Week 8 Walkthrough: Row Manipulations with `dplyr`

2025-02-21

Functions Covered

In this walkthrough, we'll be focusing on the following functions:

- `as_tibble()`: Make **data frames** into **tibbles** (fancy **data frames**)
  - **Tibbles** make for cleaner printing mostly, they're kinda nice.
- `filter()`: **Filters** rows to only the ones you want!
  - Usually, we'll do **logical conditioning** for how we want to **filter** the data, back to our **indexing vectors**!
- `arrange()`: **Arranges** rows based on the values of a given column
  - We can pick a column and choose whether to sort the data in **ascending** or **descending**[1] order.

`dplyr`

First things first, let's talk about the `dplyr` package (pronounced DEE-ply-er)! The `dplyr` package is an important part of the `tidyverse` and lets us do a lot of convenient data manipulations! It also gives us `as_tibble()`, a way to make our **data frames** a little prettier.

The function `as_tibble()` just tells `R` to treat a given **data frame** as a **tibble**. Once done, we get slightly cleaner printing and keep all the functionality of **data frames**! I've included a brief example below, just know that after using `as_tibble()` everything works just as before. Don't forget to load `dplyr` for all the code in this section!

```r
library(dplyr)
iris_tibble<-as_tibble(iris)
print(iris_tibble)
```

```
## # A tibble: 150 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
##  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
```

Now using square brackets works just the same as ever, but we'll teach some other (more easily understandable) tools for these manipulations as well.

[1] Requires 'desc()'!

```r
iris_tibble[1:3,1:3]
```

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

`filter()`

Alright! So `filter()` is *the* critical function for picking out or eliminating specific rows from your data. This requires **logical conditioning**, though, so let's start with that.

Logical Conditioning

Logical statements are everything that returns either `TRUE` or `FALSE` (you can think of these as "yes" and "no").

I'll throw some examples in a code chunk below, look to the comments for a verbal description!

```r
2 == 1 ; 'SUNDAY' == 'SUNDAY' ; 10 == '10'      #Check if two values are the same
```

```
## [1] FALSE

## [1] TRUE

## [1] TRUE
```

```r
3 != 2 ; 'Soccer' != 'Adult'                    #Check if two values are different
```

```
## [1] TRUE

## [1] TRUE
```

```r
3 >= 1 ; 3 <= 3 ; 3 < 3                         #Compare two values (only numeric!)
```

```
## [1] TRUE

## [1] TRUE

## [1] FALSE
```

```r
is.numeric('Two') ; is.character(10) ; is.na(1) #Check type
```

```
## [1] FALSE

## [1] FALSE

## [1] FALSE
```

It's important to note that if a value is `NA`, it will return `NA` from any of these except `is.na()`!

```r
NA == 'Missing' ; NA != 'Missing' ; NA >= 10 #Missing values returned!
```

```
## [1] NA

## [1] NA

## [1] NA
```

We can also then **compound** (combine) logical operations with `&` (and) and `|` (or, on your keyboard as Shift + \).

```
'a' != 'b' | 'a' == 'c'
```

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

```
5 >= 0 & 5 <= 10
```

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

There's a lot going on here! My recommendation is to play around with these as you can, *especially* `&` and `|`! Logic is not something which is natural for a lot of people, so reference this document and the slides when you need a reminder of what symbols to use. If you want to get really deep into it, you can follow **this link**, but that's beyond the scope of this course!

On to Filtering!

So now that we have an idea of **logical statements**, we can get to **filtering**! When using `filter()`, we need a **logical indexing vector** of the same length as our data, so a **vector** of `TRUE` and `FALSE` values. The way we will *usually* generate this is using the columns of our data! As an example, let's look in the `carData` library for `AMSsurvey`. If you want to know more about this dataset, you can type `?carData::AMSsurvey` into your console, but put simply this is about PhD students in the mathematical sciences (like your instructor!) from 2008-2012. First things first, let's load the library and take a look at the data.

```
library(carData)
head(AMSsurvey)
```

```
##    type    sex citizen count count11
## 1 I(Pu)   Male      US   132     148
## 2 I(Pu) Female      US    35      40
## 3 I(Pr)   Male      US    87      63
## 4 I(Pr) Female      US    20      22
## 5    II   Male      US    96     161
## 6    II Female      US    47      53
```

So our columns here are `type`, `sex`, `citizen`, `count`, and `count11`. The last two are the number of graduates in 2008-09 and 2011-12 respectively and the others are mostly self-explanatory. Since a `type` of `IV` is for statistics and biostatistics programs, let's restrict our data to just that using `filter()`! To do this, we first need to open the `dplyr` library and then put the name of the variable and our **logical operator**, in this case `==`.

```
library(dplyr)
filter(AMSsurvey, type == 'IV')
```

```
##   type    sex citizen count count11
## 1   IV   Male      US    71      89
## 2   IV Female      US    54      55
## 3   IV   Male  Non-US   122     153
## 4   IV Female  Non-US   105     115
```

Like that! Now that we've done that, let's add another condition. Let's restrict only to the US citizens and look there at the data. We can do this by including our `&` operator! I'll write three lines below, know that they all produce the *exact same* output!

```
filter(AMSsurvey, type == 'IV' & citizen != 'Non-US')
filter(AMSsurvey, type == 'IV', citizen == 'US')
```

```
filter(AMSsurvey, type == 'IV' & citizen == 'US')
```

```
##   type    sex citizen count count11
## 1   IV   Male      US    71      89
## 2   IV Female      US    54      55
```

Great! Next let's add more complexity and say that we want people who are in type `IV` (statistics/biostatistics) and now also people in type `Va`. We *could* make this much more longwinded like the first line below, or use `%in%` and save ourselves a lot of typing! The `%in%` operator checks whether the left-hand side matches *any* element of the **vector** on the right-hand side. Again, these two lines of code produce the exact same output!

```
filter(AMSsurvey, (type == 'IV' | type == 'Va') & citizen == 'US')
```

```
filter(AMSsurvey, type %in% c('IV','Va') & citizen == 'US')
```

```
##   type    sex citizen count count11
## 1   IV   Male      US    71      89
## 2   IV Female      US    54      55
## 3   Va   Male      US    34      42
## 4   Va Female      US    14      21
```

`arrange()`

That was a lot! I'll try to make this portion quick. We use `arrange()` to reorder our rows, usually using values in our data. We'll use the same `AMSsurvey` data in the `carData` library. Let's start by ordering `AMSsurvey` by the `sex` value so that we can group results by `sex`.

```
arrange(AMSsurvey,sex)
```

```
##      type    sex citizen count count11
## 1  I(Pu) Female      US    35      40
## 2  I(Pr) Female      US    20      22
## 3     II Female      US    47      53
## 4    III Female      US    32      28
## 5     IV Female      US    54      55
## 6     Va Female      US    14      21
## 7  I(Pu) Female  Non-US    29      32
## 8  I(Pr) Female  Non-US    25      26
## 9     II Female  Non-US    50      56
## 10   III Female  Non-US    39      30
## 11    IV Female  Non-US   105     115
## 12    Va Female  Non-US    12      17
## 13 I(Pu)   Male      US   132     148
## 14 I(Pr)   Male      US    87      63
## 15    II   Male      US    96     161
## 16   III   Male      US    47      71
## 17    IV   Male      US    71      89
## 18    Va   Male      US    34      42
## 19 I(Pu)   Male  Non-US   130     136
## 20 I(Pr)   Male  Non-US    79      82
## 21    II   Male  Non-US    89     116
## 22   III   Male  Non-US    53      61
## 23    IV   Male  Non-US   122     153
## 24    Va   Male  Non-US    28      27
```

The default is **ascending** order (A-Z, 1-9), so `Female` shows up first. But now the order of `type` is a mess! I think we would *also* like to see the appropriate `type`s together, so we can just add that to our `arrange()` and sort by it, too!

```r
arrange(AMSsurvey,sex,type)
```

```
##      type    sex citizen count count11
## 1  I(Pr) Female      US    20      22
## 2  I(Pr) Female  Non-US    25      26
## 3  I(Pu) Female      US    35      40
## 4  I(Pu) Female  Non-US    29      32
## 5     II Female      US    47      53
## 6     II Female  Non-US    50      56
## 7    III Female      US    32      28
## 8    III Female  Non-US    39      30
## 9     IV Female      US    54      55
## 10    IV Female  Non-US   105     115
## 11    Va Female      US    14      21
## 12    Va Female  Non-US    12      17
## 13 I(Pr)   Male      US    87      63
## 14 I(Pr)   Male  Non-US    79      82
## 15 I(Pu)   Male      US   132     148
## 16 I(Pu)   Male  Non-US   130     136
## 17    II   Male      US    96     161
## 18    II   Male  Non-US    89     116
## 19   III   Male      US    47      71
## 20   III   Male  Non-US    53      61
## 21    IV   Male      US    71      89
## 22    IV   Male  Non-US   122     153
## 23    Va   Male      US    34      42
## 24    Va   Male  Non-US    28      27
```

But I still like groups `IV` and `Va` best, so I'd like to see them *first* instead of *last*. For this, we use the `desc()` function to sort by **descending** order rather than **ascending**. Like the below!

```r
arrange(AMSsurvey,sex,desc(type))
```

```
##      type    sex citizen count count11
## 1     Va Female      US    14      21
## 2     Va Female  Non-US    12      17
## 3     IV Female      US    54      55
## 4     IV Female  Non-US   105     115
## 5    III Female      US    32      28
## 6    III Female  Non-US    39      30
## 7     II Female      US    47      53
## 8     II Female  Non-US    50      56
## 9  I(Pu) Female      US    35      40
## 10 I(Pu) Female  Non-US    29      32
## 11 I(Pr) Female      US    20      22
## 12 I(Pr) Female  Non-US    25      26
## 13    Va   Male      US    34      42
## 14    Va   Male  Non-US    28      27
## 15    IV   Male      US    71      89
## 16    IV   Male  Non-US   122     153
## 17   III   Male      US    47      71
## 18   III   Male  Non-US    53      61
## 19    II   Male      US    96     161
## 20    II   Male  Non-US    89     116
## 21 I(Pu)   Male      US   132     148
## 22 I(Pu)   Male  Non-US   130     136
## 23 I(Pr)   Male      US    87      63
## 24 I(Pr)   Male  Non-US    79      82
```

Much better. With that we'll draw this walkthrough to a close, sorry it got so long-winded but we've got a lot going on this week!