

Exploring *tibbles* & *dplyr*

July 25, 2023 V6 (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].

Basic functions in 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]

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: Using dplyr on mtcars data

We will now illustrate the crucial functions from the `dplyr` package, on the `mtcars` dataset.

1. Loading required R packages

```
# 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))
```

2. 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, 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, 1, 1, 1, 1, 0,~
$ am <dbl> 1, 1, 1, 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, 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.

```
# 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.

```
# 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, 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, 1, 1, 1, 1, 0, ~
$ am <fct> 1, 1, 1, 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.

3. Using dplyr to explore the mtcars tibble

- **filter():** Recall that 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. Suppose we want to filter the mtcars dataset for rows where the mpg is greater than 20.

```
filtered_data <- tb %>% filter(mpg > 20)
filtered_data
```

```
# A tibble: 14 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  24.4   4   147.    62  3.69  3.19  20    1     0     4     2
6  22.8   4   141.    95  3.92  3.15  22.9 1     0     4     2
7  32.4   4    78.7   66  4.08  2.2   19.5 1     1     4     1
8  30.4   4    75.7   52  4.93  1.62  18.5 1     1     4     2
9  33.9   4    71.1   65  4.22  1.84  19.9 1     1     4     1
10 21.5   4   120.    97  3.7   2.46  20.0 1     0     3     1
11 27.3   4    79     66  4.08  1.94  18.9 1     1     4     1
12 26     4   120.    91  4.43  2.14  16.7 0     1     5     2
13 30.4   4    95.1  113  3.77  1.51  16.9 1     1     5     2
14 21.4   4   121    109  4.11  2.78  18.6 1     1     4     2
```

The tibble 'filtered_data' contains only the rows where the miles per gallon (mpg) are greater than 20.

- **select()**:: Recall that this function is used to select specific columns. Suppose we want to select 'mpg' and 'hp' columns from mtcars dataset

```
selected_data <- tb %>% select(mpg, hp)
selected_data
```

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

The tibble 'selected_data' will only contain the 'mpg' (miles per gallon) and 'hp' (horsepower) columns from the mtcars dataset.

- **arrange():** Recall that this function is used to reorder rows in a data frame by one or more variables. By default, it arranges rows in ascending order. Suppose we want to select only the mpg and hp columns from the mtcars data and sort it in descending order of mpg

```
arranged_data <- tb %>% select(mpg, hp) %>% arrange(desc(mpg))
arranged_data
```

```
# A tibble: 32 x 2
  mpg    hp
<dbl> <dbl>
1  33.9    65
2  32.4    66
3  30.4    52
4  30.4   113
5  27.3    66
6  26      91
7  24.4    62
8  22.8    93
9  22.8    95
10 21.5    97
# i 22 more rows
```

The steps in the code can be broken down as follows:

arranged_data <- tb %>% select(mpg, hp): The select function is used here to extract only the mpg and hp columns from the tb dataframe. The %>% operator is the pipe operator, which is used to chain multiple operations together in a readable manner. This part of the code will create a new dataframe containing only the mpg and hp columns.

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

- **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 in the mtcars dataset.

```
mutated_data <- tb %>% mutate(efficiency = mpg / hp)
mutated_data
```

```
# A tibble: 32 x 12
  mpg cyl  disp  hp  drat    wt  qsec vs  am  gear  carb efficiency
<dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <fct> <fct> <fct> <dbl>    <dbl>
1  21    6   160  110  3.9   2.62  16.5 0    1    4     4    0.191
2  21    6   160  110  3.9   2.88  17.0 0    1    4     4    0.191
3  22.8  4   108   93  3.85  2.32  18.6 1    1    4     1    0.245
4  21.4  6   258  110  3.08  3.22  19.4 1    0    3     1    0.195
5  18.7  8   360  175  3.15  3.44  17.0 0    0    3     2    0.107
6  18.1  6   225  105  2.76  3.46  20.2 1    0    3     1    0.172
7  14.3  8   360  245  3.21  3.57  15.8 0    0    3     4    0.0584
8  24.4  4   147.   62  3.69  3.19  20    1    0    4     2    0.394
9  22.8  4   141.   95  3.92  3.15  22.9 1    0    4     2    0.24
10 19.2  6   168.  123  3.92  3.44  18.3 1    0    4     4    0.156
# i 22 more rows
```

The tibble ‘mutated_data’ will contain a new column ‘efficiency’, which is the ratio of ‘mpg’ to ‘hp’. The original data columns in **tb** will be retained.

Remember that these functions do not modify the original dataset, they create new objects with the results. If we want to modify the original dataset, we would 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.

- **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

```
summary_data <- tb %>% summarise(mean_mpg = mean(mpg))
summary_data
```

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

The tibble ‘summary_data’ will contain a single row with the mean value of ‘mpg’ in the `mtcars` dataset.

To include additional statistical measures such as median, quartiles, minimum, and maximum in your 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.

```
summary_df <- as.data.frame(summary_data)
print(summary_df)
```

```
      N   Mean    SD Median    Q1    Q3   Min   Max
1 32 20.09062 6.026948   19.2 15.425 22.8 10.4 33.9
```

If we wanted to display only two decimal places, we could try

```
summary_df %>% 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
```

Illustration of %>%: 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:

```
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:

```
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.

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.

4. Using dplyr to explore the mtcars tibble

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

```
renamed_data <- mtcars %>% rename(MPG = mpg)
renamed_data
```

	MGP	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

The dataframe `renamed_data` now includes the MPG column, which was previously named `mpg`.

- **`group_by()`**: This function is key for performing operations within distinct groups of our data. For example, let's group the `mtcars` dataset by the `cyl` (number of cylinders) column.

```
grouped_data <- mtcars %>% group_by(cyl)
grouped_data
```

```
# A tibble: 32 x 11
# Groups:   cyl [3]
   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
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
```

The `grouped_data` dataframe is now grouped by the `cyl` column, which enables us to carry out operations on each group separately.

- **`slice()`**: Remember that this function is beneficial when we wish to choose rows based on their positions. For example, let's select the first three rows of the `mtcars` dataset.

```
sliced_data <- mtcars %>% slice(1:3)
sliced_data
```

```
      mpg cyl disp  hp drat   wt  qsec vs am gear carb
Mazda RX4     21.0   6  160 110 3.90 2.620 16.46 0  1   4    4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02 0  1   4    4
Datsun 710    22.8   4  108  93 3.85 2.320 18.61 1  1   4    1
```

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

- **`transmute()`**: Recall that if we desire to create new variables and keep only these variables, we apply the `transmute()` function. Suppose we want to create a new variable that is the ratio of horsepower (`hp`) to weight (`wt`), and keep only this new variable.

```
transmuted_data <- mtcars %>% transmute(hp_to_wt = hp/wt) %>% head()
transmuted_data
```

	hp_to_wt
Mazda RX4	41.98473
Mazda RX4 Wag	38.26087
Datsun 710	40.08621
Hornet 4 Drive	34.21462
Hornet Sportabout	50.87209
Valiant	30.34682

The transmuted_data dataframe now includes the newly created hp_to_wt variable, while the other columns have been removed.

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

```
pulled_data <- mtcars %>% pull(mpg)
pulled_data
```

```
[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
```

In the pulled_data vector, only the values from the mpg variable are retained.

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

```
distinct_count <- mtcars %>% summarise(n_distinct_cyl = n_distinct(cyl))
distinct_count
```

	n_distinct_cyl
1	3

The distinct_count dataframe shows the number of unique values in the cyl column of the mtcars dataset.

Summary

The chapter has introduced and provided a detailed overview of `tibbles` and the `dplyr` package in the R programming language.

It started with an introduction to `tibbles`, 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, the chapter shifted focus to the `dplyr` package, which has been explained as 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 `%>%`, has also been discussed in detail. This operator allows for a more readable and understandable chaining of multiple operations in R, leading to cleaner and more straightforward code.

The chapter has given a comprehensive illustration of using `dplyr` on the `mtcars` dataset. This practical demonstration has involved applying `dplyr` functions to a real 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. <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>

Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10), 1-23. <https://www.jstatsoft.org/article/>

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