# Session 3. Exploratory data analysis I: Descriptive statistics

**Antonio Paez**
**My Name**

14 June, 2022

---

**Highlights**:

This is my mini-reflection. Paragraphs must be indented.
It can contain multiple paragraphs.

**Threshold Concepts**:

threshold concept 1
threshold concept 2
threshold concept 3
threshold concept 4

---

> ""We never look beyond our assumptions and what's worse, we have given up trying to meet others; we just meet ourselves."
>
> —– Muriel Barbery

## Session Outline

- What is EDA?
- Data summaries revisited
- Appropriate summary statistics by scale of measurement
- Properties of data: central tendency and spread
- Univariate description : frequency tables, summary statistics
- Bivariate description : correlation, cross-tabulation
- Multivariate description : multiple correlation, multiple cross-tabulation??

## Reminder

Remember that literate programming asks you to do the hard work up front so that your life can be easier later.

## Preliminaries

Clear the workspace from *all* objects:

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

Load packages. Remember, packages are units of shareable code that augment the functionality of base `R`. For this session, the following package/s is/are used:

```
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(edashop)
library(kableExtra)
```

```
##
## Attaching package: 'kableExtra'

## The following object is masked from 'package:dplyr':
##
##     group_rows
```

```
library(skimr)
```

We will also load the following data frames for this session:

```
data("auctions_amf")
data("auctions_pf")
data("auctions_phy")
data("auctions_sef")
```

These data frames contain information about real estate transactions in distressed markets in Italy. You can check the documentation in the usual way:

```
?auctions_amf
```

In brief, these four tables give information about properties auctioned in Italy between 2000 and 2016 in distressed real estate markets. Each table gives information about one aspect of the issue: features of the auction market (_amf), profitability features of the property (_pf), physical features of the property (_phy), and socio-economic features of the location of the property (_sef). For convenience we will combine the tables into a single `auctions` data frame (see Session 2):

```
auctions <- auctions_amf |>
  left_join(auctions_pf,
            by = "id") |>
  left_join(auctions_phy,
            by = "id") |>
  left_join(auctions_sef,
            by = "id")
```

# What is EDA?

Measuring stuff is a lot of effort. It can be expensive too. Sometimes need special equipment. And instruments. Why do we bother?

## Data summaries revisited

In the previous session we used the function `summary()` from base `R` to obtain quick summaries of data. For example:

```
summary(auctions)
```

```
##        id      days_on_market   number_auctions    discount
##  1      :  1   Min.   : 190.0   Min.   :1.000   Min.   :-0.8109
##  2      :  1   1st Qu.: 496.8   1st Qu.:2.000   1st Qu.:-0.4400
##  3      :  1   Median : 684.0   Median :3.000   Median :-0.3423
##  4      :  1   Mean   : 831.3   Mean   :3.193   Mean   :-0.3046
##  5      :  1   3rd Qu.: 931.2   3rd Qu.:4.000   3rd Qu.:-0.2091
##  6      :  1   Max.   :4104.0   Max.   :9.000   Max.   : 0.7359
##  (Other):119   NA's   :5        NA's   :6       NA's   :2
##     premium               date                 occupancy
##  Min.   :-0.4022483   Min.   :2000-12-12   Unoccupied:30
##  1st Qu.: 0.0000208   1st Qu.:2008-10-18   Tenant    :11
##  Median : 0.0667681   Median :2010-12-20   Owner     :58
##  Mean   : 0.1814727   Mean   :2010-12-29   NA's      :26
##  3rd Qu.: 0.2302389   3rd Qu.:2012-03-21
##  Max.   : 1.7285076   Max.   :2016-10-22
##  NA's   :3
##                 type_class gross_building_area      quality
##  Residence            :90   Min.   :  11.40     Poor     :18
##  Factory              : 9   1st Qu.:  91.37     Adequate :31
##  Build-on Land        : 7   Median : 117.83     Fair     :31
##  Agricultural Building: 6   Mean   : 199.75     Good     :24
##  Mixed                : 5   3rd Qu.: 184.30     Excellent: 6
##  (Other)              : 5   Max.   :1855.00     NA's     :15
##  NA's                 : 3   NA's   :14
##  state_maintenance        location      income        delta_ntn
##  Poor     :14      Center     :25   Min.   :    0   Min.   :-0.6100
##  Adequate :18      Semi-center:27   1st Qu.:10083   1st Qu.:-0.4125
##  Fair     :31      Suburban   :70   Median :12116   Median :-0.2610
##  Good     :33      NA's       : 3   Mean   : 9963   Mean   :-0.2700
##  Excellent:11                       3rd Qu.:13894   3rd Qu.:-0.1630
##  NA's     :18                       Max.   :15405   Max.   : 0.7220
##                                     NA's   :5       NA's   :6
##  re_activity_index    population
##  Min.   :0.000000   Min.   :     0
##  1st Qu.:0.008525   1st Qu.:  4685
##  Median :0.013600   Median : 12211
##  Mean   :0.011973   Mean   : 44090
##  3rd Qu.:0.016325   3rd Qu.: 27126
##  Max.   :0.030600   Max.   :888249
##  NA's   :5          NA's   :4
```

Package {skimr} is an alternative to the basic summary. It implements tools to "skim" data, and produces reports that are easier to read because it separates variables by type, provides a larger set of summary statistics that are appropriate to the type of data, and it also generates *sparklines*. {skimr} is also pipe-friendly. The basic function is `skim()`. It is possible to skim a complete data frame or parts thereby. For example:

| skim_type | skim_variable | n_missing | complete_rate | numeric.mean | numeric.sd | numeric.p0 | numeric.p25 | numeric.p50 | numeric.p75 | numeric.p100 |
|---|---|---|---|---|---|---|---|---|---|---|
| numeric | days_on_market | 5 | 0.96 | 831.29 | 561.34 | 190 | 496.75 | 684 | 931.25 | 4104 |

```
auctions |>
  select(days_on_market) |>
  skim()
```

This is read as "pass `auctions` to `select`, retrieve `days_on_market` and skim". Try the code on your console! You will see that the output includes a summary of the data, with a high level description of the inputs: the number of rows, columns, number of columns by type of data, and any grouping variables.

To render the code in the PDF file, we will use package {kableExtra}, which includes functionality to format tables. Below is the output of skimming our selected variable; from the output we strip the variable that contains the sparklines (`numeric.hist`), which are tricky to render in PDF. This is then passed to functions `kable()` and `kable_styling()`:

```
auctions |>
  select(days_on_market) |>
  skim() |>
  select(-numeric.hist) |>
  kable("latex",
        digits = 2,
        booktabs = TRUE) |>
  kable_styling(latex_options="scale_down")
```

The arguments in `kable()` and `kable_styling()` control the appearence of the table in the output document: how to format the rows (booktabs), how many digits to use, whether to scale down a wide table so that it fits the page. Much more information about the possibilities of working with {kableExtra} can be found in the documentation.

Since {skimr} plays well with {dplyr} it is possible to combine it with other data carpentry functions. For example, the next code of chunk uses `group_by()` before skimming the table:

```
auctions |>
  select(type_class,
         days_on_market) |>
  group_by(type_class) |>
skim_without_charts()
```

Try it in your console. The chunk above is read as "take data frame `auctions`, select columns `type_class` and `days_on_market`, group by `days_on_market`, and skim". The descriptive statistics skimmed include the number of missing observations and completeness of the data, the mean, standard deviation, and *quantiles*, that is, the values that cut the sample at a certain proportion of observations (e.g., "p50" is the value where the sample is split in two equal parts, the bottom 50% and the top 50%).

As you can see, three observations lack the `type_class` category. Of the rest, properties of type "Residence" stayed in auction on average for as long as 815 days, and the quickest sale took as much as 190 days.

Skimming the full table gives the following:

```
auctions |>
  skim_without_charts()
```

The summaries are separated by type of data: dates are reported separately from factors and from quantitative (numeric) variables. Appropriate summary statistics are calculated for each type of data.

## Appropriate summary statistics by scale of measurement

Wait, what do you mean by "appropriate summary statistics"?

Recall from Session 2 that not all operations are defined for all scales of measurement. For example, variables in the nominal scale could be compared using only boolean operators "==" (exactly equal to) and "!=" (not equal to). No arithmetic operations are defined for ordinal data. And division and multiplication are not appropriate for interval data.

This has implications for the kind of statistics that are appropriate by scale of measurement.

Consider the mean of a variable. The mean is defined as follows:

$$\bar{x} = \frac{x_1 + x_2 + \cdots + x_n}{n} = \frac{1}{n}\sum_{i-1}^{n} x_i$$

Is it appropriate to calculate the mean of a categorical variable? What is the meaning of two cars plus one bicycle divided by three?

To understand which summary statistics are appropriate, we must know how various summary statistics are calculated.

## Properties of data

Summary statistics are information reduction techniques. Recall that the objective of EDA is to see the data from different perspectives. Two important properties of data are their central tendency and dispersion.

### *Central tendency*

A measure of central tendency provides a summary of the a distribution of values of a variable by expressing a "typical" value, or the one most commonly found. Mathematically, this is equivalent to organizing all data values and finding where the *center of mass* of the distribution falls. To illustrate the concept of center of mass consider the following sequence of quantitative values:

```
x <- c(20, 30, 32, 34, 41, 41, 45, 46, 48, 51, 53, 54, 54, 56, 57, 58, 58, 59,
  60, 61, 64, 65, 65, 69, 71, 74, 77, 88, 94)
```

The same sequence of values is presented below in the style of a stem-and-leaf table:

| stem | leaf |
|---|---|
| 2 | 0 |
| 3 | 024 |
| 4 | 11568 |
| 5 | 134467889 |
| 6 | 014559 |
| 7 | 147 |
| 8 | 8 |
| 9 | 4 |

Where is the distribution "heavier"? Thereabouts will be its center of mass. We also see that the most common values in the distribution tend to be around 5.

#### *Mode*

The mode of a distribution is the most frequent value in a distribution. Since it only involves counting the instances of values, it is an appropriate for nominal and ordinal variables. We can find the mode by tabulating the values. Let us do so for the variable `type_class` (factor). Here we introduce function `pull()` from {dplyr}. This function extracts a column from a data frame as a vector:

```r
auctions |>
  pull(type_class) |>
  table()
```

```
##
## Agricultural Building         Build-on Land               Factory
##                     6                     7                     9
##                 Mixed                Office             Residence
##                     5                     1                    90
##                Retail
##                     4
```

We see that the mode of the distribution is "Residence", the most frequent value of the variable in this distribution. (Notice that by default the values of the factor are sorted alphabetically; this can be changed by redefining the factor and changing the order of the levels).

Next, let us do variable `quality` (ordered factor):

```r
auctions |>
  pull(quality) |>
  table()
```

```
##
##      Poor  Adequate      Fair      Good Excellent
##        18        31        31        24         6
```

We see that the mode is "Adequate" and "Fair". Since ordinal variables have by definition a natural order, the shape of their distribution can be conveniently presented in the style of a stem-and-leaf table, with each "I" representing one instance of the value:

| stem | leaf |
|---|---|
| Poor | IIIII IIIII IIIII III |
| Adequate | IIIII IIIII IIIII IIIII IIIII IIIII I |
| Fair | IIIII IIIII IIIII IIIII IIIII IIIII I |
| Good | IIIII IIIII IIIII IIIII IIII |
| Excellent | IIIII I |

#### *Median*

#### *Mean*

### *Spread*

#### *Minimum and maximum*

#### *Inter-quartile range*

#### *Standard deviation*

### *Examples of instruments used for measurement*

- 
- 
- 

## Scales of measurement

There are several typologies that describe appropriate scales of measurement. A useful and widely used on was developed by psychologist Stanley Smith Stevens, and recognizes four scales of measurement: two categorical and two quantitative scales.

### *Categorical: Nominal scale*

This is the most basic, and in a way, the least informative scale for measuring things. It assigns unique labels/categories to things. Examples of this scale include modes of transportation (e.g., car, bus, walk, bicycle) and brands (e.g., Apple, Huawei, Nokia). The labels reduce/compress much information into a single recognizable category. The labels, on the other hand, do not have any natural order, in the sense that category "car" is not intrinsically higher than or closer to category "bicycle". Different categories can be compared with boolean operations "==" (i.e., *exactly equal to*) and "!=" (i.e., *not equal to*).

```
"car" == "car"
```

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

```
"car" == "bus"
```

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

What things are you aware of that are measured using the nominal scale?

-
-
-

### *Categorical: Ordinal scale*

Measurements in an ordinal scale are still categorical, and include items measured in Likert-style scales, for example five-point scales from "Strongly Disagree" to "Strongly Agree" with a "Neutral" point. The difference with the nominal scale is that there is a natural way of ordering the categories: "Strongly Disagree" is closer to "Disagree" than it is to "Neutral", and "Strongly Agree" is even more distant from it than "Neutral". Sometimes quantitative variables (for instance, income) are collected and/or reported using ordinal scales: income less than 20,000, income between 20,000 and 40,000, and income more than 40,000. This of course involves a loss of information, but may reduce respondent burden or satisfy confidentiality constraints when income data are collected.

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).

What things are you aware of that are measured using the ordinal scale?

-
-
-

### *Quantitative: Interval scale*

In the ordinal scale, the order of the categories is important, but the difference between categories is not a quantity. For example, the difference between category "income less than 20,000" and "income more than 40,000" is not a quantity. We can count the steps that separate these two categories but cannot impute a quantity in dollars to the difference. Attitudinal variables measured using a Likert scale relate to a subjective state of mind which is not necessarily identical for all individuals. For example, suppose that the answer to a statement such as “This mode of transportation is safe” is “Strongly Agree” by two individuals with different levels of tolerance for risk. Can we quantify the difference between this response and “Agree” in a consistent way?

The interval scale is similar to the ordinal scale in the sense that the values have a natural ordering. In addition, the differences between levels *are* meaningful. An example of an interval variable is scores from an examination; an examination is an instrument aims to measure knowledge/understanding of a subject. When a scale of 1-100 is used, the difference between a score of 80 and a score of 90 is 10 points. However, a zero does not indicate the absolute *absence* of knowledge/understanding, just as a score of 100 does not indicate absolutely *complete* knowledge/understanding of the subject.

Valid operations for variables in interval scale include all those for ordinal variables, and in addition “+” and “-” (which is how 5 points in question 1 plus 10 points in question 2 add 15 points to the final score).

What things are you aware of that are measured using the interval scale?

- 
- 
- 

### *Quantitative: Ratio scale*

The interval scale is more informative than the ordinal scale in the sense that it is possible to quantify the differences between to values in a consistent way across measurements. On the other hand, the ratio between two values is not meaningful. Since variables measured in interval scale do not have a natural origin (i.e., the value of zero does not indicate complete absence), a score of 20 does not mean infinitely more knowledge/understanding than a score of zero, just as 100 does not mean twice as much knowledge/understanding as a score of 50.

Ratio variables have a natural origin that indicates the *absence* of the thing being measured. A length of zero means the absence of this dimension; an income of zero means the absence of income. This means that we can use all the operations available for interval variables, and in addition “*” and “/” (i.e., an income of 40,000 is twice as high as an income of 20,000).

What things are you aware of that are measured using the ratio scale?

- 
- 
- 

## Data objects revisited and quick data summaries

`R` provides data classes for categorical and quantitative variables. To illustrate them, suppose that we have information about modes of transportation used by a small sample of respondents, as well as how frequently they use this mode (times per week) and responses to the statement “this mode of transportation is safe” with 1: Strongly Disagree and 5: Strongly Agree. We will create the following vectors to represent this information:

```
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 the class of these vectors:

```
class(modes)
```

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

```r
class(frequency)
```

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

```r
class(safe)
```

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

Of these, only the frequency is a quantitative variable. The class "character" is not a scale of measurement: it is a way to store alphanumeric information. The variable `safe` is stored as a numeric variable, but it should be an ordinal variable.

Categorical data in `R` is coded as *factors*. Factors can be nominal (the default) and ordinal. A factor can be created by means of the function `factor()`. To properly code `modes` and `safe` as factors we do the following:

```r
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)
```

Notice that the levels of the factor (i.e., the categories) can be of class "character" or "numeric". When the levels are numeric we can assign *labels* to them to explicitly identify the category. Argument `ordered = TRUE` is used for ordinal data.

Check again the classes of the vectors:

```r
class(modes)
```

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

```r
class(safe)
```

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

Now the variables are recognized as categorical (i.e., factors), and also as ordinal when appropriate.

Correctly defining the scale of measurement allows `R` to know which operations make sense for the kind of data at hand. For example:

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

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

Respondents 1 and 3 in the data frame do not use the same mode of transportation.

However, the sum of "car" and "walk" is not defined:

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

```
## Warning in Ops.factor(modes[1], modes[3]): '+' not meaningful for factors
```

```
## [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]
```

```
## Warning in Ops.ordered(safe[1], safe[2]): '-' is not meaningful for ordered
## factors
```

```
## [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`:

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

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

```
summary(df)
```

```
##   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

What are the things that you most commonly need to do when you are preparing/organizing data?

- 
- 
- 

## Data carpentry/wrangling and the {dplyr} package

Package {dplyr}

### *Pipes*

Piping is the process of passing information from one function to another. There are at least two pipe operators in `R`: {maggritr} implements `%>%` and base `R` implements pipes natively with `|>` since version 4.1. These two operators do essentially the same thing, but their behaviors are slightly different.

A pipe operator basically works by taking the value on the left (which could be the output of a function) and passing it to another function. For example:

```
df |>
  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
```

The chunk of code above is read as “take data frame `df` and pass it on to function `summary()`.

Pipes are great for increasing the legibility of code.

### *Subsetting a table*

In the previous session 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:

```
df$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:

```
df[1,]
```

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

Or the first two rows:

```
df[1:2,]
```

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

Or rows 1, 3, and 5:

```
df[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”:

```r
df[df$modes == "walk",]
```

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

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

```r
df |>
  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 `df` and select column `modes`". It is possible to select by negation:

```r
df |>
  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:

```r
df |>
  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:

```
df |>
  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:

```
df |>
  slice(1)
```

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

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

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

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

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

```
df |>
  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 `df` and filter all rows where the mode is equal to"walk" ":

```
df |>
  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(df[df$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:

```
df |>
  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
```

Which one is easier to read?

### *Creating new variables*

Often we wish to create and add new variables to a data frame. The verb `mutate()` is useful for this purpose. For example, our sample data frame does not include an explicit identifier for the respondents. We can add one with mutate:

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

Function `n()` returns the number of rows in the input data frame. Check the table: it now has a new column with the respondent ids. By the way, notice that we assigned the results of our data manipulation phrase back to `df`, if we had not done so, the results would not have been kept in memory.

Suppose that we wanted to convert the variable `frequency()` from days per week to proportion of the week that the mode is used. Mutate can replace an existing variable in the data frame:

```
df |>
  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
```

Another verb, `transmute()` combines the behavior of `select()` and `mutate()`. See for instance:

```
df |>
  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:

```
df |>
  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*

Sometimes we wish to work with parts of the table. For example, in the previous session you were asked to count the number of spinoffs by geography of Italy (i.e., Northern, Central, Southern). To achieve this you probably subset the table three times (one for each region), and then calculated the number of cases.

A more elegant approach is to create groups and to summarize by group. The pair of verbs `group_by()` and `summarize()` work in this way. Suppose that we would like to know what is the mean proportion of use of modes of travel but by mode:

```
df |>
  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:

```
df |>
  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.

### *Combining tables*

Related data often come in separate tables, for convenience or because the data come from different sources. When two tables are of the same size they can be combined with verb `bind_cols()`. This verb puts two tables side by side as if they were a single table. Suppose that we had a second table (which we will unimaginatively call `df_2`) with information about the personal attributes of respondents to the survey (i.e., age in years and gender):

```
df_2 <- data.frame(age = c(25, 32, 39, 28, 40, 33, 21, 32),
                   gender = factor(c("male", "female", "female", "non-binary", "male", "female", "femal
```

The two columns are combined as follows:

```
df |>
  bind_cols(df_2)
```

```
##   modes frequency              safe id age     gender
## 1   car         6    Strongly Agree  1  25       male
## 2   bus         3 Strongly Disagree  2  32     female
## 3  walk         4          Disagree  3  39     female
## 4  walk         5           Neutral  4  28 non-binary
## 5   car         4             Agree  5  40       male
## 6  walk         3          Disagree  6  33     female
## 7  walk         5           Neutral  7  21     female
## 8   car         4             Agree  8  32       male
```

This works on the assumption that the rows are arranged *in the same order* and does not match by case. Suppose that the second table had been:

```
df_2 <- data.frame(id = factor(c(5, 4, 8, 1, 7, 2, 3, 6)),
                   age = c(25, 32, 39, 28, 40, 33, 21, 32),
                   gender = factor(c("male", "female", "female", "non-binary", "male", "female", "femal
```

Notice that for whatever reason, the respondents in `df_2` are not sorted in the same order as in `df`. Using `bind_cols()` would lead to the erroneous table:

```r
df |>
  bind_cols(df_2)
```

```
## New names:
## * `id` -> `id...4`
## * `id` -> `id...5`

##   modes frequency              safe id...4 id...5 age     gender
## 1   car         6    Strongly Agree      1      5  25       male
## 2   bus         3 Strongly Disagree      2      4  32     female
## 3  walk         4          Disagree      3      8  39     female
## 4  walk         5           Neutral      4      1  28 non-binary
## 5   car         4             Agree      5      7  40       male
## 6  walk         3          Disagree      6      2  33     female
## 7  walk         5           Neutral      7      3  21     female
## 8   car         4             Agree      8      6  32       male
```

Relational joins combine tables based on one or more *keys*, that is, common variables. For example, `df` and `df_1` have a common id. Verb `left_join()` takes the rows in the table on the right and joins them to the table in the left so that the key variable(s) match(es):

```r
df |>
  left_join(df_2,
            by = c("id" = "id"))
```

```
##   modes frequency              safe id age     gender
## 1   car         6    Strongly Agree  1  28 non-binary
## 2   bus         3 Strongly Disagree  2  33     female
## 3  walk         4          Disagree  3  21     female
## 4  walk         5           Neutral  4  32     female
## 5   car         4             Agree  5  25       male
## 6  walk         3          Disagree  6  32       male
## 7  walk         5           Neutral  7  40       male
## 8   car         4             Agree  8  39     female
```

Check how now the individual attributes match correctly the rows in the left table.

Other possible joins are `right_join()`, `inner_join()`, and `full_join()`. Use `?` to check the documentation.

A different situation arises when we have more *cases* (i.e., rows) of the same variables. For example, this table has two more cases that can be combined with the original table:

```r
df_3 <- data.frame(id = factor(c(9, 10)),
                   modes = c("bus", "walk"),
                   frequency = c(2, 5),
                   safe = c("Disagree", "Agree"))
```

Now we want to combine the tables not by adding columns but by adding rows. The appropriate verb is `bind_rows`, and it combines the table by joining the second argument to the bottom of the first argument. Notice that the bind matches by column name, so it does not matter if the columns are in the same order:

```r
df |>
  bind_rows(df_3)
```

```
##    modes frequency              safe id
## 1    car         6    Strongly Agree  1
## 2    bus         3 Strongly Disagree  2
## 3   walk         4          Disagree  3
## 4   walk         5           Neutral  4
## 5    car         4             Agree  5
## 6   walk         3          Disagree  6
## 7   walk         5           Neutral  7
## 8    car         4             Agree  8
## 9    bus         2          Disagree  9
## 10  walk         5             Agree 10
```

If the number of columns does not match, missing values will be added as needed (here the bottom table has an additional random variable):

```
df |>
  bind_rows(df_3 |>
              mutate(random_variable = sample(5, n())))
```

```
##    modes frequency              safe id random_variable
## 1    car         6    Strongly Agree  1              NA
## 2    bus         3 Strongly Disagree  2              NA
## 3   walk         4          Disagree  3              NA
## 4   walk         5           Neutral  4              NA
## 5    car         4             Agree  5              NA
## 6   walk         3          Disagree  6              NA
## 7   walk         5           Neutral  7              NA
## 8    car         4             Agree  8              NA
## 9    bus         2          Disagree  9               4
## 10  walk         5             Agree 10               1
```

The classes of the variables must match! The following code will throw an error because the variable `id` in one table is a factor but in the other it is a number:

```
df |>
  bind_rows(df_3 |>
              mutate(id = c(9, 10)))
```

## Practice

1. Summarize table `auctions_phy`. What are the scales of measurement of the variables in this table? (Hint: check the documentation of the table)
2. Summarize table `auctions_amf`. What are the scales of measurement of the variables in this table?
3. Join the following tables using an appropriate key variable: `auctions_amf`, `auctions_phy`, `auctions_sef`.
4. Create a new table with only variables `id`, `type_class`, `days_on_market`, `gross_building_area`, and `location`.
5. Obtain a summary of the new table.
6. Obtain a summary of the new table but only for properties of `type_class` "Residential".
7. Obtain a summary of the new table but only for properties *not* of `type_class` "Residential".
8. What is the mean `gross_building_area` by `type_class` of property?