Pandas (continues)

-	-	-
Exercise 1 (cities)	Exercise 2 (powers of series)	Exercise 3 (municipal information)
Exercise 4 (municipalities of finland)	Exercise 5 (swedish and foreigners)	Exercise 6 (growing municipalities)
Exercise 7 (subsetting with loc)	Exercise 8 (subsetting by positions)	Exercise 9 (snow depth)
Exercise 10 (average temperature)	Exercise 11 (below zero)	Exercise 12 (cyclists)
Exercise 13 (missing value types)	Exercise 14 (special missing values)	Exercise 15 (last week)
Exercise 16 (split date)	Exercise 17 (cleaning data)	

Pandas (continues)

```
import pandas as pd
import numpy as np
```

Creation of dataframes

The DataFrame is essentially a two dimensional object, and it can be created in three different ways:

- out of a two dimensional NumPy array
- out of given columns
- out of given rows

Creating DataFrames from a NumPy array

In the following example a DataFrame with 2 rows and 3 column is created. The row and column indices are given explicitly.

```
df=pd.DataFrame(np.random.randn(2,3), columns=["First", "Second", "Third"], index=["a"
df
```

	First	Second	Third
a	1.273012	-1.645268	0.133877
b	0.742194	-0.225893	2.600842

Note that now both the rows and columns can be accessed using the special `Index` object:

```
df.index                          # These are the "row names"
```

```
Index(['a', 'b'], dtype='object')
```

```
df.columns                        # These are the "column names"
```

```
Index(['First', 'Second', 'Third'], dtype='object')
```

If either `columns` or `index` argument is left out, then an implicit integer index will be used:

```
df2=pd.DataFrame(np.random.randn(2,3), index=["a", "b"])
df2
```

	0	1	2
a	0.095937	-0.688698	0.653831
b	0.978782	0.034069	1.125257

Now the column index is an object similar to Python's builtin `range` type:

```
df2.columns
```

```
RangeIndex(start=0, stop=3, step=1)
```

Creating DataFrames from columns

A column can be specified as a list, an NumPy array, or a Pandas' Series. The names of the columns can be given either with the `columns` parameter, or if Series objects are used, then the `name` attribute of each Series is used as the column name.

```
s1 = pd.Series([1,2,3])
s1
```

```
0    1
1    2
2    3
dtype: int64
```

```
s2 = pd.Series([4,5,6], name="b")
s2
```

```
0    4
1    5
2    6
Name: b, dtype: int64
```

Give the column name explicitly:

```
pd.DataFrame(s1, columns=["a"])
```

	a
0	1
1	2
2	3

Use the `name` attribute of Series s2 as the column name:

```
pd.DataFrame(s2)
```

	b
0	4
1	5
2	6

If using multiple columns, then they must be given as the dictionary, whose keys give the column names and values are the actual column content.

```
pd.DataFrame({"a": s1, "b": s2})
```

	a	b
0	1	4
1	2	5
2	3	6

Creating DataFrames from rows

We can give a list of rows as a parameter to the DataFrame constructor. Each row is given as a dict, list, Series, or NumPy array. If we want to give names for the columns, then either the rows must be dictionaries, where the key is the column name and the values are the elements of the DataFrame on that row and column, or else the column names must be given explicitly. An example of this:

```
df=pd.DataFrame([{"Wage" : 1000, "Name" : "Jack", "Age" : 21}, {"Wage" : 1500, "Name"
df
```

	Age	Name	Wage
0	21	Jack	1000
1	29	John	1500

Or:

```
df = pd.DataFrame([[1000, "Jack", 21], [1500, "John", 29]], columns=["Wage", "Name", "
df
```

	Wage	Name	Age
0	1000	Jack	21
1	1500	John	29

In the earlier case, however, where we created DataFrames from a dictionary of columns, the order of columns should be the same as in the parameter dictionary in the recent versions of Python and Pandas.

In the sense of information content the order of columns should not matter, but sometimes you want to specify a certain order to make the Frame more readable, or to make it obey some semantic meaning of column order.

Exercise 1 (cities)

Write function `cities` that returns the following DataFrame of top Finnish cities by population:

```
                Population Total area
Helsinki        643272     715.48
Espoo           279044     528.03
Tampere         231853     689.59
Vantaa          223027     240.35
Oulu            201810     3817.52
```

Exercise 2 (powers of series)

Make function `powers_of_series` that takes a Series and a positive integer k as parameters and returns a DataFrame. The resulting DataFrame should have the same index as the input Series. The first column of the dataFrame should be the input Series, the second column should contain the Series raised to power of two. The third column should contain the Series raised to the power of three, and so on until (and including) power of k. The columns should have indices from 1 to k.

The values should be numbers, but the index can have any type. Test your

function from the `main` function. Example of usage:

```
s = pd.Series([1,2,3,4], index=list("abcd"))
print(powers_of_series(s, 3))
```

Should print:

```
   1   2   3
a  1   1   1
b  2   4   8
c  3   9  27
d  4  16  64
```

Exercise 3 (municipal information)

In the `main` function load a data set of municipal information from the `src` folder (originally from Statistics Finland). Use the function `pd.read_csv`, and note that the separator is a tabulator.

Print the shape of the DataFrame (number of rows and columns) and the column names in the following format:

```
Shape: r,c
Columns:
col1
col2
...
```

Note, sometimes file ending `tsv` (tab separated values) is used instead of `csv` if the separator is a tab.

Accessing columns and rows of a dataframe

Even though DataFrames are basically just two dimensional arrays, the way to access their elements is different from NumPy arrays. There are a couple of complications, which we will go through in this section.

Firstly, the bracket notation [ ] does not allow the use of an index pair to access a single element of the DataFrame. Instead only one dimension can be specified.

Well, does this dimension specify the rows of the DataFrame, like NumPy arrays if only one index is given, or does it specify the columns of the DataFrame?

It depends!

If an integer is used, then it specifies a column of the DataFrame in the case the **explicit** indices for the column contain that integer. In any other case an error will result. For example, with the above DataFrame, the following indexing will not work, because the explicit column index consist of the column names "Name" and "Wage" which are not integers.

```
try:
    df[0]
except KeyError:
    import sys
    print("Key error", file=sys.stderr)
```

```
Key error
```

The following will however work.

```
df["Wage"]
```

```
0    1000
1    1500
Name: Wage, dtype: int64
```

As does the fancy indexing:

```
df[["Wage", "Name"]]
```

	Wage	Name
0	1000	Jack
1	1500	John

If one indexes with a slice or a boolean mask, then the **rows** are referred to. Examples of these:

```
df[0:1]                          # slice
```

	Wage	Name	Age
0	1000	Jack	21

```
df[df.Wage > 1200]               # boolean mask
```

	Wage	Name	Age
1	1500	John	29

If some of the above calls return a Series object, then you can chain the bracket calls to get a single value from the DataFrame:

```
df["Wage"][1]                    # Note order of dimensions
```

```
1500
```

But there is a better way to achieve this, which we will see in the next section.

Exercise 4 (municipalities of finland)

Load again the municipal information DataFrame. The rows of the DataFrame correspond to various geographical areas of Finland. The first row is about Finland as a whole, then rows from Akaa to Äänekoski are municipalities of Finland in alphabetical order. After that some larger regions are listed.

Write function `municipalities_of_finland` that returns a DataFrame containing only rows about municipalities. Give an appropriate argument for `pd.read_csv` so that it interprets the column about region name as the (row) index. This way you can index the DataFrame with the names of the regions.

Test your function from the `main` function.

Exercise 5 (swedish and foreigners)

Write function `swedish_and_foreigners` that

- Reads the municipalities data set
- Takes the subset about municipalities (like in previous exercise)
- Further take a subset of rows that have proportion of Swedish speaking people and proportion of foreigners both above 5 % level
- From this data set take only columns about population, the proportions of Swedish speaking people and foreigners, that is three columns.

The function should return this final DataFrame.

Do you see some kind of correlation between the columns about Swedish speaking and foreign people? Do you see correlation between the columns about the population and the proportion of Swedish speaking people in this subset?

Exercise 6 (growing municipalities)

Write function `growing_municipalities` that gets subset of municipalities (a DataFrame) as a parameter and returns the proportion of municipalities with increasing population in that subset.

Test your function from the `main` function using some subset of the municipalities. Print the proportion as percentages using 1 decimal precision.

Example output:

```
Proportion of growing municipalities: 12.4%
```

Alternative indexing and data selection

If the explanation in the previous section sounded confusing or ambiguous, or if you didn't understand a thing, you don't have to worry.

There is another way to index Pandas DataFrames, which

- allows use of index pairs to access a single element
- has the same order of dimensions as NumPy: first index specifies rows, second columns
- is not ambiguous about implicit or explicit indices

Pandas DataFrames have attributes `loc` and `iloc` that have the above qualities. You can use `loc` and `iloc` attributes and forget everything about the previous section. Or you can use these attributes and sometimes use the methods from the previous section as shortcuts if you understand them well.

The difference between `loc` and `iloc` attributes is that the former uses explicit indices and the latter uses the implicit integer indices. Examples of use:

```
df.loc[1, "Wage"]
```

```
1500
```

```
df.iloc[-1,-1]            # Right lower corner of the DataFrame
```

```
29
```

```
df.loc[1, ["Name", "Wage"]]
```

```
Name     John
Wage     1500
Name: 1, dtype: object
```

With `iloc` everything works like with NumPy arrays: indexing, slicing, fancy indexing, masking and their combinations. With `loc` it is the same but now the names in the explicit indices are used for specifying rows and

columns. Make sure your understand why the above examples work as they do!

Exercise 7 (subsetting with loc)

Write function `subsetting_with_loc` that in one go takes the subset of municipalities from Akaa to Äänekoski and restricts it to columns: "Population", "Share of Swedish-speakers of the population, %", and "Share of foreign citizens of the population, %". The function should return this content as a DataFrame. Use the attribute `loc`.

Exercise 8 (subsetting by positions)

Write function `subsetting_by_positions` that does the following.

Read the data set of the top forty singles from the beginning of the year 1964 from the `src` folder. Return the top 10 entries and only the columns `Title` and `Artist`. Get these elements by their positions, that is, by using a single call to the `iloc` attribute. The function should return these as a DataFrame.

Summary statistics

The summary statistic methods work in a similar way as their counter parts in NumPy. By default, the aggregation is done over columns.

```
wh = pd.read_csv("https://raw.githubusercontent.com/csmastersUH/data_analysis_with_pyt
```

```
wh2 = wh.drop(["Year", "m", "d"], axis=1)  # taking averages over these is not very in
wh2.mean()
```

```
Precipitation amount (mm)    1.966301
Snow depth (cm)              0.966480
Air temperature (degC)       6.527123
dtype: float64
```

The `describe` method of the `DataFrame` object gives different summary statistics for each (numeric) column. The result is a DataFrame. This method gives a good overview of the data, and is typically used in the exploratory data analysis phase.

```
wh.describe()
```

	Year	m	d	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
count	365.0	365.000000	365.000000	365.000000	358.000000	365.000000
mean	2017.0	6.526027	15.720548	1.966301	0.966480	6.527123
std	0.0	3.452584	8.808321	4.858423	3.717472	7.183934
min	2017.0	1.000000	1.000000	-1.000000	-1.000000	-17.800000
25%	2017.0	4.000000	8.000000	-1.000000	-1.000000	1.200000
50%	2017.0	7.000000	16.000000	0.200000	-1.000000	4.800000
75%	2017.0	10.000000	23.000000	2.700000	0.000000	12.900000
max	2017.0	12.000000	31.000000	35.000000	15.000000	19.600000

Exercise 9 (snow depth)

Write function `snow_depth` that reads in the weather DataFrame from the `src` folder and returns the maximum amount of snow in the year 2017.

Print the result in the `main` function in the following form:

```
Max snow depth: xx.x
```

Exercise 10 (average temperature)

Write function `average_temperature` that reads the weather data set and returns the average temperature in July.

Print the result in the `main` function in the following form:

```
Average temperature in July: xx.x
```

Exercise 11 (below zero)

Write function `below_zero` that returns the number of days when the temperature was below zero.

Print the result in the main function in the following form:

```
Number of days below zero: xx
```

Missing data

You may have noticed something strange in the output of the `describe` method. First, the minimum value in both precipitation and snow depth fields is -1. The special value -1 means that on that day there was absolutely no snow or rain, whereas the value 0 might indicate that the value was close to zero. Secondly, the snow depth column has count 358,

whereas the other columns have count 365, one measurement/value for each day of the year. How is this possible? Every field in a DataFrame should have the same number of rows. Let's use the `unique` method of the Series object to find out, which different values are used in this column:

```
wh["Snow depth (cm)"].unique()
```

```
array([ -1.,   7.,  13.,  10.,  12.,   9.,   8.,   5.,   6.,   4.
        15.,  14.,   2.,  nan,   0.])
```

The `float` type allows a special value `nan` (Not A Number), in addition to normal floating point numbers. This value can represent the result from an illegal operation. For example, the operation 0/0 can either cause an exception to occur or just silently produce a `nan`. In Pandas `nan` can be used to represent a missing value. In the weather DataFrame the `nan` value tells us that the measurement from that day is not available, possibly due to a broken measuring instrument or some other problem.

Note that only float types allow the `nan` value (in Python, NumPy or Pandas). So, if we try to create an integer series with missing values, its dtype gets promoted to `float`:

```
pd.Series([1,3,2])
```

```
0    1
1    3
2    2
dtype: int64
```

```
pd.Series([1,3,2, np.nan])
```

```
0    1.0
1    3.0
2    2.0
3    NaN
dtype: float64
```

For non-numeric types the special value `None` is used to denote a missing value, and the dtype is promoted to `object`.

```
pd.Series(["jack", "joe", None])
```

```
0    jack
1     joe
```

```
2     None
dtype: object
```

Pandas excludes the missing values from the summary statistics, like we saw in the previous section. Pandas also provides some functions to handle missing values.

The missing values can be located with the `isnull` method:

```
wh.isnull()      # returns a boolean mask DataFrame
```

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
0	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False
5	False	False	False	False	False	False	False	False
6	False	False	False	False	False	False	False	False
7	False	False	False	False	False	False	False	False
8	False	False	False	False	False	False	False	False
9	False	False	False	False	False	False	False	False
10	False	False	False	False	False	False	False	False
11	False	False	False	False	False	False	False	False
12	False	False	False	False	False	False	False	False
13	False	False	False	False	False	False	False	False
14	False	False	False	False	False	False	False	False
15	False	False	False	False	False	False	False	False
16	False	False	False	False	False	False	False	False
17	False	False	False	False	False	False	False	False
18	False	False	False	False	False	False	False	False
19	False	False	False	False	False	False	False	False
20	False	False	False	False	False	False	False	False
21	False	False	False	False	False	False	False	False
22	False	False	False	False	False	False	False	False
23	False	False	False	False	False	False	False	False
24	False	False	False	False	False	False	False	False
25	False	False	False	False	False	False	False	False
26	False	False	False	False	False	False	False	False
27	False	False	False	False	False	False	False	False
28	False	False	False	False	False	False	False	False
29	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
335	False	False	False	False	False	False	False	False
336	False	False	False	False	False	False	False	False
337	False	False	False	False	False	False	False	False
338	False	False	False	False	False	False	False	False
339	False	False	False	False	False	False	False	False
340	False	False	False	False	False	False	False	False
341	False	False	False	False	False	False	False	False
342	False	False	False	False	False	False	False	False
343	False	False	False	False	False	False	False	False
344	False	False	False	False	False	False	False	False
345	False	False	False	False	False	False	False	False
346	False	False	False	False	False	False	False	False
347	False	False	False	False	False	False	False	False
348	False	False	False	False	False	False	False	False
349	False	False	False	False	False	False	False	False
350	False	False	False	False	False	False	False	False
351	False	False	False	False	False	False	False	False
352	False	False	False	False	False	False	False	False
353	False	False	False	False	False	False	False	False
354	False	False	False	False	False	False	False	False
355	False	False	False	False	False	False	False	False
356	False	False	False	False	False	False	False	False
357	False	False	False	False	False	False	False	False
358	False	False	False	False	False	False	False	False
359	False	False	False	False	False	False	False	False
360	False	False	False	False	False	False	False	False
361	False	False	False	False	False	False	False	False
362	False	False	False	False	False	False	False	False
363	False	False	False	False	False	False	False	False
364	False	False	False	False	False	False	False	False

365 rows × 8 columns

This is not very useful as we cannot directly use the mask to index the DataFrame. We can, however, combine it with the any method to find out all the rows that contain at least one missing value:

```
wh[wh.isnull().any(axis=1)]
```

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
74	2017	3	16	00:00	UTC	1.8	NaN	3.4

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
163	2017	6	13	00:00	UTC	0.6	NaN	12.6
308	2017	11	5	00:00	UTC	0.2	NaN	8.4
309	2017	11	6	00:00	UTC	2.0	NaN	7.5
313	2017	11	10	00:00	UTC	3.6	NaN	7.2
321	2017	11	18	00:00	UTC	11.3	NaN	5.9
328	2017	11	25	00:00	UTC	8.5	NaN	4.2

The `notnull` method works conversively to the `isnull` method.

The `dropna` method of a DataFrame drops columns or rows that contain missing values from the DataFrame, depending on the `axis` parameter.

```
wh.dropna().shape   # Default axis is 0
```

```
(358, 8)
```

```
wh.dropna(axis=1).shape # Drops the columns containing missing values
```

```
(365, 7)
```

The how and `thresh` parameters of the `dropna` method allow one to specify how many values need to be missing in order for the row/column to be dropped.

The `fillna` method allows to fill the missing values with some constant or interpolated values. The `method` parameter can be:

- None: use the given positional parameter as the constant to fill missing values with
- `ffill`: use the previous value to fill the current value
- `bfill`: use the next value to fill the current value

For example, for the weather data we could use forward fill

```
wh = wh.fillna(method='ffill')
wh[wh.isnull().any(axis=1)]
```

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)

The `interpolate` method, which we will not cover here, offers more elaborate ways to interpolate the missing values from their neighbouring non-missing values.

Exercise 12 (cyclists)

Write function `cyclists` that does the following.

Load the Helsinki bicycle data set from the `src` folder (https://hri.fi/data/dataset//helsingin-pyorailijamaarat). The dataset contains the number of cyclists passing by measuring points per hour. The data is gathered over about four years, and there are 20 measuring points around Helsinki. The dataset contains some empty rows at the end. Get rid of these. Also, get rid of columns that contain only missing values. Return the cleaned dataset.

Exercise 13 (missing value types)

Make function `missing_value_types` that returns the following DataFrame. Use the `State` column as the (row) index. The value types for the two other columns should be `float` and `object`, respectively. Replace the dashes with the appropriate missing value symbols.

State	Year of independence	President
United Kingdom	-	-
Finland	1917	Niinistö
USA	1776	Trump
Sweden	1523	-
Germany	-	Steinmeier
Russia	1992	Putin

Exercise 14 (special missing values)

Write function `special_missing_values` that does the following.

Read the data set of the top forty singles from the beginning of the year 1964 from the `src` folder. Return the rows whose singles' position dropped compared to last week's position (column LW=Last Week).

To do this you first have to convert the special values "New" and "Re" (Re-entry) to missing values (None).

Exercise 15 (last week)

This exercise can give two points at maximum!

Write function `last_week` that reads the top40 data set mentioned in the above exercise. The function should then try to reconstruct the top40 list of the previous week based on that week's list. Try to do this as well as possible. You can fill the values that are impossible to reconstruct by

missing value symbols. Your solution should work for a top40 list of any week. So don't rely on specific features of this top40 list. The column `WoC` means "Weeks on Chart", that is, on how many weeks this song has been on the top 40 list.

Hint. First create the last week's top40 list of those songs that are also on this week's list. Then add those entries that were not on this week's list. Finally sort by position.

Hint 2. The `where` method of Series and DataFrame can be useful. It can also be nested.

Hint 3. Like in NumPy, you can use with Pandas the bitwise operators `&`, `|`, and `~`. Remember that he bitwise operators have higher precedence than the comparison operations, so you may have to use parentheses around comparisons, if you combined result of comparisons with bitwise operators.

You get a second point, if you get the columns `LW` and `Peak Pos` correct.

Converting columns from one type to another

There are several ways of converting a column to another type. For converting single columns (a Series) one can use the `pd.to_numeric` function or the `map` method. For converting several columns in one go one can use the `astype` method. We will give a few examples of use of these methods/functions. For more details, look from the Pandas documentation.

```
pd.Series(["1","2"]).map(int)                    # str -> int
```

```
0    1
1    2
dtype: int64
```

```
pd.Series([1,2]).map(str)                        # int -> str
```

```
0    1
1    2
dtype: object
```

```
pd.to_numeric(pd.Series([1,1.0]), downcast="integer")   # object -> int
```

```
0    1
1    1
dtype: int8
```

```
pd.to_numeric(pd.Series([1,"a"]), errors="coerce")      # conversion error produces Na
```

```
0    1.0
1    NaN
dtype: float64
```

```
pd.Series([1,2]).astype(str)                            # works for a single series
```

```
0    1
1    2
dtype: object
```

```
df = pd.DataFrame({"a": [1,2,3], "b" : [4,5,6], "c" : [7,8,9]})
print(df.dtypes)
print(df)
```

```
a int64
b int64
c int64
dtype: object
a b c
0 1 4 7
1 2 5 8
2 3 6 9
```

```
df.astype(float)                        # Convert all columns
```

	a	b	c
0	1.0	4.0	7.0
1	2.0	5.0	8.0
2	3.0	6.0	9.0

```
df2 = df.astype({"b" : float, "c" : str})    # different types for columns
print(df2.dtypes)
print(df2)
```

```
a int64
b float64
c object
dtype: object
a b c
0 1 4.0 7
1 2 5.0 8
2 3 6.0 9
```

String processing

If the elements in a column are strings, then the vectorized versions of Python's string processing methods are available. These are accessed through the `str` attribute of a Series or a DataFrame. For example, to capitalize all the strings of a Series, we can use the `str.capitalize` method:

```
names = pd.Series(["donald", "theresa", "angela", "vladimir"])
names.str.capitalize()
```

```
0       Donald
1      Theresa
2       Angela
3    Vladimir
dtype: object
```

One can find all the available methods by pressing the tab key after the text `names.str.` in a Python prompt. Try it in below cell!

```
#names.str.
```

We can split a column or Series into several columns using the `split` method. For example:

```
full_names = pd.Series(["Donald Trump", "Theresa May", "Angela Merkel", "Vladimir Puti
full_names.str.split()
```

```
0       [Donald, Trump]
1        [Theresa, May]
2     [Angela, Merkel]
3    [Vladimir, Putin]
dtype: object
```

This is not exactly what we wanted: now each element is a list. We need to use the `expand` parameter to split into columns:

```
full_names.str.split(expand=True)
```

	0	1
0	Donald	Trump
1	Theresa	May
2	Angela	Merkel
3	Vladimir	Putin

Exercise 16 (split date)

Read again the bicycle data set from `src` folder, and clean it as in the earlier exercise. Then split the `Päivämäärä` column into a DataFrame with five columns with column names `Weekday`, `Day`, `Month`, `Year`, and `Hour`. Note that you also need to to do some conversions. To get Hours, drop the colon and minutes. Convert field `Weekday` according the following rule:

```
ma -> Mon
ti -> Tue
ke -> Wed
to -> Thu
pe -> Fri
la -> Sat
su -> Sun
```

Convert the `Month` column according to the following mapping

```
tammi 1
helmi 2
maalis 3
huhti 4
touko 5
kesä 6
heinä 7
elo 8
syys 9
loka 10
marras 11
joulu 12
```

Create function `split_date` that does the above and returns a DataFrame with five columns. You may want to use the `map` method of Series objects.

So the first element in the `Päivämäärä` column of the original data set should be converted from `ke 1 tammi 2014 00:00` to `Wed 1 1 2014 0` . Test your solution from the `main` function.

Exercise 17 (cleaning data)

This exercise can give two points at maximum!

The entries in the following table of US presidents are not uniformly formatted. Make function `cleaning_data` that reads the table from the

tsv file `src/presidents.tsv` and returns the cleaned version of it. Note, you must do the edits programmatically using the string edit methods, not by creating a new DataFrame by hand. The columns should have `dtypes` `object`, `integer`, `float`, `integer`, `object`. The `where` method of DataFrames can be helpful, likewise the string methods of Series objects. You get an additional point, if you manage to get the columns President and Vice-president right!

President	Start	Last	Seasons	Vice-president
donald trump	2017 Jan	-	1	Mike pence
barack obama	2009	2017	2	joe Biden
bush, george	2001	2009	2	Cheney, dick
Clinton, Bill	1993	2001	two	gore, Al

Additional information

We covered subsetting of DataFrames with the indexers `[]`, `.loc[]`, and `.iloc[]` quite concisely. For a more verbose explanation, look at the tutorials at Dunder Data. Especially, the problems with chained indexing operators (like `df["a"][1]`) are explained well there (tutorial 4), which we did not cover at all. As a rule of thumb: one should avoid chained indexing combined with assignment! See Pandas documentation.

Summary (week 4)

- You can create DataFrames in several ways:
    - By reading from a csv file
    - Out out two dimensional NumPy array
    - Out of rows
    - Out of columns
- You know how to access rows, columns and individual elements of DataFrames
- You can use the `describe` method to get a quick overview of a DataFrame
- You know how missing values are represented in Series and DataFrames, and you know how to manipulate them
- There are similarities between Python's string methods and the vectorized forms of string operations in Series and DataFrames
- You can do complicated text processing with the `str.replace`

method combined with regular expressions

- The powerful `where` method is the vectorized form of Python's `if-else` construct
- We remember that with NumPy arrays we preferred vectorized operations instead of, for instance, `for` loops. Same goes with Pandas. It may first feel that things are easier to achieve with loops, but after a while vectorized operations will feel natural.

You have reached the end of this section!
Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI