Programming for Data Analytics

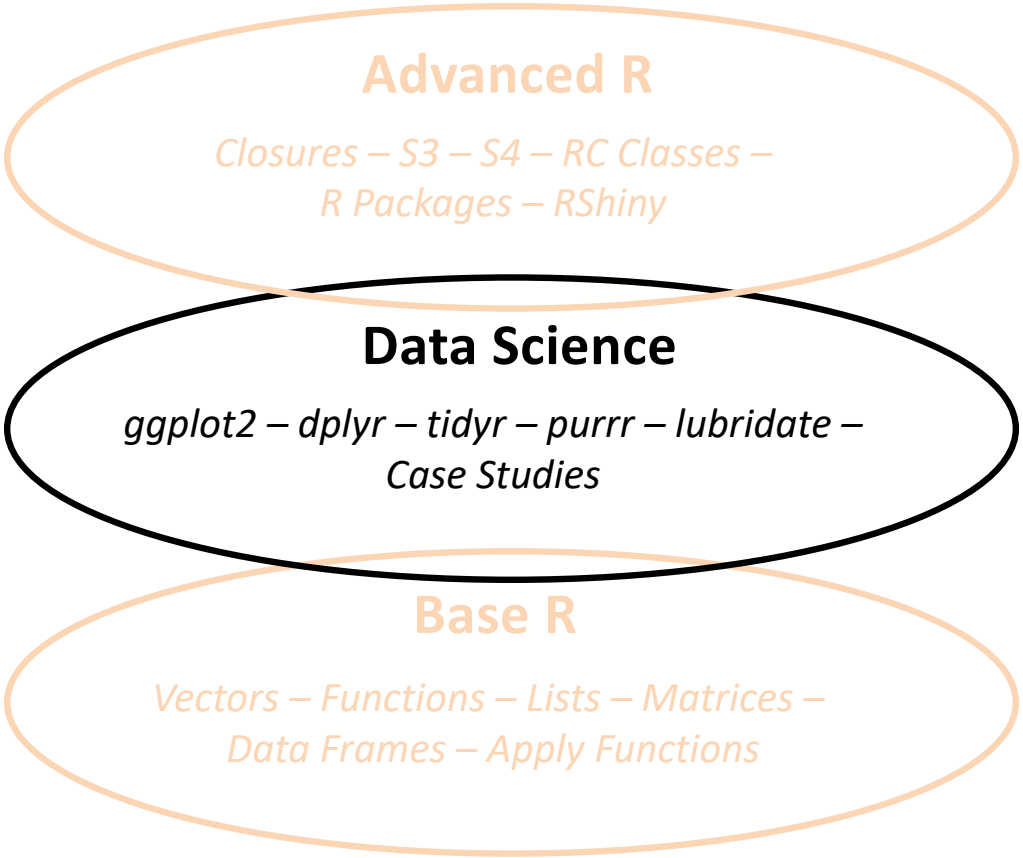
8. An Introduction to purrr

Prof. Jim Duggan,
School of Computer Science
National University of Ireland Galway.

https://github.com/JimDuggan/CT5102

Lecture Overview

- map Functions
- ~ anonymous function
- Statistical analysis
- nest() function (tidyr)
- Generating multiple plots



Introduction to purrr

http://www.rebeccabarter.com/blog/2019-08-19_purrr/

- purrr is all about iteration.
- purrr introduces map functions (the tidyverse's answer to base R's apply functions, but more in line with functional programming practices) as well as some new functions for manipulating lists

- While the workhorse of dplyr is the data frame, the workhorse of purrr is the list.
  - A **vector** is a way of storing many individual elements (a single number or a single character or string) of the same type together in a single object,
  - A **data frame** is a way of storing many vectors of the same length but possibly of different types together in a single object
  - A **list** is a way of storing many objects of any type (e.g. data frames, plots, vectors) together in a single object

(1) Map functions – beyond apply

- A **map function** is one that applies the same action/function to every element of an object (e.g. each entry of a list or a vector, or each of the columns of a data frame)
- The naming convention of the map functions are such that the type of the **output** is specified by the term that follows the underscore in the function name
- Consistent with the way of the tidyverse, the **first argument** of each mapping function is always the **data object** that you want to map over, and the **second argument** is always the ***function*** that you want to iteratively apply to each element of the input object

- `map(.x, .f)` is the main mapping function and returns a list
- `map_df(.x, .f)` returns a data frame
- `map_dbl(.x, .f)` returns a numeric (double) vector
- `map_chr(.x, .f)` returns a character vector
- `map_lgl(.x, .f)` returns a logical vector

The input object

- The **input** object to any map function is always either
  - a *vector* (of any type), in which case the iteration is done over the entries of the vector,
  - a *list*, in which case the iteration is performed over the elements of the list,
  - a *data frame*, in which case the iteration is performed over the columns of the data frame (which, since a data frame is a special kind of list, is technically the same as the previous point).

```
library(purrr)

x <- list(a=1:10, b=11:20)

> map(x,min)
$a
[1] 1

$b
[1] 11

$c
[1] 21
```

```
> map_dbl(x,min)
 a  b  c
 1 11 21
```

Iteration example

```
library(purrr)

addTen <- function(.x) {
  cat("Calling addTen with .x=",.x,"\n")
  .x + 10
}

map(.x = c(1, 4, 7),
    .f = addTen)
```

```
> map(.x = c(1, 4, 7),
+     .f = addTen)
Calling addTen with .x= 1
Calling addTen with .x= 4
Calling addTen with .x= 7
[[1]]
[1] 11

[[2]]
[1] 14

[[3]]
[1] 17
```

The output

- The first element of the output is the result of applying the function to the first element of the input (1),
- The second element of the output is the result of applying the function to the second element of the input (4),
- The third element of the output is the result of applying the function to the third element of the input (7).

```
> map(.x = c(1, 4, 7),
+     .f = addTen)
Calling addTen with .x= 1
Calling addTen with .x= 4
Calling addTen with .x= 7
[[1]]
[1] 11

[[2]]
[1] 14

[[3]]
[1] 17
```

map() always returns a list

```
> map(list(1,4,7),addTen)
Calling addTen with .x= 1
Calling addTen with .x= 4
Calling addTen with .x= 7
[[1]]
[1] 11

[[2]]
[1] 14

[[3]]
[1] 17
```

```
> map(tibble(a=1,b=4,c=7),addTen)
Calling addTen with .x= 1
Calling addTen with .x= 4
Calling addTen with .x= 7
$a
[1] 11

$b
[1] 14

$c
[1] 17
```

Challenge 8.1

- Start with the tibble mpg
- Select the columns cty, hwy and displ
- Get the mean values for each of these, using map()

Adding Anonymous Functions

```
> map(c(1,4,7),function(.x).x+10)
[[1]]
[1] 11

[[2]]
[1] 14

[[3]]
[1] 17
```

```
> map(list(1,4,7),function(.x).x+10)
[[1]]
[1] 11

[[2]]
[1] 14

[[3]]
[1] 17
```

Modifying the output from map

- If we wanted the output of map to be some other object type, we need to use a different function.
- For instance to map the input to a numeric (double) vector, you can use the map_dbl() (“map to a double”) function

- `map_df(.x, .f)` returns a data frame
- `map_dbl(.x, .f)` returns a numeric (double) vector
- `map_chr(.x, .f)` returns a character vector
- `map_lgl(.x, .f)` returns a logical vector

```
> map_dbl(c(1,4,7),function(.x).x+10)
[1] 11 14 17
>
> map_chr(c(1,4,7),function(.x).x+10)
[1] "11.000000" "14.000000" "17.000000"
```

map_df()

- If you want to return a data frame, then you would use the map_df() function.
- However, you need to make sure that in each iteration you're returning a data frame which has consistent column names.
- map_df will automatically bind the rows of each iteration.

```
> map_df(c(1,4,7),function(.x){
+   tibble(Old=.x,New=.x+10)
+ })
# A tibble: 3 x 2
    Old   New
  <dbl> <dbl>
1     1    11
2     4    14
3     7    17
```

(2) The tilde-dot shorthand for functions

- To make the code more concise you can use the tilde-dot shorthand for anonymous functions (the functions that you create as arguments of other functions).
- ~ indicates that you have started an anonymous function, and the argument of the anonymous function can be referred to using **.x** (or simply .).
- Unlike normal function arguments that can be anything that you like, ***the tilde-dot function argument is always .x.***

```
> map_dbl(c(1,4,7),~.x+10)
[1] 11 14 17
>
> map_dbl(c(1,4,7),~{.x+10})
[1] 11 14 17
>
> map_dbl(c(1,4,7),~.+10)
[1] 11 14 17
```

Challenge 8.2

- Write a map function that takes in a list of elements, and then returns (as an integer) the number of elements in each list
- Use the ~ operator to perform the same task

gapminder package

- The gapminder dataset has 1704 rows containing information on population, life expectancy and GDP per capita by year and country.
- It's in tidy data format

```
> gapminder
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333       779.
 2 Afghanistan Asia       1957    30.3  9240934       821.
 3 Afghanistan Asia       1962    32.0 10267083       853.
 4 Afghanistan Asia       1967    34.0 11537966       836.
 5 Afghanistan Asia       1972    36.1 13079460       740.
 6 Afghanistan Asia       1977    38.4 14880372       786.
 7 Afghanistan Asia       1982    39.9 12881816       978.
 8 Afghanistan Asia       1987    40.8 13867957       852.
 9 Afghanistan Asia       1992    41.7 16317921       649.
10 Afghanistan Asia       1997    41.8 22227415       635.
# … with 1,694 more rows
```

http://www.rebeccabarter.com/blog/2019-08-19_purrr/

Analysing gapminder

- Since gapminder is a data frame, the map_ functions will iterate over each column.
- An example of simple usage of the map_ functions is to summarize each column.
- For instance, you can identify the type of each column by applying the class() function to each column.
- Since the output of the class() function is a character, we will use the map_chr() function:

```
> gapminder %>% map_chr(class)
  country continent      year   lifeExp       pop gdpPercap
 "factor"  "factor" "integer" "numeric" "integer" "numeric"
```

Further analysis

http://www.rebeccabarter.com/blog/2019-08-19_purrr/

- Similarly, if you wanted to identify the number of distinct values in each column, you could apply the n_distinct() function from the dplyr package to each column.
- Since the output of n_distinct() is a numeric (a double), you might want to use the map_dbl() function so that the results of each iteration (the application of n_distinct() to each column) are concatenated into a numeric vector:

```
> gapminder %>% map_dbl(n_distinct)
  country continent      year   lifeExp       pop gdpPercap
      142         5        12      1626      1704      1704
```

- If you want to do something a little more complicated, such return a few different summaries of each column in a data frame, you can use **map_df().**
- You typically need to define an anonymous function that you want to apply to each column.
- Using the tilde-dot notation, the anonymous function calculates the number of distinct entries and the type of the current column (which is accessible as .x), and then combines them into a two-column data frame.
- Once it has iterated through each of the columns, the map_df function combines the data frames row-wise into a single data frame.

```
> gapminder %>%
+   map_df(~(tibble(n_distinct = n_distinct(.x),
+                   class = class(.x))))
# A tibble: 6 x 2
  n_distinct class
       <int> <chr>
1        142 factor
2          5 factor
3         12 integer
4       1626 numeric
5       1704 integer
6       1704 numeric
> gapminder %>%
+   map_df(~(tibble(n_distinct = n_distinct(.),
+                   class = class(.))))
# A tibble: 6 x 2
  n_distinct class
       <int> <chr>
1        142 factor
2          5 factor
3         12 integer
4       1626 numeric
5       1704 integer
6       1704 numeric
```

Adding the variable name

http://www.rebeccabarter.com/blog/2019-08-19_purrr/

- The variable names correspond to the names of the objects over which we are iterating (in this case, the column names), and these are not automatically included as a column in the output data frame.
- You can tell map_df() to include them using the .id argument of map_df().
- This will automatically take the name of the element being iterated over and include it in the column corresponding to whatever you set .id to

```
> gapminder %>%
+   map_df(~(tibble(n_distinct = n_distinct(.x),
+                   class = class(.x))),
+          .id="variable")
# A tibble: 6 x 3
  variable  n_distinct class
  <chr>          <int> <chr>
1 country          142 factor
2 continent          5 factor
3 year              12 integer
4 lifeExp         1626 numeric
5 pop             1704 integer
6 gdpPercap       1704 numeric
```

modify_if()

- **modify_if(),** only applies the function to elements that satisfy a specific criteria (specified by a "predicate function", the second argument called .p).
- For instance, the following example only modifies the third entry since it is greater than 5.

```
modify_if(c(1,4,7),
          function (x) x > 5,
          ~.x+10)
```

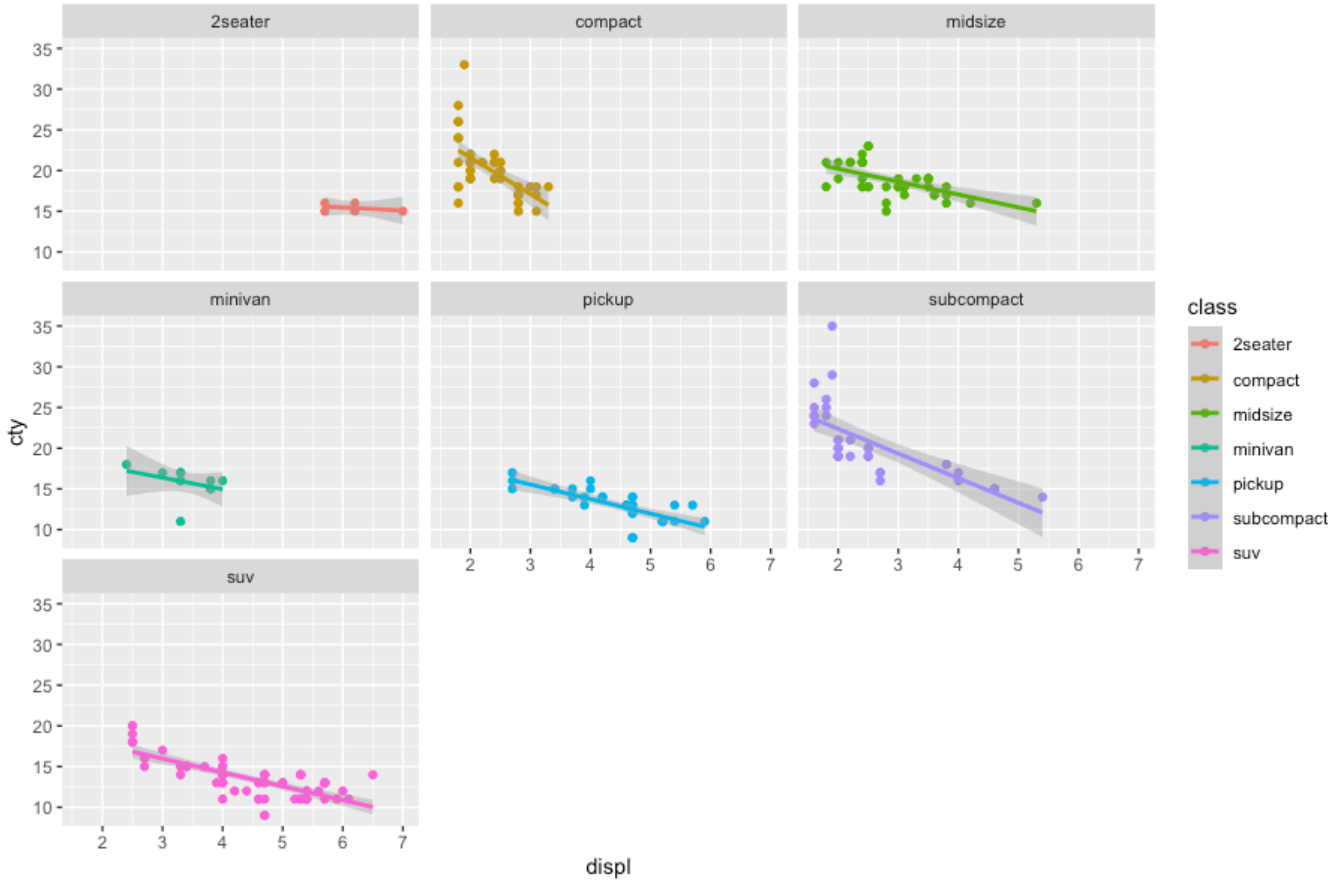
```
> modify_if(c(1,4,7),
+           function (x) x > 5,
+           ~.x+10)
[1]  1  4 17
```

Challenge 8.3

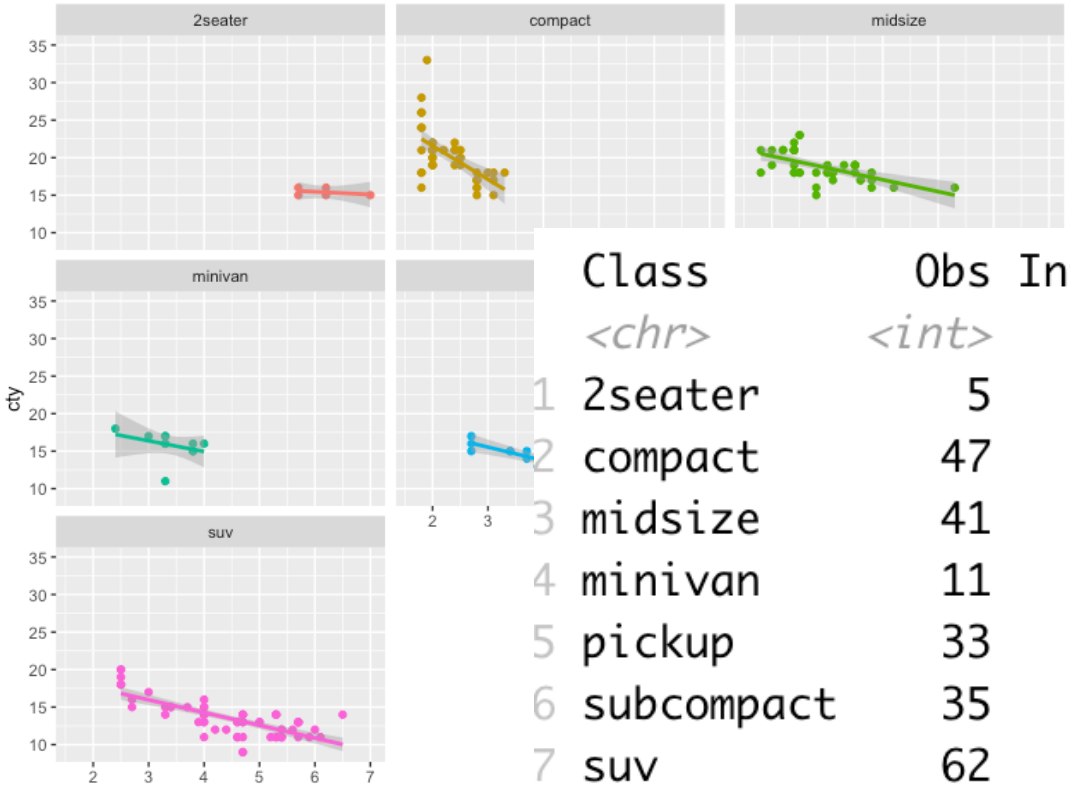
- Generate the following tibble from aimsir17::observations, making use of **map_df()**

```
# A tibble: 13 x 4
   Column  class     length ndistinct
   <chr>   <chr>      <int>     <int>
 1 station character 219000        25
 2 year    numeric   219000         1
 3 month   numeric   219000        12
 4 day     integer   219000        31
 5 hour    integer   219000        24
 6 date    POSIXct   219000      8760
 7 date    POSIXt    219000      8760
 8 rain    numeric   219000       109
 9 temp    numeric   219000       330
10 rhum    numeric   219000        82
11 msl     numeric   219000       740
12 wdsp    numeric   219000        54
13 wddir   numeric   219000        38
```

(3) Statistical Analysis - mpg



Getting to the details…



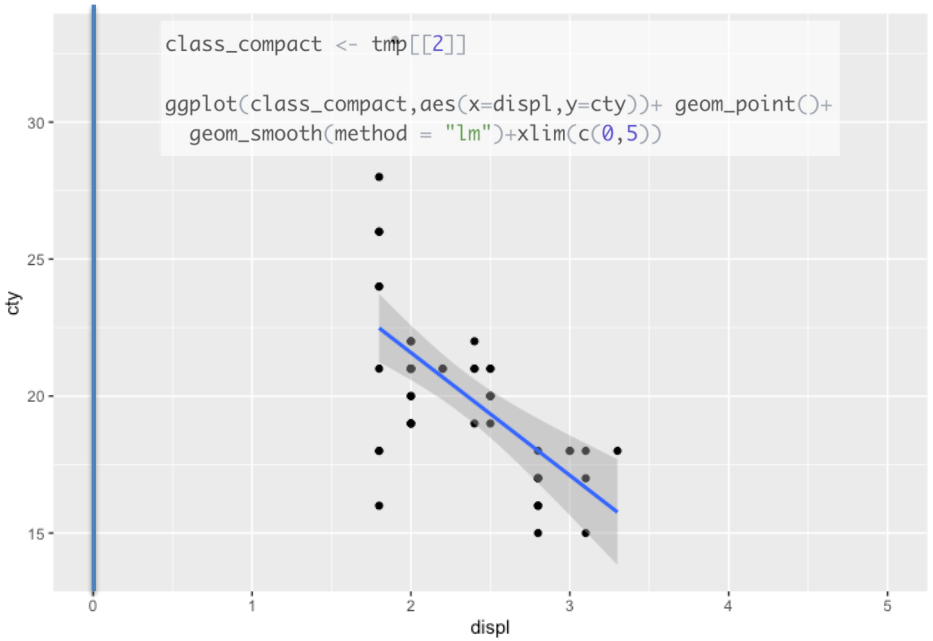
	Class	Obs	Intercept	Slope	RSquared	AdjRSquared
	<chr>	<int>	<dbl>	<dbl>	<dbl>	<dbl>
1	2seater	5	17.7	-0.371	0.130	-0.160
2	compact	47	30.5	-4.48	0.358	0.344
3	midsize	41	23.4	-1.58	0.339	0.322
4	minivan	11	20.6	-1.42	0.124	0.0262
5	pickup	33	20.9	-1.79	0.525	0.509
6	subcompact	35	28.4	-3.03	0.527	0.512
7	suv	62	21.1	-1.70	0.558	0.550

dplyr and purrr

```
> mpg %>% group_split(class) -> tmp
> str(tmp)
list<tbl_df[,11]> [1:7]
$ : tibble [5 × 11] (S3: tbl_df/tbl/
 ..$ manufacturer: chr [1:5] "chevrc
 ..$ model       : chr [1:5] "corvet
 ..$ displ       : num [1:5] 5.7 5.7
 ..$ year        : int [1:5] 1999 19
 ..$ cyl         : int [1:5] 8 8 8 8
 ..$ trans       : chr [1:5] "manual
 ..$ drv         : chr [1:5] "r" "r"
 ..$ cty         : int [1:5] 16 15 1
 ..$ hwy         : int [1:5] 26 23 2
 ..$ fl          : chr [1:5] "p" "p"
 ..$ class       : chr [1:5] "2seate
```

```
> select(tmp[[2]],class,everything())
# A tibble: 47 x 11
   class manufacturer model displ  year   cyl trans drv     cty   hwy
   <chr> <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int>
 1 comp… audi         a4      1.8  1999     4 auto… f        18    29
 2 comp… audi         a4      1.8  1999     4 manu… f        21    29
 3 comp… audi         a4      2    2008     4 manu… f        20    31
 4 comp… audi         a4      2    2008     4 auto… f        21    30
 5 comp… audi         a4      2.8  1999     6 auto… f        16    26
 6 comp… audi         a4      2.8  1999     6 manu… f        18    26
 7 comp… audi         a4      3.1  2008     6 auto… f        18    27
 8 comp… audi         a4 q…   1.8  1999     4 manu… 4        18    26
> select(tmp[[7]],class,everything())
# A tibble: 62 x 11
   class manufacturer model displ  year   cyl trans drv     cty   hwy
   <chr> <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <int>
 1 suv   chevrolet    c150… 5.3   2008     8 auto… r        14    20
 2 suv   chevrolet    c150… 5.3   2008     8 auto… r        11    15
 3 suv   chevrolet    c150… 5.3   2008     8 auto… r        14    20
 4 suv   chevrolet    c150… 5.7   1999     8 auto… r        13    17
 5 suv   chevrolet    c150… 6     2008     8 auto… r        12    17
 6 suv   chevrolet    k150… 5.3   2008     8 auto… 4        14    19
```

A linear model, with some model results




```
> mod <- lm(cty~displ,data=class_compact)
> mod$coefficients
(Intercept)       displ
   30.54586    -4.47992
> summ <- summary(mod)
> summ$r.squared
[1] 0.3581767
> summ$adj.r.squared
[1] 0.3439139
```

Overall Approach

- Group the data by class
- For each class, calculate and record:
  - Number of observations
  - Class name
  - Intercept
  - Slope
  - RSquared
  - AdjRSquared
- Return as a tibble

	Class	Obs	Intercept	Slope	RSquared	AdjRSquared
	<chr>	<int>	<dbl>	<dbl>	<dbl>	<dbl>
1	2seater	5	17.7	-0.371	0.130	-0.160
2	compact	47	30.5	-4.48	0.358	0.344
3	midsize	41	23.4	-1.58	0.339	0.322
4	minivan	11	20.6	-1.42	0.124	0.0262
5	pickup	33	20.9	-1.79	0.525	0.509
6	subcompact	35	28.4	-3.03	0.527	0.512
7	suv	62	21.1	-1.70	0.558	0.550

R Script

```
mpg %>% group_split(class) %>%
      map_df(~{
        mod <- lm (.$cty~.$displ)
        summ <- summary(mod)
        tibble(Class=first(.$class),
               Obs=nrow(.),
               Intercept=mod$coefficients[1],
               Slope=mod$coefficients[2],
               RSquared=summ$r.squared,
               AdjRSquared=summ$adj.r.squared)
      })
```

Challenge 8.4

- Generate a similar tibble for the manufacturers in mpg.
- For this example, use hwy as the dependent variable, and cyl as the independent variable
- Check the results using ggplot() and geom_smooth()

	Class	Obs	Intercept	Slope	RSquared	AdjRSquared
	<chr>	<int>	<dbl>	<dbl>	<dbl>	<dbl>
1	2seater	5	17.7	-0.371	0.130	-0.160
2	compact	47	30.5	-4.48	0.358	0.344
3	midsize	41	23.4	-1.58	0.339	0.322
4	minivan	11	20.6	-1.42	0.124	0.0262
5	pickup	33	20.9	-1.79	0.525	0.509
6	subcompact	35	28.4	-3.03	0.527	0.512
7	suv	62	21.1	-1.70	0.558	0.550

(4) Nesting of Tibbles (tidyr)

- Tibble columns can be lists (as opposed to vectors, which is what they usually are)..
- For instance, a tibble can be "nested" where the tibble is essentially split into separate data frames based on a **grouping variable**, and these separate data frames are stored as entries of a list (that is then stored in the data column of the data frame).

```
> mpg_nested <- mpg %>%
+               group_by(class) %>%
+               nest()
> mpg_nested
# A tibble: 7 x 2
# Groups:   class [7]
  class      data
  <chr>      <list>
1 compact    <tibble [47 × 10]>
2 midsize    <tibble [41 × 10]>
3 suv        <tibble [62 × 10]>
4 2seater    <tibble [5 × 10]>
5 minivan    <tibble [11 × 10]>
6 pickup     <tibble [33 × 10]>
7 subcompact <tibble [35 × 10]>
```

Extracting data (pluck function)

```
> mpg_nested %>% pluck("data", 1)
# A tibble: 47 x 10
   manufacturer model  displ  year   cyl trans drv     cty   hwy fl
   <chr>        <chr>  <dbl> <int> <int> <chr> <chr> <int> <int> <chr
 1 audi         a4       1.8  1999     4 auto… f        18    29 p
 2 audi         a4       1.8  1999     4 manu… f        21    29 p
 3 audi         a4       2    2008     4 manu… f        20    31 p
 4 audi         a4       2    2008     4 auto… f        21    30 p
 5 audi         a4       2.8  1999     6 auto… f        16    26 p
 6 audi         a4       2.8  1999     6 manu… f        18    26 p
 7 audi         a4       3.1  2008     6 auto… f        18    27 p
 8 audi         a4 qu…   1.8  1999     4 manu… 4        18    26 p
 9 audi         a4 qu…   1.8  1999     4 auto… 4        16    25 p
10 audi         a4 qu…   2    2008     4 manu… 4        20    28 p
# … with 37 more rows
```

Storing model results as a column

```
lm_mpg <- mpg_nested %>%
            mutate(lm_obj = map(data,
                                ~lm(cty ~ displ,data = .)))
```

```
> lm_mpg
# A tibble: 7 x 3
# Groups:   class [7]
  class      data               lm_obj
  <chr>      <list>             <list>
1 compact    <tibble [47 × 10]> <lm>
2 midsize    <tibble [41 × 10]> <lm>
3 suv        <tibble [62 × 10]> <lm>
4 2seater    <tibble [5 × 10]>  <lm>
5 minivan    <tibble [11 × 10]> <lm>
6 pickup     <tibble [33 × 10]> <lm>
7 subcompact <tibble [35 × 10]> <lm>
```

```
> lm1 <- lm_mpg %>% pluck("lm_obj",1)
>
> lm1

Call:
lm(formula = cty ~ displ, data = .)

Coefficients:
(Intercept)        displ
      30.55        -4.48
```

```r
mpg_models<- mpg %>%
              group_by(class) %>%
              nest()    %>%
              mutate(lm_obj = map(data,
                                  ~lm(cty ~ displ,data = .)))
```

```
> mpg_models
# A tibble: 7 x 3
# Groups:   class [7]
  class      data               lm_obj
  <chr>      <list>             <list>
1 compact    <tibble [47 × 10]> <lm>
2 midsize    <tibble [41 × 10]> <lm>
3 suv        <tibble [62 × 10]> <lm>
4 2seater    <tibble [5 × 10]>  <lm>
5 minivan    <tibble [11 × 10]> <lm>
6 pickup     <tibble [33 × 10]> <lm>
7 subcompact <tibble [35 × 10]> <lm>
```

```r
mods <- pull(mpg_models,"lm_obj")
mods[[5]]
```

```
> mods[[5]]

Call:
lm(formula = cty ~ displ, data = .)

Coefficients:
(Intercept)        displ
     20.647       -1.424
```

Challenge 8.5

- Generate a similar set of models (25) for observations (by station), where wdsp is the dependent variable, and msl is the independent variable

```
> lm_mpg
# A tibble: 7 x 3
# Groups:   class [7]
  class      data               lm_obj
  <chr>      <list>             <list>
1 compact    <tibble [47 × 10]> <lm>
2 midsize    <tibble [41 × 10]> <lm>
3 suv        <tibble [62 × 10]> <lm>
4 2seater    <tibble [5 × 10]>  <lm>
5 minivan    <tibble [11 × 10]> <lm>
6 pickup     <tibble [33 × 10]> <lm>
7 subcompact <tibble [35 × 10]> <lm>
```

```
> lm1 <- lm_mpg %>% pluck("lm_obj",1)
>
> lm1

Call:
lm(formula = cty ~ displ, data = .)

Coefficients:
(Intercept)        displ
      30.55        -4.48
```

(5) Generating Multiple Plots

```
library(purrr)
library(aimsir17)
library(dplyr)

summ <- observations %>% group_by(station,month,day) %>%
  summarise(MeanWind=mean(wdsp,na.rm=T),MeanMSL=mean(msl,na.rm=T)) %>%
  ungroup()

plots <- summ %>%
          group_split(station) %>%
          map(~{
            ggplot(.,aes(x=MeanMSL,y=MeanWind,colour=station))+
              geom_point()+geom_smooth()+
              ggtitle(.$station)
          })
```

```
> plots[[1]]
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
> plots[[10]]
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

Summary

- map functions
- ~ anonymous function
- Statistical analysis
- nest() function
- Generating multiple plots

- `map(.x, .f)` is the main mapping function and returns a list
- `map_df(.x, .f)` returns a data frame
- `map_dbl(.x, .f)` returns a numeric (double) vector
- `map_chr(.x, .f)` returns a character vector
- `map_lgl(.x, .f)` returns a logical vector

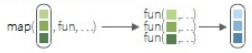
Apply functions with purrr : : **CHEAT SHEET**

Map Functions

ONE LIST

map(.x, .f, ...) Apply a function to each element of a list or vector, return a list.

```
x <- list(1:10, 11:20, 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)
```



map_dbl(.x, .f, ...) Return a double vector.
map_dbl(x, mean)

map_int(.x, .f, ...) Return an integer vector.
map_int(x, length)

map_chr(.x, .f, ...) Return a character vector.
map_chr(l1, paste, collapse = "")

map_lgl(.x, .f, ...) Return a logical vector.
map_lgl(x, is.integer)

map_dfc(.x, .f, ...) Return a data frame created by column-binding.
map_dfc(l1, rep, 3)

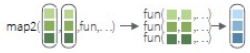
map_dfr(.x, .f, ..., .id = NULL) Return a data frame created by row-binding.
map_dfr(x, summary)

walk(.x, .f, ...) Trigger side effects, return invisibly.
walk(x, print)

TWO LISTS

map2(.x, .y, .f, ...) Apply a function to pairs of elements from two lists or vectors, return a list.

```
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y, ~ .x * .y)
```



map2_dbl(.x, .y, .f, ...) Return a double vector.
map2_dbl(y, z, ~ .x / .y)

map2_int(.x, .y, .f, ...) Return an integer vector.
map2_int(y, z, `+`)

map2_chr(.x, .y, .f, ...) Return a character vector.
map2_chr(l1, l2, paste, collapse = ",", sep = ":")

map2_lgl(.x, .y, .f, ...) Return a logical vector.
map2_lgl(l2, l1, `%in%`)

map2_dfc(.x, .y, .f, ...) Return a data frame created by column-binding.
map2_dfc(l1, l2, ~ as.data.frame(c(.x, .y)))

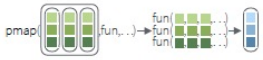
map2_dfr(.x, .y, .f, ..., .id = NULL) Return a data frame created by row-binding.
map2_dfr(l1, l2, ~ as.data.frame(c(.x, .y)))

walk2(.x, .y, .f, ...) Trigger side effects, return invisibly.
walk2(objs, paths, save)

MANY LISTS

pmap(.l, .f, ...) Apply a function to groups of elements from a list of lists or vectors, return a list.

```
pmap(list(x, y, z), ~ ..1 * (..2 + ..3))
```

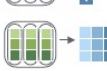


pmap_dbl(.l, .f, ...) Return a double vector.
pmap_dbl(list(y, z), ~ .x / .y)

pmap_int(.l, .f, ...) Return an integer vector.
pmap_int(list(y, z), `+`)

pmap_chr(.l, .f, ...) Return a character vector.
pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")

pmap_lgl(.l, .f, ...) Return a logical vector.
pmap_lgl(list(l2, l1), `%in%`)



pmap_dfc(.l, .f, ...) Return a data frame created by column-binding.
pmap_dfc(list(l1, l2), ~ as.data.frame(c(.x, .y)))

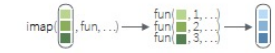
pmap_dfr(.l, .f, ..., .id = NULL) Return a data frame created by row-binding.
pmap_dfr(list(l1, l2), ~ as.data.frame(c(.x, .y)))

pwalk(.l, .f, ...) Trigger side effects, return invisibly.
pwalk(list(objs, paths), save)

LISTS AND INDEXES

imap(.x, .f, ...) Apply .f to each element and its index, return a list.

```
imap(y, ~ paste0(.y, ": ", .x))
```



imap_dbl(.x, .f, ...) Return a double vector.
imap_dbl(y, ~ .y)

imap_int(.x, .f, ...) Return an integer vector.
imap_int(y, ~ .y)

imap_chr(.x, .f, ...) Return a character vector.
imap_chr(y, ~ paste0(.y, ": ", .x))

imap_lgl(.x, .f, ...) Return a logical vector.
imap_lgl(l1, ~ is.character(.y))

imap_dfc(.x, .f, ...) Return a data frame created by column-binding.
imap_dfc(l2, ~ as.data.frame(c(.x, .y)))

imap_dfr(.x, .f, ..., .id = NULL) Return a data frame created by row-binding.
imap_dfr(l2, ~ as.data.frame(c(.x, .y)))

iwalk(.x, .f, ...) Trigger side effects, return invisibly.
iwalk(z, ~ print(paste0(.y, ": ", .x)))

Function Shortcuts

Use ~ . with functions like **map()** that have single arguments.

```
map(l, ~ . + 2)
becomes
map(l, function(x) x + 2 ))
```

Use ~ .x .y with functions like **map2()** that have two arguments.

```
map2(l, p, ~ .x +.y)
becomes
map2(l, p, function(l, p) l + p)
```

Use ~ **..1 ..2 ..3** etc with functions like **pmap()** that have many arguments.

```
pmap(list(a, b, c), ~ ..3 + ..1 - ..2)
becomes
pmap(list(a, b, c), function(a, b, c) c + a -b)
```

Use ~ .x .y with functions like **imap()**. .x will get the list value and .y will get the index.

```
imap(list(a, b, c), ~ paste0(.y, ": ", .x)
outputs "index: value" for each item
```

Use a **string** or an **integer** with any map function to index list elements by name or position. `map(l, "name")` becomes `map(l, function(x) x[["name"]])`

Work with Lists

Filter

keep(.x, .p, ...)
Select elements that pass a logical test.
Conversely, **discard()**.
keep(x, is.na)

compact(.x, .p = identity)
Drop empty elements.
compact(x)

head_while(.x, .p, ...)
Return head elements until one does not pass.
Also **tail_while()**.
head_while(x, is.character)

detect(.x, .f, ..., dir = c("forward", "backward"), .right = NULL, .default = NULL)
Find first element to pass.
detect(x, is.character)

detect_index(.x, .f, ..., dir = c("forward", "backward"), .right = NULL) Find index of first element to pass.
detect_index(x, is.character)

every(.x, .p, ...)
Do all elements pass a test?
every(x, is.character)

some(.x, .p, ...)
Do some elements pass a test?
some(x, is.character)

none(.x, .p, ...)
Do no elements pass a test?
none(x, is.character)

has_element(.x, .y)
Does a list contain an element?
has_element(x, "foo")

vec_depth(x)
Return depth (number of levels of indexes).
vec_depth(x)

Index

pluck(.x, ..., .default=NULL)
Select an element by name or index. Also **attr_getter()** and **chuck()**.
pluck(x, "b")
x %>% pluck("b")

assign_in(x, where, value)
Assign a value to a location using pluck selection.
assign_in(x, "b", 5)
x %>% assign_in("b", 5)

modify_in(.x, .where, .f)
Apply a function to a value at a selected location.
modify_in(x, "b", abs)
x %>% modify_in("b", abs)

Reshape

flatten(.x) Remove a level of indexes from a list.
Also **flatten_chr()** etc.
flatten(x)

array_tree(array, margin = NULL) Turn array into list.
Also **array_branch()**.
array_tree(x, margin = 3)

cross2(.x, .y, .filter = NULL)
All combinations of .x and .y.
Also **cross()**, **cross3()**, and **cross_df()**.
cross2(1:3, 4:6)

transpose(.l, .names = NULL)
Transposes the index order in a multi-level list.
transpose(x)

set_names(x, nm = x)
Set the names of a vector/list directly or with a function.
set_names(x, c("p", "q", "r"))
set_names(x, tolower)

Modify

modify(.x, .f, ...) Apply a function to each element. Also **modify2()**, and **imodify()**.
modify(x, ~ + 2)

modify_at(.x, .at, .f, ...) Apply a function to selected elements.
Also **map_at()**.
modify_at(x, "b", ~ + 2)

modify_if(.x, .p, .f, ...) Apply a function to elements that pass a test. Also **map_if()**.
modify_if(x, is.numeric, ~ +2)

modify_depth(.x, .depth, .f, ...) Apply function to each element at a given level of a list. Also **map_depth()**.
modify_depth(x, 2, ~ + 2)

Combine

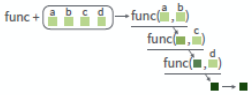
append(x, values, after = length(x)) Add values to end of list.
append(x, list(d = 1))

prepend(x, values, before = 1)
Add values to start of list.
prepend(x, list(d = 1))

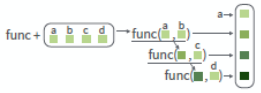
splice(...) Combine objects into a list, storing S3 objects as sub-lists.
splice(x, y, "foo")

Reduce

reduce(.x, .f, ..., .init, .dir = c("forward", "backward")) Apply function recursively to each element of a list or vector. Also **reduce2()**.
reduce(x, sum)



accumulate(.x, .f, ..., .init) Reduce a list, but also return intermediate results. Also **accumulate2()**.
accumulate(x, sum)



List-Columns

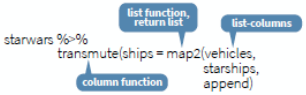
max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

List-columns are columns of a data frame where each element is a list or vector instead of an atomic value. Columns can also be lists of data frames. See **tidyr** for more about nested data and list columns.

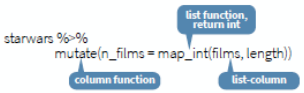
WORK WITH LIST-COLUMNS

Manipulate list-columns like any other kind of column, using **dplyr** functions like **mutate()** and **transmute()**. Because each element is a list, use **map functions** within a column function to manipulate each element.

map(), map2(), or pmap() return lists and will **create new list-columns.**



Suffixed map functions like **map_int()** return an atomic data type and will **simplify list-columns into regular columns.**



RStudio