Pandas

- - -

Exercise 13 (read series) Exercise 14 (operations on series) Exercise 15 (inverse series)

Pandas

In the NumPy section we dealt with some arrays, whose columns had each a special meaning. For example, the column number 0 could contain values interpreted as years, and column 1 could contain a month, and so on. It is possible to handle the data this way, but in can be hard to remember, which column number corresponds to which variable. Especially, if you later remove some column from the array, then the numbering of the remaining columns changes. One solution to this is to give a descriptive name to each column. These column names stay fixed and attached to their corresponding columns, even if we remove some of the columns. In addition, the rows can be given names as well, these are called *indices* in Pandas.

The Pandas library is built on top of the NumPy library, and it provides a special kind of two dimensional data structure called `DataFrame`. The `DataFrame` allows to give names to the columns, so that one can access a column using its name in place of the index of the column.

First we will quickly go through a few examples to see what is possible with Pandas. You may need to check some details from the Pandas documentation in order to complete the exercises. We start by doing some standard imports:

```
import pandas as pd    # This is the standard way of importing the Pandas library
import numpy as np
```

Let's import some weather data that is in text form in a csv (Commma Separated Values) file. The following call will fetch the data from the internet and convert it to a DataFrame:

```
wh = pd.read_csv("https://raw.githubusercontent.com/csmastersUH/data_analysis_with_pyt
wh.head()   # The head method prints the first 5 rows
```

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
0	2017	1	1	00:00	UTC	-1.0	-1.0	0.6
1	2017	1	2	00:00	UTC	4.4	-1.0	-3.9
2	2017	1	3	00:00	UTC	6.6	7.0	-6.5
3	2017	1	4	00:00	UTC	-1.0	13.0	-12.8
4	2017	1	5	00:00	UTC	-1.0	10.0	-17.8

We see that the DataFrame contains eight columns, three of which are actual measured variables. Now we can refer to a column by its name:

```
wh["Snow depth (cm)"].head()     # Using the tab key can help enter long column names
```

```
0    -1.0
1    -1.0
2     7.0
3    13.0
4    10.0
Name: Snow depth (cm), dtype: float64
```

There are several summary statistic methods that operate on a column or on all the columns. The next example computes the mean of the temperatures over all rows of the DataFrame:

```
wh["Air temperature (degC)"].mean()    # Mean temperature
```

```
6.5271232876712331
```

We can drop some columns from the DataFrame with the `drop` method:

```
wh.drop("Time zone", axis=1).head()    # Return a copy with one column removed, the or
```

	Year	m	d	Time	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
0	2017	1	1	00:00	-1.0	-1.0	0.6
1	2017	1	2	00:00	4.4	-1.0	-3.9
2	2017	1	3	00:00	6.6	7.0	-6.5
3	2017	1	4	00:00	-1.0	13.0	-12.8
4	2017	1	5	00:00	-1.0	10.0	-17.8

```
wh.head()     # Original DataFrame is unchanged
```

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)
0	2017	1	1	00:00	UTC	-1.0	-1.0	0.6
1	2017	1	2	00:00	UTC	4.4	-1.0	-3.9
2	2017	1	3	00:00	UTC	6.6	7.0	-6.5
3	2017	1	4	00:00	UTC	-1.0	13.0	-12.8
4	2017	1	5	00:00	UTC	-1.0	10.0	-17.8

In case you want to modify the original DataFrame, you can either assign the result to the original DataFrame, or use the `inplace` parameter of the `drop` method. Many of the modifying methods of the DataFrame have the `inplace` parameter.

Addition of a new column works like adding a new key-value pair to a dictionary:

```
wh["Rainy"] = wh["Precipitation amount (mm)"] > 5
wh.head()
```

	Year	m	d	Time	Time zone	Precipitation amount (mm)	Snow depth (cm)	Air temperature (degC)	Rainy
0	2017	1	1	00:00	UTC	-1.0	-1.0	0.6	False
1	2017	1	2	00:00	UTC	4.4	-1.0	-3.9	False
2	2017	1	3	00:00	UTC	6.6	7.0	-6.5	True
3	2017	1	4	00:00	UTC	-1.0	13.0	-12.8	False
4	2017	1	5	00:00	UTC	-1.0	10.0	-17.8	False

In the next sections we will systematically go through the DataFrame and its one-dimensional version: *Series*.

Creation and indexing of series

One can turn any one-dimensional iterable into a Series, which is a one-dimensional data structure:

```
s=pd.Series([1, 4, 5, 2, 5, 2])
s
```

```
0    1
1    4
2    5
3    2
4    5
5    2
dtype: int64
```

The data type of the elements in this Series is `int64`, integers representable in 64 bits.

We can also attach a name to this series:

```
s.name = "Grades"
s
```

```
0    1
1    4
2    5
3    2
4    5
5    2
Name: Grades, dtype: int64
```

The common attributes of the series are the `name`, `dtype`, and `size`:

```
print(f"Name: {s.name}, dtype: {s.dtype}, size: {s.size}")
```

```
Name: Grades, dtype: int64, size: 6
```

In addition to the values of the series, also the row indices were printed. All the accessing methods from NumPy arrays also work for the Series: indexing, slicing, masking and fancy indexing.

```
s[1]
```

```
4
```

```
s2=s[[0,5]]                    # Fancy indexing
print(s2)
```

```
0 1
5 2
Name: Grades, dtype: int64
```

```
t=s[-2:]                    # Slicing
t
```

```
4    5
5    2
Name: Grades, dtype: int64
```

Note that the indices stick to the corresponding values, they are not renumbered!

```
t[4]                        # t[0] would give an error
```

```
5
```

The values as a NumPy array are accessible via the `values` attribute:

```
s2.values
```

```
array([1, 2])
```

And the indices are available through the `index` attribute:

```
s2.index
```

```
Int64Index([0, 5], dtype='int64')
```

The index is not simply a NumPy array, but a data structure that allows fast access to the elements. The indices need not be integers, as the next example shows:

```
s3=pd.Series([1, 4, 5, 2, 5, 2], index=list("abcdef"))
s3
```

```
a    1
b    4
c    5
d    2
e    5
f    2
dtype: int64
```

```
s3.index
```

```
Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')
```

```
s3["b"]
```

```
4
```

```
s3["b":"e"]
```

```
b    4
c    5
d    2
e    5
dtype: int64
```

It is still possible to access the series using NumPy style *implicit integer indices*:

```
s3[1]
```

```
4
```

This can be confusing though. Consider the following series:

```
s4 = pd.Series(["Jack", "Jones", "James"], index=[1,2,3])
s4
```

```
1      Jack
2     Jones
3     James
dtype: object
```

What do you think `s4[1]` will print? For this ambiguity Pandas offers attributes `loc` and `iloc`. The attributes `loc` always uses the explicit index, while the attribute `iloc` always uses the implicit integer index:

```
print(s4.loc[1])
print(s4.iloc[1])
```

```
Jack
Jones
```

Exercise 13 (read series)

Write function `read_series` that reads input lines from the user and return a Series. Each line should contain first the index and then the corresponding value, separated by whitespace. The index and values are strings (in this case `dtype` is `object`). An empty line signals the end of Series. Malformed input should cause an exception. An input line is malformed, if it is non-empty and, when split at whitespace, does not result in two parts.

Test your function from the `main` function.

Exercise 14 (operations on series)

Write function `create_series` that gets two lists of numbers as parameters. Both lists should have length 3. The function should first create two Series, `s1` and `s2`. The first series should have values from the first parameter list and have corresponding indices `a`, `b`, and `c`. The second series should get its values from the second parameter list and have again the corresponding indices `a`, `b`, and `c`. The function should return the pair of these Series.

Then, write a function `modify_series` that gets two Series as parameters. It should add to the first Series `s1` a new value with index `d`. The new value should be the same as the value in Series `s2` with index `b`. Then delete the element from `s2` that has index `b`. Now the first Series

should have four values, while the second list has only two values. Adding a new element to a Series can be achieved by assignment, like with dictionaries. Deletion of an element from a Series can be done with the `del` statement.

Test these functions from the main function. Try adding together the Series returned by the `modify_series` function. The operations on Series use the indices to keep the element-wise operations *aligned*. If for some index the operation could not be performed, the resulting value will be `NaN` (Not A Number).

Exercise 15 (inverse series)

Write function `inverse_series` that get a Series as a parameter and returns a new series, whose indices and values have swapped roles. Test your function from the `main` function.

What happens if some value appears multiple times in the original Series? What happens if you use this value to index the resulting Series?

One may notice that there are similarities between Python's dictionaries and Pandas' Series, both can be thought to access values using keys. The difference is that Series requires that the indices have all the same type, and similarly, all the values have the same type. This restriction allows creation of fast data structures.

As a mark of the similaries between these two data structures, Pandas allows creation of a `Series` object from a dictionary:

```
d = { 2001 : "Bush", 2005: "Bush", 2009: "Obama", 2013: "Obama", 2017 : "Trump"}
s4 = pd.Series(d, name="Presidents")
s4
```

```
2001      Bush
2005      Bush
2009     Obama
2013     Obama
2017     Trump
Name: Presidents, dtype: object
```

Summary (week 3)

- You found that comparisons are also vectorized operations, and that the result of a comparison can be used to mask (i.e. restrict) further operations on arrays

- You can select a list of columns using fancy indexing
- An application of NumPy arrays: basic linear algebra operations and solving systems of linear equations
- You know the building blocks of matplotlib's figures. You can create figures based on NumPy arrays and you can adjust the attributes of figures
- You understand how (raster) images are organized as NumPy arrays. You can manipulate images using Numpy's array operations.
- In Pandas it is standard to use rows of DataFrames as samples and columns as variables
- Both rows and columns of Pandas DataFrames can have names, i.e. indices
    - Operations maintain these indices even when adding or removing rows or columns
    - Indices also allow several operations to be combined meaningfully and easily

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