Session 2. Data

2026-02-13

Highlights:

This is my mini-reflection. Paragraphs must be indented.

It can contain multiple paragraphs.

Write the concepts that in your opinion are threshold concepts in this exercise. A threshold concept is a key idea that once you grasp it, it changes your understanding of a topic, phenomenon, subject, method, etc. Write between three and five threshold concepts that apply to your learning experience working on this exercise.

> "If you think education is expensive, try ignorance."
>
> —— Robert Orben

Session outline

- Why measuring things?
- Scales of measurement
- Data objects revisited
- Quick data summaries
- Data manipulation
- `Pandas`

Reminder

NOTE: This is an Quarto Markdown document. This type of document is a plain text file that can recognize chunks of code. When you execute code within the document, the results are displayed beneath. Quarto Markdown files are *computational notebooks* which implement a coding philosophy called *literate programming*. Literate computing emphasizes the use of natural language to communicate with humans and chunks of code to communicate with the computer. By making the main audience other humans, this style of coding flips around the usual way in which code is written (computer is main audience, humans come second). This helps to make learning how to code more intuitive and accessible.

Preliminaries

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

```
import pandas as pd

# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)    # Show all rows
pd.set_option('display.max_columns', None) # Show all columns
pd.set_option('display.width', None)       # Auto-detect width
pd.set_option('display.max_colwidth', None) # Show full column content
```

We also will utilize some data from the `edashop` R package. To convert these R data files to `Python` files, we will use the `reticulate` package:

```
library(edashop)  # A Package for a Workshop on Exploratory Data Analysis
library(reticulate)
```

From `edashop`, we will also load the following data frames for this session:

```
data("cntr_sp_basico")
data("cntr_sp_head")
data("cntr_sp_ipvs")
```

These data frames contain information about census tracts in the state of São Paulo. You can check the documentation in the usual way:

```
?cntr_sp_basico
```

To be able to convert these datasets in Python, we need to convert them into structures that `Python` recognizes. Following chunk transform them into a Pandas DataFrame:

```
cntr_sp_basico = r.cntr_sp_basico
cntr_sp_head = r.cntr_sp_head
cntr_sp_ipvs = r.cntr_sp_ipvs
```

Why measuring?

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

Why do we measure?

-
-
-

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"
```

```
True
```

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

```
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

`Python` 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 lists to represent this information:

```
modes = ["car", "bus", "walk", "walk", "car", "walk", "walk", "car"]
frequency = [6, 3, 4, 5, 4, 3, 5, 4]
safe = [5, 1, 2, 3, 4, 2, 3, 4]
```

Pandas contains a function, named `CategoricalDtype`, to work with categorical data. To properly code `modes` and `safe`, we first create a pandas DataFrame from these lists:

```
df = pd.DataFrame({'modes': modes, 'frequency': frequency, 'safe': safe})
```

To check the data types of these columns, we use the dtypes attribute:

```
df.dtypes
```

```
modes         object
frequency      int64
safe           int64
dtype: object
```

Of these, only the frequency is a numerical variable. The modes are stored as object (string) type, and safe is stored as integer, but it should be treated as ordinal categorical data. Of these, only the frequency is a numerical variable. The modes are stored as object (string) type, and safe is stored as integer, but it should be treated as ordinal categorical data.

```
from pandas.api.types import CategoricalDtype

# convert `modes` to categorical
modes_type = CategoricalDtype(categories = ["bus", "car", "walk"], ordered = False)
df['modes'] = df['modes'].astype(modes_type)

# create ordered categorical dtype for `safe`
ordered_labels = ["Strongly Disagree", "Disagree", "Neutral", "Agree", "Strongly Agree"]
safe_type = CategoricalDtype(categories = ordered_labels, ordered = True)

# map numeric safe values to labels and convert to categorical
safe_labels = {1: "Strongly Disagree", 2: "Disagree", 3: "Neutral", 4: "Agree", 5: "Strongly Agree"}
df['safe_label'] = df['safe'].replace(safe_labels).astype(safe_type)
```

```
df
```

```
  modes  frequency  safe         safe_label
0   car          6     5     Strongly Agree
1   bus          3     1  Strongly Disagree
2  walk          4     2           Disagree
3  walk          5     3            Neutral
4   car          4     4              Agree
5  walk          3     2           Disagree
6  walk          5     3            Neutral
7   car          4     4              Agree
```

Now check the data types again:

```
df.dtypes
```

```
modes         category
frequency        int64
safe             int64
safe_label    category
dtype: object
```

```
df["modes"]
```

```
0     car
1     bus
2    walk
3    walk
4     car
5    walk
6    walk
7     car
Name: modes, dtype: category
Categories (3, object): [’bus’, ’car’, ’walk’]
```

```
df["safe_label"]
```

```
0       Strongly Agree
1    Strongly Disagree
```

```
2             Disagree
3              Neutral
4                Agree
5             Disagree
6              Neutral
7                Agree
Name: safe_label, dtype: category
Categories (5, object): ['Strongly Disagree' < 'Disagree' < 'Neutral' < 'Agree' < 'Strongly Agree']
```

The label variables are recognized as categorical, and `safe_label` is marked as ordered when appropriate. Correctly defining the scale of measurement allows pandas to know which operations make sense for the kind of data at hand. For example, we can compare categorical values:

```
df['modes'].iloc[0] == df['modes'].iloc[2]
```

```
False
```

```
df['safe_label'].iloc[0]
```

```
'Strongly Agree'
```

```
df['safe_label'].iloc[1]
```

```
'Strongly Disagree'
```

Now, to compare the categories we use the function `.cat.codes` with `.iloc` to specify the index:

```
df['safe_label'].cat.codes.iloc[0] > df['safe_label'].cat.codes.iloc[1]
```

```
True
```

With ordinal variables we can compare the relative values of levels, but arithmetic operations between categories are not meaningful (unlike with numerical variables).

Numerical operations work on numerical columns:

```
df['frequency'].iloc[0]/7
```

```
0.8571428571428571
```

The above is the proportion of the week that Respondent 0 uses the mode indicated.

Since pandas understands the data types of variables, it is possible to obtain quick summaries of the data:

```
df.describe(include='all')
```

```
       modes  frequency      safe safe_label
count      8   8.000000  8.000000          8
unique     3        NaN       NaN          5
top     walk        NaN       NaN   Disagree
freq       4        NaN       NaN          2
mean     NaN   4.250000  3.000000        NaN
std      NaN   1.035098  1.309307        NaN
min      NaN   3.000000  1.000000        NaN
25%      NaN   3.750000  2.000000        NaN
50%      NaN   4.000000  3.000000        NaN
75%      NaN   5.000000  4.000000        NaN
max      NaN   6.000000  5.000000        NaN
```

The describe() method uses appropriate summary statistics for each data type.

Data Manipulation with pandas

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

-
-
-

Column Selection

As briefly explained in the previous session, in pandas, we select columns using different notation. To select a single column:

```
df['modes']
```

```
0     car
1     bus
2    walk
3    walk
4     car
5    walk
6    walk
7     car
Name: modes, dtype: category
Categories (3, object): [’bus’, ’car’, ’walk’]
```

To select multiple columns

```
df[['modes', 'safe']]
```

```
  modes  safe
0   car     5
1   bus     1
2  walk     2
3  walk     3
4   car     4
5  walk     2
6  walk     3
7   car     4
```

To select all columns except specific ones:

```
df.drop(columns=['modes'])
```

```
   frequency  safe         safe_label
0          6     5     Strongly Agree
1          3     1  Strongly Disagree
2          4     2           Disagree
3          5     3            Neutral
4          4     4              Agree
5          3     2           Disagree
6          5     3            Neutral
7          4     4              Agree
```

To reorder columns:

```
df = df[['modes', 'frequency', 'safe', 'safe_label']]
df
```

```
  modes  frequency  safe         safe_label
0   car          6     5     Strongly Agree
1   bus          3     1  Strongly Disagree
2  walk          4     2           Disagree
3  walk          5     3            Neutral
4   car          4     4              Agree
5  walk          3     2           Disagree
6  walk          5     3            Neutral
7   car          4     4              Agree
```

Creating new variables

Often we wish to create and add new variables to a data frame. For example, our sample data frame does not include an explicit identifier for the respondents. We can add one by doing:

```
df['id'] = pd.Categorical(range(1, len(df) + 1))
```

Function `len()` 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.

```
df
```

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

Suppose that we wanted to convert the variable `frequency()` from days per week to proportion of the week that the mode is used. To create a new calculated column:

```
df['frequency_prop'] = df['frequency'] / 7
```

To change the values of an existing column:

```
#df['frequency'] = df['frequency'] / 7
```

Row Selection and Filtering

In Pandas, we also select rows using different notation.

Suppose that we want the first row from the table:

```
df.iloc[[0]]
```

```
  modes  frequency  safe      safe_label id  frequency_prop
0   car          6     5  Strongly Agree  1        0.857143
```

Or the first two rows:

```
df.iloc[0:2]
```

```
  modes  frequency  safe         safe_label id  frequency_prop
0   car          6     5     Strongly Agree  1        0.857143
1   bus          3     1  Strongly Disagree  2        0.428571
```

Or rows 1, 3, and 5:

```
df.iloc[[0, 2, 4]]  # rows 1, 3, and 5 (0 indexed)
```

```
  modes  frequency  safe      safe_label id  frequency_prop
0   car          6     5  Strongly Agree  1        0.857143
2  walk          4     2        Disagree  3        0.571429
4   car          4     4           Agree  5        0.571429
```

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

```
df[df['modes'] == 'walk']
```

```
  modes  frequency  safe safe_label id  frequency_prop
2  walk          4     2   Disagree  3        0.571429
3  walk          5     3    Neutral  4        0.714286
5  walk          3     2   Disagree  6        0.428571
6  walk          5     3    Neutral  7        0.714286
```

Summarizing based on specific conditions

Now, let's say we are interested in obtaining statistics only for cases of passengers who mentioned walk as transportation mode. We can do this by filtering the table based on the `modes` value and then asking pandas to describe the resulting selection:

```
df[df['modes'] == 'walk'].describe(include = 'all')
```

```
       modes  frequency     safe safe_label   id  frequency_prop
count      4   4.000000  4.00000          4  4.0        4.000000
unique     1        NaN      NaN          2  4.0             NaN
top     walk        NaN      NaN   Disagree  3.0             NaN
freq       4        NaN      NaN          2  1.0             NaN
mean     NaN   4.250000  2.50000        NaN  NaN        0.607143
std      NaN   0.957427  0.57735        NaN  NaN        0.136775
min      NaN   3.000000  2.00000        NaN  NaN        0.428571
25%      NaN   3.750000  2.00000        NaN  NaN        0.535714
50%      NaN   4.500000  2.50000        NaN  NaN        0.642857
75%      NaN   5.000000  3.00000        NaN  NaN        0.714286
max      NaN   5.000000  3.00000        NaN  NaN        0.714286
```

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 add up the number of permanent private domiciles in some municipalities in São Paulo. To achieve this you probably subset the table three times (one for each municipality), and then calculated the sum of private domiciles.

A more elegant approach is to create groups and to summarize by group. Suppose that we would like to obtain general statistics about my data set but by mode:

```
df.groupby('modes', observed = True).describe().reset_index()
```

```
  modes frequency                                                     safe  \
            count      mean       std  min   25%  50%  75%  max count
0   bus       1.0  3.000000       NaN  3.0  3.00  3.0  3.0  3.0   1.0
1   car       3.0  4.666667  1.154701  4.0  4.00  4.0  5.0  6.0   3.0
2  walk       4.0  4.250000  0.957427  3.0  3.75  4.5  5.0  5.0   4.0

                                                 frequency_prop            \
       mean      std  min  25%  50%  75%  max          count      mean
0  1.000000      NaN  1.0  1.0  1.0  1.0  1.0            1.0  0.428571
1  4.333333  0.57735  4.0  4.0  4.0  4.5  5.0            3.0  0.666667
2  2.500000  0.57735  2.0  2.0  2.5  3.0  3.0            4.0  0.607143


        std       min       25%       50%       75%       max
0       NaN  0.428571  0.428571  0.428571  0.428571  0.428571
1  0.164957  0.571429  0.571429  0.571429  0.714286  0.857143
2  0.136775  0.428571  0.535714  0.642857  0.714286  0.714286
```

The reset_index() method converts the groupby result back to a regular DataFrame.

Now, suppose that we would like to know what is the mean proportion of use of modes of travel but by mode:

```
df.groupby('modes', observed = True)['frequency'].mean().reset_index()
```

```
  modes  frequency
0   bus   3.000000
1   car   4.666667
2  walk   4.250000
```

It is possible to group by various variables, for example:

```
df.groupby(['modes', 'safe_label'], observed = True)['frequency'].mean().reset_index()
```

```
  modes         safe_label  frequency
0   bus  Strongly Disagree        3.0
1   car              Agree        4.0
2   car     Strongly Agree        6.0
3  walk           Disagree        3.5
4  walk            Neutral        5.0
```

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 method `concat()` with `axis = 1`. This function 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 = pd.DataFrame({
    'age': [25, 32, 39, 28, 40, 33, 21, 32],
    'gender': pd.Categorical(['male', 'female', 'female', 'non-binary', 'male', 'female', 'female', 'mal
})
```

The two columns are combined as follows:

```
df_combined = pd.concat([df, df_2], axis=1)
df_combined
```

```
  modes  frequency  safe         safe_label id  frequency_prop  age  \
0   car          6     5     Strongly Agree  1        0.857143   25
1   bus          3     1  Strongly Disagree  2        0.428571   32
2  walk          4     2           Disagree  3        0.571429   39
3  walk          5     3            Neutral  4        0.714286   28
4   car          4     4              Agree  5        0.571429   40
5  walk          3     2           Disagree  6        0.428571   33
6  walk          5     3            Neutral  7        0.714286   21
7   car          4     4              Agree  8        0.571429   32

       gender
0        male
1      female
2      female
3  non-binary
4        male
5      female
6      female
7        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 = pd.DataFrame({
    'id': pd.Categorical([5, 4, 8, 1, 7, 2, 3, 6]),
    'age': [25, 32, 39, 28, 40, 33, 21, 32],
    'gender': pd.Categorical(['male', 'female', 'female', 'non-binary', 'male', 'female', 'female', 'mal
})

df_2
```

```
  id  age      gender
0  5   25        male
1  4   32      female
2  8   39      female
3  1   28  non-binary
4  7   40        male
5  2   33      female
6  3   21      female
7  6   32        male
```

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

```
pd.concat([df, df_2], axis=1)
```

```
  modes  frequency  safe         safe_label id  frequency_prop id  age  \
0   car          6     5     Strongly Agree  1        0.857143  5   25   
1   bus          3     1  Strongly Disagree  2        0.428571  4   32   
2  walk          4     2           Disagree  3        0.571429  8   39   
3  walk          5     3            Neutral  4        0.714286  1   28   
4   car          4     4              Agree  5        0.571429  7   40   
5  walk          3     2           Disagree  6        0.428571  2   33   
6  walk          5     3            Neutral  7        0.714286  3   21   
7   car          4     4              Agree  8        0.571429  6   32   

       gender  
0        male  
1      female  
2      female  
3  non-binary  
4        male  
5      female  
6      female  
7        male  
```

Merge functions combine tables based on one or more *keys*, that is, common variables. For example, `df` and `df_1` have a common id. The method `merge()` 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):

```
df_merged = df.merge(df_2, on='id', how='left')
df_merged
```

```
  modes  frequency  safe         safe_label id  frequency_prop  age  \
0   car          6     5     Strongly Agree  1        0.857143   28   
1   bus          3     1  Strongly Disagree  2        0.428571   33   
2  walk          4     2           Disagree  3        0.571429   21   
3  walk          5     3            Neutral  4        0.714286   32   
4   car          4     4              Agree  5        0.571429   25   
5  walk          3     2           Disagree  6        0.428571   32   
6  walk          5     3            Neutral  7        0.714286   40   
7   car          4     4              Agree  8        0.571429   39   

       gender  
0  non-binary  
1      female  
2      female  
3      female  
4        male  
5        male  
6        male  
7      female  
```

Check how now the individual attributes match correctly the rows in the left table. Other types of joins are available: `how = 'right'`, `how = 'inner'`, and `how = 'outer'`.

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:

```
df_3 = pd.DataFrame({'id': pd.Categorical([9, 10]), 'modes': pd.Categorical(['bus', 'walk'], categories=


# add safe labels and frequency proportion
df_3['safe_label'] = df_3['safe'].map(safe_labels)
df_3['safe_label'] = pd.Categorical(df_3['safe_label'],
                                   categories=["Strongly Disagree", "Disagree", "Neutral", "Agree", "St
                                   ordered=True)
df_3['frequency_prop'] = df_3['frequency'] / 7

df_3
```

```
   id modes  frequency safe safe_label  frequency_prop
0   9   bus          2    2   Disagree        0.285714
1  10  walk          5    4      Agree        0.714286
```

Now we want to combine the tables not by adding columns but by adding rows. The appropriate function is `concat()`, with `axis = 0`, and it combines the table by joining the second data frame to the first. Notice that the bind matches by column name, so it does not matter if the columns are in the same order:

```
df_all = pd.concat([df, df_3], axis = 0, ignore_index=True)
df_all
```

```
  modes  frequency  safe         safe_label  id  frequency_prop
0   car          6     5     Strongly Agree   1        0.857143
1   bus          3     1  Strongly Disagree   2        0.428571
2  walk          4     2           Disagree   3        0.571429
3  walk          5     3            Neutral   4        0.714286
4   car          4     4              Agree   5        0.571429
5  walk          3     2           Disagree   6        0.428571
6  walk          5     3            Neutral   7        0.714286
7   car          4     4              Agree   8        0.571429
8   bus          2     2           Disagree   9        0.285714
9  walk          5     4              Agree  10        0.714286
```

Pandas is generally flexible about data type mismatches when combining tables, often converting to a common type. However, for categorical variables, it's best to ensure consistent category definitions:

```
df_3_mismatch = df_3.copy()
df_3_mismatch['id'] = [9, 10]
```

In the previous chunk, we created a new DataFrame with the same data of da_3. However, we set the `id` variable as integer instead of categorical.

Now, performing the concat to merge the rows:

```
df_mismatched = pd.concat([df, df_3_mismatch], axis = 0, ignore_index=True)
df_mismatched
```

```
  modes  frequency  safe         safe_label  id  frequency_prop
0   car          6     5     Strongly Agree   1        0.857143
1   bus          3     1  Strongly Disagree   2        0.428571
2  walk          4     2           Disagree   3        0.571429
3  walk          5     3            Neutral   4        0.714286
4   car          4     4              Agree   5        0.571429
```

```
5  walk          3     2           Disagree   6        0.428571
6  walk          5     3            Neutral   7        0.714286
7   car          4     4              Agree   8        0.571429
8   bus          2     2           Disagree   9        0.285714
9  walk          5     4              Agree  10        0.714286
```

Pandas was able to perform the concat. However, the variable id was transformed into a numerical variable. We can check this evaluating the types of each column:

```
df_mismatched.dtypes
```

```
modes             category
frequency            int64
safe                 int64
safe_label        category
id                   int64
frequency_prop     float64
dtype: object
```

Practice

1. Summarize (describe) table `cntr_sp_head`. What are the scales of measurement of the variables in this table?
2. Summarize table `cntr_sp_basico`. What are the scales of measurement of the variables in this table?
3. Join the following tables using an appropriate key variable: `cntr_sp_ipvs`, `cntr_sp_basico`, `cntr_sp_head`.
4. Create a new table with only variables `code_tract`, `situacao`, `AGSN`, `IPVS`, and `b_V005`.
5. Describe statistics of the new table.
6. Obtain statistics of the new table but only for properties of `IPVS` "Baixíssima vulnerabilidade" ("Extremely low vulnerability").
7. Obtain a summary of the new table but only for properties *not* of `IPVS` "Baixíssima vulnerabilidade" ("Extremely low vulnerability").
8. What is the mean `b_V005` by `IPVS` of census tract?