Sumendar

Python Pandas Tutorial: Series

A Series can be constructed with a plain list, dict, or scalar. Many operations on a Series have concise expression and are useful for powerful data analysis and munging.

Pandas is a powerful toolkit providing data analysis tools and structures for the Python programming language.

Among the most important artifacts provided by pandas is the Series. In this article, we introduce the Series class from a beginner's perspective. That means you do not need to know anything about pandas or data analysis to understand this tutorial.

What Is a Series?

A Series is similar to a list or an array in Python. It represents a series of values (numeric or otherwise) such as a column of data. Think of it as a Python list on steroids. It provides additional functionality, methods, and operators, which make it a more powerful version of a list.

To get started using a Series, you need to import the pandas toolkit into your Python program. Add the following line to the beginning of your program and you are good to go:

```
import pandas as pd
```

Create Series From List

Let's now learn how to create a Series. Since a Series is similar to a list, let's use a list to create a Series.

```
ser = pd.Series([1, 3, 5, 7])

print ser

0    1

1    3

2    5

3    7

dtype: int64
```

Notice that when the Series is printed, two columns of numbers are printed. The first column is called the index. It normally starts from 0 and runs all the way to (*N*-1) where *N* is the size of the list.

The second column is the actual data that we specified.

4. Use a Dict to Initialize Series

It is also possible to use a Python dictionary (called a dict) to initialize a Series. In this case, the Series takes its index from the keys of the dict as shown by this example.

```
prices = {'apple': 4.99,

         'banana': 1.99,

         'orange': 3.99,

         'grapes': 0.99}

ser = pd.Series(prices)

print ser

apple     4.99

banana    1.99

grapes    0.99

orange    3.99

dtype: float64
```

5. Initialize Series from Scalar

You can also use a scalar to initialize a Series. In this case, all elements of the Series are initialized to the same value. When used with a scalar for initialization, an index array can be specified. In this case, the size of the Series is the same as the size of the index array.

In the following example, we use the `range()` function to specify the index (and thus the size of the Series).

```
ser = pd.Series(2, index=range(0, 5))

print ser

0    2

1    2

2    2

3    2

4    2

dtype: int64
```

6. Some Other Ways of Creating a Series

Here are some additional ways of initializing a Series.

A Series of Odd Numbers

We use the `range()` function to create a Series of odd numbers.

```
print pd.Series(range(1, 10, 2))

0    1

1    3

2    5

3    7

4    9

dtype: int64
```

With an Alphabetic Index

```
print pd.Series(range(1, 15, 3), index=[x for x in 'abcde'])

a     1

b     4

c     7

d    10

e    13

dtype: int64
```

A Series With Random Numbers

Use a random range to initialize a Series:

```
print pd.Series(random.sample(xrange(100), 6))

0    61

1    81

2    11

3    78

4    29

5    92

dtype: int64
```

Combining Lists

If you have the data in one list and the index in another list, there are a couple of ways to create a Series from them.

Using a Dict

```
x = dict(zip([x for x in 'abcdefg'], xrange(1, 8)))

print x

y = pd.Series(x)

print y

{'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 7, 'f': 6}

a    1

b    2

c    3

d    4

e    5

f    6

g    7

dtype: int64
```

Specifying an Index

Skip defining a dict and create the Series directly.

```
print pd.Series(xrange(1,8), index=[x for x in 'abcdefg'])

a    1

b    2

c    3

d    4

e    5

f    6

g    7

dtype: int64
```

7. Naming a Series

You can also assign a name to the Series. When used in the context of a DataFrame (to be covered next), this name serves as the column name.

```
a = [1, 3, 5, 7]
```

```
print pd.Series(a, name='joe')

0    1

1    3

2    5

3    7

Name: joe, dtype: int64
```

8. Comparing List with Series

A Series is like a list. The usual array indexing works as expected, as does the array slicing. Notice that the slice operator returns a Series itself.

```
ser = pd.Series(random.sample(xrange(100), 10))

print ser

print

print '4th element: ', ser[4]

print 'Slice: ', ser[3:8]

0    79

1     4

2    71

3    20

4    19

5    24

6    82

7    74

8    17

9    48

dtype: int64

4th element:  19

Slice:  3    20

4    19

5    24

6    82

7    74

dtype: int64
```

Using a Predicate for Extraction

While list elements can be extracted using indexing and slicing, Series elements can also be extracted using a predicate function (a function returning `True` or `False`) as shown in this example.

```
x = random.sample(range(100), 10)

print x

y = pd.Series(x)

print y[y > 30]

[22, 5, 16, 56, 38, 48, 64, 41, 71, 63]

3    56

4    38

5    48

6    64

7    41

8    71

9    63

dtype: int64
```

Indexing Using a List

In addition to extraction using a predicate function, items can also be retrieved from a Series using a list as the index.

```
x = random.sample(xrange(100), 10)

print x

y = pd.Series(x)

print y[[2, 0, 1, 2]]

[29, 1, 27, 54, 2, 90, 25, 96, 45, 34]

2    27

0    29

1     1

2    27

dtype: int64
```

Executing a Function on a Series

Unlike a list, a function accepting a scalar argument can be invoked with a Series. The result will be another Series with the function applied to each element of the Series. This allows more flexible and concise ways in which operations can be combined — just what is needed in a data analysis toolkit!

```
def joe(x): return x + 10

x = random.sample(xrange(100), 10)

print 'Data => ', x, '\n'

y = pd.Series(x)

print 'Applying pow => \n', pow(y, 2), '\n'

print 'Applying joe => \n', joe(y)
```

```
Data =>  [96, 63, 79, 11, 49, 41, 12, 26, 20, 62]

Applying pow =>

0    9216

1    3969

2    6241

3     121

4    2401

5    1681

6     144

7     676

8     400

9    3844

dtype: int64

Applying joe =>

0    106

1     73

2     89

3     21

4     59

5     51

6     22

7     36

8     30
```

```
9     72

dtype: int64
```

9. Comparing a Dict With a Series

A Series also behaves like a Python dictionary.

Indexing With Label

For example, you can extract items using the index label.

```
x = pd.Series(xrange(1,8), index=[x for x in 'abcdefg'])

print x, '\n'

print 'element "d" => ', x['d']

a    1

b    2

c    3

d    4

e    5

f    6

g    7

dtype: int64

element "d" =>  4
```

Checking for Membership

Use the `in` operator to check whether a Series contains a specific label.

```
x = pd.Series(xrange(1,8), index=[x for x in 'abcdefg'])

print x, '\n'

print 'is "j" in x?', 'j' in x

print 'is "d" in x?', 'd' in x

a    1

b    2

c    3

d    4

e    5

f    6
```

```
g    7

dtype: int64

is "j" in x? False

is "d" in x? True
```

Summary

This article was a gentle introduction to pandas Series. A Series can be constructed using a plain list, a dict, or a scalar. Many operations on a Series allow concise expression and are useful for powerful data analysis and munging.

Sumendar

Python Pandas Tutorial: Series Methods

Learn commonly used methods to deal with a Series object, including methods to retrieve general information about a Series, modifying a Series, selection, and sorting.

The Series is one of the most common Pandas data structures. It is similar to a Python list and is used to represent a column of data. After looking into the basics of creating and initializing a pandas Series object, we now delve into some common usage patterns and methods.

Series Information

After a Series is created, it is most important to look into various details of its structure. These include the size of the series, whether there are NaNs in it, etc. Here are some commonly used methods which help clarify the situation.

Size of the Series

There are several methods to determine how big the Series is.

The first is the attribute `shape`, which returns a tuple.

```
a = pd.Series(random.sample(xrange(100), 6))

print a.shape

(6,)
```

We also have the `count()` method which returns the size of the Series as an integer.

```
print a.count()

6
```

However, note that `count()` only reports the number of non-NaN elements, while `shape` reports both.

Another attribute for getting the count of elements is `size`. It reports the count as an integer and includes NaN elements if any.

```
a = pd.Series(random.sample(xrange(100), 6))

print 'count of a =>', a.count(), '\n'
```

```
b = a.append(pd.Series(np.nan, index=list('abcd')), ignore_index=True)

print 'b => ', b, '\n'

print 'count of b =>', b.count(), '\n'

print 'shape of b =>', b.shape, '\n'

print 'size of b =>', b.size

count of a => 6

b =>  0    76.0

1    92.0

2    75.0

3    60.0

4    42.0

5    44.0

6     NaN

7     NaN

8     NaN

9     NaN

dtype: float64

count of b => 6

shape of b => (10,)

size of b => 10
```

Series Details

Get some detailed stats on the Series using `describe()`. This method returns a Series object with the index (or labels) as shown.

```
x = pd.Series(random.sample(xrange(100), 6))

x.describe()

count     6.000000

mean     60.500000

std      30.742479

min      20.000000

25%      44.000000

50%      53.500000

75%      86.250000
```

```
max      98.000000

dtype: float64
```

Head and Tail

Show the first 5 or last 5 rows of the Series using `head()` or `tail()`.

```
x = pd.Series(random.sample(xrange(100), 10))

print x, '\n'

print x.head(), '\n'

print x.tail(), '\n'

0    24

1    39

2    56

3    77

4    81

5    26

6     8

7    87

8    34

9    68

dtype: int64

0    24

1    39

2    56

3    77

4    81

dtype: int64

5    26

6     8

7    87

8    34

9    68

dtype: int64
```

Add Elements to Series

Adding elements to a Series is accomplished by using `append()`. The argument must be a single Series object, or a list (or tuple) of Series objects.

```
x  = pd.Series(random.sample(xrange(100), 6))

print x, '\n'

print 'appended =>\n', x.append([pd.Series(2), pd.Series([3, 4, 5])])

0    62

1    29

2    20

3    69

4    53

5    22

dtype: int64

appended =>

0    62

1    29

2    20

3    69

4    53

5    22

0     2

0     3

1     4

2     5

dtype: int64
```

You might notice the oddball labels after appending. Each Series is appended with a default index starting from 0, regardless of whether this creates duplicate labels. One way to fix this is to specify `ignore_index=True` to ensure re-labeling.

```
print 'appended =>\n', x.append([pd.Series(2), pd.Series([3, 4, 5])],
ignore_index=True)

appended =>

0    62

1    29
```

```
2    20
3    69
4    53
5    22
6     2
7     3
8     4
9     5
dtype: int64
```

What if you don't want to re-label but ensure that `append()` succeeds only if the labels are unique? Keep your precious labels intact and unique by specifying `verify_integrity=True.`

```
print 'appended =>\n', x.append([pd.Series(2), pd.Series([3, 4, 5])],
verify_integrity=True)
```

```
ValueError: Indexes have overlapping values: [0, 1, 2]
```

Delete Elements

You can delete elements from a Series using the following methods.

By Label

Use `drop()` and specify a single label or a list of labels to drop.

```
x = pd.Series(random.sample(xrange(100), 6), index=list('ABCDEF'))
print x, '\n'
print 'drop one =>\n', x.drop('C'), '\n'
print 'drop many =>\n', x.drop(['C', 'D'])
```

```
A    67
B    18
C     1
D    54
E    38
F     3
dtype: int64
drop one =>
A    67
```

```
B    18
D    54
E    38
F     3
dtype: int64
drop many =>
A    67
B    18
E    38
F     3
dtype: int64
```

Duplicate Elements

Get rid of duplicate elements by invoking `drop_duplicates()`.

```
x = pd.Series([1, 2, 2, 4, 5, 7, 3, 4])
print x, '\n'
print 'drop duplicates =>\n', x.drop_duplicates(), '\n'
0    1
1    2
2    2
3    4
4    5
5    7
6    3
7    4
dtype: int64
drop duplicates =>
0    1
1    2
3    4
4    5
5    7
6    3
```

```
dtype: int64
```

By default, the method retains the first repeated value. Get rid of all duplicates (including the first) by specifying `keep=False`.

```
drop all duplicates =>

0    1

4    5

5    7

6    3

dtype: int64
```

NaN Elements

Use the `dropna()` to drop elements without a value (NaN).

```
x = pd.Series([1, 2, 3, 4, np.nan, 5, 6])

print x, '\n'

print 'drop na =>\n', x.dropna()

0    1.0

1    2.0

2    3.0

3    4.0

4    NaN

5    5.0

6    6.0

dtype: float64

drop na =>

0    1.0

1    2.0

2    3.0

3    4.0

5    5.0

6    6.0

dtype: float64
```

Replace NaN Elements

When you want to replace NaN elements in a Series, use `fillna()`.

```
x = pd.Series([1, 2, 3, 4, np.nan, 5, 6])

print x, '\n'

print 'fillna w/0 =>\n', x.fillna(0)

0    1.0

1    2.0

2    3.0

3    4.0

4    NaN

5    5.0

6    6.0

dtype: float64

fillna w/0 =>

0    1.0

1    2.0

2    3.0

3    4.0

4    0.0

5    5.0

6    6.0

dtype: float64
```

Select Elements

Select elements from a Series based on various conditions as follows.

In a Range

Use the `between()` method, which returns a Series of boolean values indicating whether the element lies within the range.

```
a = pd.Series(random.sample(xrange(100), 10))

print a

print a.between(30, 50)

0    85

1    42
```

```
2    63
3    69
4    81
5    45
6    50
7    72
8    66
9    34
dtype: int64
0    False
1     True
2    False
3    False
4    False
5     True
6     True
7    False
8    False
9     True
dtype: bool
```

You can use this the returned boolean Series as a predicate into the original Series.

```
print a[a.between(30, 50)]
1    42
5    45
6    50
9    34
dtype: int64
```

Using a Function

Select elements using a predicate function as the argument to `select()`.

```
x = pd.Series(random.sample(xrange(100), 6))
print x, '\n'
```

```
print 'select func =>\n', x.select(lambda a: x.iloc[a] > 20)

0    83

1    96

2    29

3    15

4    28

5    12

dtype: int64

select func =>

0    83

1    96

2    29

4    28

dtype: int64
```

By List of Labels

Use `filter(items=[..])` with the labels to be selected in a list.

```
x = pd.Series([1, 2, 3, 4, np.nan, 5, 6])

print x, '\n'

print 'filtered =>\n', x.filter(items=[1, 2, 6])

0    1.0

1    2.0

2    3.0

3    4.0

4    NaN

5    5.0

6    6.0

dtype: float64

filtered =>

1    2.0

2    3.0

6    6.0

dtype: float64
```

Regex Match on Label

Select labels to filter using a regular expression match with `filter(regex='..')`.

```
x = pd.Series({'apple': 1.99,

              'orange': 2.49,

              'banana': 0.99,

              'grapes': 1.49,

              'melon': 3.99})

print x, '\n'

print 'regex filter =>\n', x.filter(regex='a

)

apple     1.99

banana    0.99

grapes    1.49

melon     3.99

orange    2.49

dtype: float64

regex filter =>

banana    0.99

dtype: float64
```

Substring Match on Label

Use the `filter(like='..')` version to perform a substring match on the labels to be selected.

```
print 'like filter =>\n', x.filter(like='an')

like filter =>

banana    0.99

orange    2.49

dtype: float64
```

Sorting

Ah! Sorting. The all important functionality when playing with data.

Here is how you can sort a Series by labels or by value.

By Index (or Labels)

Use `sort_index()`.

```
x = pd.Series(random.sample(xrange(100), 6), index=random.sample(map(chr,
xrange(ord('a'), ord('z'))), 6))

print x, '\n'

print 'sort by index: =>\n', x.sort_index(), '\n'

p    37

e    44

b    93

l    75

n     4

s    83

dtype: int64

sort by index: =>

b    93

e    44

l    75

n     4

p    37

s    83

dtype: int64
```

By Values

Use `sort_values()` to sort by the values.

```
print 'sort by value: =>\n', x.sort_values()

sort by value: =>

n     4

p    37

e    44

l    75

s    83

b    93

dtype: int64
```

Summary

This article covers some commonly used methods to deal with a Series object, including methods to retrieve general information about a Series, modifying a Series, selection, and sorting.

Sumendar

Python Pandas Tutorial: DataFrame Basics

The most commonly used data structures in pandas are DataFrames, so it's important to know at least the basics of working with them.

The DataFrame is the most commonly used data structures in pandas. As such, it is very important to learn various specifics about working with the DataFrame. After of creating a DataFrame, let's now delve into some methods for working with it.

Getting Started

Import these libraries: `pandas`, `mattplotlib` for plotting, `numpy`.

```
import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import random
```

If you are working with a Jupyter (or iPython) notebook and want to show graphs inline, use this definition.

```
%matplotlib inline
```

Let's now load some CSV data into our DataFrame for working with it. The data we have loaded is the World Happiness Report 2016.

```
x = pd.read_csv('2016.csv')
```

DataFrame Details

Index

The attribute `index` shows the row index labels.

```
x = pd.read_csv('2016.csv')

print x.index

RangeIndex(start=0, stop=157, step=1)
```

The index is a `RangeIndex` if the labels are contiguous integers.

Columns

Get the columns using the attribute `columns`.

```
print x.columns
```

```
Index([u'Country', u'Region', u'Happiness Rank', u'Happiness Score',
       u'Lower Confidence Interval', u'Upper Confidence Interval',
       u'Economy (GDP per Capita)', u'Family', u'Health (Life Expectancy)',
       u'Freedom', u'Trust (Government Corruption)', u'Generosity',
       u'Dystopia Residual'],
      dtype='object')
```

Values

The raw values array can be extracted using `values`.

```
print x.values
```

```
[['Denmark' 'Western Europe' 1 ..., 0.44453000000000004 0.36171 2.73939]
 ['Switzerland' 'Western Europe' 2 ..., 0.41203 0.28083 2.69463]
 ['Iceland' 'Western Europe' 3 ..., 0.14975 0.47678000000000004 2.83137]
 ...,
 ['Togo' 'Sub-Saharan Africa' 155 ..., 0.11587 0.17517 2.1354]
 ['Syria' 'Middle East and Northern Africa' 156 ..., 0.17232999999999998
  0.48396999999999996 0.81789]
 ['Burundi' 'Sub-Saharan Africa' 157 ..., 0.09419 0.2029 2.1040400000000004]]
```

Shape

Get a tuple of the number of rows and columns of the DataFrame using the `shape` attribute.

```
x.shape
```

```
(157, 13)
```

Size

Use the `count()` method to retrieve a count of (non-NaN) elements in each column. This method ignores any NaN elements in the column.

```
print x.count()
```

```
Country                          157
Region                           157
```

```
Happiness Rank                   157
Happiness Score                  157
Lower Confidence Interval        157
Upper Confidence Interval        157
Economy (GDP per Capita)         157
Family                           157
Health (Life Expectancy)         157
Freedom                          157
Trust (Government Corruption)    157
Generosity                       157
Dystopia Residual                157
dtype: int64
```

And the `size` attribute returns the total number of elements (including NaNs) in the DataFrame. This means the value (nrows * ncols).

```
print x.size
```

```
2041
```

Statistics

Get detailed statistics of the DataFrame using the method `describe()`. Returns various details such as mean, min, max, etc. for each column.

```
print x.describe()
```

```
       Happiness Rank  Happiness Score  Lower Confidence Interval  \
count      157.000000       157.000000                 157.000000
mean        78.980892         5.382185                   5.282395
std         45.466030         1.141674                   1.148043
min          1.000000         2.905000                   2.732000
25%         40.000000         4.404000                   4.327000
50%         79.000000         5.314000                   5.237000
75%        118.000000         6.269000                   6.154000
max        157.000000         7.526000                   7.460000
       Upper Confidence Interval  Economy (GDP per Capita)      Family  \
count                 157.000000                157.000000  157.000000
mean                    5.481975                  0.953880    0.793621
```

```
std                        1.136493                    0.412595     0.266706

...
```

Head and Tail

The `head()` method retrieves the first five rows from the DataFrame.

```
x = pd.read_csv('big-data/Salaries.csv')

print x.head()

   yearID teamID lgID   playerID  salary

0    1985    ATL   NL  barkele01  870000

1    1985    ATL   NL  bedrost01  550000

2    1985    ATL   NL  benedbr01  545000

3    1985    ATL   NL   campri01  633333

4    1985    ATL   NL  ceronri01  625000
```

And the tail method retrieves the last five rows.

```
print x.tail()

       yearID teamID lgID   playerID    salary

26423    2016    WSN   NL  strasst01  10400000

26424    2016    WSN   NL  taylomi02    524000

26425    2016    WSN   NL  treinbl01    524900

26426    2016    WSN   NL  werthja01  21733615

26427    2016    WSN   NL  zimmery01  14000000
```

The cumulative methods return a DataFrame with the appropriate cumulative function applied to the rows. Some of the operations are not valid for non-numeric columns.

Cumulative Sum

`cumsum()` (cumulative sum): Value of each row is replaced by the sum of all prior rows including this row. String value rows use concatenation as shown below.

```
y = pd.DataFrame({'one': pd.Series(range(5)),

                  'two': pd.Series(range(5, 10)),

                  'three': pd.Series(list('abcde'))})

print 'head =>\n', y.head(), '\n'

print 'cumsum =>\n', y.cumsum(), '\n'

head =>
```

```
   one three  two
0    0     a    5
1    1     b    6
2    2     c    7
3    3     d    8
4    4     e    9
cumsum =>
  one  three two
0   0      a   5
1   1     ab  11
2   3    abc  18
3   6   abcd  26
4  10  abcde  35
```

Cumulative Product

`cumprod()` **(cumulative product)**: Row value is replaced by product of all prior rows. This method is not applicable to non-numeric rows. If there are non-numeric rows in the DataFrame, you will need to extract a subset of the DataFrame as shown.

```
print 'cumprod =>\n', y[['one', 'two']].cumprod(), '\n'
cumprod =>
   one    two
0    0      5
1    0     30
2    0    210
3    0   1680
4    0  15120
```

Cumulative Maximum

`cummax()` **(cumulative max)**: Value of the row is replaced by the maximum value of all prior rows till now. In the example below, for demonstrating this method, we use this method on reversed rows of the original DataFrame.

```
print 'rev =>\n', y.iloc[::-1], '\n',
print 'cummax =>\n', y.iloc[::-1].cummax(), '\n'
rev =>
```

```
   one three  two
4    4     e    9
3    3     d    8
2    2     c    7
1    1     b    6
0    0     a    5
cummax =>
  one three two
4   4     e   9
3   4     e   9
2   4     e   9
1   4     e   9
0   4     e   9
```

Cumulative Minimum

`cummin()` :Similar to cummax, except computes the minimum of values till this row.

```
print 'cummin =>\n', y.cummin(), '\n'
cummin =>
  one three two
0   0     a   5
1   0     a   5
2   0     a   5
3   0     a   5
4   0     a   5
```

Index of Min and Max Values

Use the methods `idxmin()` and `idxmax()` to obtain the index label of the rows containing minimum and maximum values. Applicable only to numeric columns, so non-numeric columns need to be filtered out.

```
y = pd.DataFrame({'one': pd.Series(random.sample(xrange(100), 5),
index=list('abcde')),
                 'two': pd.Series(random.sample(xrange(100), 5),
index=list('abcde')),
                 'three': pd.Series(list('ABCDE'), index=list('abcde'))})
```

```
print y, '\n'

print 'idxmax =>\n', y[['one', 'two']].idxmax(), '\n'

print 'idxmin =>\n', y[['one', 'two']].idxmin(), '\n'

   one three  two

a   48     A   25

b   38     B   13

c   62     C   91

d   79     D   32

e    2     E   42

idxmax =>

one    d

two    c

dtype: object

idxmin =>

one    e

two    b

dtype: object
```

Value Counts

The method `value_counts()` returns the number of times each value is repeated in the column. Note: this is not a DataFrame method; rather it is applicable on a column (which is a Series object).

```
x = pd.read_csv('big-data/Salaries.csv')

print 'top 10 =>\n', x.head(10), '\n'

print 'value_counts =>\n', x['yearID'].value_counts().head(10)

top 10 =>

   yearID teamID lgID   playerID  salary

0    1985    ATL   NL  barkele01  870000

1    1985    ATL   NL  bedrost01  550000

2    1985    ATL   NL  benedbr01  545000

3    1985    ATL   NL   campri01  633333

4    1985    ATL   NL  ceronri01  625000

5    1985    ATL   NL  chambch01  800000
```

```
6    1985    ATL   NL  dedmoje01  150000

7    1985    ATL   NL  forstte01  483333

8    1985    ATL   NL  garbege01  772000

9    1985    ATL   NL  harpete01  250000

value_counts =>

1999    1006

1998     998

1995     986

1996     931

1997     925

1993     923

1994     884

1990     867

2001     860

2008     856

Name: yearID, dtype: int64
```

Summary

We covered a few aspects of the DataFrame in this article. Ways of learning various details of the DataFrame including size, shape, statistics, etc. were presented.