Aske Svane Qvist

W39Wed: Data with Pandas

By now, you are comfortable working with numbers in Python. You can add, divide, subtract, multiply numerical variables and that's all fine and well. But what if, instead of single values, you had a big table of numbers you needed to perform some operation on? For that, we need a framework for working with tabular data, and for this the Python state-of-the-art is calles *Pandas*.

0. Introduction to tabular data: `DataFrame`s

Pandas is in fact very simple. It let's you represent a table of data as something called a `DataFrame`. Then, when you have your data stored in a `DataFrame`, you can do all sorts of neat things with it, with very little code.

Here is an example of a Pandas `DataFrame`:

In [28]:
```
# Import `pandas` and give it the shorter, more convenient, alias `pd`
import pandas as pd

# Create a `DataFrame` with 3 rows and 3 columns, containing the numbers from 1 t
df = pd.DataFrame([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# Display `DataFrame`. Note that, in Jupyter notebooks, we don't use `print` wher
# displaying dataframes. Try it and you will understand why!
df
```

Out[28]:

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

The **bold** numbers are the *indices* and *columns*. Everything else is data. An index refers to a **horizontal row** and a column referes to a **vertical column**. In Pandas we can assign names to indices and columns. That's pretty useful, because often rows and columns actually represent real things that can be named. For example:

In [29]:
```
# Assign index names
df.index = ['Gertrude', 'Ludwig', 'Bertrand']

# Assign column names
df.columns = ['Age', 'Height', 'Weight']

# Display dataframe
df
```

Out[29]:

	Age	Height	Weight
Gertrude	1	2	3
Ludwig	4	5	6
Bertrand	7	8	9

Almost always, the dogma is that rows contain "observations", and columns represent "attributes". For your reference, here are some of the words that are used interchangably for rows and columns:

Row are sometimes called:

- Observations
- Measurements
- Datapoints
- Samples
- Examples

Columns are sometimes called:

- Features
- Attributes
- Dimensions
- Variable

I like to think of rows as individual datapoints, and columns as the attributes that describe them. Together, the datapoints make up your **dataset**. For example, we observe one 'Gertrude' datapoint that is described by age=1, height=2 and weight=3. We can describe all other datapoints by these attributes, and that is our dataset.

1. Exploring different ways of creating pandas DataFrames

As you may have heard already there are multiple approaches to create pandas DataFrames. Here, we will explore some of them.

Problem 1.1: Create an empty dataframe

In [4]:
```
df = pd.DataFrame()
```

Out[4]: —

Problem 1.2: Create a dataframe from a list of strings or from a list of lists. Then, rename the column(s).

In [9]:
```
# A list of lists
listA = [[234,543,576,98,32],["23","54","87","45","1"]]

# to df
df = pd.DataFrame(listA)
```

In [11]:
```
# Create column names
df.columns = ["a","b","c","d","e"]

df
```

Out[11]:

	a	b	c	d	e
0	234	543	576	98	32
1	23	54	87	45	1

Problem 1.3: Create a dataframe from a dictionary. Then, change the row names.

In [18]:
```
# Create Dictionary
aske = {"name": ["Aske","Andreas"], "age": [24,28], "city": ["Aarhus","Copenhager

# Change to df
df_from_dict = pd.DataFrame(aske)

df_from_dict
```

Out[18]:

	name	age	city
0	Aske	24	Aarhus
1	Andreas	28	Copenhagen

In [20]:
```
# use .index to change the row names
df_from_dict.index = ["guy1","guy2"]

df_from_dict
```

Out[20]:

	name	age	city
guy1	Aske	24	Aarhus
guy2	Andreas	28	Copenhagen

Problem 1.4: Create a dataframe from the sales.csv file available on Absalon.

In [24]:
```
# Import data
data = pd.read_csv("data/sales.csv")

# check content
data[:2]
```

Out[24]:

	product_name	number_of_sales	price	total_amount
0	product1	2	3000	6000
1	product2	4	450	1800

Problem 1.5: What is the difference between read_csv() and read_table(). How would you apply this two methods to get the same result.

Try read_table() method to load the file that you used previously.

The difference is the format of the file you are trying to import; the read_csv() will expect comma-seperated data, while the read_table() will expect tabl-seperated data

In [26]:
```
# Import data with read_table()
data = pd.read_table("data/sales.csv")

# check
data
```

Out[26]:

	product_name,number_of_sales,price,total_amount
0	product1,2,3000,6000
1	product2,4,450,1800
2	product3,5,300,67
3	product4,1,34,68

> Since read_table() does not consider a comma as seperating values, every row will just be considered as belonging to a single column.

2. Selecting

2.1. Rows columns and values

On a more practical note, how do we retrieve a specific row, column or value from your dataframe? **Answer:** We use the `loc` property on the `DataFrame` type object, `df`, that we created. Read about how to use this in the Pandas documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html). Alternatively, use this as an opportunity to practice Googling.

The approach to using the `loc` property (which admittedly behaves a bit like a method) is like this: `df.loc[index_name, column_name]`

> **Problem 2.1.1:** Select the "Gertrude" row from `df` - the dataframe created in "0. Introduction to tabular data: Dataframes" - using the `loc` property.

In [32]:
```
# Create a `DataFrame` with 3 rows and 3 columns, containing the numbers from 1 t
df = pd.DataFrame([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
# Assign index names
df.index = ['Gertrude', 'Ludwig', 'Bertrand']
# Assign column names
df.columns = ['Age', 'Height', 'Weight']
df
```

Out[32]:

	Age	Height	Weight
Gertrude	1	2	3
Ludwig	4	5	6
Bertrand	7	8	9

In [41]:
```
# With the hard brackets, I can select an index from the df. I get all the values
df.loc['Gertrude']
```

Out[41]:
```
Age       1
Height    2
Weight    3
Name: Gertrude, dtype: int64
```

`loc` can also be used to select a single desired value.

Problem 2.1.2: Select the "Weight" attribute of "Ludwig".

In [37]:
```
# The first hard bracket is the index, the second is the column
df.loc['Ludwig']["Weight"]
```

Out[37]:
```
6
```

And of course you can also select all the values of a column. You can do it like this:

In [38]:
```
df.loc[:, 'Height']
```

Out[38]:
```
Gertrude    2
Ludwig      5
Bertrand    8
Name: Height, dtype: int64
```

Problem 2.1.3: Explain `df.loc[:, 'Height']`. What does `:` mean?

Note: alternatively you can select the column like this: `df['Height']`, or even `df.Height`.

In [42]:
```
# The : indicates the index range which we want to extract.
# Since there are no values specified - it must be all
df.loc["Gertrude":"Ludwig", "Height"]
```

Out[42]:
```
Gertrude    2
Ludwig      5
Name: Height, dtype: int64
```

Problem 2.1.4: Select simultaneously, the two columns 'Height' and 'Weight'.

In [47]:
```
# We have hard brackets inside - it is a list of columns which we want to extract
df[["Height","Weight"]]
```

Out[47]:

	Height	Weight
Gertrude	2	3
Ludwig	5	6
Bertrand	8	9

2.2. Conditional selection

To communicate this, let's load some slightly more interesting data. Note that for this data, no index names are available. Therefore, we just keep the default row numbering from 0 and up (who cares what wines are called?).

In [154]:
```
# Import a dataset of wines from the `sklearn` library
from sklearn.datasets import load_wine

# Load it
data = load_wine()

# Put it into a `DataFrame`, and assign column names
wines_df = pd.DataFrame(
    data['data'],
    columns=data['feature_names']
)

# Display the data
wines_df
```

Out[154]:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflav
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	
...	...	...	...	...	...	...	...	
173	13.71	5.65	2.45	20.5	95.0	1.68	0.61	
174	13.40	3.91	2.48	23.0	102.0	1.80	0.75	
175	13.27	4.28	2.26	20.0	120.0	1.59	0.69	
176	13.17	2.59	2.37	20.0	120.0	1.65	0.68	
177	14.13	4.10	2.74	24.5	96.0	2.05	0.76	

Remember the `:` operator? I'll give it away here assuming you figured it out above: it means "everything" or "all indices". So, for example, `wine_df.loc[:]` would select all rows. But what if we only wanted to select some of the rows, maybe the wines with high alcohol content or low magnesium?

To do that, rather than selecting "all indices" we substitute `:` with an array of `bool`s, which at each index says whether to include the row or not in the selection. If that sounds abstract, here's an example:

Notice two things about how Pandas displays the data:

1. When you try to display a big `DataFrame`, it only displays enough for you to get idea of what the data looks like. Imagine your dataframe had millions of rows and thousands of columns (not an uncommon scenario you will learn), obviously you should not display everything.
2. It summarizes the size of the `DataFrame` at the bottom of the table. **Pretty useful!**

In [81]:
```
# List of booleans telling us if a datapoint has above 14% alcohol
strong_wines = wines_df['alcohol'] > 14.0

# Display
strong_wines
```

Out[81]:
```
0       True
1      False
2      False
3       True
4      False
       ...
173    False
174    False
175    False
176    False
177     True
Name: alcohol, Length: 178, dtype: bool
```

Problem 2.2.1:

1. Output the number of rows and columns of the wines_df to get a better idea of the data you are working with.
2. Check for the missing values and return if there are missing values in any of the columns.

In [82]:
```
wines_df.shape
# 178 rows and 13 columns
```

Out[82]:
```
(178, 13)
```

In [83]:
```
# isnull() will check if there is a value (true/false)
# sum() count the number of times true appears.
wines_df.isnull().sum()
```

Out[83]:
```
alcohol                         0
malic_acid                      0
ash                             0
alcalinity_of_ash               0
magnesium                       0
total_phenols                   0
flavanoids                      0
nonflavanoid_phenols            0
proanthocyanins                 0
color_intensity                 0
hue                             0
od280/od315_of_diluted_wines    0
proline                         0
dtype: int64
```

There are no missing values

In [84]:
```
wines_df.isnull().values.any()
```

Out[84]:
```
False
```

> **Problem 2.2.2:** Use the `strong_wines` list of `bool`s to select only the strong wines in `wine_df`. Explain how using a bool array to select row indices works in your intuition.

In [85]:
```
# subset using the boolean list of strong wines
strong_wines_df = wines_df.loc[strong_wines]
```

In [86]:
```
# Print minimum percentage
strong_wines_df["alcohol"].min()
```

Out[86]:
```
14.02
```

All the rows with a corresponding 'True' value in the strong_wines list will be selected from wines_df and stored in the dataframe strong_wines_df

> **Problem 2.2.3:** Print the malic acid and flavanoids for wines 98-109 (both included).

In [90]:
```
wines_df[98:109][["malic_acid","flavanoids"]]
```

Out[90]:

	malic_acid	flavanoids
98	1.07	3.75
99	3.17	2.99
100	2.08	2.17
101	1.34	1.36
102	2.45	2.11
103	1.72	1.64
104	1.73	1.92
105	2.55	1.84
106	1.73	2.03
107	1.75	1.76
108	1.29	2.04

Problem 2.2.4: Select the "alcohol" column of wines that have "flavanoids" < 0.5.

In [98]:
```
# For wines_df select the rows where the values in flavanoids (in wines_df) is sm
# then only take the column alcohol
wines_df[wines_df["flavanoids"] < 0.5]["alcohol"]
```

Out[98]:
```
136    12.25
138    13.49
146    13.88
165    13.73
170    12.20
Name: alcohol, dtype: float64
```

Problem 2.2.5: Think about what the `loc` property lets you do. You have used it to select specific indices by name, and selections of indices with a list of booleans. Do you think you could do the same with column selection? Explain how this works. What if you wanted to select wines on multiple conditions, for example an alcohol content higher than some value **and** a flavanoid content lower than some other value? Would that be possible, and if so, how?

In [133]:
```
# loc can be used for column selection as well, but since it considers rows first
# add a range of rows, then a comma, and finally the columns we are interested in
selected_wines = wines_df.loc[:5,["alcohol","flavanoids"]]
print(selected_wines)

# Additionally, one can select columns with booleans as well.
selected_wines = selected_wines.loc[:3, [False, True]]
print(selected_wines)
```

```
   alcohol  flavanoids
0    14.23        3.06
1    13.20        2.76
2    13.16        3.24
3    14.37        3.49
4    13.24        2.69
5    14.20        3.39
   flavanoids
0        3.06
1        2.76
2        3.24
3        3.49
```

In [108]:
```
# Each condition has to be in parentheses
wines_df[(wines_df["flavanoids"] < 0.6) & (wines_df["alcohol"] > 13.0)]
```

Out[108]:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavano
138	13.49	3.59	2.19	19.5	88.0	1.62	0.48	
141	13.36	2.56	2.35	20.0	89.0	1.40	0.50	
142	13.52	3.17	2.72	23.5	97.0	1.55	0.52	
145	13.16	3.57	2.15	21.0	102.0	1.50	0.55	
146	13.88	5.04	2.23	20.0	80.0	0.98	0.34	
161	13.69	3.26	2.54	20.0	107.0	1.83	0.56	
165	13.73	4.36	2.26	22.5	88.0	1.28	0.47	

Problem 2.2.6: Write a program that looks for the wine attributes alcohol, color intensity and alcalinity of ash. If the attribute matches any of these, compute the range - that is, the maximum value in that column for all the wines minus the minimum value - and the average. Print the obtained results.

In [141]:
```
# Function to calculate teh range and average
def calculator(attribute):
    # get values in column
    column = wines_df[attribute]
    # Calculate range and average
    cRange = column.max() - column.min()
    cAvg = column.mean()
    # Return values
    return (cRange, cAvg)

# Which attributes
attributes = ["alcohol", "color_intensity","alcalinity_of_ash"]

# Loop over all attributes and print calculations in a comprehensible manner
for attribute in attributes:
    Range, Avg = calculator(attribute)
    print(f"For {attribute}, the range is {Range} and the average is {Avg}.")
```

```
For alcohol, the range is 3.8000000000000007 and the average is 13.000617977528
083.
For color_intensity, the range is 11.72 and the average is 5.058089882022473.
For alcalinity_of_ash, the range is 19.4 and the average is 19.49494382022472.
```

Problem 2.2.7: Write a program that classifies the wines into good quality - that is, 1 - and bad quality - that is, 0 - according to alcohol and color intensity. If alcohol higher or equal to 13.7 and color intensity higher or equal to 7.3 the wine has good quality. On the other hand, if those values are lower than the mentioned, the wine has bad quality. **Do this by creating a new column on the existing dataframe.** Then, output how many wines have a good and bad quality in total.

In [181]:
```
# Container for evaluation
good_quality = []

# Loop over every row
for index, row in wines_df.iterrows():

    # check conditions and append dummies to list outside the loop
    if (row["alcohol"] >= 13.7) & (row["color_intensity"] >= 7.3):
        good_quality.append(1)
    else:
        good_quality.append(0)

# append the list as a new column
wines_df["good_quality"] = good_quality

# Quick check
wines_df[:10][["alcohol", "color_intensity", "good_quality"]]
```

Out[181]:

	alcohol	color_intensity	good_quality
0	14.23	5.64	0
1	13.20	4.38	0
2	13.16	5.68	0
3	14.37	7.80	1
4	13.24	4.32	0
5	14.20	6.75	0
6	14.39	5.25	0
7	14.06	5.05	0
8	14.83	5.20	0
9	13.86	7.22	0

In [179]:
```
# Count number of good and bad wines
wines_df["good_quality"].value_counts()
```

Out[179]:
```
0    168
1     10
Name: good_quality, dtype: int64
```

Problem 2.2.8: Now try deleting from the dataframe all those wines that have bad quality.

In [191]:
```
# Use ["good_quality"] == 1 as condition to select only the good wines
good_wines = wines_df.loc[wines_df["good_quality"] == 1]

# check
good_wines[:5]
```

Out[191]:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavano
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	
14	14.38	1.87	2.38	12.0	102.0	3.30	3.64	
18	14.19	1.59	2.48	16.5	108.0	3.30	3.93	
49	13.94	1.73	2.27	17.4	108.0	2.88	3.54	
156	13.84	4.12	2.38	19.5	89.0	1.80	0.83	

Problem 2.2.9: From this new dataframe, if magnesium is below 98 and if flavanoids are below 0.70, rechange the quality score to "high quality".

In [192]:
```
# using loc to conditionally overwrite ones in the quality column that live up to
good_wines.loc[(good_wines['magnesium'] < 98) & (good_wines['flavanoids'] < 0.70)

# Check
good_wines
```

```
C:\Users\askes\anaconda3\lib\site-packages\pandas\core\indexing.py:966: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  self.obj[item] = s
```

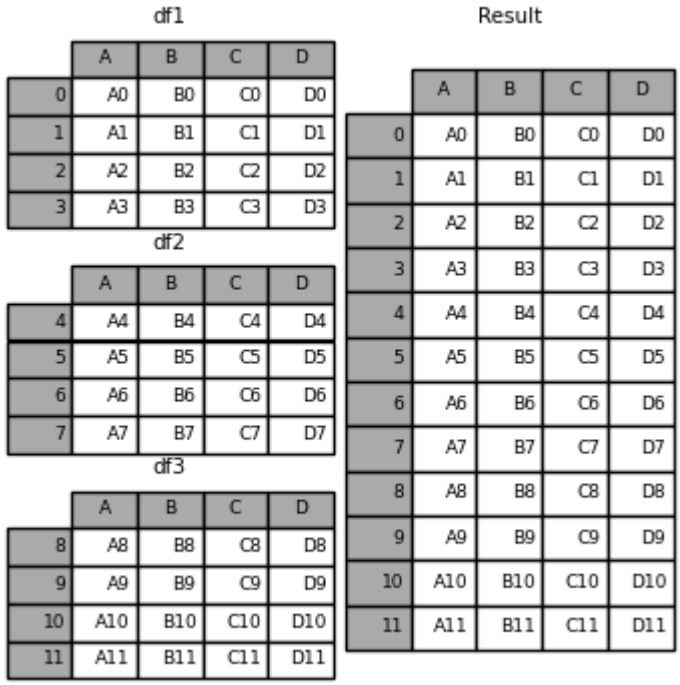
Out[192]:

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavano
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	
14	14.38	1.87	2.38	12.0	102.0	3.30	3.64	
18	14.19	1.59	2.48	16.5	108.0	3.30	3.93	
49	13.94	1.73	2.27	17.4	108.0	2.88	3.54	
156	13.84	4.12	2.38	19.5	89.0	1.80	0.83	
158	14.34	1.68	2.70	25.0	98.0	2.80	1.31	
164	13.78	2.76	2.30	22.0	90.0	1.35	0.68	
172	14.16	2.51	2.48	20.0	91.0	1.68	0.70	
173	13.71	5.65	2.45	20.5	95.0	1.68	0.61	
177	14.13	4.10	2.74	24.5	96.0	2.05	0.76	

3. Merging `DataFrame`s

It's not uncommon that we need to merge tables together into a new, bigger table. Sometimes all pieces have the same shape and column names so we can **join them vertically**.

```
pd.concat([df1, df2, df3], axis=0)
```



	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

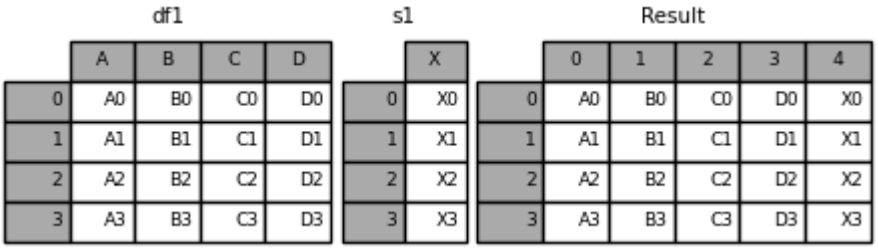
	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Of course you could also **join them horizontally**:

```
pd.concat([df1, s1], axis=1)
```



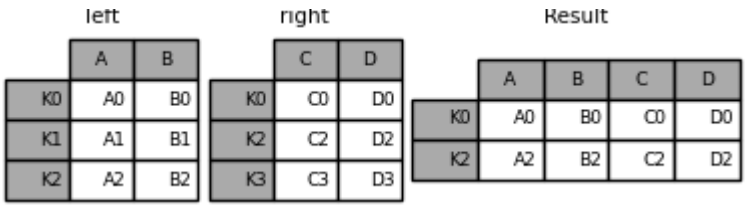
	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

	X
0	X0
1	X1
2	X2
3	X3

	0	1	2	3	4
0	A0	B0	C0	D0	X0
1	A1	B1	C1	D1	X1
2	A2	B2	C2	D2	X2
3	A3	B3	C3	D3	X3

These are straight forward examples, but there are all sorts of things that could complicate your decision about how exactly you want to join `DataFrame`s. For example, what if you wanted to join horizontally (maybe to add more attributes to your datapoints), but only keep the rows that existed in all `DataFrame`s? This is called an **inner join**.

```
pd.concat([left, right], axis=1, join='inner')
```



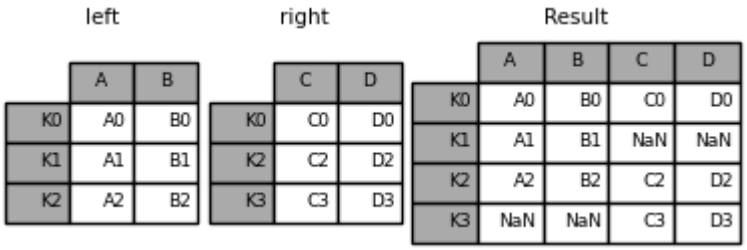
	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

	A	B	C	D
K0	A0	B0	C0	D0
K2	A2	B2	C2	D2

The opposite of an inner join is called an **outer join**. It keeps all rows from the joined tables and fills in `NaN` s where data does not exist (`NaN` is short for "not a number").

```
pd.concat([left, right], axis=1, join='outer')
```



	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

Check out [the official Pandas documentation (https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html)](https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html) for merging, joining and concatenating `DataFrame` s.

Problem 3.1: Take a moment to carefully understand the above examples, and in particular reflect on the meaning of setting `axis=0` or `1` , and setting `join='inner'` or `'outer'` . Make sure you understand this well enough that you can explain it to someone.

- `axis=0` is merging vertically
- `axis=1` is merging horizontally
- `join='inner'` Only keep the rows/columns with information from both dataframes
- `join='outer'` Keep all information and fill in NA's in case there are missing values

Problem 3.2: Take the `left` and `right` tables below and join them, not horizontally, but vertically and outer. Explain the result you get.

In [194]:
```
left = pd.DataFrame({
    'A': ['A0', 'A1', 'A2', 'A3'],
    'B': ['B0', 'B1', 'B2', 'B3']
})

right = pd.DataFrame({
    'C': ['C0', 'C1', 'C2', 'C3'],
    'D': ['D0', 'D1', 'D2', 'D3']
})
```

In [203]:
```
pd.concat([left,right], axis = 0, join = "outer")
```

Out[203]:

	A	B	C	D
0	A0	B0	NaN	NaN
1	A1	B1	NaN	NaN
2	A2	B2	NaN	NaN
3	A3	B3	NaN	NaN
0	NaN	NaN	C0	D0
1	NaN	NaN	C1	D1
2	NaN	NaN	C2	D2
3	NaN	NaN	C3	D3

Since this is an outer join, all column are kept even though there is no data for many of the indices. Also index names are repeated. This has implications for selecting index values, as e.g. selecting `0` with `loc` would select both `0` rows.

> **Problem 3.3:** Try to imagine some cases where you would need to join tables. I'll inspire your thinking. Maybe you collect questionnaire data each week on some subject from a large population, and you need to continuously join the new data into the existing dataframe. Can you think of other examples? See if you can think of examples both for inner and outer joins.

- If you have different information attributes on a group of participants, but the two groups don't correspond exactly to each other; I guess you could use inner join to keep only participants appearing in both datasets. That would be for horizontal merge.
- For vertical merge, inner join could be used if you have a lot of different participants and a lot of various parameters with information on them stored in two datasets. You only want to keep the columns (hte parameters) appearing in both datasets, since these are the once you are going to employ.
- Outer join is just if you have different information in two datasets, and even though you have missing values, you don't want to throw away any data.

4. Daily reflections

> **Task 4.1** Take a moment to reflect on your learning experience today and take notes. You *can* use these prompts to inspire your reflections:
>
> - What (if anything) did you take away from the lecture and exercise today?
> - What concepts, ideas, or topics are still unclear?
> - Are there any things you would have wanted to spend less or more time on? What are they?
> - Is there anything you have become inspired to follow up on from today's lecture and exercise? Anything you are looking forward to learning more about?

- I have been working a lot with pandas - but never really knew what I was doing. Basically just googling around. I have been more used to data processing in R which is somehow still a bit more intuitive for me. However, going over the basics was really nice. Even though there has been nothing completely new, it was really helpful to actually get it explained and not just employing code that I would not intuitively write myself.

5. Bonus exercise

> **Task 5.1** Try to inspect another dataframe that suits your interests - although we recommend that you use the SODAS dataframe, as you already know much about it. You have seen in previous exercises where to find them. You can try to check the shape of it, add new columns, remove existing ones... There is a lot of stuff than can be done, so let's get to work!

In [ ]: