Data manipulation: Pandas

AAA-Python Edition

Plan

- 1- Pandas: Series
- 2- Pandas: DataFrame
- 3- Indexing and Reindexing
- 4- Some operations
- 5- Google colab help

1- Pandas Series

pandas

- **pandas is** a library that defines **data structures** and **manipulation** tools to be used in Python. It is often combined with other numerical libraries like **Numpy**.

- In **pandas** we can work with **tabular** or **heterogeneous** data by using for example its defined structures **DataFrame**.

- The other important pandas **structure** is: **Series structure**
- Each of the two previous structures are used with an other defined object in pandas: **Index** object.

```python
from pandas import Series as S, DataFrame as DF
# wecould use pandas.Series([1,2,3])
s1 = S([1,2,3])
print("s1==\n",s1)
df1 = DF([1,2,3])
print("df1==\n",df1)
```

```
s1==
 0    1
1    2
2    3
dtype: int64
df1==
    0
0  1
1  2
2  3
```

Importing the two modules corresponding to the structures as S and DF

Giving only a list as a Parameter, default indexes were created

A default column label

[By Amina Delali]

1- Pandas Series

Series

- A **Series** is a **sequence** of values of the **same type** associated with a sequence of **labels** called **index**.

```
# printing the index and the values of a series object
print("index==",s1.index)
print("values==",s1.values)
```

```
index== RangeIndex(start=0, stop=3, step=1)
values== [1 2 3]
```

The default index created

The **length** of the **index** must be **equal** to the **list's length**

```
import numpy as np
# creating a series specifying a list and an associated index
s2 = S(list("His"),index=[1,2,3])
print (s2)
s2_2 = S("His",index=[1,2])
print(s2_2)
print("----------------")
```

```
1    H
2    i
3    s
dtype: object
1    His
2    His
dtype: object
```

Creating a list from a string ==> a list of characters

"his" is one scalar value ==> the index can be greater than one

1- Pandas Series

Series

```
# creating a series with a dict object without and with an index
s3 = S({"Third":3,"Second":2,"First":1})
print(s3)
s3_2=S(d,["Third","First","Other"])
print(s3_2)
print("----------------")
```

```
First      1
Second     2
Third      3
dtype: int64
Third    3.0
First    1.0
Other    NaN
dtype: float64
---------------
```

The **sorted** dictionary keys will be the series' **Index** (the series values will be sorted according to this index)

If the **key exists**, the Corresponding value is Added.
If it **doesn't exist**, a **Nan** Value will be added

If a key **is missed**, its corresponding value will **not be added**

```
# creating a series with ndarray and an associated index
S4 = S(np.random.randn(2),range(0,2))
print(S4)
```

```
0    0.892413
1    0.170311
dtype: float64
```

Same length as ndarray length

[By Amina Delali]

2- Pandas DataFrame

DataFrame

- A **DataFrame** is **rectangular table** of data organized in **rows** and **columns** **associated** with rows and columns **indexes** respectively.

```
# creating a DataFrame from a list of lists
df2= DF([[1,2,3,4],list("Try it"),list(np.random.randn(5))])
df2
```

The lists are not of the same dtype neither the same length

Missing values

	0	1	2	3	4	5
0	1	2	3	4	None	None
1	T	r	y		i	t
2	0.422495	-1.25771	0.208447	0.590705	-0.0559943	None

List of 3 lists ==>**3** rows

The maximum lists lengths == 5 ==> 5 columns

Both lines and columns can have different dtype (**heterogeneous** data type)

In this case, each column has different dtypes

[By Amina Delali]

DataFrame

```
# creating a DataFrame from list of series:
df4 = DF([S([1,2,3],index=list("abc")),S(["Home","Work","Travel"],index=list("abc"))],index=list("GH"))
df4
```

Column indexes: **3 different** values of indexes ==> **3 columns**

	a	b	c
G	1	2	3
H	Home	Work	Travel

List of 2 Series ==> **2 rows**

```
# creating a DataFrame from a dict of lists
df3 = DF({"Verbs":["Call","Eat","Drive"],"Occurences":[25,3,12]})
df3
```

Number of **Keys == 2 ==>** Number of **Columns == 2**

Lists must have same length: Lists' **length == 3 ==> number of rows == 3**

	Occurences	Verbs
0	25	Call
1	3	Eat
2	12	Drive

One row has different dtypes

[By Amina Delali]

2- Pandas DataFrame

DataFrame

```
# creating a DataFrame from a dict of dicts
df4 = DF({"Verbs":{0:"Call",1:"Eat",2:"Drive"},"Occurences":{3:25,4:3,5:12}})
df4
```

2 Outer Keys == 2 Columns

6 different inner keys == 6 rows

Missing values: the dicts have **different keys**

	Occurences	Verbs
0	NaN	Call
1	NaN	Eat
2	NaN	Drive
3	25.0	NaN
4	3.0	NaN
5	12.0	NaN

```
# creating a DataFrame from a dict of dicts, specifying the rows and columns
df5 = DF({"Verbs":{0:"Call",1:"Eat",2:"Drive"}},index=[1,2],columns=["Verbs"])
df5
```

Selecting the rows **1** And **2**

	Verbs
1	Eat
2	Drive

Selecting the column **Verbs**

[By Amina Delali]

. Indexing and filtering in Series

```python
# creating a Series
ser= S(range(1,4),index=list("abc"))
# Selecting one element using the given index
print('ser["a"]==',ser["a"])
# Selecting the same element using the default index
print("ser[0]==",ser[0])
# Selecting a slice of elements
print('ser["a":"b"]==',ser["a":"b"])
# But using the default index, will not give the same results:
print('ser[0:1]==',ser[0:1])
# Selecting or filtering values grater than 2
print('ser[ser>2]==',ser[ser>2])
# Selecting a list of elements
print('ser[["a","c"]]==',ser[["a","c"]])
# Assigning a value to a selected slice will affect the original value
ser["a":"b"]=1000
print('ser==',ser)
```

```
ser["a"]== 1
ser[0]== 1
```

This index wasn't specified in the creation of the series

```
ser["a":"b"]== a    1
b    2
dtype: int64
```

Using the default index will not produce the same results:

```
ser[0:1]== a    1
dtype: int64
```

```
ser[["a","c"]]== a    1
c    3
dtype: int64
ser== a    1000
b    1000
c       3
dtype: int64
```

```
ser[ser>2]== c    3
dtype: int64
```

[By Amina Delali]

3- Indexing and reindexing

• **Indexing** and **filtering** in **DataFrame**

```
# creating a DataFrame
dfr= DF([["a",1],["b",2],["c",3]],index=["r1","r2","r3"],columns=["letters","digits"])
# Selecting one element using the given index
# selecting a row
print('dfr.loc["r1"]==\n',dfr.loc["r1"])

# Selecting a column
print('dfr["letters"]==\n',dfr["letters"])
print('dfr.letters==\n',dfr.letters)
print('dfr.loc[:,"letters"]==\n',dfr.loc[:,"letters"])

# Selecting the same column using the default index for columns
print("dfr.iloc[:,0]==\n",dfr.iloc[:,0])
```

```
dfr.loc["r1"]==
 letters    a
digits     1
Name: r1, dtype: object
```

```
dfr["letters"]==
 r1    a
r2    b
r3    c
Name: letters, dtype: object
```

Access to a column as attribute

```
dfr.letters==
 r1    a
r2    b
r3    c
Name: letters, dtype: object
```

```
dfr.iloc[:,0]==
 r1    a
r2    b
r3    c
Name: letters, dtype: object
```

```
dfr.loc[:,"letters"]==
 r1    a
r2    b
r3    c
Name: letters, dtype: object
```

```
dfr[:1]==
    letters  digits
r1        a       1
```

```
dfr.iloc[0:1]==
    letters  digits
r1        a       1
```

```
For rows, if we want to use the default index, we can use :
a slice or iloc (the iloc for the same slice will produce the same result)
print("dfr[:1]==\n",dfr[:1])
print("dfr.iloc[0]==\n",dfr.iloc[0])
print("dfr.iloc[0:1]==\n",dfr.iloc[0:1])
```

```
dfr.iloc[0]==
 letters    a
digits     1
Name: r1, dtype: object
```

[By Amina Delali]

• **Indexing** and **filtering** in **DataFrame**

3- Indexing and reindexing

```python
# Selecting a slice of elements: for columns we can use :
print("dfr.iloc[:,0:2]==\n",dfr.iloc[:,0:2])
print('dfr.loc[:,"letters":"letters"]==\n',dfr.loc[:,"letters":"letters"])


# Selecting a slice of elements: for rows we can use row labels

print('dfr["r1":"r2"]==\n',dfr["r1":"r2"])
#           with loc

print('dfr.loc["r1":"r2"]==\n',dfr.loc["r1":"r2"])

# or  default indexes
print('dfr[0:2]==\n',dfr[0:2])
#            with iloc
print('dfr.iloc[0:2]==\n',dfr.iloc[0:2])
```

```
dfr.iloc[:,0:2]==
     letters  digits
r1         a       1
r2         b       2
r3         c       3
```

```
dfr.loc[:,"letters":"letters"]==
     letters
r1         a
r2         b
r3         c
```

```
dfr.loc["r1":"r2"]==
     letters  digits
r1         a       1
r2         b       2
```

```
dfr["r1":"r2"]==
     letters  digits
r1         a       1
r2         b       2
```

```
dfr[0:2]==
     letters  digits
r1         a       1
r2         b       2
```

```
dfr.iloc[0:2]==
     letters  digits
r1         a       1
r2         b       2
```

```
dfr[dfr["digits"]>2]==
     letters  digits
r3         c       3
```

```python
# Selecting or filtering values greater than 2: selecting rows
print('dfr[dfr["digits"]>2]==\n',dfr[dfr["digits"]>2])
# Selecting or filtering values greater than 2:

print('dfr>2==\n',dfr>2)
print('dfr[dfr>2]==\n',dfr[dfr>2])
```

```
dfr>2==
     letters  digits
r1      True   False
r2      True   False
r3      True    True
```

```
dfr[dfr>2]==
     letters  digits
r1         a     NaN
r2         b     NaN
r3         c     3.0
```

Selecting values corresponding to True

[By Amina Delali]

3- Indexing and reindexing

• Indexing and filtering in DataFrame

```python
# Selecting a list of rows
#     with labels: only with loc
print('dfr.loc[["r1","r3"]]==\n',dfr.loc[["r1","r3"]])
#     with default indexes: only with iloc
print('dfr.iloc[[0,2]]==\n',dfr.iloc[[0,2]])
```

```
dfr.loc[["r1","r3"]]==
   letters  digits
r1       a       1
r3       c       3
```

```
dfr.iloc[[0,2]]==
   letters  digits
r1       a       1
r3       c       3
```

```python
# Selecting a list of columns:
#     with labels
print('dfr[["digits","letters"]]==\n',dfr[["digits","letters"]])
print('dfr.loc[:,["digits","letters"]]==\n',dfr.loc[:,["digits","letters"]])
#     with default indexes: only with iloc
print('dfr.iloc[:,[1,0]]==\n',dfr.iloc[:,[1,0]])
```

```
dfr.iloc[:,[1,0]]==
    digits letters
r1       1       a
r2       2       b
r3       3       c
```

```
dfr.loc[:,["digits","letters"]]==
    digits letters
r1       1       a
r2       2       b
r3       3       c
```

```
dfr[["digits","letters"]]==
    digits letters
r1       1       a
r2       2       b
r3       3       c
```

```python
# selecting one value using the labels and default indexes with at and iat
print('dfr.at["r1","digits"]==\n',dfr.at["r1","digits"])
print('dfr.iat[0,1]==\n',dfr.iat[0,1])
```

```
dfr.at["r1","digits"]==
 1
dfr.iat[0,1]==
 1
```

```python
# selecting one value using the labels and default indexes with loc and iloc
print('dfr.loc["r1","digits"]==\n',dfr.loc["r1","digits"])
print('dfr.iloc[0,1]==\n',dfr.iloc[0,1])
```

```
dfr.loc["r1","digits"]==
 1
dfr.iloc[0,1]==
 1
```

[By Amina Delali]

3- Indexing and reindexing

• Indexing and filtering in DataFrame

```
# Assigning a value to a selected slice will affect the original value
dfr["r1":"r2"]=1000
dfr
```

	letters	digits
r1	1000	1000
r2	1000	1000
r3	c	3

• The following table will summarize the indexation possibilities:

Indexing	Using labels		Using default indexes	
	directly	loc	directly	iloc
On value		X and at method		X and iat method
One row		X	Using a slice	X
One column	X	X		X
A slice of rows	X	X	X	X
A slice of columns		X		X
A portion		X		X
A list of rows		X		X
A list of columns	X	X		X

[By Amina Delali]

3- Indexing and reindexing

- **Reindexing: creating** a **new Series or DataFrame** by **changing** the **order** of a given Series or DataFrame values.

```
#reindexing a series filling the missed values with
# a forward fill method
print(s1)
rs1=s1.reindex([3,2,1,0],method="ffill")
print(rs1)
```

Before

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

After

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

A new value created with the ffill method

```
#reindexing a DataFrame filling the missing values
# with a given argument value
rdf4 =df4.reindex(list("HGI"),columns=["c","b"],fill_value=-1)
rdf4
```

After

	c	b
H	Travel	Work
G	3	2
I	-1	-1

Before

	a	b	c
G	1	2	3
H	Home	Work	Travel

A new value created with the given fill_value

[By Amina Delali]

4- Some Operations

- **Dropping: creating** a **new Series or DataFrame** by **dropping** the **rows** or **columns** of a given Series or DataFrame.

```
# creating a new series
newS= S(np.random.randn(3), index=list("abc"))
print(newS)
# Dropping the first and last values
print(newS.drop(['a','c']))
```

Before

```
a   -1.731791
b   -0.026798
c    0.698285
dtype: float64
```

After

```
b   -0.026798
dtype: float64
```

Deleted rows

```
# creating a new series
newDF= DF(np.random.randn(6).reshape(2,3), index=list("ab"),columns=list("ABC"))
print(newDF)
# Dropping the second Column
print(newDF.drop('B',axis=1))
```

Creating a DataFrame specifying a **2** dimensional ndarray as argument

Deleted column

Before

```
          A         B         C
a -0.502287  0.897991  1.442152
b -0.427633  0.465693  0.200721
```

After

```
          A         C
a -0.502287  1.442152
b -0.427633  0.200721
```

[By Amina Delali]

4- Some Operations

Some other operations

- We can apply **arithmetic operations** using **operators** or **defined** methods:

```
df1 = DF(np.arange(6).reshape(2,3),index=["r1","r2"],columns=["c1","c2","c3"])
df1
```

	c1	c2	c3
r1	0	1	2
r2	3	4	5

```
df2 = DF(np.ones((3,3)),index=["r1","r2","r3"],columns=["c1","c2","c3"] )
df2
```

	c1	c2	c3
r1	1.0	1.0	1.0
r2	1.0	1.0	1.0
r3	1.0	1.0	1.0

```
# the rows and columns will be aligned
df1 + df2
```

	c1	c2	c3
r1	1.0	2.0	3.0
r2	4.0	5.0	6.0
r3	NaN	NaN	NaN

df1 doesn't have r3 row

```
# using the add method : we can fill the missing values
# the fill value will replace the missing values before applying the operation
df1.add(df2,fill_value=5)
```

	c1	c2	c3
r1	1.0	2.	
r2	4.0	5.	
r3	6.0	6.0	6.0

The missing values in df1 were replace by 1 then added to r3 df2's row

4- Some Operations

Some other operations

```
1 ser1 = S(range(6),index=list("abcdef"))
2 ser1
```

```
a    0
b    1
c    2
d    3
e    4
f    5
dtype: int64
```

```
1 ser2= S([1]*5,index=list("abcde"))
2 ser2
```

```
a    1
b    1
c    1
d    1
e    1
dtype: int64
```

The same method exists in DataFrame

```
1 # appliying a division between two series
2 ser1.div(ser2)
3
```

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
f    NaN
dtype: float64
```

Missing value in ser2

```
1 ser3 = S([5,6],index=["c1","c2"])
2 ser3
```

```
c1    5
c2    6
dtype: int64
```

If axis =0, will add column by column Matching rows labels

```
1 # applying a reversed division between two series
2 ser1.rdiv(ser2)
```

```
a         inf
b    1.000000
c    0.500000
d    0.333333
e    0.250000
f         NaN
dtype: float64
```

Dividing ser2 values by ser1 values

```
1 # operation between a Series and a DataFrame
2 df1.add(ser3) # or df1.add(ser3,axis=1)
```

	c1	c2	c3
r1	5.0	7.0	NaN
r2	8.0	10.0	NaN

Add row by row, matching columns labels

[By Amina Delali]

4- Some Operations

- we can **apply** functions on pandas structures just by using the structures as arguments or by using the: **apply** , **map** or the **applymap** method.

```
# call of function mean
np.mean(df1)
```

```
c1     1.5
c2     2.5
c3     3.5
dtype: float64
```

```
def f1(x):
    return S(np.sum(x),index=["sum"])

df1.apply(f1,axis=0)
```

	c1	c2	c3
sum	3	5	7

Apply to each column

```
def f2(x):
    return float(x)
df1.applymap(f2)
```

	c1	c2	c3
r1	0.0	1.0	2.0
r2	3.0	4.0	5.0

Apply to each element

```
def f3(x):
    return np.where(x>3,"Yes","No")

df1.loc["r2"].map(f3)
```

```
c1      No
c2     Yes
c3     Yes
Name: r2, dtype: object
```

Defined for Series

[By Amina Delali]

Sorting and Ranking

- The pandas structures can be **sorted** either by **indexes** or by **values**
- The values can also be **ranked** considering their **position** in a **sorting**

```
df1.sort_index(ascending=False)
```

	c1	c2	c3
r2	3	4	5
r1	0	1	2

The indexes were sorted (so their corresponding rows)

The column c2 was sorted

```
df1.iat[0,1]=1000
df1.sort_values(by=["c2"])
```

	c1	c2	c3
r2	3	4	5
r1	0	1000	2

```
ser4=S([6,5,1,6,9,0,-3])
ser4.sort_values()
```

```
6    -3
5     0
2     1
1     5
0     6
3     6
4     9
dtype: int64
```

6 is at the 5th and 6th position,so it is ranked the mean of those Positions (5 + 6)/2 == 5.5

```
ser4.rank()
```

```
0    5.5
1    4.0
2    3.0
3    5.5
4    7.0
5    2.0
6    1.0
dtype: float64
```

4- Some Operations

4- Some Operations

Descriptive operations

- There is a set of **methods** and **functions** that produce some **descriptive** values about the **data** contained in the corresponding structure.

```
df1
```

	c1	c2	c3
r1	0	1000	2
r2	3	4	5

```
# some descriptive values
df1.describe()
```

	c1	c2	c3
count	2.00000	2.000000	2.00000
mean	1.50000	502.000000	3.50000
std	2.12132	704.278354	2.12132
min	0.00000	4.000000	2.00000
25%	0.75000	253.000000	2.75000
50%	1.50000	502.000000	3.50000
75%	2.25000	751.000000	4.25000
max	3.00000	1000.000000	5.00000

```
ser2
```

```
a    1
b    1
c    1
d    1
e    1
dtype: int64
```

The unique different Value is **1**

```
# return all the uniques values
ser2.unique()
```

```
array([1])
```

```
# count the number of each unique value
ser2.value_counts()
```

```
1    5
dtype: int64
```

There are **5** “1”

[By Amina Delali]

Descriptive operations

```
# check if the DataFrame values are in the argument values
df1.isin([2,3])
```

	c1	c2	c3
r1	False	False	True
r2	True	False	False

[r1,c3] and[r2,c1] are in [2,3]

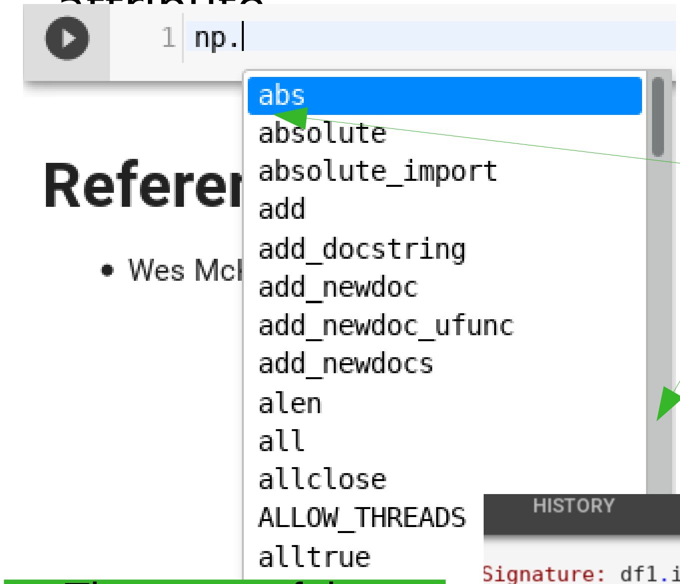
c and d are In [2,3]

```
#  check if the Series values are in the argument values
ser1.isin([2,3])
```

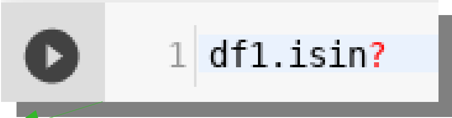
```
a     False
b     False
c      True
d      True
e     False
f     False
dtype: bool
```

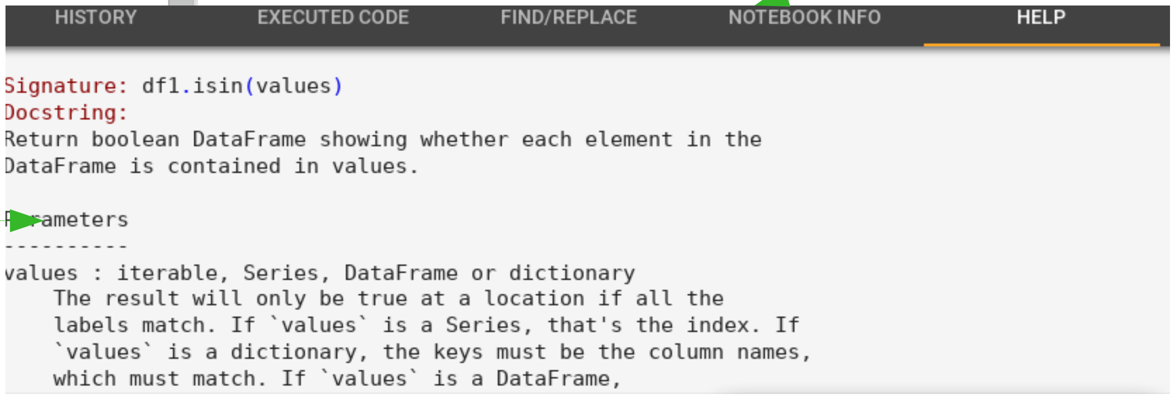
5- Google Colab Help

- Google Colab allows us to see the list of the available modules and function in a given module.
- It allows us also to access to the help of a given function or attribute.



The name of the module: **np** followed by a dot: **.** So: **np.** followed by a tabulation (key)





The name of the function or attribute followed by the "**?**" and run, will display this help

[By Amina Delali]

References

- Wes McKinney. Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. O’Reilly Media, Inc, 2018.

Thank you!

FOR ALL YOUR TIME