Tips on the `dplyr` package.

The `dplyr` package in R is part of the tidyverse collection, and it's primarily used for data manipulation. It provides a set of tools for efficiently performing common data wrangling tasks, especially on data frames and tibbles. The key idea behind `dplyr` is that it allows for writing readable and expressive code using simple, consistent verbs that represent common data operations.

Here are the **main verbs** in `dplyr` and the logic behind each:

1. `filter()` : Subsetting rows based on conditions

- **Purpose**: It filters rows in a data frame by keeping only those that meet specified logical conditions.
- **Logic**: `filter()` works on rows and returns a new data frame with only the rows that satisfy the condition.

```
df_filtered <- filter(df, condition)
```

- **Example**: Keep rows where `age > 30`.

  ```
  df_filtered <- df %>%
    filter(age > 30)
  ```

2. `select()` : Selecting columns

- **Purpose**: Selects a subset of columns from a data frame.
- **Logic**: You provide the names of the columns you want to keep, and `select()` returns a new data frame with only those columns.

```
df_selected <- select(df, column1, column2)
```

- **Example**: Select only the `name` and `age` columns.

  ```
  df_selected <- df %>%
    select(name, age)
  ```

3. `mutate()` : Creating new columns or modifying existing ones

- **Purpose**: Adds new columns or modifies existing ones by applying functions to the data.
- **Logic**: `mutate()` operates column-wise. You provide the transformations (e.g., math operations, function applications), and `mutate()` returns the data frame with the new or changed columns.

```
df_mutated <- mutate(df, new_column = existing_column * 2)
```

- **Example**: Add a new column `age_in_5_years` that adds 5 to the current `age` column.

```
df_mutated <- df %>%
  mutate(age_in_5_years = age + 5)
```

4. `summarise()` : Summarizing data

- **Purpose**: Reduces multiple rows down to a single summary statistic, such as a mean or a total.
- **Logic**: `summarise()` works by collapsing data based on the computation you specify (e.g., mean, sum, max).

```
df_summary <- summarise(df, summary_column = mean(column1))
```

- **Example**: Calculate the average `age`.

```
df_summary <- df %>%
  summarise(avg_age = mean(age))
```

5. `group_by()` : Grouping data for summary purposes

- **Purpose**: Groups rows by one or more columns, which is useful for performing grouped operations (e.g., calculating the mean for each group).
- **Logic**: `group_by()` doesn't change the data by itself but sets the stage for other operations (like `summarise()`) to act on groups.

```
df_grouped <- group_by(df, group_column)
```

- **Example**: Group the data by the `gender` column.

```
df_grouped <- df %>%
  group_by(gender)
```

You typically follow up a `group_by()` with a summarizing operation:

```
df_group_summary <- df %>%
  group_by(gender) %>%
  summarise(avg_age = mean(age))
```

6. `arrange()` : Sorting rows

- **Purpose**: Reorders the rows of a data frame based on the values of one or more columns.
- **Logic**: `arrange()` sorts rows in ascending (default) or descending order.

```
df_arranged <- arrange(df, column1)
```

- **Example**: Arrange rows in ascending order of `age`.

```
df_arranged <- df %>%
  arrange(age)
```

To sort in descending order:

```
df_arranged_desc <- df %>%
  arrange(desc(age))
```

7. `join` functions: Combining data frames

- **Purpose**: Combines two data frames based on matching keys.
- **Logic**: There are several `join` functions, each serving a different purpose. Common ones include:
  - `inner_join()`: Keeps only the rows with matching keys in both data frames.
  - `left_join()`: Keeps all rows from the left data frame and matches from the right, filling in `NA` when there is no match.
  - `right_join()`, `full_join()`: These work similarly, depending on which side (left or right) or whether you want to include all rows from both.

```
df_joined <- left_join(df1, df2, by = "key_column")
```

There are two cases where you might want to count rows using `dplyr` verbs (commands):

1. **Counting rows after filtering without grouping** (using `nrow()`).
2. **Counting rows within groups** (using `n()` inside `summarise()`).

Case 1: Counting Rows After Filtering Without Grouping (Using `nrow()`)

When you simply want to count the number of rows in a data frame that meet certain logical conditions, but you don't need to group the data, you can use the base R function `nrow()` after a filtering operation. This approach is straightforward and works well if you're not trying to summarize within groups but just need to see how many rows meet the condition.

How it Works:

- `filter()`: First, apply a logical condition to filter the data frame down to the rows that meet your criteria.
- `nrow()`: After the filtering step, use `nrow()` to count how many rows are left in the resulting data frame.

Example:

Imagine you have a data frame `df` and you want to count how many rows have `age > 30`:

```r
df_filtered_count <- df %>%
  filter(age > 30) %>%
  nrow()
```

Explanation:

1. `filter(age > 30)` selects the rows where the `age` column is greater than 30.
2. `nrow()` counts how many rows remain after filtering.

Multiple Logical Conditions:

You can also apply multiple logical conditions in the `filter()` step:

```r
df_filtered_count <- df %>%
  filter(age > 30, income > 50000) %>%
  nrow()
```

This filters the data frame to include only rows where both `age > 30` and `income > 50000` are true, and then `nrow()` counts the number of remaining rows.

Key Points for Case 1:

- Use `nrow()` when you're working with the whole data frame and just want to count rows after filtering or applying some logical condition.
- `nrow()` returns the total number of rows in the data frame that match your filter conditions.
- This method is simple and doesn't require `group_by()` or `summarise()`.

Case 2: Counting Rows Within Groups (Using `n()` Inside `summarise()`)

When you need to count the number of rows within different groups of data, use `n()` inside `summarise()` in combination with `group_by()`. This is a common pattern in `dplyr` when you want to summarize data for each group (e.g., counting rows per group).

How it Works:

- `group_by()`: This step groups the data by one or more columns, dividing the data frame into subsets (one for each group).
- `summarise()`: After grouping, `summarise()` collapses each group to a single row, typically calculating summary statistics like counts, means, sums, etc.
- `n()`: Inside `summarise()`, `n()` counts the number of rows in each group.

Example:

Let's say you want to count how many rows exist for each value of `gender` in your data frame `df`:

```
df_grouped_count <- df %>%
  group_by(gender) %>%
  summarise(count = n())
```

Explanation:

1. `group_by(gender)` divides the data into groups based on the values in the `gender` column.
2. `summarise(count = n())` creates a new summary column called `count` that holds the number of rows for each group (i.e., for each gender).

Grouping with Multiple Variables:

You can group by more than one variable:

```
df_grouped_count <- df %>%
  group_by(gender, country) %>%
  summarise(count = n())
```

Here, the data is grouped by both `gender` and `country`, and `n()` counts the number of rows for each unique combination of `gender` and `country`.

Key Points for Case 2:

- Use `n()` **inside** `summarise()` to count the number of rows within groups of data.
- `group_by()` defines how the data is grouped (e.g., by `gender`, `region`, etc.).
- This method is useful when you need row counts per group or for specific categories in the data.
- The result is a summarized data frame with one row per group, showing the count of rows in that group.

Piping (`%>%`)

One of the key concepts in `dplyr` is the use of the pipe operator `%>%`, which allows you to chain multiple operations together in a readable sequence. Instead of nesting functions, you pass the result of one function directly to the next, which improves readability.

Example using several dplyr verbs:

```
result <- df %>%
  filter(age > 30) %>%
  group_by(gender) %>%
  summarise(avg_income = mean(income)) %>%
  arrange(desc(avg_income))
```

In this example, the data is first filtered to keep rows where `age > 30`, then grouped by `gender`, summarized to calculate the average income for each gender, and finally sorted in descending order by `avg_income`.

Summary of the Main Verbs:

1. **`filter()`**: Subsets rows.
2. **`select()`**: Subsets columns.
3. **`mutate()`**: Adds or modifies columns.
4. **`summarise()`**: Collapses rows into a summary.
5. **`group_by()`**: Groups data for summary or other grouped operations.
6. **`arrange()`**: Sorts rows.
7. **`join` functions**: Combine data frames based on common keys.

These verbs enable a fluent, readable style of data manipulation when used together with the pipe operator `%>%`.