|Sustainable water management

Python for Hydrology

Session 4 – Pandas

Objective:

Interact with Pandas to manipulate any information in tabular format, make mathematical operations in them, create new columns or rows, and export data.

Pandas

Pandas is one of the most used libraries in Python. It allows you to manipulate any tabular information, even if the data has a strange format, Pandas allows you to read it correctly.

To start, create a folder called "**Session4**" and a notebook with the same name

In the session's folder, you have a folder called "**Data**" copy all the files inside "Data" to the notebook's folder.

Let's **import pandas and all the tools needed** for this session.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Let's **open the file of the previous session**. Anytime Pandas opens a multi-column table, it is called a "**DataFrame**" and each column is a "**Serie**"

```
pd.read_csv('temp.txt')
```

```
pd.read_csv('temp.txt')
```

	Min and Max values of temperature in celcius degrees
0	Source: Hatari
1	Month Min Max
2	1 3.89 9.81
3	2 5.68 12.54
4	3 7.79 15.39
5	4 11.21 19.47
6	5 14.10 24.65
7	6 18.52 28.37
8	7 20.79 29.17
9	8 21.89 32.28
10	9 17.92 26.47
11	10 14.84 21.45
12	11 8.51 16.31
13	12 6.55 13.13

As you can see, it returns kind of a table, but it is unformatted. You can **specify to pandas the format** in which you want to open the file. As the first two lines of the file are a kind of commentary, let' **skip two lines**.

```
pd.read_csv('temp.txt',skiprows=2)
```

```
pd.read_csv('temp.txt',skiprows=2)
```

	Month Min Max
0	1 3.89 9.81
1	2 5.68 12.54
2	3 7.79 15.39
3	4 11.21 19.47
4	5 14.10 24.65
5	6 18.52 28.37
6	7 20.79 29.17
7	8 21.89 32.28
8	9 17.92 26.47
9	10 14.84 21.45
10	11 8.51 16.31
11	12 6.55 13.13

Automatically Pandas detects a header. We can specify **not to read the header**.

```
df = pd.read_csv('temp.txt',skiprows=2,header=None)
df
```

```
df = pd.read_csv('temp.txt',skiprows=2,header=None)
df
```

	0
0	Month Min Max
1	1 3.89 9.81
2	2 5.68 12.54
3	3 7.79 15.39
4	4 11.21 19.47
5	5 14.10 24.65
6	6 18.52 28.37
7	7 20.79 29.17
8	8 21.89 32.28
9	9 17.92 26.47
10	10 14.84 21.45
11	11 8.51 16.31
12	12 6.55 13.13

You can specify the table's delimiter with the option "delimiter," but a more useful tool is "**delim_whitespace**" with this tool Pandas detects that spaces delimit the table.

```
df = pd.read_csv('temp.txt',skiprows=2,delim_whitespace=True)
df
```

```
df = pd.read_csv('temp.txt',skiprows=2,delim_whitespace=True)
df
```

	Month	Min	Max
0	1	3.89	9.81
1	2	5.68	12.54
2	3	7.79	15.39
3	4	11.21	19.47
4	5	14.10	24.65
5	6	18.52	28.37
6	7	20.79	29.17
7	8	21.89	32.28
8	9	17.92	26.47
9	10	14.84	21.45
10	11	8.51	16.31
11	12	6.55	13.13

You can **call only a column** of the following way.

```
df.Month
```

```
df.Month
```

```
0      1
1      2
2      3
3      4
4      5
5      6
6      7
7      8
8      9
9     10
10    11
11    12
Name: Month, dtype: int64
```

If the table's name has a strange format with spaces or special symbols, you can call it as a string inside brackets.

```
df['Month']
```

```
df['Month']
```

```
0      1
1      2
2      3
3      4
4      5
5      6
6      7
7      8
8      9
9     10
10    11
11    12
Name: Month, dtype: int64
```

This kind of format allows you **to call the specific columns** you want.

```
df[['Month','Min']]
```

```
df[['Month','Min']]
```

	Month	Min
0	1	3.89
1	2	5.68
2	3	7.79
3	4	11.21
4	5	14.10
5	6	18.52
6	7	20.79
7	8	21.89
8	9	17.92
9	10	14.84
10	11	8.51
11	12	6.55

You can **slice** the values by selecting the column and then call the index value.

```
df['Month'][1]
```

```
df['Month'][1]
```

2

Or you can **slice by a range of indexes**.

```
df['Month'][1:5]
```

```
df['Month'][1:5]
```

```
1    2
2    3
3    4
4    5
Name: Month, dtype: int64
```

With the function “**keys**” you can obtain an array of the indexes.

```
df.keys()
```

```
df.keys()
```

```
Index(['Month', 'Min', 'Max'], dtype='object')
```

With “**values**” you can get an array of the values

```
df.values
```

```
df.values
```

```
array([[ 1.  ,  3.89,  9.81],
       [ 2.  ,  5.68, 12.54],
       [ 3.  ,  7.79, 15.39],
       [ 4.  , 11.21, 19.47],
       [ 5.  , 14.1 , 24.65],
       [ 6.  , 18.52, 28.37],
       [ 7.  , 20.79, 29.17],
       [ 8.  , 21.89, 32.28],
       [ 9.  , 17.92, 26.47],
       [10.  , 14.84, 21.45],
       [11.  ,  8.51, 16.31],
       [12.  ,  6.55, 13.13]])
```

You can call the first 5 rows of the array using the function “**head**”

```
df.head()
```

```
df.head()
```

	Month	Min	Max
0	1	3.89	9.81
1	2	5.68	12.54
2	3	7.79	15.39
3	4	11.21	19.47
4	5	14.10	24.65

Or the last 5 values of the DataFrame with the function “**tail**”

```
df.tail()
```

```
df.tail()
```

	Month	Min	Max
7	8	21.89	32.28
8	9	17.92	26.47
9	10	14.84	21.45
10	11	8.51	16.31
11	12	6.55	13.13

When opening the text file, you can also specify the names of the columns with “**names**”

```
df = pd.read_csv('temp.txt',skiprows=3,delim_whitespace=True,names
=['Month','Minimum Temperature','Maximum temperature'])
df
```

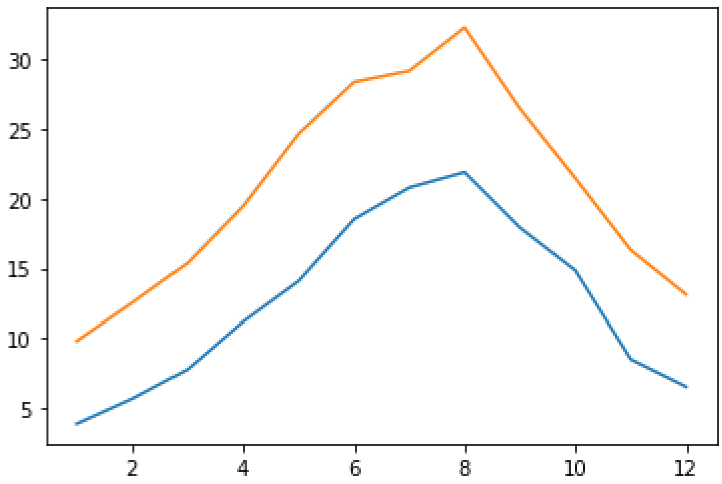
```
df = pd.read_csv('temp.txt',skiprows=3,delim_whitespace=True,names =['Month','Minimum Temperature','Maximum temperature'])
df
```

	Month	Minimum Temperature	Maximum temperature
0	1	3.89	9.81
1	2	5.68	12.54
2	3	7.79	15.39
3	4	11.21	19.47
4	5	14.10	24.65
5	6	18.52	28.37
6	7	20.79	29.17
7	8	21.89	32.28
8	9	17.92	26.47
9	10	14.84	21.45
10	11	8.51	16.31
11	12	6.55	13.13

You can easily **plot** the values following the same structure learned in the Session 3

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(df.Month,df['Minimum Temperature'])
ax.plot(df.Month,df['Maximum temperature'])
plt.show()
```


```
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(df.Month,df['Minimum Temperature'])
ax.plot(df.Month,df['Maximum temperature'])
plt.show()
```

To read Excel tables, you need to **install** and extra package:

```
!pip install xlrd
```

```
!pip install xlrd
Requirement already satisfied: xlrd in c:\users\lrbk\miniconda3\lib\site-packages (1.2.0)
```

As done before, open the Excel table, but this time use "**read_excel**"

```
pd.read_excel('temp.xlsx')
```

```
: pd.read_excel('temp.xlsx')
```

	Month	Min	Max
0	1	3.89	9.81
1	2	5.68	12.54
2	3	7.79	15.39
3	4	11.21	19.47
4	5	14.10	24.65
5	6	18.52	28.37
6	7	20.79	29.17
7	8	21.89	32.28
8	9	17.92	26.47
9	10	14.84	21.45
10	11	8.51	16.31
11	12	6.55	13.13

Pandas can support many kinds of **Date formats**. They are related to the library "**datetime**", so let's import it.

```
import datetime as dt
```

With “datetime” we can print the actual time, which returns the year, month, day, hour, minute, seconds, etc.

```
t0 = dt.datetime.now()
t0 #year,month,day,hour,minute,second ...
```

```
t0 = dt.datetime.now()
t0 #year,month,day,hour,minute,second ...
```

```
datetime.datetime(2020, 10, 3, 17, 37, 7, 707616)
```

Based on that time, you can **change its time format.**

```
t0.strftime('%Y-%m-%d')
```

```
t0.strftime('%Y-%m-%d')
```

```
'2020-10-03'
```

```
t0.strftime('%d-%m-%Y')
```

```
t0.strftime('%d-%m-%Y')
```

```
'03-10-2020'
```

```
t0.strftime('%d-%b-%Y')
```

```
t0.strftime('%d-%b-%Y')
```

```
'03-Oct-2020'
```

```
t0.strftime('%H:%M:%S')
```

```
t0.strftime('%H:%M:%S')
```

```
'17:37:07'
```

```
t0.strftime('%Y-%B-%d %H:%M:%S')
```

```
t0.strftime('%Y-%B-%d %H:%M:%S')
```

```
'2020-October-03 17:37:07'
```

To create a serie you can use:

```
ht = pd.Series([160.0-4.9*t*t for t in range(6)])
ht
```

```
#Series
ht = pd.Series([160.0-4.9*t*t for t in range(6)])
ht
```

```
0    160.0
1    155.1
2    140.4
3    115.9
4     81.6
5     37.5
dtype: float64
```

You can **slice it** too, as we did with before.

```
ht[0]
```

```
ht[0]
```

```
160.0
```

You can **get its values** too

```
ht.values
```

```
ht.values
```

```
array([160. , 155.1, 140.4, 115.9,  81.6,  37.5])
```

To get its **indexes**

```
ht.index
```

```
ht.index
```

```
RangeIndex(start=0, stop=6, step=1)
```

You can declare a Serie with its index, so you don’t need to change it later.

```
heights = pd.Series([188, 157, 173, 169, 155],
                    index=['Juan', 'Pedro', 'Ian','Mario', 'Gonzalo'])
heights
```

```
heights = pd.Series([188, 157, 173, 169, 155],
                    index=['Juan', 'Pedro', 'Ian','Mario', 'Gonzalo'])

heights
```

```
Juan       188
Pedro      157
Ian        173
Mario      169
Gonzalo    155
dtype: int64
```

You can easily convert the Serie of a DataFrame **to a dictionary**

```
heights.to_dict()
```

```
heights.to_dict()
```

```
{'Juan': 188, 'Pedro': 157, 'Ian': 173, 'Mario': 169, 'Gonzalo': 155}
```

Pandas allows us to create **ranges of dates**. Commonly the frequency is Days, but you can change it as you want., for example:

```
dtr = pd.date_range('2017-07-22', periods=5)
dtr
```


```
dtr = pd.date_range('2017-07-22', periods=5)
dtr
```


```
DatetimeIndex(['2017-07-22', '2017-07-23', '2017-07-24', '2017-07-25',
               '2017-07-26'],
              dtype='datetime64[ns]', freq='D')
```

And we can **insert this information** as the index of another DataFrame or Serie.

```
heights.index = dtr
heights
```


```
heights.index = dtr
heights
```


```
2017-07-22    188
2017-07-23    157
2017-07-24    173
2017-07-25    169
2017-07-26    155
Freq: D, dtype: int64
```

Dictionaries can be converted to DataFrames too. Suppose you have the following information regarding materials.

```
optmat = {'mat': ['silica', 'titania', 'PMMA', 'PS'],
          'index': [1.46, 2.40, 1.49, 1.59],
```

```
           'density': [2.03, 4.2, 1.19, 1.05]}
optmat
```

```
optmat = {'mat': ['silica', 'titania', 'PMMA', 'PS'],
          'index': [1.46, 2.40, 1.49, 1.59],
          'density': [2.03, 4.2, 1.19, 1.05]}
optmat
```

```
{'mat': ['silica', 'titania', 'PMMA', 'PS'],
 'index': [1.46, 2.4, 1.49, 1.59],
 'density': [2.03, 4.2, 1.19, 1.05]}
```

You can **automatically transform it to a DataFrame**:

```
omdf = pd.DataFrame(optmat)
omdf
```

```
omdf = pd.DataFrame(optmat)
omdf
```

	mat	index	density
0	silica	1.46	2.03
1	titania	2.40	4.20
2	PMMA	1.49	1.19
3	PS	1.59	1.05

If you want to, create the DataFrame with **columns names given**.

```
omdf = pd.DataFrame(optmat, columns=['mat', 'density','index'])
omdf
```

```
omdf = pd.DataFrame(optmat, columns=['mat', 'density','index'])
omdf
```

	mat	density	index
0	silica	2.03	1.46
1	titania	4.20	2.40
2	PMMA	1.19	1.49
3	PS	1.05	1.59

An **empty DataFrame** is useful when you want to insert values later.

```
omdf1 = pd.DataFrame(index=['silica', 'titania','PMMA', 'PS'],
                          columns={'density', 'index'})
omdf1
```

```
omdf1 = pd.DataFrame(index=['silica', 'titania','PMMA', 'PS'],
                          columns={'density', 'index'})
omdf1
```

	density	index
silica	NaN	NaN
titania	NaN	NaN
PMMA	NaN	NaN
PS	NaN	NaN

If you want to **insert data**, you can use"**loc**" to locate in which column and row you want to insert the data.

```
omdf1.loc['PS', ('index', 'density')] = (1.05, 1.59)
omdf1
```

```
omdf1.loc['PS', ('index', 'density')] = (1.05, 1.59)
omdf1
```

	density	index
silica	NaN	NaN
titania	NaN	NaN
PMMA	NaN	NaN
PS	1.59	1.05

If you see the **type** of the previous DataFrame, you could see that it is of the "object" type

```
omdf1.dtypes
```

```
omdf1.dtypes
```

```
density    object
index      object
dtype: object
```

You can **select the columns and change their format**. In the next example, we are transforming to the "numeric" type

```
omdf1[['index', 'density']] = omdf1[['index',
'density']].apply(pd.to_numeric)
omdf1.dtypes
```

```
omdf1[['index', 'density']] = omdf1[['index', 'density']].apply(pd.to_numeric)
omdf1.dtypes
```

```
density    float64
index      float64
dtype: object
```

Now, let's **work with an Excel** file with instantaneous flow measurements in cubic meters per second

|Sustainable water management

```
pd.read_excel('PuenteDiablo.xlsx')
```

```
pd.read_excel('PuenteDiablo.xlsx')
```

	Nota: Información primaria - sin control de calidad	Unnamed: 1	Unnamed: 2	Unnamed: 3	Unnamed: 4	Unnamed: 5	Unnamed: 6
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	STATION	OPERATOR	VARIABLE	DATE	HOUR	VALUE	MEASUREMENT UNIT
2	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-08-01 00:00:00	06:00:00	5.832	m^3/s
3	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-10-01 00:00:00	06:00:00	6.13	m^3/s
4	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-10-01 00:00:00	08:00:00	6.13	m^3/s
...	...	...	...	...	...	...	...
1176	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	25/09/2020	06:00:00	8	m^3/s
1177	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	28/09/2020	06:00:00	8.3	m^3/s
1178	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	30/09/2020	06:00:00	8	m^3/s

As you could see, there are many records, from 2016 to 2020, but the header is not placed correctly. We can use "**skiprows**" here too.

```
ws =pd.read_excel('PuenteDiablo.xlsx',skiprows=2)
ws
```

```
ws =pd.read_excel('PuenteDiablo.xlsx',skiprows=2)
ws
```

	STATION	OPERATOR	VARIABLE	DATE	HOUR	VALUE	MEASUREMENT UNIT
0	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-08-01 00:00:00	06:00:00	5.832	m^3/s
1	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-10-01 00:00:00	06:00:00	6.130	m^3/s
2	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-10-01 00:00:00	08:00:00	6.130	m^3/s
3	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2016-12-01 00:00:00	08:00:00	6.011	m^3/s
4	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	13/01/2016	08:00:00	5.801	m^3/s
...	...	...	...	...	...	...	...
1174	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	25/09/2020	06:00:00	8.000	m^3/s
1175	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	28/09/2020	06:00:00	8.300	m^3/s
1176	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	30/09/2020	06:00:00	8.000	m^3/s
1177	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2020-01-10 00:00:00	06:00:00	8.000	m^3/s
1178	PUENTE DEL DIABLO	SERVICIO NACIONAL METEOROLOGÍA E HIDROLOGÍA	CAUDAL INS	2020-02-10 00:00:00	06:00:00	8.000	m^3/s

1179 rows × 7 columns

If you don't want to use all the columns, we can use "**usecols**" to filter them. The filter applied is the column in the excel.

```
ws =pd.read_excel('PuenteDiablo.xlsx',skiprows=2,usecols='A,D,F')
ws
```

	STATION	DATE	VALUE
0	PUENTE DEL DIABLO	2016-08-01 00:00:00	5.832
1	PUENTE DEL DIABLO	2016-10-01 00:00:00	6.130
2	PUENTE DEL DIABLO	2016-10-01 00:00:00	6.130
3	PUENTE DEL DIABLO	2016-12-01 00:00:00	6.011
4	PUENTE DEL DIABLO	13/01/2016	5.801
...	...	...	...
1174	PUENTE DEL DIABLO	25/09/2020	8.000
1175	PUENTE DEL DIABLO	28/09/2020	8.300
1176	PUENTE DEL DIABLO	30/09/2020	8.000
1177	PUENTE DEL DIABLO	2020-01-10 00:00:00	8.000
1178	PUENTE DEL DIABLO	2020-02-10 00:00:00	8.000

1179 rows × 3 columns

If you have dates and times columns separated or year and month columns separated, you can use “**parse_dates**” to merge them.

```
ws =
pd.read_excel('PuenteDiablo.xlsx',skiprows=2,usecols='A,D,E,F',parse_d
ates=[['DATE','HOUR']])
ws
```

```
ws = pd.read_excel('PuenteDiablo.xlsx',skiprows=2,usecols='A,D,E,F',parse_dates=[['DATE','HOUR']])
ws
```

	DATE_HOUR	STATION	VALUE
0	2016-08-01 06:00:00	PUENTE DEL DIABLO	5.832
1	2016-10-01 06:00:00	PUENTE DEL DIABLO	6.130
2	2016-10-01 08:00:00	PUENTE DEL DIABLO	6.130
3	2016-12-01 08:00:00	PUENTE DEL DIABLO	6.011
4	2016-01-13 08:00:00	PUENTE DEL DIABLO	5.801
...	...	...	...
1174	2020-09-25 06:00:00	PUENTE DEL DIABLO	8.000
1175	2020-09-28 06:00:00	PUENTE DEL DIABLO	8.300
1176	2020-09-30 06:00:00	PUENTE DEL DIABLO	8.000
1177	2020-01-10 06:00:00	PUENTE DEL DIABLO	8.000
1178	2020-02-10 06:00:00	PUENTE DEL DIABLO	8.000

1179 rows × 3 columns

Pandas allows us to filter data with “loc” and “**iloc**”, the latter one is used to filter based on the index position. For example, if we want to filter the first 500 values based in its index, we do the following:

```
ws.iloc[0:500]
```

```
ws.iloc[0:500]
```

	DATE_HOUR	STATION	VALUE
0	2016-08-01 06:00:00	PUENTE DEL DIABLO	5.832
1	2016-10-01 06:00:00	PUENTE DEL DIABLO	6.130
2	2016-10-01 08:00:00	PUENTE DEL DIABLO	6.130
3	2016-12-01 08:00:00	PUENTE DEL DIABLO	6.011
4	2016-01-13 08:00:00	PUENTE DEL DIABLO	5.801
...	...	...	...
495	2017-05-19 08:00:00	PUENTE DEL DIABLO	7.450
496	2017-05-20 08:00:00	PUENTE DEL DIABLO	7.690
497	2017-05-21 08:00:00	PUENTE DEL DIABLO	7.450
498	2017-05-22 08:00:00	PUENTE DEL DIABLO	8.311
499	2017-05-23 08:00:00	PUENTE DEL DIABLO	7.700

500 rows × 3 columns

Using "iloc" we can **filter rows and columns**, for example, if we want to get the value of the flow, which index is 2 in the index 45, we do the following.

```
ws.iloc[45, 2]
```

```
21.381
```

To place the date and hour column as index you can use "**set_index**"

```
ws = ws.set_index('DATE_HOUR')
ws
```

```
ws = ws.set_index('DATE_HOUR')
ws
```

	STATION	VALUE
DATE_HOUR		
2016-08-01 06:00:00	PUENTE DEL DIABLO	5.832
2016-10-01 06:00:00	PUENTE DEL DIABLO	6.130
2016-10-01 08:00:00	PUENTE DEL DIABLO	6.130
2016-12-01 08:00:00	PUENTE DEL DIABLO	6.011
2016-01-13 08:00:00	PUENTE DEL DIABLO	5.801
...	...	...
2020-09-25 06:00:00	PUENTE DEL DIABLO	8.000
2020-09-28 06:00:00	PUENTE DEL DIABLO	8.300
2020-09-30 06:00:00	PUENTE DEL DIABLO	8.000
2020-01-10 06:00:00	PUENTE DEL DIABLO	8.000
2020-02-10 06:00:00	PUENTE DEL DIABLO	8.000

1179 rows × 2 columns

“**loc**” is more versatile than “iloc”; it allows you to filter by the values of the index, for example, if they are dates, as in our case, you can choose directly the date you want.

```
ws.loc['2016-03-31']
```

```
ws.loc['2016-03-31']
```

	STATION	VALUE
DATE_HOUR		
2016-03-31 08:00:00	PUENTE DEL DIABLO	6.009

And in the same way, we can filter indexes and columns.

```
ws.loc['2016-03-31','VALUE']
```

```
ws.loc['2016-03-31','VALUE']
```

```
DATE_HOUR
2016-03-31 08:00:00    6.009
Name: VALUE, dtype: float64
```

We can **choose values** greater than a given year too.

```
ws.loc[ws.index>='2017']
```

```
ws.loc[ws.index>='2017']
```

	STATION	VALUE
DATE_HOUR		
2017-01-01 08:00:00	PUENTE DEL DIABLO	6.390
2017-02-01 08:00:00	PUENTE DEL DIABLO	6.416
2017-03-01 08:00:00	PUENTE DEL DIABLO	6.390
2017-04-01 08:00:00	PUENTE DEL DIABLO	10.000
2017-05-01 08:00:00	PUENTE DEL DIABLO	26.000
...	...	...
2020-09-25 06:00:00	PUENTE DEL DIABLO	8.000
2020-09-28 06:00:00	PUENTE DEL DIABLO	8.300
2020-09-30 06:00:00	PUENTE DEL DIABLO	8.000
2020-01-10 06:00:00	PUENTE DEL DIABLO	8.000
2020-02-10 06:00:00	PUENTE DEL DIABLO	8.000

822 rows × 2 columns

Or filter the values with a given year and month, or any combination we want.

```
ws.loc[ws.index>='2017-08']
```

```
ws.loc[ws.index>='2017-08']
```

DATE_HOUR	STATION	VALUE
2017-08-01 08:00:00	PUENTE DEL DIABLO	10.0
2017-09-01 08:00:00	PUENTE DEL DIABLO	8.0
2017-10-01 08:00:00	PUENTE DEL DIABLO	6.5
2017-11-01 08:00:00	PUENTE DEL DIABLO	7.5
2017-12-01 08:00:00	PUENTE DEL DIABLO	8.5
...	...	...
2020-09-25 06:00:00	PUENTE DEL DIABLO	8.0
2020-09-28 06:00:00	PUENTE DEL DIABLO	8.3
2020-09-30 06:00:00	PUENTE DEL DIABLO	8.0
2020-01-10 06:00:00	PUENTE DEL DIABLO	8.0
2020-02-10 06:00:00	PUENTE DEL DIABLO	8.0

609 rows × 2 columns

If for any case, your index is not in order, you can change the visualization to sort it using "**sort_index**"

```
ws.sort_index(inplace=True)
```

To **filter a column with a range** you can use the following conditional:

```
ws.loc[(ws.VALUE > 10) & (ws.VALUE < 15),'VALUE']
```

```
ws.loc[(ws.VALUE > 10) & (ws.VALUE < 15),'VALUE']
```

```
DATE_HOUR
2016-02-20 08:00:00    12.574
2017-01-14 08:00:00    13.500
2017-02-18 08:00:00    10.426
2017-02-24 08:00:00    10.370
2017-02-27 08:00:00    12.400
                        ...
2020-04-26 06:00:00    10.460
2020-04-27 06:00:00    10.860
2020-04-28 06:00:00    10.860
2020-04-29 06:00:00    10.060
2020-05-02 06:00:00    10.284
Name: VALUE, Length: 222, dtype: float64
```

You can **sort the table by any column**.

```
ws.sort_values(by='VALUE')
```

```
ws.sort_values(by='VALUE')
```

DATE_HOUR	STATION	VALUE
2018-10-23 08:00:00	PUENTE DEL DIABLO	0.090
2018-12-09 08:00:00	PUENTE DEL DIABLO	0.101
2018-08-09 08:00:00	PUENTE DEL DIABLO	0.102
2018-10-22 08:00:00	PUENTE DEL DIABLO	0.108
2018-07-09 08:00:00	PUENTE DEL DIABLO	0.117
...	...	...
2019-11-02 06:00:00	PUENTE DEL DIABLO	125.000
2019-02-22 08:00:00	PUENTE DEL DIABLO	134.637
2020-02-13 06:00:00	PUENTE DEL DIABLO	155.500
2019-09-02 17:43:00	PUENTE DEL DIABLO	175.000
2016-02-25 06:00:00	PUENTE DEL DIABLO	200.000

1179 rows × 2 columns

If you use the following expression, what you get are **Booleans** regarding the conditional.

```
ws['VALUE']>50
```

```
ws['VALUE']>50
```

```
DATE_HOUR
2016-01-02 08:00:00    False
2016-01-03 08:00:00    False
2016-01-04 08:00:00    False
2016-01-05 08:00:00    False
2016-01-06 08:00:00    False
                       ...
2020-12-02 06:00:00    False
2020-12-04 06:00:00    False
2020-12-05 06:00:00    False
2020-12-06 06:00:00    False
2020-12-08 06:00:00    False
Name: VALUE, Length: 1179, dtype: bool
```

You can use the previous Booleans to **filter the output data**.

```
ws[ws['VALUE']>50]
```

```
ws[ws['VALUE']>50]
```

	STATION	VALUE
DATE_HOUR		
2016-02-25 06:00:00	PUENTE DEL DIABLO	200.000
2016-02-26 08:00:00	PUENTE DEL DIABLO	65.000
2016-02-27 08:00:00	PUENTE DEL DIABLO	74.180
2017-02-04 08:00:00	PUENTE DEL DIABLO	58.000
2017-03-04 08:00:00	PUENTE DEL DIABLO	80.000
2017-03-15 08:00:00	PUENTE DEL DIABLO	60.000
2017-03-16 08:00:00	PUENTE DEL DIABLO	80.000
2017-03-17 08:00:00	PUENTE DEL DIABLO	65.000
2017-04-04 08:00:00	PUENTE DEL DIABLO	55.000
2018-03-22 08:00:00	PUENTE DEL DIABLO	52.800
2018-03-23 08:00:00	PUENTE DEL DIABLO	80.000
2018-03-24 08:00:00	PUENTE DEL DIABLO	54.319

Let's change the name of the column "Value" to add the units in it. To do this, we need to use "**rename**" and indicate in a dictionary the columns with its new name.

```
ws.rename(columns = {'VALUE':'VALUE (m3/d)'}, inplace = False)
ws
```

```
ws.rename(columns = {'VALUE':'VALUE (m3/d)'}, inplace = True)
ws
```

Let's create a new column with **flow values in cubic feet**.

```
ws['VALUE (ft3/d)'] = ws['VALUE (m3/d)']/(0.3048**3)
ws.head()
```

```
ws['VALUE (ft3/d)'] = ws['VALUE (m3/d)']/(0.3048**3)
ws.head()
```

DATE_HOUR	STATION	VALUE (m3/d)	VALUE (ft3/d)
2016-01-02 08:00:00	PUENTE DEL DIABLO	5.720	201.999894
2016-01-03 08:00:00	PUENTE DEL DIABLO	20.550	725.716401
2016-01-04 08:00:00	PUENTE DEL DIABLO	5.909	208.674366
2016-01-05 08:00:00	PUENTE DEL DIABLO	6.295	222.305827
2016-01-06 08:00:00	PUENTE DEL DIABLO	7.450	263.094267

We can make **many statistics** based on the columns.

For the **sum** of the values:

```
ws['VALUE (ft3/d)'].sum()
```

```
ws['VALUE (ft3/d)'].sum()
```

```
564482.5227296269
```

For the **mode**:

```
ws['VALUE (ft3/d)'].mode()
```

```
ws['VALUE (ft3/d)'].mode()
```

```
0    282.517334
dtype: float64
```

For any **quantile**:

```
ws['VALUE (ft3/d)'].quantile(0.5)
```

```
ws['VALUE (ft3/d)'].quantile(0.5)
```

```
307.23760047695066
```

For the **standard deviation**:

```
ws['VALUE (ft3/d)'].std()
```

For the **variance**:

```
ws['VALUE (ft3/d)'].var()
```

```
ws['VALUE (ft3/d)'].var()
```

```
349235.09606167115
```

You can get the **maximum and minimum values** in a column or index, or even a row.

```
ws.index.min() , ws.index.max()
```

```
ws.index.min() , ws.index.max()
```

```
(Timestamp('2016-01-02 08:00:00'), Timestamp('2020-12-08 06:00:00'))
```

As the dates are in Datetime format, you can make mathematical operations with them.

```
ws.index.max() - ws.index.min()
```

```
ws.index.max() - ws.index.min()
```

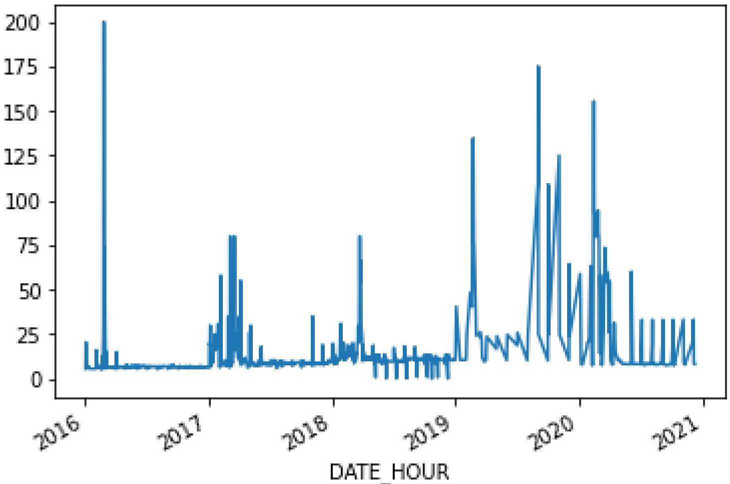
```
Timedelta('1801 days 22:00:00')
```

To have a **quick plot** of the distribution:

```
ws['VALUE (m3/d)'].plot()
```

```
ws['VALUE (m3/d)'].plot()
```

```
<AxesSubplot:xlabel='DATE_HOUR'>
```

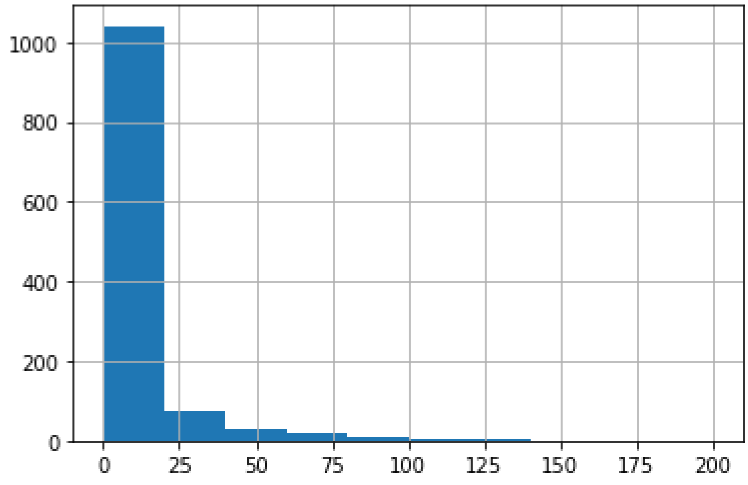


For a **histogram**:

```
ws['VALUE (m3/d)'].hist()
```

```
ws['VALUE (m3/d)'].hist()
```

```
<AxesSubplot:>
```

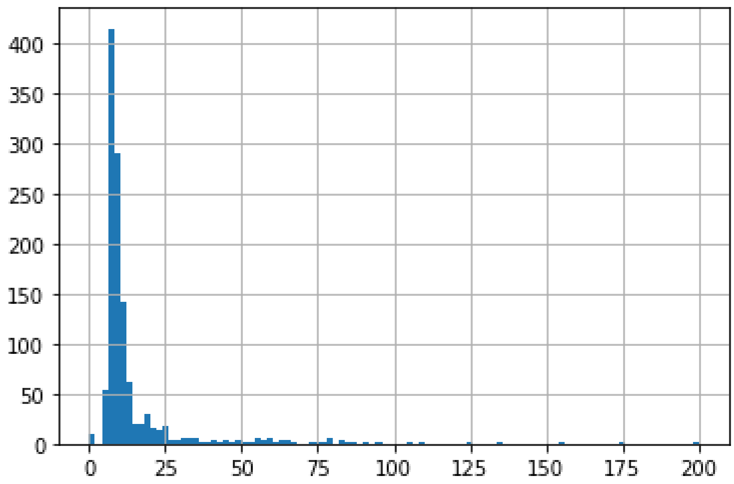


You can change the **number of bins**:

```
ws['VALUE (m3/d)'].hist(bins = 100)
```

```
ws['VALUE (m3/d)'].hist(bins = 100)
```

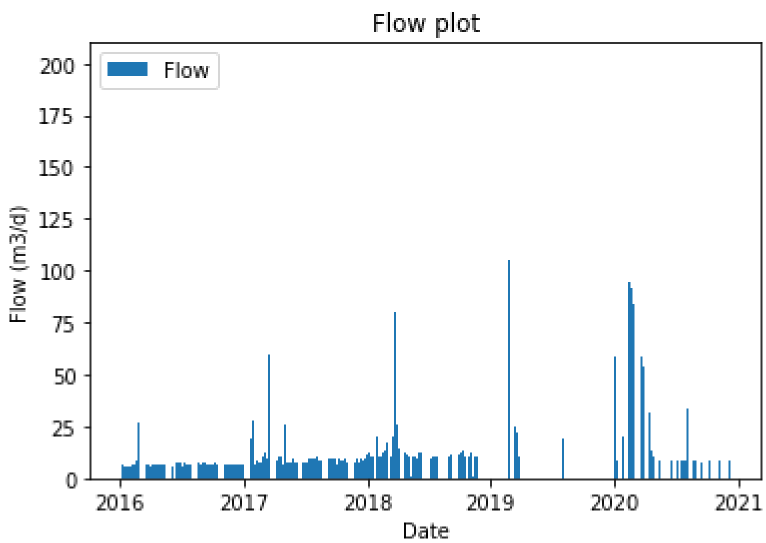
```
<AxesSubplot:>
```



A more **detailed plot** can be made with the steps of Session 3.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.bar(ws.index, ws['VALUE (m3/d)'].values, label='Flow')
ax.set_xlabel('Date')
ax.set_ylabel('Flow (m3/d)')
ax.set_title('Flow plot')
ax.legend(loc='upper left')
plt.show()
```


```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.bar(ws.index, ws['VALUE (m3/d)'].values, label='Flow')
ax.set_xlabel('Date')
ax.set_ylabel('Flow (m3/d)')
ax.set_title('Flow plot')
ax.legend(loc='upper left')
plt.show()
```

In case you want to **add a column with conditionals** based on a column of a DataFrame, you would need to use conditionals. The list or array can be created apart, or you can use a one-line statement as in the following example in which we are creating a column called “Quantile” with 3 kinds of quantiles.

```
ws['QUANTILE'] = ['q25' if x <= 6.92 else ('q50' if x<=8.7 else 'q70')
for x in ws['VALUE (m3/d)']]
ws
```

```
ws['QUANTILE'] = ['q25' if x <= 6.92 else ('q50' if x<=8.7 else 'q70') for x in ws['VALUE (m3/d)']]
ws
```

	STATION	VALUE (m3/d)	VALUE (ft3/d)	QUANTILE
DATE_HOUR				
2016-01-02 08:00:00	PUENTE DEL DIABLO	5.720	201.999894	q25
2016-01-03 08:00:00	PUENTE DEL DIABLO	20.550	725.716401	q70
2016-01-04 08:00:00	PUENTE DEL DIABLO	5.909	208.674366	q25
2016-01-05 08:00:00	PUENTE DEL DIABLO	6.295	222.305827	q25
2016-01-06 08:00:00	PUENTE DEL DIABLO	7.450	263.094267	q50
...	...	...	...	...
2020-12-02 06:00:00	PUENTE DEL DIABLO	21.061	743.762196	q70
2020-12-04 06:00:00	PUENTE DEL DIABLO	33.250	1174.212668	q70
2020-12-05 06:00:00	PUENTE DEL DIABLO	8.480	299.468374	q50
2020-12-06 06:00:00	PUENTE DEL DIABLO	8.480	299.468374	q50
2020-12-08 06:00:00	PUENTE DEL DIABLO	8.300	293.111734	q50

1179 rows × 4 columns

The last feature we explore is **Groups**. For example, if we group our new column created before, we use the following expression with returns us a “Groupby” object.

```
ws.groupby(['QUANTILE'])
```

```
ws.groupby(['QUANTILE'])
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001C32539BB80>
```

Based on the previous object, we can create **statistics of the other columns based on the groupby**.

```
ws.groupby(['QUANTILE']).mean()
```

```
ws.groupby(['QUANTILE']).mean()
```

	VALUE (m3/d)	VALUE (ft3/d)
QUANTILE		
q25	6.244923	220.537368
q50	7.876918	278.170727
q70	20.073545	708.890551

If we **"Group" by values**, we would have many groups based on the number of the values, so making "Groups" only make sense we have qualifier fields.

```
ws.groupby(['VALUE (ft3/d)']).mean()
```

```
ws.groupby(['VALUE (ft3/d)']).mean()
```

	VALUE (m3/d)
VALUE (ft3/d)	
3.178320	0.090
3.566781	0.101
3.602096	0.102
3.813984	0.108
4.131816	0.117
...	...
4414.333340	125.000
4754.660783	134.637
5491.430675	155.500
6180.066676	175.000
7062.933344	200.000

765 rows × 1 columns

Another example of "Group" would be filtering first the columns you want to analyze

```
ws[['VALUE (m3/d)','QUANTILE']].groupby(['QUANTILE']).mean()
```

```
ws[['VALUE (m3/d)','QUANTILE']].groupby(['QUANTILE']).mean()
```

QUANTILE	VALUE (m3/d)
q25	6.244923
q50	7.876918
q70	20.073545

Another example of statistic:

```
ws.groupby(['QUANTILE']).sum()
```

```
ws.groupby(['QUANTILE']).sum()
```

QUANTILE	VALUE (m3/d)	VALUE (ft3/d)
q25	1860.987	65720.135678
q50	2300.060	81225.852339
q70	11823.318	417536.534712

Finally, we can export our table to a "CSV" file that can be opened inside Jupyter Lab.

```
ws.to_csv('WeatherStation.csv')
```

Or you can export your "Group" created before:

```
ws.groupby(['QUANTILE']).mean().to_csv('WeatherStation_quantiles.csv')
```

Sesion4.ipynb | WeatherStation_quantiles.csv | temp.txt

Delimiter: ,

	QUANTILE	VALUE (m3/d)	VALUE (ft3/d)
1	q25	6.244922818791947	220.53736804705684
2	q50	7.8769178082191775	278.170727189819
3	q70	20.073544991511028	708.8905512940183

But, “CSV” is not the only format in which you can export. You can use all of the following methods too.

pandas.DataFrame.to_csv

pandas.DataFrame.to_hdf

pandas.DataFrame.to_sql

pandas.DataFrame.to_dict

pandas.DataFrame.to_excel

pandas.DataFrame.to_json

pandas.DataFrame.to_html

pandas.DataFrame.to_feather

pandas.DataFrame.to_latex

pandas.DataFrame.to_stata

pandas.DataFrame.to_gbq

pandas.DataFrame.to_records

pandas.DataFrame.to_string

pandas.DataFrame.to_clipboard

pandas.DataFrame.to_markdown

References:

Most of the references of how to use Pandas come from the documentation:

https://pandas.pydata.org/pandas-docs/stable/index.html