Intro to Tidyverse and Spain Wrangling

TC & JEPA

2018-10-30

Intro to the Tidyverse

The tidyverse is a group of R packages that are designed to work together and to have a common syntax to make learning each package easy. They all have a similar syntax and use. Check more out here: https://www.tidyverse.org/packages/

Libraries and Data

```r
#If you downloaded tidyverse
#install.packages('tidyverse')
library(tidyverse)
#Loading tidyverse loads packages: ggplot2, tidyr, dplyr, stringr, and more
#Working from the tidyverse package that includes all of them makes updates more consistent (when the p

# In addition, we are going to use the following packages

# Janitor
library(janitor)

# This amazing package helps keep consistancy on the colnames
```

SAU Data

The Sea Around Us (www.seaaroundus.org) is a valuable source of information related to fisheries, from catch Data, to economics and more! Fortunately there's a *R* package to access the Data, you can find the documentation here.

- Today we will only use one function `catchdata`

```r
# Install and load package
# install.packages("seaaroundus")
library(seaaroundus)

# Lets get Spains catch Data, shall we?

## First, we need the id

listregions(region = "fao")

Data <- catchdata(
  region = "fao",
  id = 34, # I happen to know that Spain's id is this
  measure = "tonnage",
  dimension = "country"
  )

# Explore our df
head(Data)

# Using the janitor package to fix names
Data <- clean_names(Data) # can you tell the difference?

# Explore our df
head(Data)

#### Lets also load the complete SAU Data for Spain

Spain <- read.csv("./Data/SAU_Catch_Spain.csv")

head(Spain)
```

Data Manipulation with Dplyr and Tidyr

Despite being separate, these two packages work together as one. Their main function is to manipulate Data frames and keep things "tidy". In some cases you can also make basic Data creation. Both packages follow the same syntax and can use the pipe operator, I normally don't even know which function is from what package so I often just call both.

Plus: Most functions are self explanatory like `select` or `filter`!

Tidyr

Gather and Spread

The `gather` function allows us to convert long Data to short format. This is specifically helpful for plotting since it will allow you to set categories to Data.

Note: The `spread` function is exactly the opposite to `gather` and has the same structure

Basic Structure

Data_Name <- gather(Dataset, key ="Some_Name", value ="Other_name", x:x)

```
# We can do it by number of columns
Tidy_Spain <- gather(Data,
                     key='nation',
                     value='catch',
                     2:12) # I want you to convert from column 2 to 12

head(Tidy_Spain)

# Is everything therer?
unique(Tidy_Spain$nation)
unique(Tidy_Spain$year)
```

```
#OR with column names (Better!):
Tidy_Spain <- gather(Data,
                     key='nation',
                     value='catch',
                     spain:others)

# But what if I want to leave something behaind? Wel...

Tidy_Spain_B <- gather(Data,
                     key='nation',
                     value='catch',
                     spain:latvia)
```

```
# We can do it by number of columns
Spread_Tidy_Spain <- spread(Tidy_Spain,
                     key='nation',
                     value='catch'
                     ) # I want you to convert from column 2 to 12

head(Spread_Tidy_Spain)
```

Unite and Separate

These functions are also used to unite or spread, but focused on dates

Basic Structure

Data_name <- separate(Data, TemporalColumn, c("year", "month", "day"), sep = "-")

Note: The date structure will depend on your Data, as well as the `sep =`

```
#Assuming that our Data set had a dat volumn with year/month/day this is how we would do it...
Separate_Example <- separate(Tidy_Spain,year,c("year", "month", "day"), sep = "-")

#Note: ignore the warning message, is because we don't have a month/day format

head(Separate_Example)

# And then we can also go backwords

Unite_Example <- unite(Separate_Example,"Date",year, month, day, sep = "-")

head(Unite_Example)

#Note that, because month and day are NA's, the new column has them together
```

Dplyr

Arrange

The `arrange`function allows you to, literally, arrange your Data by any value of a column

Basic structure:

New_Table <- arrange(Data, column_to_arrange_by)

Note: If you want to do from Top <- Bottom you can use `desc()` within the function

Note: when doing multiple variables the order is important since it will start with the first one

```r
head(Spain)

head(Spain[5:7], 3)
#Throughout this lesson, we use the head function in this way and select the columns we were just modif

# We can arrange by numeric factors

# In increasing order

arrange(Spain,
        year)

# And dincreasing order

arrange(Spain,
        desc(year) # use desc() for decreasing
        )

# We can do multiple columns:

arrange(Spain,
        desc(year),
        area_name
        )

#You can arrange by characters (A <- Z) using desc()

arrange(Spain,
        desc(area_name),
        year
        )

# Watch it, order mnatters!
arrange(Spain,
        year,
        desc(area_name),
        )
```

Filter

The `filter` function allows you to, literally, filter your Data by any category or number.

Basic structure:

New_Table <- filter(Data, column_to_filter_by == "category")

`filter` operators:

- `a == b` a is equal to `b`
- `a != b` a is not equal to `b`
- `a > b` a is greater than `b`
- `a < b` a is less than `b`
- `a >= b` a is greater than or equal to `b`
- `a <= b` a is less than or equal to `b`
- `a %in% b` a is an element in `b`

- `is.na(a)` values within `a` that are NA
- `!is.na(a)` values within `a` that are not NA

```
# WE can filter by character

filter(Spain,
       year == 2014)

# Note: you can do =>, <= or !=

Filter_Example <- filter(Spain,
                         year <= 1951)

unique(Filter_Example$year)

Filter_Example <- filter(Spain,
                         year >= 2010)

unique(Filter_Example$year)

# you can do multiple characters:

Selection <- c("1952","1968","1970","1994","2002")

Filter_Example <- filter(Spain,
                         year %in% Selection)

unique(Filter_Example$year)

# NOTE: remember that in R there are multiple ways to get to the same result!

#Wait! What if I want to filter by multiple columns!?

Filter_Example <- filter(Spain,
       year %in% Selection,
       tonnes > 1000)

head(Filter_Example)
unique(Filter_Example$year)
min(Filter_Example$tonnes)


#You can also filter by NA
# filter(Data,is.na(column))

# Or... remove NAs
# filter(Data,!is.na(column)) #Clear NA's


#You can also combine the filter with other tidyverse commands like str_detect (Covered later in this l
#For example, return all that have a capital 'C' (note: str_detect is case sensitive)
filter_starts_with_c <- filter(Spain, str_detect(common_name, pattern="lobster"))
#OR return all results where the common name includes the string 'salmon' OR 'crab'.
#We didn't include any spaces in this pattern search because str_detect would 'think' that was part of
filter_salmon <- filter(Spain, str_detect(common_name, pattern="cod|crab"))
```

The Piping operator %>%

Many R packages like `dplyr`, `tidyr` `ggplot2` and `leaflet`, allows you to use the pipe (`%>%`) operator to chain functions together. Chaining code allows you to streamline your workflow and make it easier to read.

When using the `%>%` operator, first specify the Data frame that all following functions will use. For the rest of the chain the Data frame argument can be omitted from the remaining functions.

NOTE: for Mac users the pipe symbol "%>%" shortcut is: command + shit + m. For windows users is: Ctrol + Shift + m

```
# I like to think about it as a series of steps to follow
Juliano_Morning <- Juliano %>%
  wakes_up %>%
  takes_shower %>%
  dresses_up %>%
  coffe %>%
  packs %>%
  coffe %>%
  leaves
```

Group_by* (plus summarize)

The `group_by`function allows you to group your Data by common variable(s) for future (immediate) calculations. This function needs the "pipe operator"

Basic structure:

New_Table <- Data %>% group_by(column_1,column_2…) %>% second_function()

```
#Simple group_by
Group_by_Example <- Spain %>%
  group_by(common_name) %>%
  summarise(n()) #tells you how many rows of each "common_name"" you have

Group_by_Example <- Spain %>%
  group_by(common_name) %>%
  summarise(agg_tonnes=sum(tonnes)) %>%
  arrange(desc(agg_tonnes))
#tells you how many rows of each "common_name"" you have
#We added in to arrange the resulting Data frame with the largest values at the top

Group_by_Example2 <- Spain %>%
  group_by(common_name,catch_type) %>%
  summarise(agg_tonnes=sum(tonnes)) %>%
  arrange(desc(agg_tonnes))
#Can also do two variables at once as ^

head(Group_by_Example, 3)

#Multiple
Group_by_Example2 <- Spain %>%
  group_by(latin_name=scientific_name,gear_type) %>% #You can rename columns within the group_by for on
  summarise(n()) %>% #tells you how many rows of each "scientific_name" you have
  arrange(gear_type)

head(Group_by_Example2)
tail(Group_by_Example2)
```

Mutate

The `mutate` function allows you to create a new column in the Data-set. The new column can have characters or numbers.

Basic structure:

New_Table <- mutate(Data, Name_New_Column = action)

```
#Functions
#Create a new variable in the Dataframe with mutate:
Mutate_Example1 <- mutate(Spain, Log = log(tonnes))
#Mutated variables can be used instantly as in this example.
#We create the variable 'Log' and then call it in the next line.
#Multiple mutate commands can be called one after the other by separating each with a comma.
Mutate_Example1 <- Spain %>%
  filter(year>1990) %>%
  mutate(Log = log(tonnes),
  Log_times_value = Log*landed_value)





head(Mutate_Example1[13:16], 3)

#In Spain calculations (per row)
Mutate_Example2 <- mutate(Spain, Price_per_Ton = (landed_value/tonnes))

head(Mutate_Example2[13:15], 3)

#Or characters...
Mutate_Example3 <- mutate(Spain, Pais = "Espana")
head(Mutate_Example3[13:15], 3)
```

Replace

The 'replace' function can be used with mutate to replace values within a column based on certain characteristics.

Basic Function

mutate(Column1 = replace(column1, Column1==Value, ReplacementValue)) OR mutate(Column1 = replace(column1, Column2==Value2, ReplacementValue))

```
Replace_Example <- mutate(Mutate_Example3, Country = paste("In",year,area_name,"harvested",
                                                        round(tonnes,2), "tonnes of", common_name))
head(Replace_Example)
tail(Replace_Example)
```

Rename

The `rename` function is another "self explanatory" it allows you to rename the columns

Basic structure:

New_Table <- rename(Data,New_Name = Old_Name)

```
Rename_Example <- rename(Spain, Weight = tonnes)
Rename_Example <- mutate(Spain, Weight = tonnes)
```

Select

The `select`function is one of those "of-course it does that" function because it allows you to, wait for it… SELECT any column you want.

Basic structure:

New_Table <- select(Data,number or name of column)

Note: Re-ordering of values happens here!

```
#Select by column number
Select_Example1 <- select(Spain, 6)
Select_Example1 <- select(Spain, latin_name= scientific_name, tonnes)

head(Select_Example1,3)

#Select by multiple column numbers
Select_Example2 <- select(Spain, 4,5,6,7)

head(Select_Example2, 3)

# You can also do (4:7) and even (4:6,15)

#Select by name
Select_Example3 <- select(Spain, area_name,year,scientific_name,tonnes)
```

```r
head(Select_Example3, 3)

#And similar to using column numbers you can select columns within a range by name
Select_Example4 <- select(Spain, area_name:tonnes)

#Note: This includes the named columns.

# You can drop columns from a Dataframe

Select_Example5 <- select(Select_Example3, -area_name,year)

head(Select_Example5, 3)

#Note, you can also drop using -

#And you can also re-order your columns!

Select_Example6 <- select(Select_Example3, scientific_name,year,tonnes,area_name)

head(Select_Example6, 3)

#And you don't have to write everything

Select_Example7 <- select(Select_Example5, scientific_name,
                          everything())

head(Select_Example7, 3)
```

###slice The `slice`function works like the `select`function but for rows. So, if you want to extract an specific row, a set of rows, or a range between values, use slice! This works similar to filter in that it keeps all columns of the Dataframe, but only certain rows. You can choose to use this or filter depending on if you know the row numbers, or if you know the values within the rows you want.

Basic Structure

New_Data <- slice(Old_Data, number)

```r
#Select by row number
Slice_Example1 <- slice(Spain, 3948)

Slice_Example1

#Select by multiple rows
Slice_Example2 <- slice(Spain, 1000:3948)

head(Slice_Example2, 3)
```

###pull

If you want to get all the values from a column you can pull those values to a new variable. Often helpful at the end of a pipe operator. You can continue to work with these values (in the pipe, or otherwise) after you have used the pull function

```r
pull_vales <- Spain %>%
  pull(scientific_name)
```

```
pull_vales <- Spain %>%
  pull(scientific_name) %>%
  unique()


mean_catch <- Spain %>%
  pull(tonnes) %>%
  mean() # not it saves it as a value not a df
```

Cleaning Data (stringr package)

Often, when you get Data (or have finished entering it yourself), you need to check the Data for unlikely values, weird things that come up or other problems. If you are working with strings (text), then you can modify it in a consistent way with stringr.

```
str_example <- c("shark", "Ray", "fish", "Whale", "FISH", "fish")

#Can use to modify strings:
str_to_upper(str_example) #If you want to yell at people
str_to_lower(str_example) #If you're whispering
str_to_title(str_example) #If you feel your text is important

#Many work with searching for a pattern with str_VERB(x, pattern)
sharks <- str_detect(str_example, pattern="shark")
#Detect returns a list of TRUE or FALSE values where it find the value of interest


#How many sharks are there? str_count
str_count(str_example, pattern="shark")

#Replace pesky values with values you want instead
str_replace(str_example, pattern="Whale", "non-fish")

str_split(str_example, pattern="i")

num_str_example <- c("1,000", "937.1", "1,000,000")
str_replace(num_str_example, pattern=",", replacement="") # replaces only first encounter
num_str_example <- str_replace(num_str_example, pattern=",", replacement="") # replaces only first enco

as.numeric(num_str_example)
str_replace_all(num_str_example, pattern=",", replacement="")


str_example <- c("fish, shark", "fish, Ray", "fish, tuna", "mammal, Whale", "fish, FISH", "fish, herrin
str_split(str_example, pattern=",")

df <- data.frame(taxon=str_example)
df$taxon <- as.character(df$taxon)
df$split<- str_split(df$taxon, pattern=",")

df$split <- str_replace(df$taxon, pattern="fish, ", replacement="")
```

```
df
```

Joining Data with dplyr

The "bind" family

These functions will help us bind two or more Data-sets in one depending on different variables.

bind_cols

The `bind_cols` function allows us to bind two Data-sets by column.

Basic Structure

New_Data <- bind_cols(Data1, Data2)

```
#Lets just asume that we have two different Data sets
Spain1 <- select(Spain, 1)
Spain2 <- select(Spain, 2)

#Now we bind the columns together
Bind_Cols_1 <- bind_cols(Spain1,Spain2)

head(Bind_Cols_1, 3)
```

bind_rows

The `bind_rows` function is a sister-function of bind_cols but for binding rows.

Basic Structure

New_Data <- bind_rows(Data1, Data2)

```
#Lets just assume that we have two different Data sets
Spain1 <- slice(Spain, 1:3)
Spain2 <- slice(Spain, 10800:10802)

#Now we bind the columns together
Bind_Row_1 <- bind_cols(Spain1,Spain2)

head(Bind_Row_1, 6)
```

The "join" family

This is similar to 'merge' in base R. However, there are more options than merge with the types of joins you can do to combine two Datasets, and for large Datasets it can be much much faster. This 'join' function is heavily based on SQL style joins. Here is a simple visual explanation of what they all mean:

anti_join

This function will allow you to select all variables that are **not** the same within two Data-sets. Note, both Data-sets must have at least one similar category/column. This can be helpful to find values that aren't matching between your Datasets (i.e., for testing).

Basic Structure

Data_Name <- anti_join(Dataset1,Dataset2, by="similar category")

Lets us know what variables from one Data-set are not present in some other Data-set

```r
Ireland <- Spain %>%
  filter(area_name == "Ireland")

Senegal <- Spain %>%
  filter(area_name == "Senegal")

#Lets asume we want to know how many species are fished in Ireland and not in Senegal

Diff_Species <- anti_join(Ireland, Senegal,
                          by="scientific_name") %>%
  pull(scientific_name) %>%
  unique()

#You can also do it by more than one variable
Diff_Species2 <- anti_join(Ireland, Senegal, by=c("scientific_name","reporting_status"))
```

semi_join

This function does the opposite as the anti join, letting you select those variables shared by two Data-sets.

Basic Structure

Data_Name <- semi_join(Dataset1, Dataset2, by="similar category")

```r
#Lets now asume we want to know the species are fished in Ireland and in Senegal
Similar_Species <- semi_join(Ireland, Senegal, by="scientific_name") %>%
  pull(scientific_name) %>%
  unique()

#Note: just like anti_join, you can do it for more than one variable
```

Left_join

Basic Structure

Spain_Name <- left_join(Dataset1, Dataset2, by="similar category")

```r
#Now we want to know how many species are fished in BOTH Spain and the continental US
Spain_Price <- Spain %>%
  group_by(scientific_name) %>%
  summarize(price=sum(landed_value))

Left_Species <- left_join(Spain, Spain_Price)
```

Right_join

Basic Structure

Data_Name <- right_join(Dataset1, Dataset2, by="similar category")

```
Right_Species <- right_join(Spain, Spain_Price, by="scientific_name")

#Note: just like anti_join, you can do it for more than one variable
```

Combo!

So now that we have all of the functions, we can do something like this…

```
Pipie_Example <- Spain %>%
  filter(year >= 2000) %>% #Lets filter the year above 2000
  select(area_name,scientific_name,
         tonnes,year) %>% #We only care about these Spain
  group_by(scientific_name,year) %>%
  summarise(Mean = mean(tonnes),
            SD = sd(tonnes),
            N = n()) %>% #Give me the mean and sd of each species each year
  mutate(Round_Mean = round(Mean,2), #create a log version of mean
         Round_SD = round(SD,2)) %>% #... and the sd
  transmute(Log_Mean = log(Round_Mean,2),
            Log_SD = log(Round_SD,2)) %>%
  left_join(Spain_Price,
            by="scientific_name")
```

One of the beauties of `tidyverse` is that you can mix several packages in one code like this graph:

```
Spain %>%
  filter(year >= 2000) %>% #Lets filter the year above 2000
  select(area_name,scientific_name,tonnes,year) %>% #We only care about these Spain
  group_by(scientific_name,year) %>%
  summarise(Mean = mean(tonnes),
            SD = sd(tonnes),
            N = n()) %>% #Give me the mean and sd of each species each year
  mutate(Round_Mean = round(Mean,2), #create a log version of mean
         Round_SD = round(SD,2)) %>% # and the sd
  transmute(Log_Mean = log(Round_Mean,2),
            Log_SD = log(Round_SD,2)) %>%
  ggplot(., #It tells ggplot2 to use the Spain you are piping
         aes(
           x=Log_Mean,
           y=Log_SD
         )) +
  geom_point()

#Pipe_Example
```