data_manip_part_II

March 28, 2025

1 Data Manipulation with Tidyverse, Part II

In the Data Foundations module, we covered Part I of Data Manipulation with the Tidyverse and began learning how to clean datasets using packages in the Tidyverse. In this lecture, we will cover a few more advanced data manipulation techniques.

Remember, **data manipulation is essential in environmental data science** because real-world data often isn't in a format that fits the models we want to use. By transforming the data, we can use it to build accurate and precise models that approximate how the world works. Without data manipulation, we may not be able to build the most accurate model.

In part I, we used the following function from the `dplyr` package

- `mutate()`
- `if_else()`
- `filter()`
- `select()`
- `group_by()`
- `summarise()`

In this part II lecture, we are going to learn about `join()` functions in the `dplyr` package, as well as `pivot()` functions in the `tidyr` package

- `left_join()`
- `pivot_longer()`
- `pivot_wider()`

First things first, let's load our packages

[8]:
```r
# Load Libraries
library(dplyr)
library(tidyr)
```

2 join() functions

In environmental data science, we often need to combine information from multiple datasets—for example, species data from one source and habitat data from another. We use `join()` functions to do this.

The most common is `left_join()`, but you may also see `right_join()`, `inner_join()`, and `full_join()`.

A `left_join()` adds variables from a second dataset to your original dataset based on a shared key. This is what I almost always use.

The most common is `left_join()`, but you may also see `right_join()`, `inner_join()`, and `full_join()`.

- A `left_join()` adds variables from a second dataset to your original dataset based on a shared key, keeping all rows from the original dataset.
- A `right_join()` is the opposite of `left_join()`, keeping all rows from the second dataset and adding variables from the original dataset where keys match.
- An `inner_join()` keeps only the rows where there is a match between the keys in both datasets.
- A `full_join()` keeps all rows from both datasets, filling in `NA` for missing matches.

Let's create a dataset of observed species and the habitat type they were found in:

[9]:
```
# ------------------------------------------------------------------------
# make a data frame of species observations
# ------------------------------------------------------------------------
species_obs <- data.frame(
  species_id = 1:5,
  species = c("Red Fox", "Black Bear", "Gray Wolf", "Moose", "Beaver"),
  habitat = c("Forest", "Mountain", "Forest", "Wetland", "Wetland"),
  count = c(3, 2, 5, 1, 7)
)

species_obs
```

A data.frame: 5 x 4

species_id <int>	species <chr>	habitat <chr>	count <dbl>
1	Red Fox	Forest	3
2	Black Bear	Mountain	2
3	Gray Wolf	Forest	5
4	Moose	Wetland	1
5	Beaver	Wetland	7

Now, let's create a second dataset with additional information about each habitat:

[10]:
```
# ------------------------------------------------------------------------
# create habitat dataset
# ------------------------------------------------------------------------
habitat_info <- data.frame(
  habitat = c("Forest", "Mountain", "Wetland"),
  protection_status = c("Protected", "Unprotected", "Protected")
)

# ------------------------------------------------------------------------
# left_join to add habitat information to species observations
# ------------------------------------------------------------------------
obs_joined <- left_join(x = species_obs, y = habitat_info, by = "habitat")
```

```
obs_joined
```

A data.frame: 5 x 5

species_id	species	habitat	count	protection_status
<int>	<chr>	<chr>	<dbl>	<chr>
1	Red Fox	Forest	3	Protected
2	Black Bear	Mountain	2	Unprotected
3	Gray Wolf	Forest	5	Protected
4	Moose	Wetland	1	Protected
5	Beaver	Wetland	7	Protected

3 Manipulating and Cleaning with `tidyr`

Just like dplyr, the tidyr package provides powerful tools for transforming data. Two of the most essential functions to know are: - pivot_longer() - pivot_wider()

3.1 `pivot_longer()`

Many models – especially regression models – work best with data in a long format, where repeated measurements or categories are stacked in a single column rather than spread across multiple ones.

Example

- Suppose you have a dataset with 12 columns for each month.

- If you want to include month as a variable in your regression, you need to reshape the data so that month values are in a single column.

- Using `pivot_longer()`, you can convert those 12 month columns into two columns: one for month and one for the corresponding value.

- After this transformation, your dataset is "long by month," which is typically much more model-friendly.

[11]:
```r
# -------------------------------------------------------------------------
# Create data set
# -------------------------------------------------------------------------
myTemps <- data.frame(
  location = c("Forest", "Desert"),
  Jan = c(5, 20),
  Feb = c(6, 22),
  Mar = c(10, 25)
)
myTemps

# -------------------------------------------------------------------------
# pivot longer
# -------------------------------------------------------------------------
myTemps_long <- myTemps %>%
  pivot_longer(
  cols = Jan:Mar, # the columns we want to pivot
  names_to = "Month", # the new variable where the names of columns will be␣
 ↪assigned
  values_to = "Temperature" # the new variable where the values will be␣
 ↪assigned
)
myTemps_long
```

A data.frame: 2 x 4

location <chr>	Jan <dbl>	Feb <dbl>	Mar <dbl>
Forest	5	6	10
Desert	20	22	25

A tibble: 6 x 3

location <chr>	Month <chr>	Temperature <dbl>
Forest	Jan	5
Forest	Feb	6
Forest	Mar	10
Desert	Jan	20
Desert	Feb	22
Desert	Mar	25

3.1.1 `pivot_wider()`

I typically use `pivot_wider()` to reshape data from a **long** format to a **wide** format when I want to **reduce granularity**—for example, when aggregating data. While we often prefer more detailed (granular) data, sometimes we need to align with another dataset that is **less granular** in order to merge them properly.

Example: Suppose you have daily rainfall data (long by day), but you want to join it with a dataset that is only long by **station and week**. Therefore you want to get the total weekly rainfall, instead of having it as daily rainfall. We can do this by using `pivot_wider()`.

[12]:
```
# ------------------------------------------------------------------------
# Create long data
# ------------------------------------------------------------------------
myRain <- data.frame(
  station = c("A", "A", "A",
              "B", "B", "B"),
  day = c("Monday", "Tuesday", "Wednesday",
          "Monday", "Tuesday", "Wednesday"),
  rainfall_mm = c(5, 10, 3,
                  0, 0, 12)
)

myRain

# ------------------------------------------------------------------------
# pivot so it's wide by station
# ------------------------------------------------------------------------

myRain_wide <- myRain %>%
  pivot_wider(names_from = day, values_from = rainfall_mm)

  myRain_wide

# ------------------------------------------------------------------------
# get weekly total rainfall
# ------------------------------------------------------------------------

myRain_weekly <- myRain_wide %>%
  mutate(weekly_rain = Monday + Tuesday + Wednesday)

myRain_weekly
```

A data.frame: 6 x 3

station <chr>	day <chr>	rainfall_mm <dbl>
A	Monday	5
A	Tuesday	10
A	Wednesday	3
B	Monday	0
B	Tuesday	0
B	Wednesday	12

A tibble: 2 x 4

station <chr>	Monday <dbl>	Tuesday <dbl>	Wednesday <dbl>
A	5	10	3
B	0	0	12

A tibble: 2 x 5

station <chr>	Monday <dbl>	Tuesday <dbl>	Wednesday <dbl>	weekly_rain <dbl>
A	5	10	3	18
B	0	0	12	12

4 Summary

- `left_join(x, y, by = "key")`: Add columns from `y` to `x` by matching rows using a shared column (called a **key**).
- `pivot_longer(cols, names_to, values_to)`: Turn multiple columns into **key-value pairs**, where the original column names become values in a new "key" column, and the original cell values become values in a new "value" column.

  What's a key-value pair?
  A **key** is a label that identifies what kind of data a value represents, and the **value** is the data itself.
  For example, if you have columns named `Jan`, `Feb`, and `Mar`, each with temperature values, `pivot_longer()` will turn those columns into:

location	Month	Temperature
Forest	Jan	5
Forest	Feb	6
Forest	Mar	10

Here, `Month` is the **key**, and `Temperature` is the **value**.

- `pivot_wider(names_from, values_from)`: Spread key-value pairs back into multiple columns.

  Example of key-value pairs for `pivot_wider()`:
  Suppose you have a dataset in long format like this:

station	day	rainfall_mm
A	Monday	5
A	Tuesday	10
A	Wednesday	3
B	Monday	0
B	Tuesday	0
B	Wednesday	12

  Using `pivot_wider(names_from = day, values_from = rainfall_mm)`, this will transform into:

station	Monday	Tuesday	Wednesday
A	5	10	3
B	0	0	12

  Here, `day` is the **key**, and `rainfall_mm` is the **value**.