



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.

```
1 from pandas import Series as S, DataFrame as DF
2 # we could use pandas.Series([1,2,3])
3 s1 = S([1,2,3])
4 print("s1==\n", s1)
5 df1 = DF([1,2,3])
6 print("df1==\n", df1)
```

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

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

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

A default column label



1- Pandas Series

Series

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

The default index created

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



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

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

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

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

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

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



1- Pandas Series

Series

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

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

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

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

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

Same length as ndarray length

0	0.892413
1	0.170311
dtype:	float64



2- Pandas DataFrame

DataFrame

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

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

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

Missing values

List of 3 lists
==> 3 rows

	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

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

In this case,
each column
has different
dtypes

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



2- Pandas DataFrame

DataFrame

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

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



2- Pandas DataFrame

DataFrame

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

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

Selecting
the rows 1
And 2

	Verbs
1	Eat
2	Drive

Selecting the
column Verbs



3- Indexing and reindexing

. Indexing and filtering in Series

```
1 # creating a Series
2 ser= S(range(1,4),index=list("abc"))
3 # Selecting one element using the given index
4 print('ser["a"]==',ser["a"])
5 # Selecting the same element using the default index
6 print("ser[0]==",ser[0])
7 # Selecting a slice of elements
8 print('ser["a":"b"]==',ser["a":"b"])
9 # But using the default index, will not give the same results:
10 print('ser[0:1]==',ser[0:1])
11 # Selecting or filtering values greater than 2
12 print('ser[ser>2]==',ser[ser>2])
13 # Selecting a list of elements
14 print('ser[["a","c"]]==',ser[["a","c"]])
15 # Assigning a value to a selected slice will affect the original value
16 ser["a":"b"]=1000
17 print('ser==',ser)
```

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

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

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

Using the default index will not produce the same results:

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

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

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



3- Indexing and reindexing

- Indexing and filtering in DataFrame

```
1 # creating a DataFrame
2 dfr= DF([["a",1],["b",2],["c",3]],index=["r1","r2","r3"],columns=["letters","digits"])
3 # Selecting one element using the given index
4 # selecting a row
5 print('dfr.loc["r1"]==\n',dfr.loc["r1"])
6
7 # Selecting a column
8 print('dfr["letters"]==\n',dfr["letters"])
9 print('dfr.letters==\n',dfr.letters)
10 print('dfr.loc[:, "letters"]==\n',dfr.loc[:, "letters"])
11
12 # Selecting the same column using the default index for columns
13 print("dfr.iloc[:,0]==\n",dfr.iloc[:,0])
14
```

Access to a column as attribute

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

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

[By Amina Delali]

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

```
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
```

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)

```
18 print("dfr[:1]==\n",dfr[:1])
19 print("dfr.iloc[0]==\n",dfr.iloc[0])
20 print("dfr.iloc[0:1]==\n",dfr.iloc[0:1])
```

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



3- Indexing and reindexing

- Indexing and filtering in DataFrame

```
21 # Selecting a slice of elements: for columns we can use :
22 print("dfr.iloc[:,0:2]==\n",dfr.iloc[:,0:2])
23 print('dfr.loc[:, "letters": "letters"]==\n',dfr.loc[:, "letters": "letters"])
24
25 # Selecting a slice of elements: for rows we can use row labels
26
27 print('dfr["r1": "r2"]==\n',dfr["r1": "r2"])
28 # with loc
29 print('dfr.loc["r1": "r2"]==\n',dfr.loc["r1": "r2"])
30
31 # or default indexes
32 print('dfr[0:2]==\n',dfr[0:2])
33 # with iloc
34 print('dfr.iloc[0:2]==\n',dfr.iloc[0:2])
35
36 dfr[0:2]==
37 letters digits
38 r1 a 1
39 r2 b 2
40
41 dfr.iloc[0:2]==
42 letters digits
43 r1 a 1
44 r2 b 2
45
46 dfr[dfr["digits"]>2]==
47 letters digits
48 r3 c 3
49
50 # Selecting or filtering values greater than 2: selecting rows
51 print('dfr[dfr["digits"]>2]==\n',dfr[dfr["digits"]>2])
52 # Selecting or filtering values greater than 2:
53
54 print('dfr>2==\n',dfr>2)
55 print('dfr[dfr>2]==\n',dfr[dfr>2])
56
57 dfr[dfr>2]==
58 letters digits
59 r1 a NaN
60 r2 b NaN
61 r3 c 3.0
62
63 dfr>2==
64 letters digits
65 r1 True False
66 r2 True False
67 r3 True True
```

Diagram illustrating DataFrame indexing and filtering operations:

- Column Selection:** `dfr.iloc[:,0:2]` and `dfr.loc[:, "letters": "letters"]` select the first two columns (letters and digits).
- Row Selection:** `dfr["r1": "r2"]` and `dfr.loc["r1": "r2"]` select rows r1 and r2.
- Default Indexing:** `dfr[0:2]` and `dfr.iloc[0:2]` select the first two rows.
- Filtering:** `dfr[dfr["digits"]>2]` and `dfr>2` filter rows where the 'digits' value is greater than 2.

Output for `dfr[dfr>2]`:

	letters	digits
r1	a	NaN
r2	b	NaN
r3	c	3.0

Output for `dfr>2`:

	letters	digits
r1	True	False
r2	True	False
r3	True	True

11



3- Indexing and reindexing

- Indexing and filtering in DataFrame

```
46 # Selecting a list of rows
47 #   with labels: only with loc
48 print('dfr.loc[["r1","r3"]]==\n',dfr.loc[["r1","r3"]])
49 #   with default indexes: only with iloc
50 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
```

```
# 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
```

```
# 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
```

```
# 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
```



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

A new value created
with the given fill_value

Before

	a	b	c
G	1	2	3
H	Home	Work	Travel



4- Some Operations

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

```
1 # creating a new series
2 newS= S(np.random.randn(3), index=list("abc"))
3 print(newS)
4 # Dropping the first and last values
5 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

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

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

Creating a DataFrame
specifying a 2 dimensional
ndarray as argument



4- Some Operations

Some other operations

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

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

	c1	c2	c3
r1	0	1	2
r2	3	4	5

```
1 df2 = DF(np.ones((3,3)),index=["r1","r2","r3"],columns=["c1","c2","c3"] )
2 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

df1 doesn't
have r3 row

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

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

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

	c1	c2	c3
r1	1.0	2.0	3.0
r2	4.0	5.0	6.0
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
```

```
1 # applying 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
```

```
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

Missing value in ser2

```
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



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.

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

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

```
1 def f1(x):
2     return S(np.sum(x),index=["sum"])
3
4 df1.apply(f1,axis=0)
5
```

	c1	c2	c3
sum	3	5	7

Apply to each column

Apply to each element

```
1 def f2(x):
2     return float(x)
3 df1.applymap(f2)
4
```

	c1	c2	c3
r1	0.0	1.0	2.0
r2	3.0	4.0	5.0

```
1 def f3(x):
2     return np.where(x>3,"Yes","No")
3
4 df1.loc["r2"].map(f3)
```

```
c1    No
c2    Yes
c3    Yes
```

Name: r2, dtype: object

Defined for Series



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**

```
1 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

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

	c1	c2	c3
r2	3	4	5
r1	0	1000	2

```
1 ser4=S([6,5,1,6,9,0,-3])
2 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 $(5 + 6)/2 == 5.5$

```
1 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

Descriptive operations

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

```
1 df1
```

	c1	c2	c3
r1	0	1000	2
r2	3	4	5

```
1 ser2
```

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

The unique different Value is 1

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

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

```
1 # some descriptive values
2 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

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

```
1 5
dtype: int64
```

There are 5 "1"



4- Some Operations

Descriptive operations

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

	c1	c2	c3
r1	False	False	True
r2	True	False	False

c and d are
In [2,3]

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

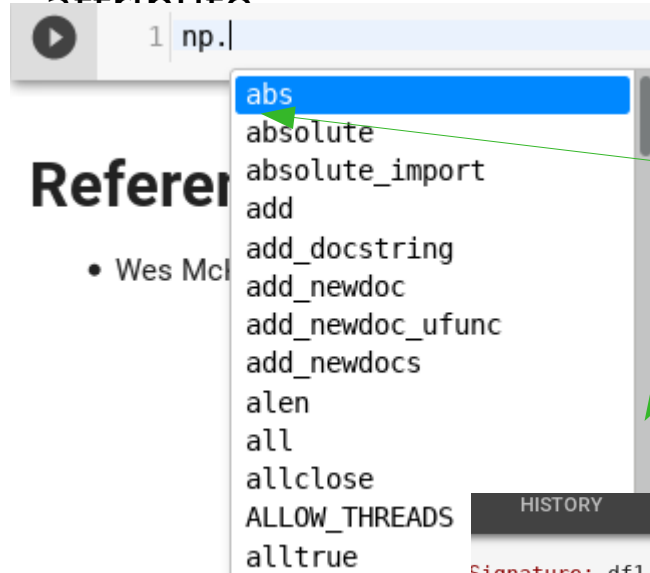
```
1 # check if the Series values are in the argument values
2 ser1.isin([2,3])
```

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



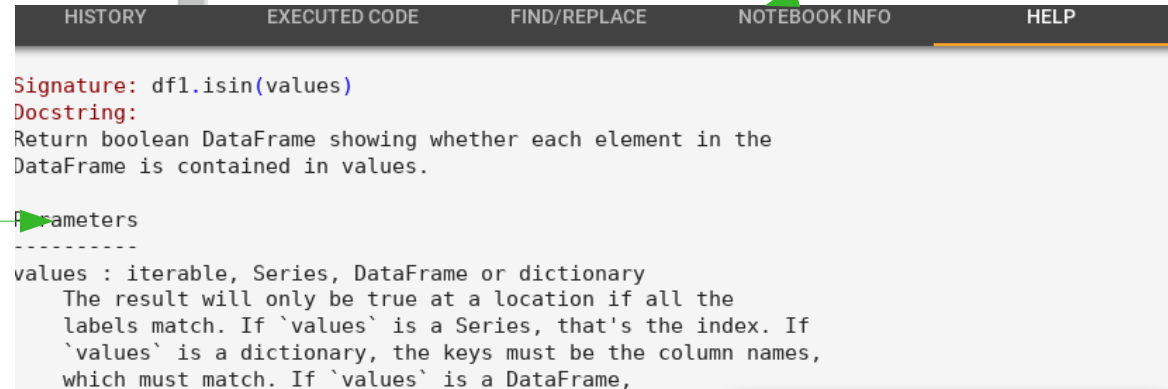
5- Google Colab Help

- 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





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