Programming technologies

Pandas

Pandas library

Pandas is a high-level Python library for data analysis. Why do I call it high-level, because it is built on top of the lower-level NumPy library (written in C), which is a big plus in performance. In the Python ecosystem, pandas is the most advanced and fastest growing data science library.

```
import pandas as pd
```

Data structures of Pandas

DataFrame

```
mydataset = {
    'model': ["Tesla Model Y", "Toyota Corolla", "Toyota RAV4",
            "Ford F-Series", "Honda CR-V"],
    'number_of_sales': [1.15, 1.13, 0.93, 0.9, 0.72]
}
df = pd.DataFrame(mydataset)
print(df)
```

	model	number_of_sales
0	Tesla Model Y	1.15
1	Toyota Corolla	1.13
2	Toyota RAV4	0.93
3	Ford F-Series	0.90
4	Honda CR-V	0.72

Series

```
a = [1, 7, 2]
srs = pd.Series(a, index=["x",
"y", "z"])
print(srs)
```

```
x    1
y    7
z    2
dtype: int64
```

Series

The Series structure/object is an object similar to a one-dimensional array (Python list, for example), but its distinctive feature is the presence of associated labels, the so-called. indexes along each element from the list. This feature makes it an associative array or dictionary in Python.

```
import pandas as pd

my_series = pd.Series([5, 6, 7, 8, 9, 10])
print(my_series)
```

```
0     5
1     6
2     7
3     8
4     9
5    10
dtype: int64
```

Series

In the string representation of a Series object, the index is on the left and the element itself is on the right. If the index is not explicitly given, then pandas automatically creates a RangeIndex from 0 to N-1, where N is the total number of elements. It is also worth noting that Series has a type of stored elements, in our case it is int64, because we passed integer values.

The Series object has attributes through which you can get a list of elements and indexes, these are values and index, respectively.

```
my_series.index
my_series.values
```

Indexes

Access to the elements of a Series object is possible by their index (remember the analogy with a dictionary and access by key).

```
my_series[4] # will return 9
```

Indexes can be set explicitly:

```
my_series2 = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
print(my_series2['f']) # will print 10
```

Indexes

Select across multiple indexes and perform group assignment:

```
my_series2[['a', 'b', 'f']]
```

```
a     5
b     6
f    10
dtype: int64
```

```
my_series2[['a', 'b', 'f']] = 0
```

```
a    0
b    0
c    7
d    8
e    9
f    0
dtype: int64
```

Filtering + Math operations

```
my_series2[my_series2 > 0]
```

```
c    7
d    8
e    9
dtype: int64
```

```
my_series2[my_series2 > 0] * 2
```

```
c    14
d    16
e    18
dtype: int64
```

Series from dict() object

```
my_series3 = pd.Series({'a': 5, 'b': 6, 'c': 7, 'd': 8})

print(my_series3)
```

```
a    5
b    6
c    7
d    8
dtype: int64
```

```
print('d' in my_series3)
```

```
True
```

Name of series and index

The Series object and its index have a name attribute, which gives the name of the object and index, respectively.

```
my_series3.name = 'numbers'

my_series3.index.name = 'letters'

print(my_series3)
```

```
letters
a    5
b    6
c    7
d    8
Name: numbers, dtype: int64
```

```
my_series3.index = ['A', 'B', 'C', 'D']

print(my_series3)
```

```
A    5
B    6
C    7
D    8
Name: numbers, dtype: int64
```

Keep in mind that the indexed list must be the same length as the number of elements in the Series.

DataFrame

The DataFrame object is best thought of as a regular table, and rightly so, because the DataFrame is a tabular data structure. Any table always has rows and columns. The columns in a DataFrame object are Series objects, the rows of which are their immediate elements.

The easiest way to construct a DataFrame is using a Python dictionary as an example:

```
df = pd.DataFrame({
    'country': ['United Kingdom', 'United States', 'Canada', 'Ukraine'],
    'population': [60.6, 298.4, 33.1, 46.7],
    'square': [244820, 9631420, 9984670, 603700]
})
print(df)
```

	country	population	square
0	United Kingdom	60.6	244820
1	United States	298.4	9631420
2	Canada	33.1	9984670
3	Ukraine	46.7	603700

Series in Dataframe

The column in the DataFrame is Series.

```
print(df['country'])
```

```
0    United Kingdom
1     United States
2            Canada
3           Ukraine
Name: country, dtype: object
```

```
print(type(df['country']))
```

```
<class 'pandas.core.series.Series'>
```

DataFrame Indexes

The DataFrame object has 2 indexes: on rows and on columns. If the index on the rows is not explicitly specified (for example, the column by which they need to be built), then pandas sets an integer index RangeIndex from 0 to N-1, where N is the number of rows in the table.

```
df.columns
```

```
Index(['country', 'population', 'square'], dtype='object')
```

```
df.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

Index defining

During defining of DataFrame:

```
df = pd.DataFrame({
    'country': ['United Kingdom', 'United States', 'Canada', 'Ukraine'],
    'population': [60.6, 298.4, 33.1, 46.7],
    'square': [244820, 9631420, 9984670, 603700]
}, index=['UK', 'US', 'CA', 'UA'])
print(df)
```

	country	population	square
UK	United Kingdom	60.6	244820
US	United States	298.4	9631420
CA	Canada	33.1	9984670
UA	Ukraine	46.7	603700

Define after Dataframe creation + define the name:

```
df.index = ['UK', 'US', 'CA', 'UA']

df.index.name = 'Country Code'
```

Country Code	country	population	square
UK	United Kingdom	60.6	244820
US	United States	298.4	9631420
CA	Canada	33.1	9984670
UA	Ukraine	46.7	603700

Row(s) access

Rows can be accessed by index in several ways:

- .loc - used for access by string label
- .iloc - used to access by numeric value (starting from 0)

```
print(df.loc['UA'])
```

```
country       Ukraine
population       46.7
square          603700
Name: UA, dtype: object
```

```
print(df.iloc[0])
```

```
country       United Kingdom
population                60.6
square                244820
Name: UK, dtype: object
```

Selection

```
print(df.loc[['UK', 'US'], 'population'])
```

```
Country Code
UK     60.6
US    298.4
Name: population, dtype: float64
```

```
print(df.loc['UK':'CA', :])
```

Country Code	country	population	square
UK	United Kingdom	60.6	244820
US	United States	298.4	9631420
CA	Canada	33.1	9984670

```
print(df[df.population > 50][['country', 'square']])
```

Country Code	country	square
UK	United Kingdom	244820
US	United States	9631420

```
df.population # the same for df['population']
```

```
Country Code
UK     60.6
US    298.4
CA     33.1
UA     46.7
Name: population, dtype: float64
```

Operation

Pandas, when operating on a DataFrame, returns a new DataFrame object.

Let's add a new column in which we divide the population (in millions) by the area of the country, thereby obtaining the density:

```
df['density'] = df['population'] / df['square'] * 1000000
```

Country Code	country	population	square	density
UK	United Kingdom	60.6	244820	247.528797
US	United States	298.4	9631420	30.981932
CA	Canada	33.1	9984670	3.315082
UA	Ukraine	46.7	603700	77.356303

```
df = df.drop(['density'], axis='columns') # del df['density']
```

Country Code	country	population	square
UK	United Kingdom	60.6	244820
US	United States	298.4	9631420
CA	Canada	33.1	9984670
UA	Ukraine	46.7	603700

```
df = df.rename(columns={'country':'Country',
                        'population': 'Population',
                        'square':'Square'})
```

Country Code	Country	Population	Square
UK	United Kingdom	60.6	244820
US	United States	298.4	9631420
CA	Canada	33.1	9984670
UA	Ukraine	46.7	603700

Export / Import data

Pandas supports all the most popular data storage formats: csv, excel, sql, clipboard, html and much more:

Most often you have to work with csv files. For example, to save our DataFrame with countries, just write:

```
df.to_csv('filename.csv')
```

The `to_csv` function is also passed various arguments (for example, the separator character between columns), about which you can find out more in the official documentation.

You can read data from a csv file and turn it into a DataFrame using the `read_csv` function.

```
df = pd.read_csv('filename.csv', sep=',')
```

The `sep` argument specifies the split columns. There are many more ways to create a DataFrame from various sources, but the most commonly used are CSV, Excel and SQL. For example, using the `read_sql` function, pandas can execute an SQL query and, based on the response from the database, generate the necessary DataFrame.

Grouping and aggregation in pandas

```
df = pd.read_csv('data/countries_of_the_world.csv')

print(df.groupby(['Region'])['Country'].count())
```

```
Region
ASIA (EX. NEAR EAST)         28
BALTICS                       3
C.W. OF IND. STATES          12
EASTERN EUROPE               12
LATIN AMER. & CARIB          45
NEAR EAST                    16
NORTHERN AFRICA               6
NORTHERN AMERICA              5
OCEANIA                      21
SUB-SAHARAN AFRICA           51
WESTERN EUROPE               28
Name: Country, dtype: int64
```