

# **Pandas\_2**

**Data analysis with Pandas**

Kunal Khurana

2024-02-16

# Table of contents

<b>Data structures</b>	<b>3</b>
<b>Functionality</b>	<b>4</b>
Series . . . . .	4
DataFrame . . . . .	8
index objects . . . . .	12
Reindexing . . . . .	14
Dropping entries from Axis . . . . .	16
Indexing, Selecting, and Filtering . . . . .	18
selection of DataFrame with loc and iloc . . . . .	22
integer indexing pitfalls . . . . .	24
Pitfalls with chained indexing . . . . .	25
Arithmetic and Data Alignment . . . . .	26
Arithmetic methods with fill values . . . . .	28
Operations between DataFrame and Series . . . . .	31
Function application and ‘mapping’ . . . . .	33
Sorting and Ranking . . . . .	37
Axis indices with duplicate labels . . . . .	42
Descriptive statistics . . . . .	43
Unique Values, Value counts, and Membership . . . . .	44
<ul style="list-style-type: none"><li>• helps in numerical computing (NumPy, SciPy)</li><li>• helps with analytical libraries (scikit-learn, and data visualizatioon,</li><li>• processes data without for loops</li></ul>	

# Data structures

- Series
- Data Frames
- index objects

# Functionality

- Reindexing
- Dropping entries from axis
- indexing, selection, and filtering
- DataFrame selection with loc and iloc
- integer indexing pitfalls
- pitfalls with chained indexing
- arithmetic and data alignment
- arithmetic methods with fill values
- Operations between DataFrame and Series
- Function application and mapping
- Sorting and Ranking
- Axis indexed with duplicate labels # Summarizing and Descriptive statistics
- correlation and variance
- unique values, counts, and memberships

## Series

```
import pandas as pd

import numpy as np

from pandas import Series, DataFrame
```

```
obj = pd.Series([4, 2, 312, -3])
```

```
obj
```

```
0      4
1      2
2    312
3     -3
dtype: int64
```

```
obj2 = pd.Series([4,2, 312, -3], index = ['a', 'b', 'c', 'd'])
```

```
obj2
```

```
a      4
b      2
c    312
d     -3
dtype: int64
```

```
obj2.index
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
obj2[obj2 > 0]
```

```
a      4
b      2
c    312
dtype: int64
```

```
np.exp(obj2)
```

```
a      5.459815e+01
b      7.389056e+00
c    3.161392e+135
d      4.978707e-02
dtype: float64
```

```
'b' in obj2
```

```
True
```

```
'e' in obj2
```

```
False
```

```
sdata = {'ohio': 232, 'Texas': 332, 'Oregon': 34343}
```

```
obj3 = pd.Series(sdata)
```

```
obj3
```

```
ohio      232
Texas     332
Oregon    34343
dtype: int64
```

```
obj3.to_dict()
```

```
{'ohio': 232, 'Texas': 332, 'Oregon': 34343}
```

```
states = ['California', 'ohio', 'orleon']
```

```
obj4 = pd.Series(sdata, index = states)
```

```
obj4
```

```
California    NaN
ohio          232.0
orleon        NaN
dtype: float64
```

```
pd.isna(obj4) # is null
```

```
California    True
ohio          False
orleon        True
dtype: bool
```

```
pd.notna(obj4) #not null
```

```
California    False
ohio          True
orleon        False
dtype: bool
```

```
obj3 + obj4
```

```
California    NaN
Oregon        NaN
Texas         NaN
ohio          464.0
orleon        NaN
dtype: float64
```

```
obj4.name = 'population'

obj4.index.name = 'state'

obj4
```

```
state
California    NaN
ohio          232.0
orleon        NaN
Name: population, dtype: float64
```

```
obj
```

```
0      4
1      2
2     312
3     -3
dtype: int64
```

```
# altering the index in place

obj.index = ['Kunal', 'Rahul', 'Raghav', 'Ryan']
```

```
obj
```

```
Kunal      4
Rahul      2
Raghav    312
Ryan      -3
dtype: int64
```

## DataFrame

```
data = {'state': ['ohio', 'ohio', 'nevada'],
        'year': [2000, 2001, 2002],
        'pop': [1.2, 1.3, 1.4]}
```

```
frame = pd.DataFrame(data)
```

```
frame
```

	state	year	pop
0	ohio	2000	1.2
1	ohio	2001	1.3
2	nevada	2002	1.4

```
frame.head()
```

	state	year	pop
0	ohio	2000	1.2
1	ohio	2001	1.3
2	nevada	2002	1.4

```
frame.tail()
```



	state	year	pop
0	ohio	2000	1.2
1	ohio	2001	1.3
2	nevada	2002	1.4

```
# passing another column in the dataframe
```

```
frame2 = pd.DataFrame(data, columns = ['state', 'year', 'pop', 'debt'])
```

```
frame2
```

	state	year	pop	debt
0	ohio	2000	1.2	NaN
1	ohio	2001	1.3	NaN
2	nevada	2002	1.4	NaN

```
# changing the order of columns
```

```
frame2 = pd.DataFrame(data, columns = [ 'year', 'pop', 'debt', 'state'])
```

```
frame2
```

	state	year	pop	debt
0	ohio	2000	1.2	NaN
1	ohio	2001	1.3	NaN
2	nevada	2002	1.4	NaN

```
frame2.year
```

```
0    2000
1    2001
2    2002
Name: year, dtype: int64
```

```
frame2.loc[1]
```

```
year      2001
pop        1.3
debt      NaN
state     ohio
Name: 1, dtype: object
```

```
frame2.iloc[2]
```

```
year      2002
pop        1.4
debt      NaN
state     nevada
Name: 2, dtype: object
```

```
frame2.pop
```

```
<bound method DataFrame.pop of      year  pop  debt  state
0  2000  1.2  NaN   ohio
1  2001  1.3  NaN   ohio
2  2002  1.4  NaN  nevada>
```

```
frame2.year
```

```
0    2000
1    2001
2    2002
Name: year, dtype: int64
```

```
# assigning values
```

```
frame2['debt'] = 14.5
```

```
frame2
```

	year	pop	debt	state
0	2000	1.2	14.5	ohio
1	2001	1.3	14.5	ohio
2	2002	1.4	14.5	nevada

```
# assiging a new column (results in new column if it does not exist before)
```

```
frame2['eastern'] = frame2['state'] == 'ohio'
```

```
frame2
```

	year	pop	debt	state	eastern
0	2000	1.2	14.5	ohio	True
1	2001	1.3	14.5	ohio	True
2	2002	1.4	14.5	nevada	False

```
# transposing
```

```
frame2.T
```

	0	1	2
year	2000	2001	2002
pop	1.2	1.3	1.4
debt	14.5	14.5	14.5
state	ohio	ohio	nevada
eastern	True	True	False

```
pd.DataFrame(data)
```

	state	year	pop
0	ohio	2000	1.2
1	ohio	2001	1.3
2	nevada	2002	1.4

```
frame2.index.name = 'year'
```

```
frame2.columns.name = 'state' # starts with state column
```

```
frame2
```

	state	year	pop	debt	state	eastern
	year					
0		2000	1.2	14.5	ohio	True
1		2001	1.3	14.5	ohio	True
2		2002	1.4	14.5	nevada	False

```
frame2.to_numpy()
```

```
array([[2000, 1.2, 14.5, 'ohio', True],  
       [2001, 1.3, 14.5, 'ohio', True],  
       [2002, 1.4, 14.5, 'nevada', False]], dtype=object)
```

## index objects

```
obj4 = pd.Series(np.arange(3), index = ['a', 'b', 'c'])
```

```
index = obj4.index
```

```
index
```

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

```
index [1:]
```

```
Index(['b', 'c'], dtype='object')
```

```
# index objects are immutable
```

```
index[1]= 'd' #type error
```

```
labels = pd.Index(np.arange(3))
```

```
labels
```

```
Index([0, 1, 2], dtype='int32')
```

```
obj2 = pd.Series([1.5, -2.5, 0], index = labels)
```

```
obj2
```

```
0    1.5
1   -2.5
2    0.0
dtype: float64
```

```
obj2.index is labels
```

```
True
```

```
frame2
```

state	year	pop	debt	state	eastern
year					
0	2000	1.2	14.5	ohio	True
1	2001	1.3	14.5	ohio	True
2	2002	1.4	14.5	nevada	False

```
frame2.columns
```

```
Index(['year', 'pop', 'debt', 'state', 'eastern'], dtype='object', name='state')
```

```
2003 in frame2.index
```

```
False
```

```
# unlike python, a pandas index can contain duplicate labels
```

```
pd.Index(['foo', 'boo', 'bar', 'baa', 'etc', 'foo'])
```

```
Index(['foo', 'boo', 'bar', 'baa', 'etc', 'foo'], dtype='object')
```

## Reindexing

```
obj = pd.Series([4.5, 48, -3, 2, 3.9], index= ['a', 'b', 'c', 'd', 'e'])
```

```
obj
```

```
a      4.5
b     48.0
c     -3.0
d      2.0
e      3.9
dtype: float64
```

```
# reindexing
obj2 = obj.reindex(['b', 'a', 'c', 'd', 'e'])
```

```
obj2
```

```
b      48.0
a       4.5
c     -3.0
d       2.0
e       3.9
dtype: float64
```

```
# time series data fill
obj3 = pd.Series(['blue', 'purple', 'yellow'], index = [0, 2, 4])
```

```
obj3
```

```
0      blue
2    purple
4    yellow
dtype: object
```

```
# forward filling the values using ffill
obj3.reindex(np.arange(6), method='ffill')
```

```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

```
# backward fill
obj3.reindex(np.arange(6), method = 'bfill')
```

```
0      blue
1    purple
2    purple
3    yellow
4    yellow
5         NaN
dtype: object
```

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                      index = ['a', 'b', 'c'],
                      columns= ['ohio', 'texas', 'burmingham'])
```

```
frame
```

	ohio	texas	burmingham
a	0	1	2
b	3	4	5
c	6	7	8

```
frame2 = frame.reindex(index=['a', 'b', 'c', 'd'])
```

```
frame2
```

	ohio	texas	burmingham
a	0.0	1.0	2.0
b	3.0	4.0	5.0
c	6.0	7.0	8.0
d	NaN	NaN	NaN

```
# reindexing columns with column keyword
```

```
states = ['london', 'texas', 'surrey']
```

```
frame.reindex(columns = states)
```

	london	texas	surrey
a	NaN	NaN	NaN
b	NaN	NaN	NaN
c	NaN	NaN	NaN

## Dropping entries from Axis

```
obj = pd.Series(np.arange(5.), index = ['a', 'b', 'c', 'd', 'e'])
```

```
obj
```

```
a    0.0
b    1.0
```



```
c    2.0
d    3.0
e    4.0
dtype: float64
```

```
new_obj = obj.drop('c')
new_obj
```

```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

```
obj.drop(['d', 'e'])
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

```
# in DataFrame
data = pd.DataFrame(np.arange(16).reshape((4,4)),
                    index=['québec', 'montréal', 'toronto', 'sainte-anne'],
                    columns = ['one', 'two', 'three', 'four'])

data
```

	one	two	three	four
québec	0	1	2	3
montréal	4	5	6	7
toronto	8	9	10	11
sainte-anne	12	13	14	15

```
# using drop method
data.drop(index=['toronto', 'sainte-anne'])
```

	one	two	three	four
québec	0	1	2	3
montréal	4	5	6	7

```
# dropping using axis method (axis = 1 = columns)
```

```
data.drop('two', axis=1)
```

	one	three	four
québec	0	2	3
montréal	4	6	7
toronto	8	10	11
sainte-anne	12	14	15

```
data.drop(['three', 'four'], axis='columns')
```

	one	two
québec	0	1
montréal	4	5
toronto	8	9
sainte-anne	12	13

## Indexing, Selecting, and Filtering

```
obj = pd.Series(np.arange(4.), index= ['a', 'b', 'c', 'd'])
```

```
obj
```

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
obj['b']
```

```
1.0
```

```
obj[1]
```

```
1.0
```

```
obj[2:4]
```

```
c    2.0  
d    3.0  
dtype: float64
```

```
obj[obj<2]
```

```
a    0.0  
b    1.0  
dtype: float64
```

```
obj.loc[['b', 'c']]
```

```
b    1.0  
c    2.0  
dtype: float64
```

```
obj1 = pd.Series([1,2,3], index = [2,0,1])
```

```
obj2 = pd.Series([1,2,3], index = ['a', 'b', 'c'])
```

```
obj1
```

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

```
obj2
```

```
a    1
b    2
c    3
dtype: int64
```

```
# loc fails as index doesnot contain integers
obj2.loc[[0, 1]]
```

```
# fix this
```

```
obj2.loc['b':'c']
```

```
b    2
c    3
dtype: int64
```

```
# so, prefer using iloc with integers
```

```
obj1.iloc[[0,1,2]]
```

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

```
obj2.iloc[[0,1,2]]
```

```
a    1
b    2
c    3
dtype: int64
```

```
# assigning values

obj2.loc['b':'c'] = 5

obj2
```

```
a    1
b    5
c    5
dtype: int64
```

```
data
```

	one	two	three	four
québec	0	1	2	3
montréal	4	5	6	7
toronto	8	9	10	11
sainte-anne	12	13	14	15

```
data[:2]
```

	one	two	three	four
québec	0	1	2	3
montréal	4	5	6	7

```
# booleans
data < 5
```

	one	two	three	four
québec	True	True	True	True
montréal	True	False	False	False
toronto	False	False	False	False
sainte-anne	False	False	False	False

```
# assigning values
data[data < 5] = 0

data
```

	one	two	three	four
québec	0	0	0	0
montréal	0	5	6	7
toronto	8	9	10	11
sainte-anne	12	13	14	15

## selection of DataFrame with loc and iloc

```
data
```

	one	two	three	four
québec	0	0	0	0
montréal	0	5	6	7
toronto	8	9	10	11
sainte-anne	12	13	14	15

```
data.loc['montréal']
```

```
one      0
two      5
three    6
four     7
Name: montréal, dtype: int32
```

```
data.loc[['montréal', 'québec']]
```

	one	two	three	four
montréal	0	5	6	7
québec	0	0	0	0

	one	two	three	four
québec	0	0	0	0

```
data.loc['montréal', ['two', 'three']]
```

```
two      5
three    6
Name: montréal, dtype: int32
```

```
# similar operations with iloc
data.iloc[2]
```

```
one      8
two      9
three    10
four     11
Name: toronto, dtype: int32
```

```
data.iloc[[2,1]] #third row and second row
```

	one	two	three	four
toronto	8	9	10	11
montréal	0	5	6	7

```
data.iloc[2,[3,0,1]] #third row (three elements in order)
```

```
four     11
one      8
two      9
Name: toronto, dtype: int32
```

```
data.iloc[[1,2],[3,0,1]]
```

	four	one	two
montréal	7	0	5
toronto	11	8	9

## integer indexing pitfalls

```
series = pd.Series(np.arange(3.))
```

```
series
```

```
0    0.0
1    1.0
2    2.0
dtype: float64
```

```
# fails here but works fine with iloc and loc
series[-1]
```

```
# value error; key error: -1
```

```
series.iloc[-1]
```

```
2.0
```

```
# non-integer doesnot do this ambiguity
```

```
series2 = pd.Series(np.arange(3.0), index = ['a', 'b', 'c'])
```

```
series2[-1]
```

```
2.0
```



## Pitfalls with chained indexing

```
data.loc[:, 'one'] = 1
```

data

	one	two	three	four
québec	1	0	0	0
montréal	1	5	6	7
toronto	1	9	10	11
sainte-anne	1	13	14	15

```
data.iloc[2] = 5
```

data

	one	two	three	four
québec	1	0	0	0
montréal	1	5	6	7
toronto	5	5	5	5
sainte-anne	1	13	14	15

```
data.loc[data['four'] > 5] = 3
```

data

	one	two	three	four
québec	1	0	0	0
montréal	3	3	3	3
toronto	5	5	5	5
sainte-anne	3	3	3	3

```
# the data gets modified, but it is not the way that was asked for
```

```
# fixing it with loc operation
```

```
data.loc[data.three == 10, "three"] = 9
```

```
data
```

	one	two	three	four
québec	1	0	0	0
montréal	3	3	3	3
toronto	5	5	5	5
sainte-anne	3	3	3	3

## Arithmetic and Data Alignment

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index = ['a', 'c', 'd', 'e'])
```

```
s2 = pd.Series([1.2, -3, -.3, -.33, -43.2], index = ['e', 'j', 'o', 't', 'y'])
```

```
s1
```

```
a      7.3
c     -2.5
d      3.4
e      1.5
dtype: float64
```

```
s2
```

```
e      1.20
j     -3.00
o     -0.30
t     -0.33
y    -43.20
dtype: float64
```

```
# adding these- missing values donot overlap
```

```
s1+s2
```

```

a    NaN
c    NaN
d    NaN
e    2.7
j    NaN
o    NaN
t    NaN
y    NaN
dtype: float64

```

```

# in case of DataFrame, alignment is performed on both rows and columns

df1 = pd.DataFrame(np.arange(9.).reshape((3,3)),
                    columns = list('abc'),
                    index = ['ferozpur', 'faridkot', 'montréal'])

df2 = pd.DataFrame(np.arange(12.).reshape((4,3)),
                    columns = list('abc'),
                    index = ['faridkot', 'toronto', 'québec', 'montréal'])

```

df1

	a	b	c
ferozpur	0.0	1.0	2.0
faridkot	3.0	4.0	5.0
montréal	6.0	7.0	8.0

df2

	a	b	c
faridkot	0.0	1.0	2.0
toronto	3.0	4.0	5.0
québec	6.0	7.0	8.0
montréal	9.0	10.0	11.0

```
df1 + df2 #because the columns were same, it added those numbers
```

	a	b	c
faridkot	3.0	5.0	7.0
ferozpur	NaN	NaN	NaN
montréal	15.0	17.0	19.0
québec	NaN	NaN	NaN
toronto	NaN	NaN	NaN

```
# changing columns names will give all NaN (null values)
df3 = pd.DataFrame(np.arange(12.).reshape((4,3)),
                    columns = list('xyz'),
                    index = ['faridkot', 'toronto', 'québec', 'montréal'])
```

```
df1 + df3
```

	a	b	c	x	y	z
faridkot	NaN	NaN	NaN	NaN	NaN	NaN
ferozpur	NaN	NaN	NaN	NaN	NaN	NaN
montréal	NaN	NaN	NaN	NaN	NaN	NaN
québec	NaN	NaN	NaN	NaN	NaN	NaN
toronto	NaN	NaN	NaN	NaN	NaN	NaN

## Arithmetic methods with fill values

```
df2
```

	a	b	c
faridkot	0.0	1.0	2.0
toronto	3.0	4.0	5.0
québec	6.0	7.0	8.0
montréal	9.0	10.0	11.0

```
df2.loc['faridkot', 'y'] = np.nan
```

```
df2
```

	a	b	c	y
faridkot	0.0	1.0	2.0	NaN
toronto	3.0	4.0	5.0	NaN
québec	6.0	7.0	8.0	NaN
montréal	9.0	10.0	11.0	NaN

```
help(pd.DataFrame._drop_axis)
```

Help on function `_drop_axis` in module `pandas.core.generic`:

```
_drop_axis(self: 'NDFrameT', labels, axis, level=None, errors: 'IgnoreRaise' = 'raise', only_
Drop labels from specified axis. Used in the ``drop`` method
internally.
```

```
Parameters
```

```
-----
```

```
labels : single label or list-like
```

```
axis : int or axis name
```

```
level : int or level name, default None
```

```
For MultiIndex
```

```
errors : {'ignore', 'raise'}, default 'raise'
```

```
If 'ignore', suppress error and existing labels are dropped.
```

```
only_slice : bool, default False
```

```
Whether indexing along columns should be view-only.
```

```
help(pd.DataFrame.drop)
```

```
print(dir(DataFrame))
```

```
['T', '_AXIS_LEN', '_AXIS_ORDERS', '_AXIS_TO_AXIS_NUMBER', '_HANDLED_TYPES', '__abs__', '__a
```

```
help(pd.DataFrame.describe)
```

```
help(pd.DataFrame._drop_axis)
```

```
df4 = df2
```

df4

	a	b	c	y
faridkot	0.0	1.0	2.0	NaN
toronto	3.0	4.0	5.0	NaN
québec	6.0	7.0	8.0	NaN
montréal	9.0	10.0	11.0	NaN

df1 + df4

	a	b	c	y
faridkot	3.0	5.0	7.0	NaN
ferozpur	NaN	NaN	NaN	NaN
montréal	15.0	17.0	19.0	NaN
québec	NaN	NaN	NaN	NaN
toronto	NaN	NaN	NaN	NaN

df4.fill\_value = 0

df4

	a	b	c	y
faridkot	0.0	1.0	2.0	NaN
toronto	3.0	4.0	5.0	NaN
québec	6.0	7.0	8.0	NaN
montréal	9.0	10.0	11.0	NaN

1/df4

	a	b	c	y
faridkot	inf	1.000000	0.500000	NaN
toronto	0.333333	0.250000	0.200000	NaN
québec	0.166667	0.142857	0.125000	NaN
montréal	0.111111	0.100000	0.090909	NaN

```
df4.rdiv(1)
```

	a	b	c	y
faridkot	inf	1.000000	0.500000	NaN
toronto	0.333333	0.250000	0.200000	NaN
québec	0.166667	0.142857	0.125000	NaN
montréal	0.111111	0.100000	0.090909	NaN

```
df4.reindex(columns = df4.columns, fill_value=0) # not working
```

	a	b	c	y
faridkot	0.0	1.0	2.0	NaN
toronto	3.0	4.0	5.0	NaN
québec	6.0	7.0	8.0	NaN
montréal	9.0	10.0	11.0	NaN

## Operations between DataFrame and Series

```
arr = np.arange(12.).reshape((3,4))
```

```
arr
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
arr[0]
```

```
array([0., 1., 2., 3.])
```

```
# broadcasting
```

```
arr - arr[0]    #subtracts from all rows
```

```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

```
frame
```

	ohio	texas	burmingham
a	0	1	2
b	3	4	5
c	6	7	8

```
help(pd.Series)
```

```
series
```

```
series1 = pd.Series(data = np.arange(3), index = ['a', 'b', 'c'])
```

```
series1
```

```
a    0
b    1
c    2
dtype: int32
```

```
frame-series1
```

	a	b	burmingham	c	ohio	texas
a	NaN	NaN	NaN	NaN	NaN	NaN
b	NaN	NaN	NaN	NaN	NaN	NaN
c	NaN	NaN	NaN	NaN	NaN	NaN

```
frame + series2
```

	a	b	burmingham	c	ohio	texas
a	NaN	NaN	NaN	NaN	NaN	NaN



	a	b	burmingham	c	ohio	texas
b	NaN	NaN	NaN	NaN	NaN	NaN
c	NaN	NaN	NaN	NaN	NaN	NaN

## Function application and ‘mapping’

```
frame2 = pd.DataFrame(np.random.standard_normal((4,3)),
                      columns = list('bde'),
                      index = ['utah', 'faridkot', 'shahkot', 'malsahian'])

frame2
```

	b	d	e
utah	-1.549165	0.443756	1.013167
faridkot	1.130587	-1.289388	-1.210530
shahkot	1.195553	0.274397	0.510043
malsahian	0.713024	-1.223282	1.857681

```
np.abs(frame2) #converts non-negative values to positive
```

	b	d	e
utah	1.549165	0.443756	1.013167
faridkot	1.130587	1.289388	1.210530
shahkot	1.195553	0.274397	0.510043
malsahian	0.713024	1.223282	1.857681

```
help(np.abs)
```

Help on ufunc:

```
absolute = <ufunc 'absolute'>
  absolute(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subol
```

Calculate the absolute value element-wise.

``np.abs`` is a shorthand for this function.

## Parameters

-----

`x` : array\_like

Input array.

`out` : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

`where` : array\_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value.

Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

**\*\*kwargs**

For other keyword-only arguments, see the `:ref:`ufunc docs <ufuncs.kwargs>``.

## Returns

-----

`absolute` : ndarray

An ndarray containing the absolute value of each element in `x`. For complex input, ```a + ib```, the absolute value is `:math:\sqrt{a^2 + b^2}`.

This is a scalar if `x` is a scalar.

## Examples

-----

```
>>> x = np.array([-1.2, 1.2])
```

```
>>> np.absolute(x)
```

```
array([ 1.2,  1.2])
```

```
>>> np.absolute(1.2 + 1j)
```

```
1.5620499351813308
```

Plot the function over ```[-10, 10]```:

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(start=-10, stop=10, num=101)
```

```
>>> plt.plot(x, np.absolute(x))
```

```
>>> plt.show()
```

Plot the function over the complex plane:

```
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10], cmap='gray')
>>> plt.show()
```

The `abs` function can be used as a shorthand for `np.absolute` on `ndarrays`.

```
>>> x = np.array([-1.2, 1.2])
>>> abs(x)
array([1.2, 1.2])
```

```
#DataFrame's apply method
```

```
def f1(x):
    return x.max() - x.min()
```

```
frame2.apply(f1)
```

```
b    2.744718
d    1.733145
e    3.068212
dtype: float64
```

```
# applying across columns
```

```
frame2.apply(f1, axis = 'columns')
```

```
utah      2.562332
faridkot   2.419976
shahkot    0.921156
malsahian  3.080963
dtype: float64
```

```
# modifying the function to return Series with multiple values
```

```
def f2(x):
    return pd.Series([x.min(), x.max()], index= ['min', 'max'])

frame.apply(f2)
```

	ohio	texas	burmingham
min	0	1	2
max	6	7	8

```
frame
```

	ohio	texas	burmingham
a	0	1	2
b	3	4	5
c	6	7	8

```
frame2.apply(f2)
```

	b	d	e
min	-1.549165	-1.289388	-1.210530
max	1.195553	0.443756	1.857681

```
# apply map function
```

```
def my_format(x):
    return f"{x:.2f}"
```

```
frame2.applymap(my_format)
```

	b	d	e
utah	-1.55	0.44	1.01
faridkot	1.13	-1.29	-1.21
shahkot	1.20	0.27	0.51
malsahian	0.71	-1.22	1.86

```
# applying map function in Series
```

```
frame2['e'].map(my_format)
```

```
utah      1.01
faridkot  -1.21
shahkot    0.51
malsahian  1.86
Name: e, dtype: object
```

## Sorting and Ranking

```
obj2 = pd.Series(np.arange(4), index = ['d', 'a', 'b', 'c'])
```

```
obj2
```

```
d    0
a    1
b    2
c    3
dtype: int32
```

```
obj2.sort_index()
```

```
a    1
b    2
c    3
d    0
dtype: int32
```

```
# sorting in DataFrame can be done with either axis
```

```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                      index = ['three', 'one'],
                      columns = ['d', 'a', 'b', 'c'])
```

```
frame
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

```
frame.sort_index()
```

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

```
frame.sort_index(axis= 'columns')
```

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

```
# data can be stored in descending order aswell
```

```
frame.sort_index(axis = 'columns', ascending = False)
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

```
# sorting a series by its values
```

```
obj = pd.Series([4, 7, -3, -2])
```

```
obj.sort_values()
```

```
2    -3
3    -2
0     4
1     7
dtype: int64
```

```
# missing values get sorted to the end by default

obj = pd.Series([4, 3, 4, np.nan, 33, np.nan, -3, 3])

obj.sort_values()
```

```
6    -3.0
1     3.0
7     3.0
0     4.0
2     4.0
4    33.0
3     NaN
5     NaN
dtype: float64
```

```
# using na_position to bring missing values to the front
obj.sort_values(na_position= 'first')
```

```
3     NaN
5     NaN
6    -3.0
1     3.0
7     3.0
0     4.0
2     4.0
4    33.0
dtype: float64
```

```
# while sorting a DataFrame

frame = pd.DataFrame({'b': [1, 2, 3, 4, 5], 'a': [3, 43, 33, 1, 5]})
```

```
frame
```

	b	a
0	1	3
1	2	43
2	3	33
3	4	1
4	5	5

```
frame.sort_values('a')
```

	b	a
3	4	1
0	1	3
4	5	5
2	3	33
1	2	43

```
# ranking
obj = pd.Series([4, 5, -5, 7, 8, 0, 4])

obj.rank()
```

```
0    3.5
1    5.0
2    1.0
3    6.0
4    7.0
5    2.0
6    3.5
dtype: float64
```

```
# ranking in order the data is observed
```

```
obj.rank(method='first')
```



```

0    3.0
1    5.0
2    1.0
3    6.0
4    7.0
5    2.0
6    4.0
dtype: float64

```

```
obj.rank(ascending = False)
```

```

0    4.5
1    3.0
2    7.0
3    2.0
4    1.0
5    6.0
6    4.5
dtype: float64

```

```
# DataFrame for rank computation
```

```
frame
```

	b	a
0	1	3
1	2	43
2	3	33
3	4	1
4	5	5

```
frame.rank(axis = 'columns')
```

	b	a
0	1.0	2.0
1	1.0	2.0
2	1.0	2.0

	b	a
3	2.0	1.0
4	1.5	1.5

## Axis indices with duplicate labels

```
obj = pd.Series(np.arange(5), index=['a', 'a', 'b','b', 'c' ])
```

```
obj
```

```
a    0
a    1
b    2
b    3
c    4
dtype: int32
```

```
obj.index.is_unique
```

```
False
```

```
obj['a']
```

```
a    0
a    1
dtype: int32
```

```
obj['c']
```

```
4
```

```
# DataFrame
df = pd.DataFrame(np.random.standard_normal((5, 3)),
                  index = ['a', 'a', 'b', 'c', 'b'])
```

```
df
```

	0	1	2
a	-1.817751	0.915854	-0.389590
a	0.603020	0.573012	1.070691
b	-0.903033	1.109707	0.874381
c	2.529357	-1.169854	0.676702
b	-0.368763	0.723758	0.375079

```
df.loc['b']
```

	0	1	2
b	-0.903033	1.109707	0.874381
b	-0.368763	0.723758	0.375079

```
df.loc['c']
```

```
0    2.529357
1   -1.169854
2    0.676702
Name: c, dtype: float64
```

## Descriptive statistics

```
df.sum()
```

```
0    0.042830
1    2.152477
2    2.607263
dtype: float64
```

```
df.sum(axis = 'columns')
```

```

a    -1.291487
a     2.246723
b     1.081055
c     2.036205
b     0.730074
dtype: float64

```

```
df.describe()
```

	0	1	2
count	5.000000	5.000000	5.000000
mean	0.008566	0.430495	0.521453
std	1.659562	0.917106	0.570471
min	-1.817751	-1.169854	-0.389590
25%	-0.903033	0.573012	0.375079
50%	-0.368763	0.723758	0.676702
75%	0.603020	0.915854	0.874381
max	2.529357	1.109707	1.070691

```
df.cumsum()
```

	0	1	2
a	-1.817751	0.915854	-0.389590
a	-1.214731	1.488866	0.681101
b	-2.117763	2.598573	1.555481
c	0.411594	1.428719	2.232183
b	0.042830	2.152477	2.607263

## Unique Values, Value counts, and Membership

```

obj = pd.Series(['c', 'd', 'a', 'b', 'n', 'm', 'g', "k", 'b', 'c', 'k'])

uniques = obj.unique()

uniques

```

```
array(['c', 'd', 'a', 'b', 'n', 'm', 'g', 'k'], dtype=object)
```

```
obj.value_counts()
```

```
c    2
b    2
k    2
d    1
a    1
n    1
m    1
g    1
Name: count, dtype: int64
```

```
pd.value_counts(obj.to_numpy(), sort = False)
```

```
c    2
d    1
a    1
b    2
n    1
m    1
g    1
k    2
Name: count, dtype: int64
```

```
# 'isin' is used for vectorized set membership
```

```
obj
```

```
0    c
1    d
2    a
3    b
4    n
5    m
6    g
7    k
```

```
8      b
9      c
10     k
dtype: object
```

```
mask = obj.isin(['b', 'c'])
```

```
mask
```

```
0      True
1     False
2     False
3      True
4     False
5     False
6     False
7     False
8      True
9      True
10     False
dtype: bool
```

```
obj[mask]
```

```
0      c
3      b
8      b
9      c
dtype: object
```

```
to_match = pd.Series(['c', 'a', 'b', 'b', 'c'])
```

```
unique_vals = pd.Series(['c', 'b', 'a'])
```

```
indices = pd.Index(unique_vals).get_indexer(to_match)
```

```
indices
```

```
array([0, 2, 1, 1, 0], dtype=int64)
```

```
data = pd.DataFrame({'ça va': [1, 2, 3, 4, 5],
                     'oui, ça va' : [43,3, 2, 4, 2],
                     'et toi': [3, 2, 44, 1, 5]})
```

```
data
```

	ça va	oui, ça va	et toi
0	1	43	3
1	2	3	2
2	3	2	44
3	4	4	1
4	5	2	5

```
# computing value counts
data['ça va'].value_counts().sort_index()
```

```
ça va
1      1
2      1
3      1
4      1
5      1
Name: count, dtype: int64
```

```
data['et toi'].value_counts().sort_index()
```

```
et toi
1      1
2      1
3      1
5      1
44     1
Name: count, dtype: int64
```

```
result = data.apply(pd.value_counts).fillna(0)
```

```
result
```

	ça va	oui, ça va	et toi
1	1.0	0.0	1.0
2	1.0	2.0	1.0
3	1.0	1.0	1.0
4	1.0	1.0	0.0
5	1.0	0.0	1.0
43	0.0	1.0	0.0
44	0.0	0.0	1.0