Exploring Dataframes (Part 2 of 2)

Chapter 6.

This chapter continues exploring dataframes. It provides an insightful look into their advantages over traditional data frames, integral to the tidyverse suite. Tibbles streamline data handling with features like partial data display for large sets, consistent structure during subsetting, support for non-standard names, and transparent data type management.

It also explores the `dplyr` package, a key part of tidyverse, highlighting its functions (e.g., `filter()`, `select()`, `arrange()`, `mutate()`, `summarise()`) that simplify data manipulation. Combined with the pipe operator `%>%`, these functions offer an efficient, readable workflow.

Using the `mtcars` dataset as an example, the chapter demonstrates transforming datasets into tibbles, applying dplyr functions for data exploration, and generating new variables. Additional functions like `rename()`, `group_by()`, `slice()`, `transmute()`, `pull()`, and `n_distinct()` are also discussed, showcasing their specific roles in data processing. Overall, thisd chapter effectively highlights the practicality and efficiency of `tibbles` and `dplyr` in data analysis, enhancing accessibility and ease in data science tasks.

tibbles

A `tibble` is essentially an updated version of the conventional data frame, providing more flexible and effective data management features.

`Tibbles`, are a component of the `tidyverse` suite, a collection of R packages geared towards making data science more straightforward. They share many properties with dataframes but also offer unique benefits that enhance our ability to work with data.

1. **Printing:** When a `tibble` is printed, only the initial ten rows and the number of columns that fit within our screen's width are displayed. This feature becomes particularly useful when dealing with extensive datasets having multiple columns, enhancing the data's readability.
2. **Subsetting:** Unlike conventional data frames, subsetting a `tibble` always maintains its original structure. Consequently, even when we pull out a single column, it remains as a one-column `tibble`, ensuring a consistent output type.

3. **Data types:** `tibbles` offer a transparent approach towards data types. They avoid hidden conversions, ensuring that the output aligns with our expectations.

4. **Non-syntactic names:** `tibbles` support columns having non-syntactic names (those not following R's standard naming rules), which is not always the case with standard data frames.

We consider `tibbles` to be a vital part of our data manipulation arsenal, especially when working within the `tidyverse` ecosystem. [1] [3]

The `dplyr` package

The `dplyr` package is very useful when we are dealing with data manipulation tasks (Wickham et al., 2021). This package offers us a cohesive set of functions, frequently referred to as "verbs," that are designed to facilitate common data manipulation activities. Below, we review some of the key "verbs" provided by the `dplyr` package:

1. **`filter()`:** When we want to restrict our data to specific conditions, we can use `filter()`. For instance, this function allows us to include only those rows in our dataset that fulfill a condition we specify.

2. **`select()`:** If we are interested in retaining specific variables (columns) in our data, `select()` is our function of choice. It is particularly useful when we have datasets with many variables, but we only need a select few.

3. **`arrange()`:** If we wish to reorder the rows in our dataset based on our selected variables, we can use `arrange()`. By default, `arrange()` sorts in ascending order. However, we can use the `desc()` function to sort in descending order.

4. **`mutate()`:** To create new variables from existing ones, we utilize the `mutate()` function. It is particularly helpful when we need to conduct transformations or generate new variables that are functions of existing ones.

5. **`summarise()`:** To produce summary statistics of various variables, we use `summarise()`. We frequently use it with `group_by()`, enabling us to calculate these summary statistics for distinct groups within our data.

Moreover, one of the significant advantages of `dplyr` is the ability to chain these functions together using the pipe operator `%>%` for a more streamlined and readable data manipulation workflow. [2] [3]

The pipe operator %>%

1. The `%>%` operator, colloquially known as the "pipe" operator, plays a vital role in enhancing the effectiveness of the `dplyr` package.
2. The purpose of this operator is to facilitate a more readable and understandable chaining of multiple operations.
3. In a typical scenario in R, when we need to carry out multiple operations on a data frame, each function call must be nested within another. This could lead to codes that are difficult to comprehend due to their complex and nested structure.
4. However, the pipe operator comes to our rescue here. It allows us to rewrite these nested operations in a linear, straightforward manner, greatly enhancing the readability of our code. [2] [3]

Illustration: Using dplyr on `mtcars` data

Loading required R packages

```
# Load the required libraries, suppressing annoying startup messages
library(dplyr, quietly = TRUE, warn.conflicts = FALSE)
```

Aside: When we load the `dplyr` package using `library(dplyr)`, R displays messages indicating that certain functions from `dplyr` are masking functions from the `stats` and `base` packages. We could instead prevent the display of package startup messages by using `suppressPackageStartupMessages(library(dplyr))` or adding `library(dplyr, quietly = TRUE, warn.conflicts = FALSE)`

Reading and Viewing the mtcars dataset as a tibble

```
# Read the mtcars dataset into a tibble called tb
tb <- as_tibble(mtcars)
```

- The `as_tibble()` function is used to convert the built-in `mtcars` dataset into a tibble object, named `tb`.

Exploring the data:

- The `head()` function is called on `tb` to display the first six rows of the dataset. This is a quick way to visually inspect the first few entries.

- The `glimpse()` function is used to provide a more detailed view of the `tb` object, showing the column names and their respective data types, along with a few entries for each column.

```
# Display the first few rows of the dataset
head(tb)
```

```
# A tibble: 6 x 11
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  21       6   160   110  3.9   2.62  16.5     0     1     4     4
2  21       6   160   110  3.9   2.88  17.0     0     1     4     4
3  22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
4  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
5  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
6  18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

```
# Display the structure of the dataset
glimpse(tb)
```

```
Rows: 32
Columns: 11
$ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
$ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
$ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
$ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

Changing data types

```
# Convert several numeric columns into factor variables
tb$cyl <- as.factor(tb$cyl)
tb$vs <- as.factor(tb$vs)
tb$am <- as.factor(tb$am)
```

```
tb$gear <- as.factor(tb$gear)
```

- Factors are used in statistical modeling to represent categorical variables.
- The `as.factor()` function is used to convert the '`cyl`', '`vs`', '`am`', and '`gear`' columns from numeric data types to factors.
- In our case, these four variables are better represented as categories rather than numerical values. For instance, '`cyl`' represents the number of cylinders in a car's engine, '`vs`' is the engine shape, '`am`' is the transmission type, and '`gear`' is the number of forward gears; all of these are categorical in nature, hence the conversion to factor.
- At this point, we can call the `glimpse()` function again to review the data structures.

```
# Display the structure of the dataset, again
glimpse(tb)
```

```
Rows: 32
Columns: 11
$ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
$ cyl  <fct> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
$ vs   <fct> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
$ am   <fct> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
$ gear <fct> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

- Notice that the datatypes are now modified and the tibble is ready for further exploration. [2] [3]

Using `dplyr` to explore the `mtcars` tibble

1. **`filter()`:** Recall that this function is used to select subsets of rows in a tibble. It takes logical conditions as inputs and returns only those rows where the conditions hold true. Suppose we wanted to filter the `mtcars` dataset for rows where the `mpg` is greater than 25.

```
tb %>%
  filter(mpg > 25)
```

```
# A tibble: 6 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  32.4 4      78.7    66  4.08  2.2   19.5 1     1     4         1
2  30.4 4      75.7    52  4.93  1.62  18.5 1     1     4         2
3  33.9 4      71.1    65  4.22  1.84  19.9 1     1     4         1
4  27.3 4      79      66  4.08  1.94  18.9 1     1     4         1
5  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
6  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
```

- This code filters the rows where the miles per gallon (mpg) are greater than 25.

2. Suppose we want to filter cars where the miles per gallon (mpg) are greater than 25 AND the number of gears is equal to 5.

```
tb %>%
  filter(mpg > 25 & gear == 5)
```

```
# A tibble: 2 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
2  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
```

- This code shows that we can impose more than one logical condition in the filter. It filters by two conditions `mpg > 25 & gear == 5`

```
tb %>%
  filter(mpg > 25, gear == 5)
```

```
# A tibble: 2 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
2  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
```

- This code shows an alternate way of setting AND conditions.

- R provides the standard suite of comparison operators: `\>`, `\>=`, `\<`, `\<=`, `!=` (not equal), and `==` (equal). It allows us to use common Boolean operators `&` (and), `|` (or) and `!` (not).

```
tb %>%
  filter(mpg > 30 | gear == 5)
```

```
# A tibble: 8 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  32.4 4      78.7    66  4.08  2.2   19.5 1     1     4         1
2  30.4 4      75.7    52  4.93  1.62  18.5 1     1     4         2
3  33.9 4      71.1    65  4.22  1.84  19.9 1     1     4         1
4  26   4     120.     91  4.43  2.14  16.7 0     1     5         2
5  30.4 4      95.1   113  3.77  1.51  16.9 1     1     5         2
6  15.8 8     351     264  4.22  3.17  14.5 0     1     5         4
7  19.7 6     145     175  3.62  2.77  15.5 0     1     5         6
8  15   8     301     335  3.54  3.57  14.6 0     1     5         8
```

- This code demonstrates the use of the `|` (or) operator.

3. **`select()`**: Recall that this function is used to select specific columns. Suppose we want to select `mpg`, `hp`, `cyl` and `am` columns.

```
tb %>%
  select(mpg, hp, cyl, am)
```

```
# A tibble: 32 x 4
     mpg    hp cyl   am
   <dbl> <dbl> <fct> <fct>
 1  21     110 6     1
 2  21     110 6     1
 3  22.8    93 4     1
 4  21.4   110 6     0
 5  18.7   175 8     0
 6  18.1   105 6     0
 7  14.3   245 8     0
 8  24.4    62 4     0
 9  22.8    95 4     0
10  19.2   123 6     0
# i 22 more rows
```

- The tibble will only contain the `mpg` (miles per gallon), `hp` (horsepower), `cyl` (cylinders) and `am` transmission columns selected from the dataset.

4. Now suppose we wanted to both filter and select. Specifically, suppose we want to:

- filter cars where the miles per gallon (mpg) are greater than 20 AND number of gears is equal to 5
- select `mpg`, `hp`, `cyl` and `am` columns for these cars.

```
filterAndSelect <- tb %>%
  filter(mpg > 20 & gear == 5) %>%
  select(mpg, hp, cyl, am)
filterAndSelect
```

```
# A tibble: 2 x 4
    mpg    hp cyl   am
  <dbl> <dbl> <fct> <fct>
1  26      91 4     1
2  30.4   113 4     1
```

- Here, we have written code that utilizes `filter()` and `select()`. These two functions, in concert with the pipe operator (`%\>%`), create a **pipeline** of operations for data transformation. Breaking this down, we observe a two-step process:
- `filter(mpg > 20 & gear == 5)`: Here, we are utilizing the `filter()` function to sift through the dataset `tb` and retain only those rows where `mpg` (miles per gallon) is more than 20 and the number of gears is equal to 5.
- `select(mpg, hp, cyl, am)`: This function is then invoked to choose specific columns from our filtered dataset. In this instance, we have picked the columns `mpg`, `hp` (horsepower), `cyl` (cylinders), and `am` (transmission type). The resulting dataset, therefore, contains only these four columns from the filtered data.

5. Suppose we wanted to select all the columns within a range. Specifically, suppose we wanted to select all the columns within cyl and wt, excluding all other columns. Recall that the original tibble has the following data columns.

```
colnames(tb)
```

```
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
```

```
tb %>%
  select(cyl:wt)
```

```
# A tibble: 32 x 5
   cyl    disp    hp  drat    wt
   <fct> <dbl> <dbl> <dbl> <dbl>
 1 6      160    110  3.9   2.62
 2 6      160    110  3.9   2.88
 3 4      108     93  3.85  2.32
 4 6      258    110  3.08  3.22
 5 8      360    175  3.15  3.44
 6 6      225    105  2.76  3.46
 7 8      360    245  3.21  3.57
 8 4      147.    62  3.69  3.19
 9 4      141.    95  3.92  3.15
10 6      168.   123  3.92  3.44
# i 22 more rows
```

- `select(cyl:wt)`: This code selects all columns in the `tb` dataframe starting from `cyl` up to and including `wt`.
- Only the five columns {cyl, disp, hp, drat, wt} get selected. This is a particularly useful feature when dealing with dataframes that have a large number of columns, and we are interested in a contiguous subset of those columns

6. Alternately, suppose instead that we wanted to select all columns except those within the range of `cyl` and `wt`.

```
tb %>%
  select(-cyl:wt)
```

```
# A tibble: 32 x 6
     mpg cyl    disp    hp  drat    wt
   <dbl> <fct> <dbl> <dbl> <dbl> <dbl>
 1  21   6      160    110  3.9   2.62
 2  21   6      160    110  3.9   2.88
 3  22.8 4      108     93  3.85  2.32
 4  21.4 6      258    110  3.08  3.22
 5  18.7 8      360    175  3.15  3.44
 6  18.1 6      225    105  2.76  3.46
 7  14.3 8      360    245  3.21  3.57
```

```
 8  24.4 4      147.    62  3.69  3.19
 9  22.8 4      141.    95  3.92  3.15
10  19.2 6      168.   123  3.92  3.44
# i 22 more rows
```

`select(-cyl:wt)`: The - sign preceding the cyl:wt range denotes exclusion. Consequently, this selects all columns in the `tb` dataframe, excluding those from `cyl` to `wt` inclusive.

7. **`arrange()`:** Recall that this function is used to reorder rows in a tibble by one or more variables. By default, it arranges rows in ascending order.

- Suppose we want to select only the `mpg` and `hp` columns, where `hp>200` and we want to sort the result in descending order of `mpg`.

```
tb %>%
  select(mpg, hp) %>%
  filter(hp>200) %>%
  arrange(desc(mpg))
```

```
# A tibble: 7 x 2
    mpg    hp
  <dbl> <dbl>
1  15.8   264
2  15     335
3  14.7   230
4  14.3   245
5  13.3   245
6  10.4   205
7  10.4   215
```

`tb %>% select(mpg, hp)`: The select function is used here to extract only the `mpg` and `hp` columns from the `tb` dataframe.

`filter(hp>200)`: The filter function is used to extract the cars where horsepower `hp>200`.

`arrange(desc(mpg))`: The arrange function is then used to order the rows in descending order (`desc`) based on the `mpg` column.

8. **Benefit from using %>%**: Suppose we wanted to subset the data as follows.

- Select cars with 6 cylinders (`cyl` == 6).
- Choose only the `mpg` (miles per gallon), `hp` (horsepower) and `wt` (weight) columns.
- Arrange in descending order by `mpg`.

Without using the pipe operator, we would have to nest the operations, as follows:

```
arrange(select(filter(tb, cyl == 6), mpg, hp, wt), desc(mpg))
```

```
# A tibble: 7 x 3
    mpg    hp    wt
  <dbl> <dbl> <dbl>
1  21.4   110  3.22
2  21     110  2.62
3  21     110  2.88
4  19.7   175  2.77
5  19.2   123  3.44
6  18.1   105  3.46
7  17.8   123  3.44
```

- Using the pipe operator, we could write the code more efficiently as follows:

```
tb %>%
  filter(cyl == 6) %>%
  select(mpg, hp, wt) %>%
  arrange(desc(mpg))
```

```
# A tibble: 7 x 3
    mpg    hp    wt
  <dbl> <dbl> <dbl>
1  21.4   110  3.22
2  21     110  2.62
3  21     110  2.88
4  19.7   175  2.77
5  19.2   123  3.44
6  18.1   105  3.46
7  17.8   123  3.44
```

- This way, the pipe operator makes the code more readable and the sequence of operations is easier to follow.

9. **`mutate()`:** Recall that this function is used to create new variables (columns) or modify existing ones.

- Suppose we want to create a new column named `efficiency`, defined as the ratio of `mpg` to `hp`.

```r
mutated_data <- tb %>%
  mutate(efficiency = mpg / hp) %>%
  select(mpg, hp, efficiency, cyl, am) %>%
  filter(mpg>30)

mutated_data
```

```
# A tibble: 4 x 5
    mpg    hp efficiency cyl   am
  <dbl> <dbl>      <dbl> <fct> <fct>
1  32.4    66      0.491 4     1
2  30.4    52      0.585 4     1
3  33.9    65      0.522 4     1
4  30.4   113      0.269 4     1
```

- The resulting tibble contains a new column `efficiency`, which is the ratio of `mpg` to `hp`.
- Note that `mutate()` does not modify the original dataset, but creates a new object `mutated_data` with the results. If we want to modify the original dataset, we would need to save the result back to the original variable.

`transmute()`: This is a variation of **`mutate()`** which does not retain the old columns.

```r
t1 <- tb %>%
  filter(mpg>30) %>%
  transmute(efficiency = mpg / hp)

t1
```

```
# A tibble: 4 x 1
  efficiency
       <dbl>
1      0.491
2      0.585
3      0.522
4      0.269
```

- Notice that `transmute()` retains only the newly created column.

10. **`summarise()`:**: Recall that this function is used to create summaries of data. It collapses a tibble to a single row. Suppose we want to calculate the mean of `mpg` in the `mtcars` dataset

```
tb %>%
  summarise(Mean_mpg = mean(mpg))
```

```
# A tibble: 1 x 1
  Mean_mpg
     <dbl>
1     20.1
```

- This code creates a tibble that contains a single row having the mean value of `mpg`.
- We can extend this code to print both the mean and standard deviation, by slightly extending the above code, as follows.

```
tb %>%
  summarise(Mean_mpg = mean(mpg),
            SD_mpg = sd(mpg)
            )
```

```
# A tibble: 1 x 2
  Mean_mpg SD_mpg
     <dbl>  <dbl>
1     20.1   6.03
```

11. To include additional statistical measures such as median, quartiles, minimum, and maximum in our summary data, we can use respective R functions within the `summarise()` function.

```
summary_data <- tb %>% summarise(
  N = n(),
  Mean = mean(mpg),
  SD = sd(mpg),
  Median = median(mpg),
  Q1 = quantile(mpg, 0.25),
  Q3 = quantile(mpg, 0.75),
  Min = min(mpg),
  Max = max(mpg)
)
summary_data
```

```
# A tibble: 1 x 8
      N  Mean    SD Median    Q1    Q3   Min   Max
  <int> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>
1    32  20.1  6.03   19.2  15.4  22.8  10.4  33.9
```

- We could convert this back into a standard dataframe and display it in a better formatted manner up to two decimal places, with the following code. [2] [3]

```
summary_data %>%
  as.data.frame() %>%
  round(2)
```

```
   N  Mean   SD Median    Q1   Q3  Min  Max
1 32 20.09 6.03   19.2 15.43 22.8 10.4 33.9
```

Additional functions in the dplyr package

1. **`rename()`:** The `rename()` function is utilized whenever we need to modify the names of some variables in our dataset. Without changing the structure of the original dataset, it allows us to give new names to chosen columns.
2. **`group_by()`:** The `group_by()` function comes into play when we need to implement operations on individual groups within our data. By categorizing our data based on one or multiple variables, we are able to apply distinct functions to each group separately.
3. **`slice()`:** To select rows by their indices, we use the `slice()` function. This is especially handy when we need specific rows, for example, the first 10 or last 10 rows, depending on a defined order.
4. **`transmute()`:** When we want to generate new variables from existing ones and keep only these new variables, we use the `transmute()` function. It is similar to `mutate()`, but it only keeps the newly created variables, making it a powerful tool when we're only interested in transformed or calculated variables.
5. **`pull()`:** The `pull()` function is used to extract a single variable as a vector from a dataframe. This function becomes very practical when we wish to isolate and work with a single variable outside its dataframe.
6. **`n_distinct()`:** To enumerate the unique values in a column or vector, we use the `n_distinct()` function. It's an essential function when we want to know the number of distinct elements within a specific categorical variable. [2] [3]

Using `dplyr` to explore the `mtcars` tibble more

1. **`rename()`:** Remember that this function is helpful in changing column names in our data. For instance, let us modify the name of the `mpg` column to `MPG` in the mtcars dataset.

```
tb %>%
  rename(MPG = mpg)
```

```
# A tibble: 32 x 11
     MPG cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
   <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
 1  21   6      160    110  3.9   2.62  16.5 0     1     4         4
 2  21   6      160    110  3.9   2.88  17.0 0     1     4         4
 3  22.8 4      108     93  3.85  2.32  18.6 1     1     4         1
 4  21.4 6      258    110  3.08  3.22  19.4 1     0     3         1
 5  18.7 8      360    175  3.15  3.44  17.0 0     0     3         2
 6  18.1 6      225    105  2.76  3.46  20.2 1     0     3         1
 7  14.3 8      360    245  3.21  3.57  15.8 0     0     3         4
 8  24.4 4      147.    62  3.69  3.19  20   1     0     4         2
 9  22.8 4      141.    95  3.92  3.15  22.9 1     0     4         2
10  19.2 6      168.   123  3.92  3.44  18.3 1     0     4         4
# i 22 more rows
```

2. **`group_by()`:** This function is key for performing operations within distinct groups of our data.

*For example, let us group the dataset by `cyl` (number of cylinders) and `am` transmission and find the mean and standard deviation for each sub-group.

```
tb %>%
  group_by(cyl, am) %>%
  summarize(MeanMPG = mean(mpg),
            StdDevMPG = sd(mpg))
```

```
`summarise()` has grouped output by 'cyl'. You can override using the `.groups`
argument.
```

```
# A tibble: 6 x 4
# Groups:   cyl [3]
  cyl   am    MeanMPG StdDevMPG
```

```
  <fct> <fct>   <dbl>     <dbl>
1 4     0        22.9     1.45
2 4     1        28.1     4.48
3 6     0        19.1     1.63
4 6     1        20.6     0.751
5 8     0        15.0     2.77
6 8     1        15.4     0.566
```

3. **slice():** Recall that this function is beneficial when we wish to choose rows based on their positions. For example, let us slice from row 3 to row 7 of the dataset.

```
tb %>%
  slice(3:7)
```

```
# A tibble: 5 x 11
    mpg cyl    disp    hp  drat    wt  qsec vs    am    gear   carb
  <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>
1  22.8 4       108    93  3.85  2.32  18.6 1     1     4         1
2  21.4 6       258   110  3.08  3.22  19.4 1     0     3         1
3  18.7 8       360   175  3.15  3.44  17.0 0     0     3         2
4  18.1 6       225   105  2.76  3.46  20.2 1     0     3         1
5  14.3 8       360   245  3.21  3.57  15.8 0     0     3         4
```

In this `sliced_data` tibble, only the first three rows from the mtcars dataset are included.

4. **pull()**: Recall that this function is employed to remove a single variable from a dataframe as a vector. Let us isolate the `mpg` (miles per gallon) variable from the dataset.

```
tb %>%
  pull(mpg)
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

5. **n_distinct()**: Recall that this function is used to count the distinct values in a column or vector. Let us count the number of distinct values in the`cyl`(cylinders) column from the dataset.

```r
tb %>%
  summarise(countDistinctCyl = n_distinct(cyl))
```

```
# A tibble: 1 x 1
  countDistinctCyl
             <int>
1                3
```

- This code shows the number of unique levels in the `cyl` column of the dataset.

Summary of Chapter 6 – Exploring Dataframes (Part 2 of 2)

This chapter provided an overview of the `tibble` data structure and the `dplyr` package in the R programming language. We started with an introduction to `tibble`, a data structure in R that is an updated version of data frames with enhanced features for flexible and effective data management. These benefits include more user-friendly printing, reliable subsetting behavior, transparent handling of data types, and support for non-syntactic column names.

Subsequently, we shifted focus to the `dplyr` package, which is a powerful tool for data manipulation in R. This package offers a cohesive set of functions, often referred to as "verbs", which allow for efficient and straightforward manipulation of data. The key "verbs" in `dplyr`— `filter()`, `select()`, `arrange()`, `mutate()`, and `summarise()`— have been explained and illustrated with examples. An integral component of the `dplyr` package, the pipe operator `%>%`, was also discussed. This operator allows for a more readable and understandable chaining of multiple operations in R, leading to cleaner and more straightforward code.

The chapter gives a comprehensive illustration of using `dplyr` on the `mtcars` dataset. This practical demonstration has involved applying `dplyr` functions to a dataset and explaining the process and results. In addition to the basics, the chapter has also touched upon additional `dplyr` functions such as `rename()`, `group_by()`, and `slice()`, enriching readers' understanding and competency in data manipulation using R.

Overall, this chapter has provided an in-depth understanding of `tibbles` and `dplyr`, their applications, and their importance in data manipulation and management in the R programming environment.

References

[1] Müller, K., & Wickham, H. (2021). tibble: Simple Data Frames. R Package Version 3.1.3. Retrieved from https://CRAN.R-project.org/package=tibble

[2] Wickham, H., François, R., Henry, L., & Müller, K. (2021). dplyr: A Grammar of Data Manipulation. R Package Version 1.0.7. Retrieved from https://CRAN.R-project.org/package=dplyr

Wickham, H. (2014). Tidy Data. Journal of Statistical Software, 59(10), 1-23. Retrieved from https://www.jstatsoft.org/article/view/v059i10

Grolemund, G., & Wickham, H. (2017). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media, Inc.

[3] Wickham, H. (2016). ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H., & Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media.

Wickham, H. (2020). ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics (Version 3.3.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=ggplot2

Wickham, H., et al. (2020). dplyr: A Grammar of Data Manipulation (Version 1.0.2) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=dplyr

Wilkinson, L. (2005). The Grammar of Graphics (2nd ed.). Springer-Verlag.

Wickham, H., et al. (2020). tibble: Simple Data Frames (Version 3.0.3) [Computer Software]. Retrieved from https://CRAN.R-project.org/package=tibble