Exploring *tibbles* & *dplyr*

July 25, 2023 V5 (Work in progress)

tibbles

A `tibble` is essentially an updated version of the conventional data frame, providing more flexible and effective data management features (Müller & Wickham, 2021).

`Tibbles`, also recognized as `tbl_df`, are a component of the tidyverse suite, a collection of R packages geared towards making data science more straightforward. They share many properties with regular data frames 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].

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]

Loading required R packages

```r
# Load the required libraries, suppressing annoying startup messages
library(dplyr)
```

```

Attaching package: 'dplyr'


The following objects are masked from 'package:stats':

    filter, lag
```

```
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

```
library(tibble)
```

Aside: When we load the `dplyr` package using `library(dplyr)`, R will display messages indicating that certain functions from dplyr are masking functions from the stats and base packages. We could instead use the `suppressPackageStartupMessages()` function to prevent the display of package startup messages.

```
# Load the required libraries, suppressing annoying startup messages
suppressPackageStartupMessages(library(dplyr))
```

Reading the mtcars dataset as a tibble

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

Creating a tibble: The `as_tibble` function is used to convert the built-in `mtcars` dataset into a tibble object, which we're naming `tb`.

```
# 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,~
```

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. Then, 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.

```r
# 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)
```

Changing data types: The `as.factor()` function is used to convert the 'cyl', 'vs', 'am', and 'gear' columns from numeric data types to factors. Factors are used in statistical modeling to represent categorical variables. 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.

```r
# 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 futher exploration.

Illustration: Using dplyr on mtcars data

Let's dive into these crucial functions from the dplyr package, providing their detailed explanations and illustrating their usage on the mtcars dataset.

filter(): This function is used to select subsets of rows in a data frame. It takes logical conditions as inputs and returns rows where the conditions hold true.

```
# Filter mtcars dataset for rows where mpg is greater than 20
filtered_data <- mtcars %>% filter(mpg > 20)

# The new dataframe 'filtered_data' contains only the rows where the miles per gallon (mpg
```

arrange(): This function is used to reorder rows in a data frame by one or more variables. By default, it arranges rows in ascending order.

```
# Arrange mtcars dataset in descending order of mpg
arranged_data <- mtcars %>% arrange(desc(mpg))

# 'arranged_data' will contain the rows of the mtcars dataset, but sorted in descending or
```

select(): This function is used to select variables (columns) by their names.

```
# Select 'mpg' and 'hp' columns from mtcars dataset
selected_data <- mtcars %>% select(mpg, hp)

# 'selected_data' will only contain the 'mpg' (miles per gallon) and 'hp' (horsepower) col
```

mutate(): This function is used to create new variables (columns) or modify existing ones.

```
# Create a new column 'efficiency' as ratio of mpg to hp in mtcars dataset
mutated_data <- mtcars %>% mutate(efficiency = mpg / hp)

# The dataframe 'mutated_data' will contain a new column 'efficiency', which is the ratio
```

transmute(): This function is similar to mutate(), but it only keeps the newly created columns.

```
# Create a new column 'efficiency' as ratio of mpg to hp in mtcars dataset and only keep t
transmuted_data <- mtcars %>% transmute(efficiency = mpg / hp)

# 'transmuted_data' only contains the 'efficiency' column which is the ratio of 'mpg' to '
```

summarise(): This function is used to create summaries of your data. It collapses a data frame to a single row.

```
# Find the mean of 'mpg' in the mtcars dataset
summary_data <- mtcars %>% summarise(mean_mpg = mean(mpg))

# 'summary_data' will contain a single row with the mean value of 'mpg' in the mtcars data
```

Remember that all these functions do not modify the original dataset, they create new objects with the results. If you want to modify the original dataset, you'd need to save the result back to the original variable, or use the mutate_at, mutate_all, mutate_if functions to modify specific columns directly.

The pipe operator %>%

As regular users of R, we often utilize the `%>%` operator, colloquially known as the "pipe" operator, which plays a vital role in the `dplyr` package. The purpose of this operator is to facilitate a more readable and understandable chaining of multiple operations. Although this operator was originally introduced by the `magrittr` package, it has since become extensively adopted in `dplyr` and other tidyverse packages.

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. 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. [3]

Illustration: This operator is best understood with an illustration. Using the `mtcars` dataset suppose we want to:

- 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 the pipe operator, we would have to nest your operations like this:

```r
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
```

Here's how we would do the same operations using the pipe operator:

```r
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
```

Here's what each line is doing:

`tb %>%` sends the mtcars data frame into the `filter()` function.

`filter(cyl == 6) %>%` filters the data frame to include only rows where cyl is equal to 6, then sends this filtered data frame to the `select()` function.

`select(mpg, hp) %>%` selects only the mpg and hp columns from the data frame, then sends this subset of the data to the `arrange()` function.

`arrange(desc(mpg))` arranges the rows of the data frame in descending order based on the mpg column.

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

References

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

[3] Bache, S. M., & Wickham, H. (2020). magrittr: A Forward-Pipe Operator for R. R package version 2.0.1. https://CRAN.R-project.org/package=magrittr