Data Science and Machine Learning for Engineering Applications

Lecture Notes 4: Pandas

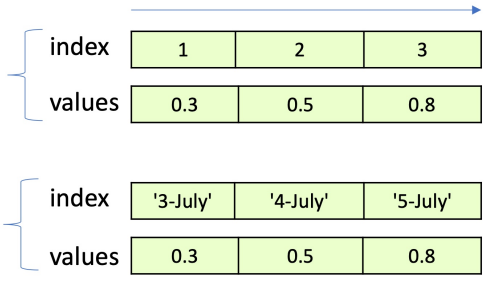
March 29, 2023 - Politecnico di Torino

1 Pandas Introduction

Pandas [1, 2] is a Python library useful for **handling** and **analyzing data structures**, particularly **bidimensional tables** and **time series** (i.e., data associated with time). It provides useful data structures (i.e., **Series** and **DataFrames**) to manage data effectively. The library provides tools for managing the **data selection**, **transforming data** with **grouping** and **pivoting operations**, **managing missing data** in the dataset, and performing **statistics** and **charts** on data. The library is **based on Numpy arrays** (efficient). It differs from Numpy because, for example, you can assign **names to the columns** of a 2-dimensional array. The two main objects provided by the **Pandas** library are **Series** (Section 2) and **DataFrames** (Section 3).

2 Pandas Series

A **Series** is a **1-dimensional** sequence of **homogeneous elements** (i.e., all with the **same type**) associated with an **explicit index**. Index elements can be either strings or integers. The main difference with respect to a 1-dimensional array is that each element is associated with an **index**. You can use the **index** to access the associated array element. The **index** can be **numerical** or **textual** (e.g., timestamp, date, etc.).



2.1 Pandas Series creation

There are many ways to create a Series. You should specify the **values** and the **index** of the **series**. If the **index is not specified**, it is set automatically with a **progressive number**.

2.1.1 Series from list

You can create a Series directly from a Python **list**. If you do not specify the index, it will automatically create a progressive number starting from 0:

```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5]) # create a series without specifying the index
print(s1)
```

Output:

```
0     2.0
1     3.1
2     4.5
dtype:  float64
```

If you want, you can **specify the index** by passing a list to the `index` parameter:

```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5], index=['a','b','c']) # create a series with specific index
print(s1)
```

Output:

```
a     2.0
b     3.1
c     4.5
dtype:  float64
```

2.1.2 Series from dictionary

You can create a Series also from a Python **dictionary**. In this case, the **keys** of the dictionary define the **index** of the Series, while the **values** of the dictionary define the **values** of the Series. The **order** of the elements in the dictionary is **preserved** when creating the series (i.e., the first key of the dictionary is the first index in the Series).

```
import pandas as pd
s1 = pd.Series({'a':2.0, 'b':3.1, 'c':4.5}) # create Series from dictionary
print(s1)
```

Output:

```
a     2.0
b     3.1
c     4.5
dtype:  float64
```

2.2 Accessing Series elements

You can access elements of a series by specifying the following:

- **Explicit index**: using the **explicit index** specified while creating the series (with the Series`.loc[]` attribute).
- **Implicit index**: using the **position** (i.e., the number) associated with the element order (similarly to Numpy arrays) (with the Series`.iloc[]` attribute).

2.2.1 Accessing Series elements by index (explicit index)

To access an element of the Series by specifying the **explicit index**, you can use the `loc[]` method of the series and specify the **index** inside the square brackets (`s1.loc[index]`).

```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5], index=['a','b','c']) # create a series with specific index
el = s1.loc['b'] # Access the element by index (element associated to index 'b')
print(el)
```

Output:

```
3.1
```

2.2.2 Accessing Series elements by position (implicit index)

To access an element of the Series by specifying the **position** (**implicit index**), you can use the `iloc[]` method of the series and specify the **position** inside the square brackets (`s1.iloc[position]`).

```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5], index=['a','b','c']) # create a series with specific index
el = s1.iloc[1] # Access the element in position 1 (second element)
print(el)
```

Output:

```
3.1
```

2.2.3 Accessing all values and index

You can obtain the **values** and the **index** of a Series with the `.values` and `.index` attributes of the Series object. Notice that the **values** are a **Numpy array**. Instead, the **Index** is a custom Python object defined in *Pandas* that allows you to perform more complex operations (e.g., union, intersection, etc. of series).

```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5], index=['a','b','c']) # create a series with specific index
print(s1.values) # s1.values returns a Numpy array
print(s1.index) # s1.index return a Index object
```

Output:

```
[2.  3.1 4.5]
Index(['a', 'b', 'c'], dtype='object')
```

2.2.4 Assign values to elements

You can also use `.loc[]` and `.iloc[]` to **assign values** to elements and modify the Series **inplace**:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc['a'])    # With explicit index
print(s1.iloc[0])     # With implicit index
s1.loc['b'] = 10      # Allows editing values (assign a value)
print(f"Series:\n{s1}")
```

Output:

```
2.0
2.0
Series:
a     2.0
b     10.0
c     4.5
dtype:  float64
```

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc['a'])    # With explicit index
print(s1.iloc[0])     # With implicit index
s1.iloc[1] = 10      # Allows editing values (assign a value)
print(f"Series:\n{s1}")
```

Output:

```
2.0
2.0
Series:
a     2.0
b     10.0
c     4.5
dtype:  float64
```

2.2.5 Slicing a Series

You can also use `.loc[]` and `.iloc[]` to access a **slice** of the elements of the Series. With the **implicit index** (`iloc`), it works as Numpy arrays and lists. You have to specify the start position (included) and the end position (**excluded**). Instead with **explicit index** (`loc`), you should specify the starting and stop index, **both included**. After slicing, you get a **new Series** containing the sliced elements.

Example with **explicit index** `loc` (both indices are included):

```
s1 = pd.Series([2.0, 3.1, 4.5, 1.1, 7.7, 2.4], index=['a', 'b', 'c', 'd', 'e', 'f'])
print(s1.loc['c':'e'])    # Slicing with explicit index (both included)
```

`Output:`

```
c     4.5
d     1.1
e     7.7
dtype:  float64
```

Example with **implicit index** `iloc` (start position included and stop position excluded):

```
s1 = pd.Series([2.0, 3.1, 4.5, 1.1, 7.7, 2.4], index=['a', 'b', 'c', 'd', 'e', 'f'])
print(s1.loc[2:5])    # Slicing with implicit index (start included and stop exclued)
```

`Output:`

```
c     4.5
d     1.1
e     7.7
dtype:  float64
```

2.2.6 Masking a Series

You can also access Series elements with **masking**. The **masking** will create a **boolean Series** with `True` if the condition is satisfied and `False` if not satisfied. When using **masking**, you can **avoid** using the `loc` function.

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
mask = (s1>2) & (s1<10) # the AND operator is &
print(mask)
```

`Output:`

```
a     False
b     True
c     True
dtype:  bool
```

As for Numpy, you can exploit the **mask** to access the **Series elemnts** and/or modifying if they satisfy a condition:

This example shows how to **access** (read) Series elements with a mask:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
mask = (s1>2) & (s1<10) # the AND operator is &
print(s1[mask]) # access elements of s1 where mask is True
```

Output:

```
b     3.1
c     4.5
dtype:  float64
```

This example shows how to **modify** Series elements with a mask:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
mask = (s1>2) & (s1<10)
s1[mask] = 0 # modify elements of s1 where mask is True
print(s1)
```

Output:

```
a     2.0
b     0.0
c     0.0
dtype:  float64
```

2.2.7 Accessing a Series with Fancy Indexing

Fancy Indexing allows you to access a subset of a Series by specifying the list of indices (e.g., you want to access rows with indices 'a' and 'b'). It is an access method also available for *Numpy* arrays (but we didn't cover it). However, with Series and DataFrame (we will see it later), it is really useful. The syntax is simple: when you access the series, you have to put inside the square brackets a list of index values that you want to access (e.g., `s1.loc[['a', 'b']]` or `s1.iloc[[1, 3]]`).

This example shows how to access the elements of the Series with (explicit) index `'a'` and `'c'`:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc[['a', 'c']]) # Access index 'a' and columns 'c'
```

Output:

```
a     2.0
c     4.5
dtype:  float64
```

This example shows how to access with (implicit) index the first (position `0`) and the third (position `2`) elements of the Series:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.iloc[[0, 2]])
```

Output:

```
a     2.0
c     4.5
dtype:  float64
```

Notice that you are not accessing from 0 to 2, but only row 0 and row 2.

3 Pandas DataFrame

DataFrame represents a **2-dimensional array** (i.e., a **Table**). It can be viewed as a table where **columns are Series** objects that share the **same index**. Each column has a **name**.

Index	'Price'	'Quantity'	'Liters'
'Water'	1.0	5	1.5
'Beer'	1.4	10	0.3
'Wine'	5.0	8	1

3.1 Pandas DataFrame creation

3.1.1 DataFrame from Series

You can create a **DataFrame** starting from existing **Series** with the same **Index** for all of them. You should use the `pd.DataFrame()` constructor by passing as a parameter a **dictionary** with the **column names** as **keys**, and the **Series** as **values**:

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
print(df)
```

Output:

	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1.0

If you have Series that don't contain exactly the same **Index**, the values of the index that **do not match** will be inserted only for the Series that contain those values, and for the other Series (i.e., columns), will be inserted a **Null** value (i.e., `NaN`).

```
price = pd.Series([1.0, 1.4, 5, 2], index=['a', 'b', 'c', 'd'])# Added  'd' in the Index
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
print(df)
```

Output:

	Price	Quantity	Liters
a	1.0	5.0	1.5
b	1.4	10.0	0.3
c	5.0	8.0	1.0
d	2.0	NaN	NaN

3.1.2 DataFrame from list of dictionaries

You can create a **DataFrame** from a **list of dictionaries**. Each **dictionary** in the list represents a **row** in the **DataFrame**. The **Index** is automatically set to a **progressive number** (unless explicitly passed as a parameter, e.g., `index=['p1', 'p2', 'p3']`).

```
df = pd.DataFrame([{'a':1, 'b':0.5, 'c': 2.2},
                   {'a':1.1, 'b':0.7, 'c': 1.8},
                   {'a':1.5, 'b':0.2, 'c': 2.5}])
print(df)
```

Output:

	a	b	c
0	1.0	0.5	2.2
1	1.1	0.7	1.8
2	1.5	0.2	2.5

If you specify the **Index** parameter:

```
df = pd.DataFrame([{'a':1, 'b':0.5, 'c': 2.2},
                   {'a':1.1, 'b':0.7, 'c': 1.8},
                   {'a':1.5, 'b':0.2, 'c': 2.5}],
                   index=['p1', 'p2', 'p3'])
print(df)
```

Output:

	a	b	c
p1	1.0	0.5	2.2
p2	1.1	0.7	1.8
p3	1.5	0.2	2.5

3.1.3 DataFrame from a dictionary of key-list pairs

You can create a **DataFrame** from a **dictionary of key-list** pairs. In this case, each **value** of the dictionary is a **list**, and it is associated to a **column**. The **column name** is given by the corresponding **key** in the dictionary. The **Index** of the DataFrame is automatically set to a progressive number unless explicitly passed as a parameter, e.g., `index=['p1', 'p2', 'p3']`).

```
my_dict = { "c1": [0, 1, 2], "c2": [0, 2, 4] }
df = pd.DataFrame(my_dict)
print(df)
```

Output:

	c1	c2
0	0	0
1	1	2
2	2	4

3.1.4 DataFrame from a 2D Numpy array

You can create a **DataFrame** from a **2-dimensional Numpy array** by specifying the name of the columns and, optionally, the Index.

```
arr = np.arange(6).reshape((3,2))
df = pd.DataFrame(arr, columns=['c1', 'c2'],
                       index=['a', 'b', 'c'])
print(df)
```

Output:

	c1	c2
a	0	1
b	2	3
c	4	5

3.2 Accessing DataFrames

3.2.1 Accessing column names and index

You can obtain **all the column names** and the **index** of a DataFrame with the `.columns` and `.index` attributes of the DataFrame object. In this case, both attributes return an `Index` object. The `.columns` attribute returns an **Index object** with the column names (i.e., index of the columns. you can access columns as with rows). Instead, the `.index` attribute returns the Index for the rows.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
print(df.columns) # Index object with column names
print(df.index) # Index object
```

`Output:`

```
Index(['Price', 'Quantity', 'Liters'], dtype='object')
Index(['a', 'b', 'c'], dtype='object')
```

3.2.2 Accessing DataFrame data as Numpy array

You can get the DataFrame **data** into a **Numpy array** with the `.values` attribute.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
my_arr = df.values  # Numpy array with data
my_arr
```

`Output:`

```
array([[ 1.  , 5.  , 1.5],
[ 1.4, 10.  , 0.3],
[ 5.  , 8.  , 1.  ]])
```

3.2.3 Accessing DataFrame columns

You can access DataFrame **a column** by specifying in **square brackets** `[]` the **column name**. It returns a **Series** with the selected column.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
print(df["Quantity"]) # access by column name -> returns a Series
```

`Output:`

```
a     5
b     10
c     8
Name:  Quantity, dtype:  int64
```

3.2.4 Accessing a single DataFrame row by index

You can access a **single DataFrame row** with the same methods as for Series: `.loc` for **explicit indexing** and `.iloc` for **implicit indexing**. It returns a **Series** with an element for each column. As Index, it contains the names of the columns.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
print(df.loc['a'])
print(df.iloc[0])
```

Output:

```
Price      1.0
Quantity   5.0
Liters     1.5
Name:  a, dtype:  int64
Price      1.0
Quantity   5.0
Liters     1.5
Name:  a, dtype:  int64
```

3.2.5 Accessing DataFrames with slicing

You can access DataFrames with **slicing** by selecting **rows** and/or **columns**. Between square brackets `[]`, you have to put the **rows slice**, then a `comma ','`, and then the **columns slice**. however you cannot mix **implicit** with **explicit** indexing.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
print(df.loc['b':'c', 'Quantity':'Liters']) # access columns from 'Quantity' to 'Liters' and
    rows from 'b' to 'c'
```

Output:

	Quantity	Liters
b	10	0.3
c	8	1.0

3.2.6 Accessing DataFrames with masking

You can also use **masking** to select **rows based on a condition**. You can also combine **masking** with **slicing**. You have to specify a **mask** to select the rows based on a condition and then slicing to select only some columns.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
mask = (df['Quantity']<10) & (df['Liters']>1)
print(df.loc[mask, 'Quantity':])   # Use masking and slicing
```

Output:

	Quantity	Liters
a	5	1.5

Index	Price	Quantity	Liters
a	1.0	**5**	**1.5**
b	1.4	10	0.3
c	5.0	8	1

```
mask = (df['Quantity']<10) & (df['Liters']>1)
df.loc[mask, 'Quantity':]   # Use masking and slicing
```

Or you can combine the mask with **Fancy Indexing**.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
mask = (df['Quantity']<10) & (df['Liters']>1)
print(df.loc[mask, ['Price','Liters']])   # Use masking and fancy
```

Output:

```
   Quantity  Liters
a         5     1.5
```

Index	Price	Quantity	Liters
a	**1.0**	5	**1.5**
b	1.4	10	0.3
c	5.0	8	1

```
mask = (df['Quantity']<10) & (df['Liters']>1)
df.loc[mask, ['Price','Liters']]   # Use masking and fancy
```

3.2.7 Accessing DataFrame with only fancy indexing

You can use **Fancy Indexing** to select only some rows and/or only some columns. You have to specify two lists with the list of Index values and the list of column names.

```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})
mask = (df['Quantity']<10) & (df['Liters']>1)
print(df.loc[['a', 'c'], ['Price','Liters']])   # Use only fancy
```

Output:

```
   Price  Liters
a    1.0     1.5
c    5.0     1.0
```

Index	Price	Quantity	Liters
a	**1.0**	5	**1.5**
b	1.4	10	0.3
c	**5.0**	8	**1**

```
df.loc[['a', 'c'], ['Price','Liters']]
```

You can also use `.loc` to **assign a value**:

```
In [1]: df.loc[['a', 'c'], ['Price','Liters']] = 0
```

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

3.2.8 Adding a new column to DataFrame

You can **add a new column** from a **Series** to a DataFrame. The added Series should have the same Index. The DataFrame is modified **inplace**. If the DataFrame already has a column with the specified name, then it is **replaced**.

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

→

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

```
In [1]: df['Available'] = pd.Series([True, False, True],
                                    index=['a', 'b', 'c'])
```

You can add a new column directly from a **List** to a DataFrame. The DataFrame is modified **inplace**. If the DataFrame already has a column with the specified name, then it is **replaced**. The order of the elements in the list will be preserved.

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

→

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

```
In [1]: df['Available'] = [True, False, True]
```

3.2.9 Drop columns from a DataFrame

You can **delete** some **columns** from the DataFrame with the `drop` method and specify the list of columns to delete as a parameter. E.g., `df.drop(columns=['column 1', 'column 2'])`. The `drop` method returns a **copy** of the DataFrame (the DataFrame is **not** modified inplace). Therefore you have to assign the returned DataFrame to the old one if you want to modify it inplace. An alternative to obtain the same results is `df.drop(columns=['column 1', 'column 2'], inplace=True)`. This last option modifies the DataFrame inplace.

Index	Price	~~Quantity~~	~~Liters~~	Available
a	1.0	~~5~~	~~1.5~~	True
b	1.4	~~10~~	~~0.3~~	False
c	5.0	~~8~~	~~1~~	True

```
In [1]: df = df.drop(columns=['Quantity', 'Liters'])
```

3.2.10 Rename columns of a DataFrame

You can **rename** some columns of a DataFrame by passing a **dictionary** which **maps old names with new names** as a parameter of the `df.rename()` method. The old names of the DataFrame are specified in the **keys** of the dictionary, while the new names are in the **values** of the dictionary. Also, the `rename` method returns a copy of the DataFrame. Therefore, if you want to modify the DataFrame **inplace**, you should reassign the returned DataFrame to the old one or pass `inplace=True` as a parameter of the `rename` function.

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

→

Index	Price	**nItems**	**[L]**	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

```
In [1]: df = df.rename(columns={'Quantity': 'nItems',
                                'Liters': '[L]'})
```

4 Computation with Pandas

4.1 Unary operations on Series and DataFrames

The **unary operations** on Series and DataFrames work with any **Numpy** unary function. The specified operation is applied to each element of the Series/DataFrame. Also, **broadcasting** works in the same way. You can sum/divide/multiply **each element** of a Series or a DataFrame by a scalar with the `+`, `/`, or `*` operators. Or you can compute the absolute value, the exponent, etc., of each element of a Series or a DataFrame with the corresponding Numpy functions `np.abs(s1)`, `np.abs(df)`, `np.exp(s1)`, `np.exp(df)`, etc.

4.2 Operations between Series and DataFrames

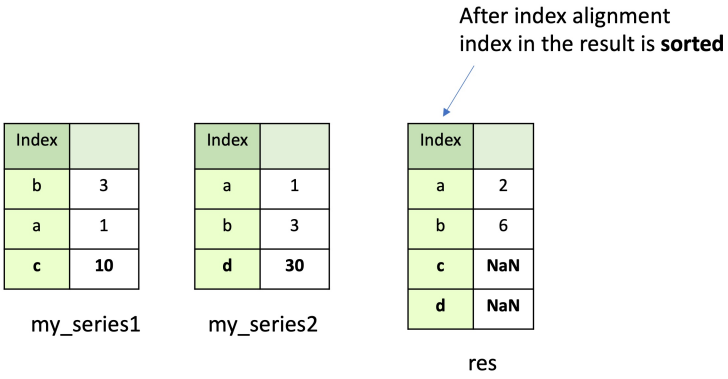
You can apply operations between two series. The operation is applied **element-wise** after **aligning indices**. The Index elements which do **not match** are set to **NaN** (i.e., not a number). After the alignment, the index in the result is **sorted** (only if they do not match).

```
import pandas as pd

s1 = pd.Series([3, 1, 10], index=['b', 'a', 'c'])
s2 = pd.Series([1, 3, 30], index=['a', 'b', 'd'])

res = s1 + s2
print(res)
```

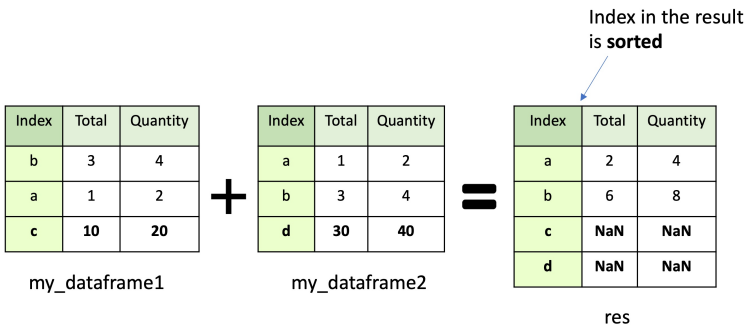
Output:

```
a     2.0
b     6.0
c     NaN
d     NaN
dtype:  float64
```

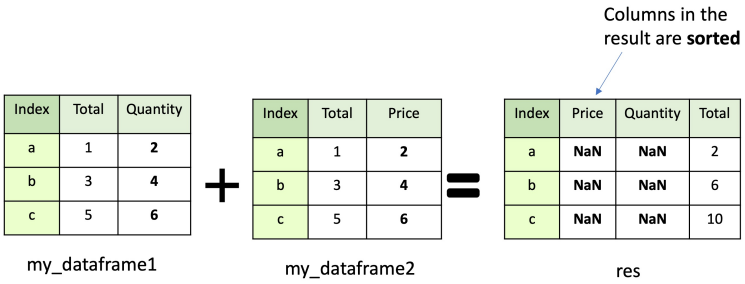
For the Index values with no match in both series, it puts the **NaN** values. Moreover, in this case, the index and the columns are ordered (only if they do not match).



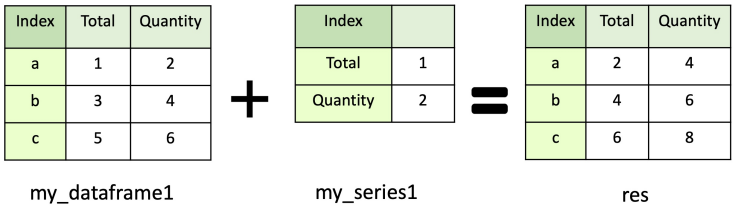
To perform operations with two DataFrames, you not only have to align Index but also the columns. Therefore, the operation is applied **element-wise** after **aligning indices** and **columns**.



If the columns are not aligned, it inserts **NaN** values in all the rows of the not aligned columns.



You can also apply operations between DataFrames and Series. The operation is applied between **the Series and each row of the DataFrame**. The operation follows the **broadcasting rules**. You have to consider the **Series** as a **row vector** where each column became an index.



4.3 Aggregations

You can perform aggregate functions (for both Series and DataFrames) to compute the mean `df.mean()`, the standard deviation `df.std()`, the minimum value `df.min()`, the maximum value `df.max()`, and the sum `df.sum()`.

An aggregate function applied to a Series returns a single value with the mean/sum/etc. of the series elements.

```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5]) # create a series without specifying the index
print(s1.mean())
```

`Output:`

```
3.1999999999999997
```

Instead, for DataFrames, aggregate functions are applied **column-wise** and return a Series with the mean/sum/etc. of each column separately.

```
import pandas as pd

price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
```

```
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])

df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})

print(df.mean())
```

Output:

```
Price      2.466667
Quantity     7.666667
Liters      0.933333
dtype:  float64
```

If you want to perform the **Z-Score normalization** with pandas of each column separately, you can do the following:

```
import pandas as pd

price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])

df = pd.DataFrame({'Price':price, 'Quantity':quantity, 'Liters':liters})

mean_series = df.mean()
std_series = df.std()

df_norm = (df - mean_series) / std_series
print(df_norm)
```

Output:

```
          Price      Quantity      Liters
a      -0.665750       -1.059626       0.940102
b      -0.484182        0.927173      -1.050702
c      1.149932        0.132453       0.110600
```

5 Handling missing values

Missing values in Pandas are represented with **sentinel** values. They can be represented with the Python null value `None` or the Numpy not a number `np.Nan`. The difference is that `None` is a python object, instead `np.Nan` is a floating point number. Using `NaN` achieves better **performances** when performing numerical computations. Pandas supports both types and automatically converts between them when appropriate.

5.1 Check if there are Null elements

You can check if a Series or a DataFrame **contain null values** with the `.isnull()` method (e.g., `s1.isnull()` or `df.isnull()`). It returns a boolean mask indicating null values (i.e., a boolean mask with `True` if the element is Null, `False` otherwise). The opposite function is `.notnull()`, which returns a boolean mask indicating **not** `Null` values (i.e., `True` of the element is not Null, `False` otherwise).

```
import pandas as pd
import numpy as np

s1 = pd.Series([4, None, 5, np.nan])
s1.isnull()
```

Output:

```
0     False
1     True
2     False
3     True
dtype=bool
```

5.2 Remove Null elements

You can also **remove Null elements** with the `.dropna()` method.

```
import pandas as pd
import numpy as np

s1 = pd.Series([4, None, 5, np.nan])
s1.dropna()
```

Output:

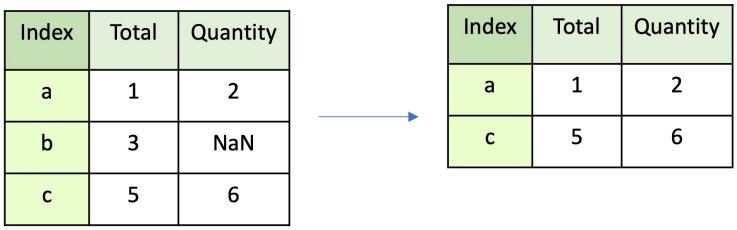
```
0     4.0
2     5.0
dtype=float64
```

When working with DataFrames, `.dropna()` removes **rows** that contain **at least one** missing value (as a default behavior). However, if you pass the parameter `how=all`, it removes rows only if they contain **all** `Nan`.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Total': [1, 3, 5], 'Quantity': [2, np.nan, 6]}, index=['a', 'b', 'c'])
df.dropna()
```

Output:

```
Index     Total     Quantity
a           1           2
c           5           6
```



Removing columns by specifying `axis='columns'` is also possible. E.g., `df.dropna(axis='columns')`.

5.3 Fill missing values

You can fill `Null` values with a **specified value** with the `.fillna()` method. E.g., `s1.fillna(0)` or `df.fillna(0)`.

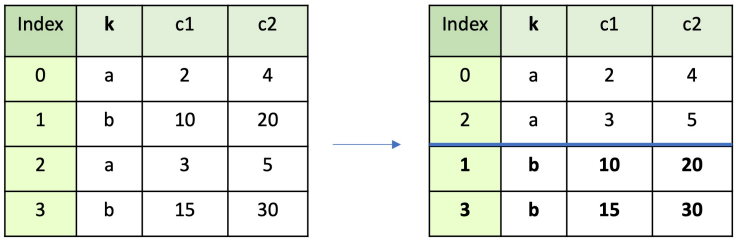
```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Total': [1, 3, 5], 'Quantity': [2, np.nan, 6]}, index=['a', 'b', 'c'])
df.fillna(0)
```

Output:

```
Index      Total      Quantity
a              1           2.0
b              3           0.0
c              5           6.0
```

6 Grouping data inside a DataFrame

Pandas provides the equivalent of the SQL **group by** statement. It allows **iterating** on groups, **aggregating** the values of each group (e.g., mean, sum, min, max, etc.), and **filtering** groups according to a condition. The `.groupby()` method returns a `DataFrameGroupBy` object. You have to specify the **column(s)** where you want to group (key).



Index	k	c1	c2
0	a	2	4
1	b	10	20
2	a	3	5
3	b	15	30

Index	k	c1	c2
0	a	2	4
2	a	3	5
1	**b**	**10**	**20**
3	**b**	**15**	**30**

After the creation of a `DataFrameGroupBy` object, you can **iterate** on groups:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'k' : ['a','b','a','b'], 'c1': [2,10,3,15], 'c2' : [4,20,5,30]})
grouped_df = df.groupby('k')   # 2 groups: 'a' and 'b'

for key, group_df in grouped_df:
    print(key)
    print(group_df)
```

Output:

```
a
   k  c1  c2
0  a   2   4
2  a   3   5
b
   k  c1  c2
1  b  10  20
3  b  15  30
```

Or you can **aggregate** by groups (e.g., with `min`, `max`, `sum`, `mean`, `std`).

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'k' : ['a','b','a','b'], 'c1': [2,10,3,15], 'c2' : [4,20,5,30]})
grouped_df = df.groupby('k')   # 2 groups: 'a' and 'b'

grouped_df.mean().reset_index() # Mean, separately for each group
```

Output:

	k	c1	c2
0	a	2.5	4.5
1	b	12.5	25.0

Notice that you should also use the `.reset_index()` to return a single-level DataFrame. Otherwise, it will return a **multi-level** DataFrame on the columns.

You can also **aggregate** a **single column** by group. The output is a Series with the result of the aggregation of each group.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'k' : ['a','b','a','b'], 'c1': [2,10,3,15], 'c2' : [4,20,5,30]})
grouped_df = df.groupby('k')   # 2 groups: 'a' and 'b'

grouped_df['c1'].mean().reset_index() # Mean for only the 'c1' column for each group
```

`Output:`

	k	c1
0	a	2.5
1	b	12.5

7 Load from a CSV

You can load a DataFrame from a **csv** file. You could specify the **delimiter** (`sep`). The function automatically reads the **header** from the **first line** of the file after **skipping** the specified number of rows (e.g., `skiprows=1`). The column data types are inferred.

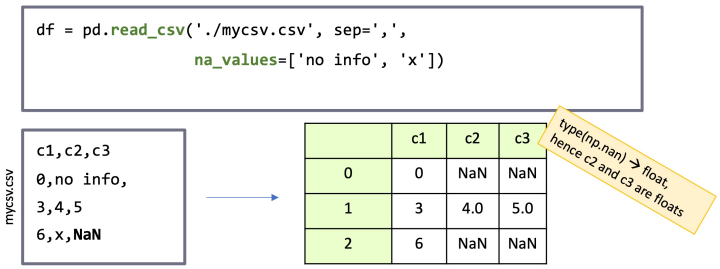
```
df = pd.read_csv('./mycsv.csv', sep=',', skiprows=1)
```

mycsv.csv

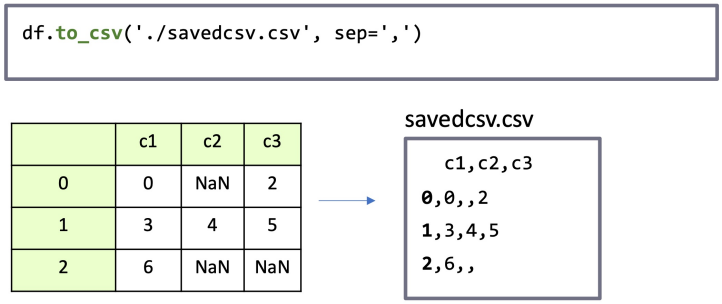
```
MyTitle
c1,c2,c3
0,1,2
3,4,5
6,7,8
```

	c1	c2	c3
0	0	1	2
1	3	4	5
2	6	7	8

If it contains **null** values, you can specify how to recognize them. By default, empty columns are converted to `NaN` (i.e., not a number Numpy datatype). The string `'NaN'` is automatically recognized as a null value.



You can also save an existing DataFrame to a CSV file. If you specify `index=False` as a parameter, it avoids writing the index.



References

[1] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[2] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.