Summary Sheet

AUTHOR
Julia Gallucci

Class 4

Pivot

- Refers to the process of transforming data from a “long” format to a “wide” format or vice versa. This reshaping operation is used to restructure the data in a way that makes it easier to analyze or present.

pivot_longer:

`pivot_longer()` : used to transform data from a wide format to a long format. It “lengthens” data, increasing the number of rows and decreasing the number of columns. In the long format, each row represents a unique observation or data point, and different variables are stored in separate columns. This format is typically used when each row contains a single observation and there are multiple variables associated with each observation. The long format is often more suitable for data analysis and manipulation.

```r
student_scores <-
  tibble(
  Student = c("Alice", "Julia", "Max", "Alice", "Julia", "Max"),
  Subject = c("Math", "Math", "Math", "Science", "Science", "Science"),
  Score = c(87, 77, 90, 85, 65, 95)
)
student_scores
```

```
# A tibble: 6 × 3
  Student Subject Score
  <chr>   <chr>   <dbl>
1 Alice   Math       87
2 Julia   Math       77
3 Max     Math       90
4 Alice   Science    85
5 Julia   Science    65
6 Max     Science    95
```

In this example, the long format has three columns: **`Student`**, **`Subject`**, and **`Score`**. Each row represents a student’s score in a particular subject.

Spread vs pivot_wider:

The `spread()` function works in the same way as `pivot_wider()`, widening the data, increasing the number of columns while decreasing the number of rows. This format is useful when summarizing or presenting data in a more compact and concise manner. Some differences:

spread(data, key, value)	**pivot_wider(data, names_from, values_from, <optional> values_fill)**
Converts a data frame from long to wide format based on key-value pairs.	Converts a data frame from long to wide format based on key-value pairs. It offers more flexibility and options compared to `spread()`, for ex. it allows for additional customization, such as specifying the default values for missing cells and handling duplicate values.
Arguments: • `data`: The input data frame. • `key`: The column that contains the keys used to create new columns. • `value`: The column that contains the values to be spread into the new columns.	Arguments: • `data`: The input data frame. • `names_from`: The column that contains the values to become the new column names. • `values_from`: The column that contains the values to be spread into the new columns. • `values_fill`: (optional) Specifies the default value for cells with missing values.

`pivot_wider()` is generally recommended for more advanced and flexible reshaping operations.

```
student_scores_wide <-
  student_scores %>%
  pivot_wider(names_from = Subject, values_from = Score)
student_scores_wide
```

```
# A tibble: 3 × 3
  Student  Math Science
  <chr>   <dbl>   <dbl>
1 Alice      87      85
2 Julia      77      65
3 Max        90      95
```

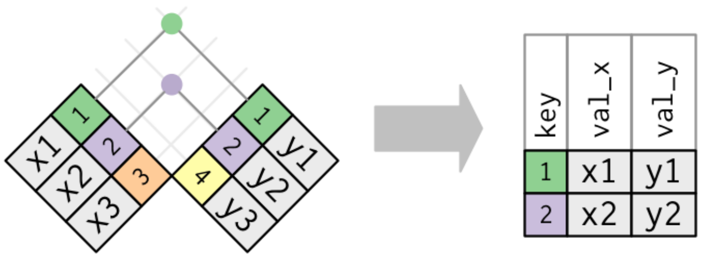
In this example, we specify `names_from = Subject` to indicate that the unique values in the "Subject" column will become the new column names. Similarly, we specify `values_from = Score` to indicate that the values in the "Score" column will populate the new columns.

The resulting **`student_scores_wide`** tibble has three columns: "Student", "Math", and "Science". Each row represents a student, and their scores in different subjects are displayed in the respective columns.

Join

used to combine or merge multiple data frames based on common variables or keys. There are several types of joins available, including inner join, left join, right join, and full join. The join operation brings together rows from different data frames that have matching values in the specified variables. **Keys** are the columns or variables **used to match and merge the data**

`inner_join()` returns **only the rows with matching values in both data frames**. It combines the rows from two data frames based on a common variable or key, excluding any rows that don't have a match in the other data frame.

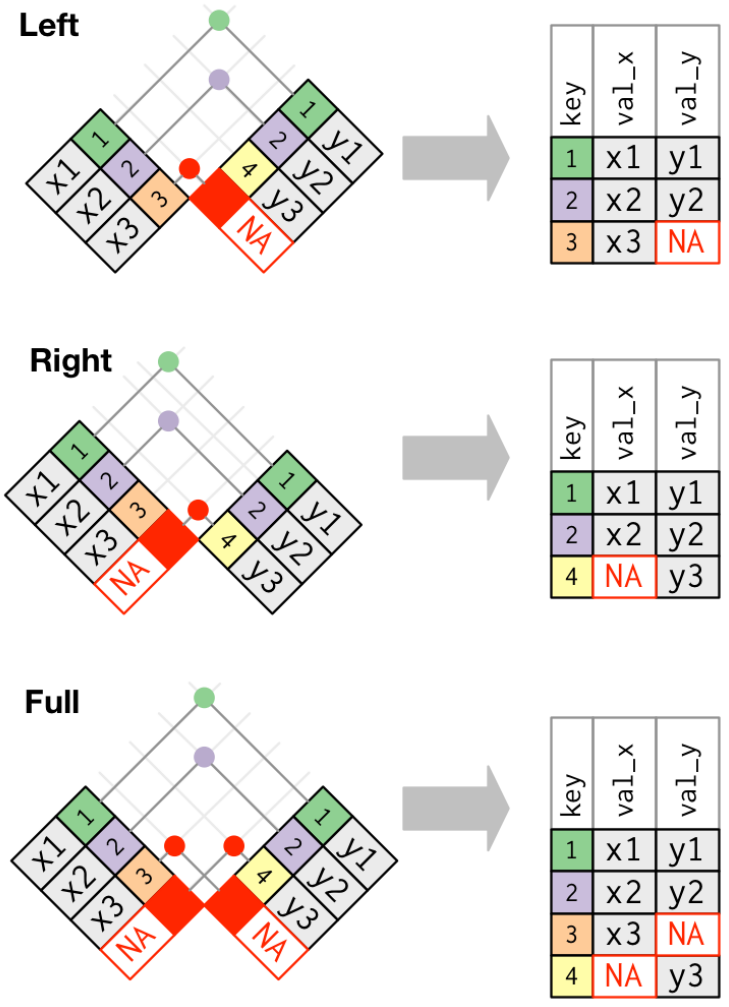


https://r4ds.had.co.nz/relational-data.html

`left_join()` returns **all rows from the left (or first) data fram**e and the matching rows from the right (or second) data frame based on a common variable or key. If there is **no match** for a row in the left data frame, the corresponding values from the **right data frame will be filled with NAs**.

`right_join()` returns **all rows from the right (or second) data frame** and the matching rows from the left (or first) data frame based on a common variable or key. If there is **no match** for a row in the right data frame, the corresponding values from the **left data frame will be filled with NAs.**

`full_join()` returns **all rows from both the left (or first) and right (or second) data frames**, combining them based on a common variable or key. If there is **no match for a row in either** data frame, the corresponding values from the other data frame will be **filled with NAs.**



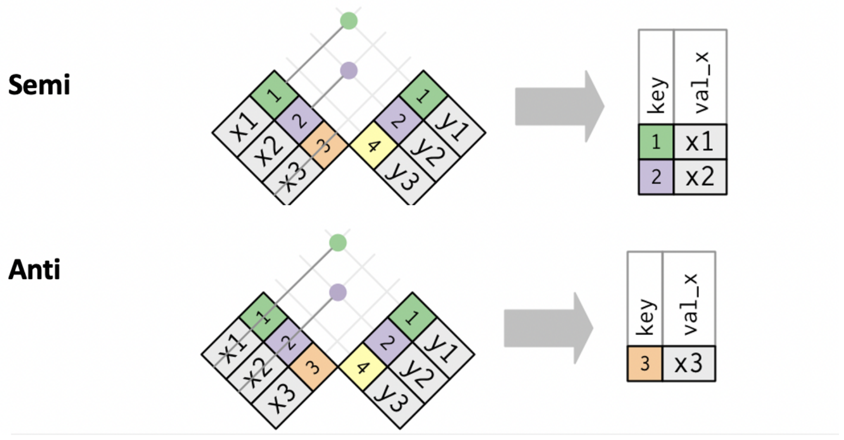
https://r4ds.had.co.nz/relational-data.html

Filter Joins

Allow you to filter one data frame based on matching values in another data frame. Filter joins are useful when you want to keep only the rows in one data frame that have matches in another data frame, effectively filtering the data based on the matching criteria.

`semi_join()` returns only the rows from the left (or first) data frame that have matching values in the right (or second) data frame.

`anti_join()` returns only the rows from the left (or first) data frame that do not have any matching values in the right (or second) data frame. It is essentially the *opposite of a regular join, as it keeps only the non-matching rows from the left data frame.*



https://r4ds.had.co.nz/relational-data.html

data.table

from data.table library, data tables are faster and more memory efficient than data.frames.

- `sapply(data_table, class)` can get info able class of data in your data table
- `haskey(data_table)` to check if your data table has a key
- `setkey(data_table, column_to_set)` to set a specified column in your table as a key
- `data_table[column_name with a certain condition]` to subset rows
- `data_table[, column_name]` to select columns
- `data_table[, list(column_name)]` to return column back as a data table

A **key** refers to one or more columns that are sorted in ascending order. The key helps in efficient indexing and sorting of the data. Here's an example:

```
dt <- data.table(ID = c(1, 2, 3),
                 Name = c("Alice", "Julia", "Max"),
                 Age = c(30, 25, 35))

# Set the key on the "ID" column
setkey(dt, ID)
dt
```

```
   ID  Name Age
1:  1 Alice  30
2:  2 Julia  25
3:  3   Max  35
```

After setting the key, the **data.table** is physically sorted by the "ID" column in ascending order. Once a key is set, it allows us to perform various operations that take advantage of the sorted order, leading to faster performance. For example, using the key, you can perform a **binary search** to efficiently locate rows based on the key column(s), as shown below.

```r
# Perform binary search using the key
search_id <- 3
result <- dt[ID == search_id]
result
```

```
   ID Name Age
1:  3  Max  35
```

Loading different types of data

- CSV `readr::read_csv()`
- XL `readxl::read_excel()`
- SPSS `haven::read_sav()`
- Stata `haven::read_dta()`
- SAS `haven::read_sas()`

can also give specifications, like skip lines using the `skip =` argument, change separator using the `sep =` argument, data does not have column names using the `col_names = FALSE` argument

to write data, use the following format:

`write.csv(variable_name, "what_you_want_to_save_your_output_file_as.csv")`

data.table computations

Here are some common computations that can be performed using data.table:

- **Subsetting:** You can subset data.table based on certain conditions using the square bracket [] notation. For example, to select rows where a certain column value is greater than a threshold, you can use:

  ```r
  dt <- data.table(X = 1:10, Y = letters[1:10])
  dt
  ```

  ```
       X Y
   1:  1 a
   2:  2 b
   3:  3 c
   4:  4 d
   5:  5 e
   6:  6 f
   7:  7 g
   8:  8 h
   9:  9 i
  10: 10 j
  ```

  ```r
  subset <- dt[X > 5]
  subset
  ```

```
    X Y
1:  6 f
2:  7 g
3:  8 h
4:  9 i
5: 10 j
```

- **Aggregation:** you can compute summary statistics like mean, sum, count, etc., grouped by one or more columns. For example, to calculate the average value of column “x” for each unique value in column “y,” and count the number of occurences of each group, you can use the `by` argument:

  ```
  dt <- data.table(x = 1:10, y = rep(c("a","b"),5))
  dt
  ```

  ```
       x y
   1:  1 a
   2:  2 b
   3:  3 a
   4:  4 b
   5:  5 a
   6:  6 b
   7:  7 a
   8:  8 b
   9:  9 a
  10: 10 b
  ```

  ```
  summary <- dt[, .(mean_x = mean(x), count = .N), by = y]
  summary
  ```

  ```
     y mean_x count
  1: a      5     5
  2: b      6     5
  ```

  You can also group by a *condition* rather than a column! For example, lets say we wanted to calculate the average value of column “x” for each unique value in column “y,” and count the number of occurrences for each group based on instances where “x” was greater/less than 4.

  ```
  dt[, .(mean_x = mean(x), count = .N), .(x >4)]
  ```

  ```
     x > 4 mean_x count
  1: FALSE    2.5     4
  2:  TRUE    7.5     6
  ```

- **Sorting:** You can sort a data.table based on one or more columns using the `order()` function. For example, to sort a data.table in ascending order based on column “x”

  ```
  sorted <- dt[order(x)]
  sorted
  ```

```
     x y
 1:  1 a
 2:  2 b
 3:  3 a
 4:  4 b
 5:  5 a
 6:  6 b
 7:  7 a
 8:  8 b
 9:  9 a
10: 10 b
```

can perform computations on multiple columns in a table using the lapply() function. For example,

```
dt_2 <- data.table(X = rep(c("a","b"),5), Y = 11:20, Z = 21:30)
dt_2
```

```
    X  Y  Z
 1: a 11 21
 2: b 12 22
 3: a 13 23
 4: b 14 24
 5: a 15 25
 6: b 16 26
 7: a 17 27
 8: b 18 28
 9: a 19 29
10: b 20 30
```

```
dt_2[, lapply(.SD, mean), by =X]
```

```
   X  Y  Z
1: a 15 25
2: b 16 26
```

Here, the `lapply(.SD, mean)` applies the `mean()` function to each column in the Subset of Data (`.SD`) for each group defined by the “X” column. The result is returned as a new data.table with the columns “X”, “Y”, and “Z” and their respective means for each group.

Functions

Functions provide a more robust and versatile approach to automating repetitive tasks compared to copy-and-pasting. Creating a function offers three significant benefits over copy-and-paste:

1. By assigning a descriptive name to a function, you enhance the comprehensibility of your code, making it easier for others to understand its purpose.

2. When requirements evolve, modifying a function in a single location suffices, instead of having to update multiple instances scattered throughout the code.

3. Copy-and-paste errors, such as forgetting to update a variable name in one instance but not in another, are eliminated when using functions, ensuring greater accuracy and reliability.

Basic function structure

`function_name <- function(arguments){`

`some sort of operation you wish to perform`

`return(output)`

`}`

3 main components:

1. name of function
2. inputs needed (also known as arguments)
3. body of the function that performs the operation

For example, lets say we wanted to create a function that calculates the square of a number.

```r
square <- function(num) {
  result <- num^2
  return(result)
}
square(2)
```

```
[1] 4
```

Loops

Iteration is an additional technique that aids in minimizing redundancy by allowing you to perform identical operations on multiple inputs. It serves as a valuable tool when you encounter scenarios requiring repetitive actions on various elements.

Basic loop structure

For loops

`for(sequence to iterate over){`

`<code to execute>`

`}`

While loops

`while(iterator condition){`

`<code to execute>`

```
}
```

While loop	For loop
repeatedly executes a block of code as long as a specified condition remains true.	repeatedly executes a block of code for a **specific number of iterations.**
The **number of iterations is not predetermined, and the loop continues until the condition is no longer satisfied**.	It takes on each value in the specified sequence, and the code block is executed for each iteration

Example of while loop:

```
x <- 1
while (x <= 3) {
  print(x)
  x <- x + 1
}
```

```
[1] 1
[1] 2
[1] 3
```

Example of for loop:

```
for (x in 1:3) {
  print(x)
}
```

```
[1] 1
[1] 2
[1] 3
```

To save the output of a loop into a vector, you can initialize an empty vector before the loop and then append the desired values to the vector within each iteration of the loop

```
# Initialize empty vector
output_vector <- c()

# Perform the loop and save output into the vector
for (i in 1:3) {
  output <- i*2
  output_vector <- c(output_vector, output) #append to vector
}
output_vector
```

```
[1] 2 4 6
```

If/else

If/else statements allows you to execute different blocks of code based on a specific condition.

Basic condition structure

`if(condition){`

`<code to execute if condition is TRUE>`

`}``else{`

`<code to execute if condition is FALSE>`

`}`

can combine multiple conditions:

Operator	Use
||	condition 1 OR condition 2
&&	condition 1 AND condition 2

`any()` to find out if any of the conditions are true

`all()` to find out if all the conditions are true

`ifelse()` function writes out the condition in a single line

for example, if we wanted to characterize numbers as odd or even

```
x <- 1:5
if_else(x%%2 == 0, "Even", "Odd")
```

```
[1] "Odd"  "Even" "Odd"  "Even" "Odd"
```

```
#if number is divisable by 2, return odd, else return even
```

`case_when()` function to write out multiple conditions

for example, if we wanted to assign letter grades based on numeric scores.

```
grade <- c(90,70,60,50,40)
case_when(
  grade >= 90 ~ "A",
  grade >= 70 ~ "B",
  grade >= 60 ~ "C",
  grade >= 50 ~ "D",
  TRUE ~ "Fail"
)
```

```
[1] "A"    "B"    "C"    "D"    "Fail"
```

Here,

- If the grade is greater than or equal to 90, it is assigned the letter grade "A".
- If the grade is between 70 and 89 (inclusive), it is assigned the letter grade "B".
- If the grade is between 60 and 69 (inclusive), it is assigned the letter grade "C".
- If the grade is between 50 and 59 (inclusive), it is assigned the letter grade "D".
- For any other grade, the condition **`TRUE`** is used as a catch-all, and the letter grade "Fail" is assigned.

Map

The **`map()`** family of functions from the `purrr library` allows you to apply a function to each element of a list or vector, and return the results in a new list.

- `map()` for lists. Note, `lapply()` is identical in functionality to `map()`
- `map_lgl()` for logical vectors
- `map_int()` for integer vectors
- `map_dbl()` for double vectors
- `map_chr()` for character vectors

Example of usage:

```
# Vector of numbers 1 to 5
numbers <- 1:5

# Square each number using map_dbl()
squared_numbers <- map_dbl(numbers, ~ .x^2)

# Print the squared numbers
print(squared_numbers)
```

```
[1]  1  4  9 16 25
```

Here, the **`map_dbl()`** function takes a numeric (double) vector to iterate over and apply the function to apply to each element. In this example, we use an anonymous function (**`~ .x^2`**) to square each element.

Note: *An anonymous function*, is a function defined without a specific name. Instead, it is defined inline within the code where it is needed. In the function above, we used the **`~`** (tilde) symbol followed by the expression **`.x^2`**. The **`.x`** represents the input parameter of the function.

We can also iterate over two lists simultaneously and apply a function to each corresponding pair of elements.

Example:

```
x <- list(100, 200, 300)
y <- list(10, 20, 30)
map2(x, y, ~ .x - .y)
```

```
[[1]]
[1] 90

[[2]]
[1] 180

[[3]]
[1] 270
```

In this example, we have two lists, `x` and `y`, each containing three elements. We want to calculate the difference between the corresponding elements of `x` and `y`.

Here, the anonymous function (`~ .x - .y`) takes two arguments, `.x` and `.y`, which represent the corresponding elements from `x` and `y` in each iteration. The function subtracts `.y` from `.x` to calculate the difference.