>% head(2)
```

```
## # A tibble: 2 × 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
```

Now we can filter multiple columns for multiple conditions simultaneously!

```r
mtcars %>%
   filter(if_all(c(cyl, gear, carb), ~.x > 3 & .x < 8))
```

```
##                 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
## 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
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

`purrr` is also a super helpful package!

"Designed to make your functions purrr."

`dplyr` is designed for data frames `purrr` 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

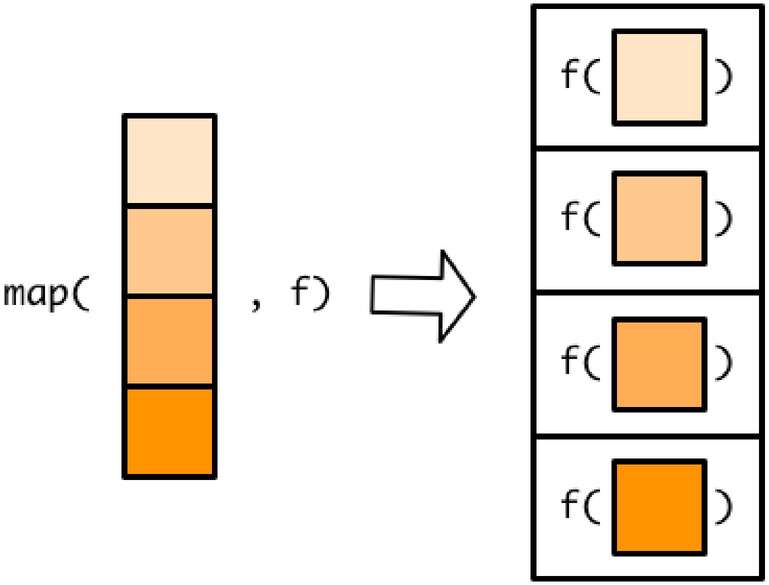
`map` and `map_*` and `modify`
- **applies function** to each element of an vector or object (map returns a list, modify returns the same object type)

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

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

the _* options specify the type of data output

map (and modify)



[source]

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

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

map (and modify)

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

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

```
modify(x, round)
```

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

`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 × 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)
```

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

It’s a bit simpler than across…

```
mtcars %>%
  modify(format, digits = 1) %>%
  head(n = 2)
```

```
##   mpg cyl disp  hp drat wt qsec vs am gear carb
## 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 %>%
  mutate(across(.cols = everything(), format, digits = 1)) %>%
  head(n = 2)
```

```
## Warning: There was 1 warning in `mutate()`.
## ℹ In argument: `across(.cols = everything(), format, digits = 1)`.
## Caused by warning:
## ! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.
## Supply arguments directly to `.fns` through an anonymous function instead.
##
##   # Previously
##   across(a:b, mean, na.rm = TRUE)
##
##   # Now
##   across(a:b, \(x) mean(x, na.rm = TRUE))
```

```
##                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
```

`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 × 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 × 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"
```

modify_if vs mutate/across/where

```
system.time(iris %>%
              modify_if(is.factor, as.character))
```

```
##    user  system elapsed 
##   0.001   0.000   0.001
```

```
system.time(iris %>%
               mutate(across(.cols = where(is.factor), as.character)))
```

```
##    user  system elapsed 
##   0.002   0.000   0.002
```

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 lists, 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
```

… Why lists?

great example with `split()`

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
str(mtcars %>% split(.$cyl))
```

```
## 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
```

great example with split()

```
mtcars %>%
  split(.$cyl) %>% # creates split of data for each unique cyl value
  map(~lm(mpg ~ wt, data = .)) %>% # apply linear model to each
  map(summary) %>%
  map_dbl("r.squared")
```

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

Lists from multiple files!

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/avahoffman/Dropbox/JHSPH/Data-Wrangling_SISBID/data/iris/iris_q1.csv"
## [2] "/Users/avahoffman/Dropbox/JHSPH/Data-Wrangling_SISBID/data/iris/iris_q4.csv"
## [3] "/Users/avahoffman/Dropbox/JHSPH/Data-Wrangling_SISBID/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 × 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
## # ℹ 140 more rows
```

Reading in multiple files

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

```
## # A tibble: 150 × 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
## # ℹ 140 more rows
```

Reading in multiple files

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

```
## # A tibble: 150 × 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
## # ℹ 140 more rows
```

Fixing the second file

First, separating by the `:`.

```
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 × 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
```

Fixing the second file

Second, making sure values are numeric.

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

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

```
## # A tibble: 3 × 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
```

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 × 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!

Lab

https://sisbid.github.io/Data-Wrangling/labs/functional-program-lab.Rmd