Basics of R and Data

Bruno Santos & Antonio Paez

2025-07-02

This Rmarkdown file is part of the **CommuteCA** package. This package was created in conjunction with the office of the *Research Data Center* at *McMaster University*, the *Sherman Centre for Digital Scholarship* and the *Mobilizing Justice*[1].

The **CommuteCA** R package was created to develop standardized methods for transport analysis in research, particularly for analysis using the *2021 Census of Population* from Statistics Canada. We focused our efforts on the *Commuting Reference Guide*, which provides valuable variables and information on commuting for the Canadian population aged 15 and older living in private households.

[1] The Mobilizing Justice project is a multidisciplinary and multi-sector collaboration with the objective of understand and address transportation poverty in Canada and to improve the well-being of Canadians at risk of transport poverty. The Social Sciences and Humanities Research Council (SSRHC) has provided funding for the project, which was created by an unprecedented alliance of academics from various Canadian provinces and institutions, transportation firms, and nonprofit organizations

Introduction

This R markdown aims to provide a brief introduction of `R` language and the concepts of data. This notebook[2] is an updated and reduced version of the computational notebooks available in the *edashop* package. The *edashop* package is an open educational resource to teach a workshop on EDA using R and computational notebooks, created and maintained by Dr. Antonio Paez, co-author of the *CommuteCA* package. For those interested in learning EDA in more deep, we suggest installing and studying the *edashop* package.

[2] We will explain what a R markdown and a notebook are in the following sections.

In this first R markdown, we will learn about:

- Literate programming
- Data objects and basic operations
- Ways of measuring stuff
- Data manipulation with {dplyr}

Literate programming

This is an R Markdown document, a plain text file that recognizes code chunks and displays results beneath them. R Markdown files are computational notebooks following literate programming, which emphasizes natural language for human communication and code for computer communication, making coding more intuitive.

Structure of an R Markdown Document:

- Header (YAML Header): At the top, between —, includes metadata like title, author(s), and processing instructions.
- Body: Main content, including text and code.

Code chunks are essential in computational notebooks and can be executed to see results. Try the example below:

```r
print("Hello, CommuteCA")
```

```
## [1] "Hello, CommuteCA"
```

The print() function displays the argument on the screen.

R Markdown documents prioritize humans and use R for illustrations. The computer assists in demonstrating concepts, like code chunks.

Each R markdown of this packages serves as lecture notes and can be knit into a `pdf` for study. These documents can be the foundation for experiments and annotations, allowing you to customize notes with your understanding and additional information. The original notebook is available for fresh starts.

It's good practice to clear the working memory in RStudio before starting new work. This ensures no extraneous information is in memory that might confuse your tasks. Use the rm command in R to clear the workspace by removing items. To clear all objects, run:

```r
rm(list = ls())
```

Here, ls() lists all objects in the workspace. Alternatively, click the broom icon in the 'Environment' tab in RStudio.

Data objects and basic operations

R can handle different data types, including scalars, vectors, and matrices. The following code chunks illustrate these data types:

- *Scalar*: A scalar is an object with one element (one row and one column):

```r
1
```

```
## [1] 1
```

- *Vector*: A vector is a data object with one row and multiple columns:

```r
c(1, 2, 3, 4)
```

```
## [1] 1 2 3 4
```

- *Matrix*: A matrix has multiple rows and columns. This example has three rows and three columns:

```r
matrix(c(1, 2, 3, 4, 0, 0, 0, 0, 1), nrow = 3, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    0
## [2,]    2    0    0
## [3,]    3    0    1
```

In general, matrices can be of size $n \times m$, meaning they have n rows and m columns. The c() command concatenates inputs, while matrix() creates a matrix with specified nrow and ncol.

An important data class in R is the data frame, a rectangular table of rows and columns, often consisting of vectors combined for convenience. Data frames store data digitally and resemble tables in spreadsheet software.

Data frames can hold large amounts of information (up to several billion items, depending on your computer's memory). They can include numeric, alphanumeric (characters), logical (TRUE/FALSE) data, etc. Each cell has an address identified by its row and column, allowing R to perform mathematical operations. R typically labels columns alphabetically and rows numerically.

To illustrate a data framework, we will create the following vectors for some Canadian provinces. This data comes from the Statistics Canada. The first vector includes the names of the provinces; the second vector contains the population (2021); the second contains the population percentage change from 2016 to 2021; the next is the number of active workers (in April 2021); and finally, there is a vector with the area of the province in square kilometers:

```
province_name <- c("Alberta",
                   "British Columbia",
                   "Ontario",
                   "Quebec",
                   "Newfoundland and Labrador",
                   "Manitoba",
                   "Saskatchewan")
population <- c(4262635,
                5000879,
                14223942,
                8501833,
                510550,
                1342153,
                1132505)
population_change  <- c(4.8,
                         7.6,
                        5.8,
                        4.1,
                        -1.8,
                    5.0,
                    3.1)
workers <- c(1897337,
             2263247,
             6279827,
             3693457,
             209649,
             599534,
             470931)
area_km2 <- c(634658.27,
             920686.55,
             892411.76,
             1298599.75,
             358170.37,
             540310.19,
             577060.40)
```

Note that <- is an assignment operator in R, used to assign the value on the right to the name on the left. This allows easy retrieval of values later. After executing the previous code chunk, five vectors appear in your Environment (upper right pane tab). These vectors are of size 1:7, consisting of one alphanumeric ('chr', for 'character') and four numeric ('num') vectors.

Creating a Data Frame

Vectors can be assembled into a data frame, which is convenient when data are related. To create a data frame, vectors must have the same length; data frames do not support vectors with differing row numbers. Let's create a data frame named `df` (you can choose a different name). Some rules for naming objects include starting with a letter and using letters, digits, periods, and underscores. Names should be intuitive, easy to remember, and concise.

Use the data.frame() function to create a data frame, with vectors as its arguments:

```
df <- data.frame(province_name,
                 population,
                 population_change,
                 workers,
                 area_km2)
```

Running the above chunk creates a new object in your environment: a data frame called `df`. Data frames are rectangular tables, so all vectors must be the same length. For instance, if we have seven municipalities but data for only six, we can exclude the one missing information or include it with missing values, ensuring vectors remain the same size.

Double-click `df` in the Environment tab to see that it has four columns (labeled `province_name`, `population`, `population_change`, `workers`, `area_km2`), and 7 rows. Row numbers and column names identify specific cells. You can enter data into a data frame and use R's built-in functions for analysis. Display the data frame by typing its name as an R command:

```
df
```

```
##               province_name population population_change workers  area_km2
## 1                   Alberta    4262635               4.8 1897337  634658.3
## 2          British Columbia    5000879               7.6 2263247  920686.6
## 3                   Ontario   14223942               5.8 6279827  892411.8
## 4                    Quebec    8501833               4.1 3693457 1298599.8
## 5 Newfoundland and Labrador     510550              -1.8  209649  358170.4
## 6                  Manitoba    1342153               5.0  599534  540310.2
## 7              Saskatchewan    1132505               3.1  470931  577060.4
```

The variable `province_name` at the moment is a character vector. If we check the structure of the data frame we can see this:

```
str(df)
```

```
## 'data.frame':    7 obs. of  5 variables:
##  $ province_name    : chr  "Alberta" "British Columbia" "Ontario" "Quebec" ...
##  $ population       : num  4262635 5000879 14223942 8501833 510550 ...
##  $ population_change: num  4.8 7.6 5.8 4.1 -1.8 5 3.1
##  $ workers          : num  1897337 2263247 6279827 3693457 209649 ...
##  $ area_km2         : num  634658 920687 892412 1298600 358170 ...
```

The summary of the names of the provinces is:

```
summary(df$province_name)
```

```
##    Length     Class      Mode
##         7 character character
```

This shows that the column class is 'character'. A character variable can be converted into a factor. Factors are used for categorical data (e.g., labels, names, classes) and typically have two or more levels. In this case, `province_name` has seven levels, representing Canada's provinces. With multiple time periods, each province might appear more than once. Convert `province_name` to a factor:

```
df$province_name <- factor(df$province_name)
```

Reference columns in a data frame using `$` after the table name. The `<-` operator assigns the result of `factor(df$province_name)` to the province_name column in `df`. We could store it in a new data frame, but it's unnecessary now. The `factor()` function converts a variable to a factor:

```
str(df)
```

```
## 'data.frame':    7 obs. of  5 variables:
##  $ province_name    : Factor w/ 7 levels "Alberta","British Columbia",..: 1 2 5 6 4 3 7
##  $ population       : num  4262635 5000879 14223942 8501833 510550 ...
##  $ population_change: num  4.8 7.6 5.8 4.1 -1.8 5 3.1
##  $ workers          : num  1897337 2263247 6279827 3693457 209649 ...
##  $ area_km2         : num  634658 920687 892412 1298600 358170 ...
```

Compare the factor summary to the character variable summary:

```
summary(df$province_name)
```

```
##                   Alberta          British Columbia                  Manitoba
##                         1                         1                         1
## Newfoundland and Labrador                   Ontario                    Quebec
##                         1                         1                         1
##              Saskatchewan
##                         1
```

Basic operations

R can perform various operations, including arithmetic (sums, multiplications) and logical (TRUE/FALSE) operations. To perform operations effectively, it's important to understand how R locates information, such as in a data frame. Each grid cell has an address, known as an index, which can be referenced in several ways. For example, to reference the first value in the data frame, specifically row 1 of the column `province_name`, use square brackets with row and column indices separated by a comma:

```
df[1, 1]
```

```
## [1] Alberta
## 7 Levels: Alberta British Columbia Manitoba ... Saskatchewan
```

This recalls the element in the first row and first column of `df`. Alternatively, you can type:

```
df$province_name[1]
```

```
## [1] Alberta
## 7 Levels: Alberta British Columbia Manitoba ... Saskatchewan
```

Both methods yield the same result. The `$` symbol is used to reference columns in a data frame, so R calls the first element of `province_name` in `df`. To reference the second province in the table:

```
df$province_name[2]
```

```
## [1] British Columbia
## 7 Levels: Alberta British Columbia Manitoba ... Saskatchewan
```

Requesting `df[1,1]` is the same as `df$province_name[1]`. To recall the full column of province names, type `df$province_name`.

Indexing for operations

Indexing is useful for operations. To calculate the total number of workers in two provinces, say Alberta and British Columbia, execute:

```
df$workers[1] + df$workers[2]
```

```
## [1] 4160584
```

An issue with indexing cells by row number is that if new provinces are added, the row numbers might change, leading to incorrect references. Additionally, in large tables, it's difficult to know which province corresponds to which row. A better approach is using logical operators to index data frame cells, as shown below. Here, we ask for the sum of "workers of (province which is Alberta)" and "workers of (province which is British Columbia)":

```
df$workers[df$province_name == "Alberta"] + df$workers[df$province_name == "British Columbia"]
```

```
## [1] 4160584
```

Inside the square brackets, R checks for the row with the specified province name (indexing by province name rather than row number), and the text before the brackets refers to the workers column in the indexed row.

To calculate the total area (in square kilometers) in the data frame, use the sum function, identifying the column to sum with $:

```
sum(df$area_km2)
```

```
## [1] 5221897
```

As demonstrated, R can function as a calculator, but it's much more powerful. You can store instruction results in memory using assignment (`<-`). Many other operations are possible, such as finding the maximum value in a set:

```
max(df$area_km2)
```

```
## [1] 1298600
```

The max() function can be used to find the maximum of any set of values, not just a single column:

```
max(df$area_km2[df$province_name == "Ontario"], df$area_km2[df$province_name == "Quebec"])
```

```
## [1] 1298600
```

To find the province with the largest area:

```
df$province_name[df$area_km2 == max(df$area_km2)]
```

```
## [1] Quebec
## 7 Levels: Alberta British Columbia Manitoba ... Saskatchewan
```

This shows that Quebec has the largest territorial area. Similarly, use `min()` to find the minimum value:

```
min(df$area_km2)
```

```
## [1] 358170.4
```

To find the province with the smallest area:

```
df$province_name[df$area_km2 == min(df$area_km2)]
```

```
## [1] Newfoundland and Labrador
## 7 Levels: Alberta British Columbia Manitoba ... Saskatchewan
```

Exploring data with calculations

R has a high flexibility and allows for various calculations to explore and analyze data. You can perform operations on single columns or use multiple columns in a data frame. For instance, the sample data frame contains information about Canadian provinces. To discover the population density of the provinces, we can perform calculations across several columns.

To define the population density by province, calculate the population by area in squared kilometers in each province:

$$PD_i = \frac{population_i}{area\_km_i}$$

Where PD_i is the population density in province i.

The text enclosed by `$$` is *LaTeX* code, allowing you to type mathematical formulas in R Markdown. Inline mathematical expressions can be written using the notation x. Learning to write these expressions is optional, and numerous online resources online can help you get started.

To calculate the population density :

```
PD <- df$population/df$area_km2
```

The code above creates a new vector `PD`. To add this vector as a new column in the data frame:

```
df$PD <- df$population/df$area_km2
```

You can add new columns in real time, ensuring the assigned data size matches the data frame (same number of rows). Verify the new column in your `df` data frame:

```
df
```

```
##               province_name population population_change workers   area_km2
## 1                   Alberta    4262635               4.8 1897337   634658.3
## 2          British Columbia    5000879               7.6 2263247   920686.6
## 3                   Ontario   14223942               5.8 6279827   892411.8
## 4                    Quebec    8501833               4.1 3693457  1298599.8
## 5 Newfoundland and Labrador     510550              -1.8  209649   358170.4
## 6                  Manitoba    1342153               5.0  599534   540310.2
## 7              Saskatchewan    1132505               3.1  470931   577060.4
##          PD
## 1  6.716426
## 2  5.431685
## 3 15.938766
## 4  6.546923
## 5  1.425439
## 6  2.484042
## 7  1.962542
```

To round off numeric data values, use the round() function. Here, we round to two decimals:

```
df$PD <- round(df$population/df$area_km2, 2)
```

Ways of measuring stuff

Merriam-Webster[3] defines data as *a factual information (such as measurements or statistics) used as a basis for reasoning, discussion, or calculation.* Data arise from observations, and measurement theory is the mathematical framework for recording these observations. Measurements focus on specific attributes of an object, like weight, density, height, or price, rather than encompassing all its characteristics. This distinction highlights that a measurement captures a particular aspect rather than the entirety of the subject being measured.

[3] https://www.merriam-webster.com/dictionary/data

There are several typologies that describe appropriate scales of measurement. Stanley Smith Stevens, a notable psychologist, identified four such scales, two of which are categorical and two are quantitative.

Categorical: Nominal Scale

The nominal scale is the most basic form of measurement, used to assign unique labels or categories to items. It compresses complex information into simple, recognizable categories. Examples include modes of transportation such as "car," "bus," "walk," and "bicycle," or brands like "Apple," "Huawei," and "Nokia." Importantly, nominal categories do not have an inherent order; for example, "car" is not inherently higher or closer to "bicycle."

These categories can be compared using Boolean operations:

```
"car" == "car"  # returns TRUE
```

```
## [1] TRUE
```

```
"car" == "bus"  # returns FALSE
```

```
## [1] FALSE
```

Categorical: Ordinal scale

Ordinal scale measurements are categorical but include a natural order among categories. These are often used in Likert-style scales, such as five-point scales ranging from "Strongly Disagree" to "Strongly Agree," with a midpoint of "Neutral." The key difference from the nominal scale is that ordinal categories have a meaningful sequence: "Strongly Disagree" is closer to "Disagree" than to "Neutral," and "Strongly Agree" is further from "Neutral."

Sometimes quantitative variables, like income, are collected and reported using ordinal scales, such as income brackets:

- Less than 20,000
- Between 20,000 and 40,000
- More than 40,000

Like nominal variables, different categories can be compared with boolean operations "==" and "!=" (i.e., not equal to). In addition, the following operations are also valid: "<" and "<=" (i.e., *less than*) and ">" (i.e., greater than).

Quantitative: Interval scale

In the ordinal scale, categories have a natural order, but the differences between them are not quantified. For example, income categories like “less than 20,000” and “more than 40,000” can be ordered, but the difference between them isn’t a specific dollar amount. Likewise, Likert scale responses (e.g., “Strongly Agree”) reflect subjective attitudes that vary among individuals and can’t be precisely measured.

The interval scale also has an order, but unlike the ordinal scale, the differences between values are meaningful. Examination scores, such as on a 1-100 scale, exemplify this: the difference between 80 and 90 is 10 points. However, zero doesn’t imply a total absence of knowledge, nor does 100 indicate complete understanding. Interval scales allow operations like addition and subtraction, providing more quantitative insight than ordinal scales.

Quantitative: Ratio scale

The interval scale is more informative than the ordinal scale because it allows consistent quantification of differences between values. However, the ratio between two values is not meaningful because interval variables lack a natural origin. For instance, a score of 20 does not represent infinitely more knowledge than a score of zero, nor does 100 indicate twice the knowledge of a score of 50.

Ratio variables have a natural origin, meaning a value of zero indicates the absence of what is being measured. For example, a length of zero means no length, and an income of zero means no income. This allows all operations valid for interval variables and additionally enables multiplication and division. For instance, an income of 40,000 is twice as much as an income of 20,000.

Classes of objects in R

In `R`, data classes differentiate between categorical and quantitative variables. Consider these vectors:

```r
modes <- c("car", "bus", "walk", "walk", "car", "walk", "walk", "car")
frequency <- c(6, 3, 4, 5, 4, 3, 5, 4)
safe <- c(5, 1, 2, 3, 4, 2, 3, 4)
```

Check their classes:

```r
class(modes)
```

```
## [1] "character"
```

```
class(frequency)
```

```
## [1] "numeric"
```

```
class(safe)
```

```
## [1] "numeric"
```

We can see that modes is a character vector (nominal, categorical); frequency is a numeric vector (quantitative); safe is numeric but should be ordinal.

To properly code modes and safe as factors:

```
modes <- factor(modes, levels = c("bus", "car", "walk"))
safe <- factor(safe,
               levels = c(1, 2, 3, 4, 5),
               labels = c("Strongly Disagree", "Disagree", "Neutral", "Agree", "Strongly Agree"),
               ordered = TRUE)
```

Check the updated classes:

```
class(modes)
```

```
## [1] "factor"
```

```
class(safe)
```

```
## [1] "ordered" "factor"
```

Now, modes and safe are factors, with safe being an ordinal factor. Properly defining scales allows R to perform appropriate operations. For example:

```
modes[1] == modes[3]
```

```
## [1] FALSE
```

This checks if the modes are the same for respondents 1 and 3, while operations like summing categorical data (e.g., “car” + “walk”) are not defined.

```
modes[1] + modes[3]
```

```
## [1] NA
```

With ordinal variables we can compare the relative values of levels:

```
safe[1] >= safe[2]
```

```
## [1] TRUE
```

As noted above, though, the difference between levels is not meaningful:

```
safe[1] - safe[2]
```

```
## [1] NA
```

In contrast, frequency is a ratio variable, and this is a meaningful operation:

```
frequency[1]/7
```

```
## [1] 0.8571429
```

The above is the proportion of the week that Respondent 1 uses the mode indicated. Here we collect the vectors in a data frame, which we are going to call unimaginatively `df`:

```
df2 <- data.frame(modes, frequency, safe)
```

Since `R` understands the classes of the variables, it is possible to obtain quick summaries of the data:

```
summary(df2)
```

```
##   modes     frequency                    safe
##  bus :1   Min.   :3.00   Strongly Disagree:1
##  car :3   1st Qu.:3.75   Disagree         :2
##  walk:4   Median :4.00   Neutral          :2
##           Mean   :4.25   Agree            :2
##           3rd Qu.:5.00   Strongly Agree   :1
##           Max.   :6.00
```

Function `summary()` uses appropriate methods for the data.

Data manipulation with dplyr

A package is a shareable code unit that enhances `R` capabilities. Packages are commonly installed from {CRAN} (Comprehensive R Archive Network), which manages dependencies to ensure packages work together smoothly. Install packages from `CRAN` using `install.packages()`. A valuable package set is {dplyr} (the name is a riff on "data plier"), it provides numerous functions for data "carpentry" or data "wrangling". What distinguishes the package is that functions are implemented following a consistent *grammar* of data manipulation. Data are manipulated using "verbs" that act on data objects. Verbs can be linked by means of *pipes* to form complete *phrases* of data manipulation.[4]

Load package:

[4] You can install `dplyr` directly from CRAN using `install.packages(dplyr)`. If you have the `CommuteCA` package installed, all the packages required for all sessions will be already installed.

```
library(dplyr) # A Grammar of Data Manipulation
```

The R community has developed consistent standards for sharing packages, emphasizing the importance of documentation. Good documentation is a hallmark of a quality package. To consult the help pages for a package or its contents, use the following function:

```
?dplyr
```

One way of manipulate Data Frames in `R` is using pipe. Piping passes information from one function to another, enhancing code readability. R supports two pipe operators: `%>%` from the {magrittr} package and `|>` natively in base R (version 4.1 and later). For example:

```
df2 |>
  summary()
```

```
##   modes     frequency                    safe
##  bus :1   Min.   :3.00   Strongly Disagree:1
##  car :3   1st Qu.:3.75   Disagree         :2
##  walk:4   Median :4.00   Neutral          :2
##           Mean   :4.25   Agree            :2
##           3rd Qu.:5.00   Strongly Agree   :1
##           Max.   :6.00
```

This code means *"pass data frame df2 to the summary() function."* Pipes improve code clarity by chaining functions in a straightforward manner.

Previously we saw how indexing works to call *parts* of data frames, in other words, to *subset* data frames. Recall that we can retrieve a column from a data frame by naming it:

```
df2$modes
```

```
## [1] car  bus  walk walk car  walk walk car
## Levels: bus car walk
```

Similarly, we can retrieve rows as follows. Suppose that we want the first row from the table:

```
df2[1,]
```

```
##   modes frequency           safe
## 1   car         6 Strongly Agree
```

Or the first two rows:

```
df2[1:2,]
```

```
##   modes frequency              safe
## 1   car         6    Strongly Agree
## 2   bus         3 Strongly Disagree
```

Or rows 1, 3, and 5:

```
df2[c(1, 3, 5),]
```

```
##   modes frequency           safe
## 1   car         6 Strongly Agree
## 3  walk         4       Disagree
## 5   car         4          Agree
```

As an alternative scenario, suppose that we want to extract only the rows corresponding to "walk":

```
df2[df2$modes == "walk",]
```

```
##   modes frequency     safe
## 3  walk         4 Disagree
## 4  walk         5  Neutral
## 6  walk         3 Disagree
## 7  walk         5  Neutral
```

{dplyr} implements several verbs that are useful to subset a data frame. The verb `select()` acts on columns. For instance, with piping:

```
df2 |>
  select(modes)
```

```
##   modes
## 1   car
## 2   bus
## 3  walk
## 4  walk
## 5   car
## 6  walk
## 7  walk
## 8   car
```

The above is read as "take data frame `df2` and select column `modes`". It is possible to select by negation:

```
df2 |>
  select(-modes)
```

```
##   frequency              safe
## 1         6    Strongly Agree
## 2         3 Strongly Disagree
## 3         4          Disagree
## 4         5           Neutral
## 5         4             Agree
## 6         3          Disagree
## 7         5           Neutral
## 8         4             Agree
```

As well, it is possible to select multiple columns:

```
df2 |>
  select(modes, safe)
```

```
##   modes              safe
## 1   car    Strongly Agree
## 2   bus Strongly Disagree
## 3  walk          Disagree
## 4  walk           Neutral
## 5   car             Agree
## 6  walk          Disagree
## 7  walk           Neutral
## 8   car             Agree
```

Or:

```
df2 |>
  select(-modes, -safe)
```

```
##   frequency
## 1         6
## 2         3
## 3         4
## 4         5
## 5         4
## 6         3
## 7         5
## 8         4
```

Verb `slice()` acts on rows. For example:

```
df2 |>
  slice(1)
```

```
##   modes frequency           safe
## 1   car         6 Strongly Agree
```

The above is read as *"take data frame `df2` and slice the first row"*. The following chunk implements *"take data frame `df2` and slice rows one to two"*:

```
df2 |>
  slice(1:2)
```

```
##   modes frequency               safe
## 1   car         6    Strongly Agree
## 2   bus         3 Strongly Disagree
```

Or *"slice rows 1, 3, and 5"*:

```
df2 |>
  slice(c(1, 3, 5))
```

```
##   modes frequency           safe
## 1   car         6 Strongly Agree
## 2  walk         4       Disagree
## 3   car         4          Agree
```

The following variations of `slice()` are available: `slice_head()`, `slice_tail()`, `slice_min()`, `slice_max()`, and `slice_sample()`. Use `?` to check the documentation.

Another verb, `filter()`, also acts on rows, but instead of retrieving them by position does it by some condition. The following *phrase* implements *"take data frame `df2` and filter all rows where the `mode` is equal to `walk`"*:

```
df2 |>
  filter(modes == "walk")
```

```
##   modes frequency     safe
## 1  walk         4 Disagree
## 2  walk         5  Neutral
## 3  walk         3 Disagree
## 4  walk         5  Neutral
```

The value of a grammatical approach with piping becomes more evident when we wish to form more complex *phrases* of data manipulation and analysis. For example:

```
summary(df2[df2$modes == "walk",])
```

```
##   modes     frequency                   safe
##  bus :0   Min.   :3.00   Strongly Disagree:0
##  car :0   1st Qu.:3.75   Disagree         :2
##  walk:4   Median :4.50   Neutral          :2
##           Mean   :4.25   Agree            :0
##           3rd Qu.:5.00   Strongly Agree   :0
##           Max.   :5.00
```

In the above, the arguments are nested and the phrase has to be read from inside out, as it were. Compare to the more linear grammar of the following chunk:

```
df2 |>
  filter(modes == "walk") |>
  summary()
```

```
##   modes     frequency                     safe
##  bus :0   Min.   :3.00   Strongly Disagree:0
##  car :0   1st Qu.:3.75   Disagree         :2
##  walk:4   Median :4.50   Neutral          :2
##            Mean   :4.25   Agree            :0
##            3rd Qu.:5.00   Strongly Agree   :0
##            Max.   :5.00
```

Creating new variables

To create a new column `id` with unique identifiers:

```
df2 <- df2 |>
  mutate(id = factor(1:n()))
```

Here, `n()` returns the number of rows, and `mutate()` adds the new id column to `df2`. Remember to reassign the result to `df2` to retain the changes. To convert `frequency` from days per week to the proportion of the week:

```
df2 |>
  mutate(frequency = frequency/7)
```

```
##   modes frequency              safe id
## 1   car 0.8571429    Strongly Agree  1
## 2   bus 0.4285714 Strongly Disagree  2
## 3  walk 0.5714286          Disagree  3
## 4  walk 0.7142857           Neutral  4
## 5   car 0.5714286             Agree  5
## 6  walk 0.4285714          Disagree  6
## 7  walk 0.7142857           Neutral  7
## 8   car 0.5714286             Agree  8
```

This replaces the existing frequency column with its proportionate value. Another verb, `transmute()` combines the behavior of `select()` and `mutate()`. See for instance:

```
df2 |>
  transmute(id,
            frequency = frequency/7)
```

```
##   id frequency
## 1  1 0.8571429
## 2  2 0.4285714
## 3  3 0.5714286
## 4  4 0.7142857
## 5  5 0.5714286
## 6  6 0.4285714
## 7  7 0.7142857
## 8  8 0.5714286
```

Verb `relocate()` changes the order of columns:

```
df2 |>
  mutate(frequency = frequency/7) |>
  relocate(id,
           .before = "modes")
```

```
##   id modes frequency              safe
## 1  1   car 0.8571429    Strongly Agree
## 2  2   bus 0.4285714 Strongly Disagree
## 3  3  walk 0.5714286          Disagree
## 4  4  walk 0.7142857           Neutral
## 5  5   car 0.5714286             Agree
## 6  6  walk 0.4285714          Disagree
## 7  7  walk 0.7142857           Neutral
## 8  8   car 0.5714286             Agree
```

Working on groups of cases

To summarize data by groups, you can use `group_by()` in conjunction with `summarize()` from the {dplyr} package. Here's how you can calculate the mean proportion of mode use by travel mode:

```
df2 |>
  group_by(modes) |>
  summarize(mean_frequency = mean(frequency),
            .groups = "drop")
```

```
## # A tibble: 3 x 2
##   modes mean_frequency
##   <fct>          <dbl>
## 1 bus             3
## 2 car             4.67
## 3 walk            4.25
```

The argument `.groups = "drop"` ungroups the output of the phrase. It is possible to group by various variables, for example:

```
df2 |>
  group_by(modes,
           safe) |>
  summarize(mean_frequency = mean(frequency),
            .groups = "drop")
```

```
## # A tibble: 5 x 3
##   modes safe              mean_frequency
##   <fct> <ord>                      <dbl>
## 1 bus   Strongly Disagree            3
## 2 car   Agree                        4
## 3 car   Strongly Agree               6
## 4 walk  Disagree                     3.5
## 5 walk  Neutral                      5
```

Grouping is a powerful way to work simultaneously on separate parts of a data frame.