

Curiosity

Musings on observations.

- [RSS](#)

- [Blog](#)
- [Archives](#)

Pandas and Python: Top 10

Mar 7th, 2013 1:43 pm

I recently discovered the high-performance [Pandas](#) library written in Python while performing data munging in a machine learning project. Using simple examples, I want to highlight my favorite (and sometimes hard to find) features.

Apart from serving as a quick reference, I hope this post will help new users to quickly start extracting value from Pandas. For a good overview of Pandas and its advanced features, I highly recommended Wes McKinney's [Python for Data Analysis](#) book and the [documentation](#) on the website.

Here is my top 10 list:

1. [Indexing](#)
2. [Renaming](#)
3. [Handling missing values](#)
4. [map\(\).apply\(\).applymap\(\)](#)
5. [groupby\(\)](#)
6. [New Columns = f\(Existing Columns\)](#)
7. [Basic stats](#)
8. [Merge, join](#)
9. [Plots](#)
10. [Scikit-learn conversion](#)

Example DataFrame

I will use a simple data frame for explanation.

```
1 In [1]: import pandas as pd
2
3 In [2]: import numpy as np
4
5 In [3]: from pandas import DataFrame, Series
6
7 In [4]: df = DataFrame({'int_col' : [1,2,6,8,-1], 'float_col' : [0.1, 0.2,0.2,10.1,None], 'str_col' : ['a','b',None,'c','a']})
8
9 In [5]: df
10 Out[5]:
11 float_col  int_col  str_col
12 0         0.1      1      a
13 1         0.2      2      b
14 2         0.2      6     None
15 3        10.1      8      c
16 4         NaN     -1      a
```

Indexing

Selecting a subset of columns

It is one of the simplest features but was surprisingly difficult to find. The [ix](#) method works elegantly for this purpose. Suppose you wanted to index only using columns *int_col* and *string_col*, you would use the advanced indexing **ix** method as shown below.

```
1 In [6]: df.ix[:,['float_col','int_col']]
2 Out[6]:
3 float_col  int_col
4 0         0.1      1
5 1         0.2      2
6 2         0.2      6
7 3        10.1      8
8 4         NaN     -1
```

EDIT Suggestion by Dan in the comments below. Another (probably more elegant) syntax for indexing multiple columns is given below.

```
1 In [9]: df[['float_col','int_col']]
2 Out[9]:
3 float_col  int_col
4 0         0.1      1
```

5	1	0.2	2
6	2	0.2	6
7	3	10.1	8
8	4	NaN	-1

Conditional indexing

One can index using [boolean indexing](#)

```
1 In [7]: df[df['float_col'] > 0.15]
2 Out[7]:
3   float_col  int_col str_col
4 1         0.2      2      b
5 2         0.2      6     None
6 3        10.1      8      c
```

```
1 In [8]: df[df['float_col'] == 0.1]
2 Out[8]:
3   float_col  int_col str_col
4 0         0.1      1      a
```

EDIT Suggestion by Roby Levy in the comments below. One can select multiple boolean operators (| for or, & for and, and ~ for not) and group them by parenthesis.

```
1 In [10]: df[(df['float_col'] > 0.1) & (df['int_col']>2)]
2 Out[10]:
3   float_col  int_col str_col
4 2         0.2      6     None
5 3        10.1      8      c
6
7 In [11]: df[(df['float_col'] > 0.1) | (df['int_col']>2)]
8 Out[11]:
9   float_col  int_col str_col
10 1         0.2      2      b
11 2         0.2      6     None
12 3        10.1      8      c
13
14 In [12]: df[~(df['float_col'] > 0.1)]
15 Out[12]:
16   float_col  int_col str_col
17 0         0.1      1      a
18 4         NaN     -1      a
```

Renaming columns

Use the [rename](#) method to rename columns. It copies the data to another DataFrame.

```
1 In [9]: df2 = df.rename(columns={'int_col' : 'some_other_name'})
2
3 In [10]: df2
4 Out[10]:
5   float_col  some_other_name str_col
6 0         0.1              1      a
7 1         0.2              2      b
8 2         0.2              6     None
9 3        10.1              8      c
10 4         NaN             -1      a
```

Set the **inplace = True** flag incase you want to modify the existing DataFrame.

```
1 In [11]: df2.rename(columns={'some_other_name' : 'int_col'}, inplace = True)
2 Out[11]:
3   float_col  int_col str_col
4 0         0.1      1      a
5 1         0.2      2      b
6 2         0.2      6     None
7 3        10.1      8      c
8 4         NaN     -1      a
```

Handling missing values

[Handling of missing values](#) can be performed beautifully using pandas.

Drop missing values

The [dropna](#) can be used to drop rows or columns with missing data (NaN). By default, it drops all rows with any missing entry.

```
1 n [12]: df2
2 Out[12]:
3   float_col  int_col str_col
4 0         0.1      1      a
5 1         0.2      2      b
```

```

6 2      0.2      6  None
7 3     10.1      8    c
8 4      NaN     -1    a
9
10 In [13]: df2.dropna()
11 Out[13]:
12 float_col  int_col  str_col
13 0         0.1      1      a
14 1         0.2      2      b
15 3         10.1     8      c

```

Fill missing values

The [fillna](#) method on the other hand can be used to fill missing data (NaN). The example below shows a simple replacement using the mean of the available values.

```

1 In [14]: df3 = df.copy()
2
3 In [15]: mean = df3['float_col'].mean()
4
5 In [16]: df3
6 Out[16]:
7 float_col  int_col  str_col
8 0         0.1      1      a
9 1         0.2      2      b
10 2         0.2      6    None
11 3         10.1     8      c
12 4         NaN     -1      a
13
14 In [17]: df3['float_col'].fillna(mean)
15 Out[17]:
16 0         0.10
17 1         0.20
18 2         0.20
19 3         10.10
20 4         2.65
21 Name: float_col

```

Map, Apply

Forget writing *for loops* while using pandas. One can do beautiful vectorized computation by applying function over rows and columns using the **map**, **apply** and **applymap** methods.

map

The [map](#) operation operates over each element of a Series.

```

1 In [18]: df['str_col'].dropna().map(lambda x : 'map_' + x)
2 Out[18]:
3 0      map_a
4 1      map_b
5 3      map_c
6 4      map_a
7 Name: str_col

```

apply

The [apply](#) is a pretty flexible function which, as the name suggests, applies a function along any axis of the DataFrame. The examples show the application of the *sum* function over columns. (Thanks to Mindey in the comments below to use np.sum instead of np.sqrt in the example)

```

1 In [19]: df.ix[:,['int_col','float_col']].apply(np.sqrt)
2 Out[19]:
3 int_col      16.0
4 float_col    10.6
5 dtype: float64

```

applymap

The [applymap](#) operation can be used to apply the function to each element of the DataFrame.

```

1 In [39]: def some_fn(x):
2 ....:     if type(x) is str:
3 ....:         return 'applymap_' + x
4 ....:     elif x:
5 ....:         return 100 * x
6 ....:     else:
7 ....:         return
8 ....:
9
10 In [40]: df.applymap(some_fn)
11 Out[40]:
12 float_col  int_col  str_col
13 0         10      100  applymap_a
14 1         20      200  applymap_b
15 2         20      600      None

```

```

16 3      1010      800  applymap_c
17 4      NaN     -100  applymap_a

```

Vectorized mathematical and string operations

HT: [@janschulz](#)

One can perform vectorized calculations using simple operators and numpy functions.

```

1 In [4]: df = pd.DataFrame(data={"A":[1,2], "B":[1.2,1.3]})
2
3 In [5]: df["C"] = df["A"]+df["B"]
4
5 In [6]: df
6 Out[6]:
7      A      B      C
8  0  1  1.2  2.2
9  1  2  1.3  3.3
10
11 In [7]: df["D"] = df["A"]*3
12
13 In [8]: df
14 Out[8]:
15      A      B      C      D
16  0  1  1.2  2.2  3
17  1  2  1.3  3.3  6
18
19 In [9]: df["E"] = np.sqrt(df["A"])
20
21 In [10]: df
22 Out[10]:
23      A      B      C      D      E
24  0  1  1.2  2.2  3  1.000000
25  1  2  1.3  3.3  6  1.414214

```

Also, [vectorized string operations](#) are easy to use.

```

1 In [11]: df = pd.DataFrame(data={"A":[1,2], "B":[1.2,1.3], "Z":["a","b"]})
2
3 In [12]: df
4 Out[12]:
5      A      B      Z
6  0  1  1.2  a
7  1  2  1.3  b
8
9 In [13]: df["F"] = df.Z.str.upper()
10
11 In [14]: df
12 Out[14]:
13      A      B      Z      F
14  0  1  1.2  a  A
15  1  2  1.3  b  B

```

GroupBy

The [groupby](#) method let's you perform SQL-like grouping operations. The example below shows a grouping operation performed with *str_col* columns entries as keys. It is used to calculate the mean of the *float_col* for each key. For more details, please refer to the [split-apply-combine](#) description on the pandas website.

```

1 In [41]: grouped = df['float_col'].groupby(df['str_col'])
2
3 In [42]: grouped.mean()
4 Out[42]:
5 str_col
6 a          0.1
7 b          0.2
8 c         10.1

```

New Columns = f(Existing Columns)

Generating new columns from existing columns in a data frame is an integral part of my workflow. This was one of the hardest parts for me to figure out. I hope these examples will save time and effort for other people.

I will try to illustrate it in a piecemeal manner – multiple columns as a function of a single column, single column as a function of multiple columns, and finally multiple columns as a function of multiple columns.

multiple columns as a function of a single column

I often have to generate multiple columns of a DataFrame as a function of a single columns. [Related Stack Overflow question](#)

```

1 In [43]: df4 = df.copy()
2
3 In [44]: def two_three_strings(x):
4     ....:     return x*2, x*3
5     ....:
6
7 In [45]: df4['twice'],df4['thrice'] = zip(*df4['int_col'].map(two_three_strings))

```

```

8
9 In [46]: df4
10 Out[46]:
11 float_col  int_col  str_col  twice  thrice
12 0         0.1      1       a      2      3
13 1         0.2      2       b      4      6
14 2         0.2      6      None     12     18
15 3        10.1      8       c      16     24
16 4         NaN     -1       a      -2     -3

```

single column as a function of multiple columns

It's sometimes useful to generate multiple DataFrame columns from a single column. It comes in handy especially when methods return tuples. [Related Stack Overflow question](#)

```

1 In [47]: df5 = df.copy()
2
3 In [48]: def sum_two_cols(series):
4     ....:     return series['int_col'] + series['float_col']
5     ....:
6
7 In [49]: df5['sum_col'] = df5.apply(sum_two_cols,axis=1)
8
9 In [50]: df5
10 Out[50]:
11 float_col  int_col  str_col  sum_col
12 0         0.1      1       a      1.1
13 1         0.2      2       b      2.2
14 2         0.2      6      None     6.2
15 3        10.1      8       c     18.1
16 4         NaN     -1       a      NaN

```

multiple columns as a function of multiple columns

Finally, a way to generate a new DataFrame with multiple columns based on multiple columns in an existing DataFrame. [Related Stack Overflow question](#)

```

1 In [51]: import math
2
3 In [52]: def int_float_squares(series):
4     ....:     return pd.Series({'int_sq' : series['int_col']**2, 'flt_sq' : series['float_col']**2})
5     ....:
6
7 In [53]: df.apply(int_float_squares, axis = 1)
8 Out[53]:
9     flt_sq  int_sq
10 0     0.01      1
11 1     0.04      4
12 2     0.04     36
13 3    102.01    64
14 4      NaN      1

```

Stats

Pandas provides nifty methods to understand your data. I am highlighting the [describe](#), [correlation](#), [covariance](#), and [correlation](#) methods that I use to quickly make sense of my data.

describe

The *describe* method provides quick stats on all suitable columns.

```

1 In [54]: df.describe()
2 Out[54]:
3     float_col  int_col
4 count      4.00000  5.000000
5 mean        2.65000  3.200000
6 std         4.96689  3.701351
7 min         0.10000 -1.000000
8 25%         0.17500  1.000000
9 50%         0.20000  2.000000
10 75%         2.67500  6.000000
11 max        10.10000  8.000000

```

covariance

The *cov* method provides the covariance between suitable columns.

```

1 In [55]: df.cov()
2 Out[55]:
3     float_col  int_col
4 float_col    24.670000  12.483333
5 int_col      12.483333  13.700000

```

correlation

The *corr* method provides the correlation between suitable columns.

```
1 In [56]: df.corr()
2 Out[56]:
3          float_col  int_col
4 float_col    1.000000  0.760678
5 int_col      0.760678  1.000000
```

Merge and Join

Pandas supports database-like [joins](#) which makes it easy to link data frames.

I will use the simple example to highlight the joins using the merge command.

```
1 n [57]: df
2 Out[57]:
3          float_col  int_col  str_col
4 0          0.1         1         a
5 1          0.2         2         b
6 2          0.2         6        None
7 3         10.1         8         c
8 4          NaN        -1         a
9
10 In [58]: other = DataFrame({'str_col' : ['a','b'], 'some_val' : [1, 2]})
11
12 In [59]: other
13 Out[59]:
14          some_val  str_col
15 0              1         a
16 1              2         b
```

The inner, outer, left and right joins are show below. The data frames are joined using the *str_col* keys.

```
1 In [60]: pd.merge(df,other,on='str_col',how='inner')
2 Out[60]:
3          float_col  int_col  str_col  some_val
4 0          0.1         1         a          1
5 1          NaN        -1         a          1
6 2          0.2         2         b          2
7
8 In [61]: pd.merge(df,other,on='str_col',how='outer')
9 Out[61]:
10          float_col  int_col  str_col  some_val
11 0          0.1         1         a          1
12 1          NaN        -1         a          1
13 2          0.2         2         b          2
14 3          0.2         6        None        NaN
15 4         10.1         8         c          NaN
16
17 In [62]: pd.merge(df,other,on='str_col',how='left')
18 Out[62]:
19          float_col  int_col  str_col  some_val
20 0          0.1         1         a          1
21 1          NaN        -1         a          1
22 2          0.2         2         b          2
23 3          0.2         6        None        NaN
24 4         10.1         8         c          NaN
25
26 In [63]: pd.merge(df,other,on='str_col',how='right')
27 Out[63]:
28          float_col  int_col  str_col  some_val
29 0          0.1         1         a          1
30 1          NaN        -1         a          1
31 2          0.2         2         b          2
```

Plot

I was thoroughly surprised by the plotting capabilities of the pandas library. There are [several plotting methods](#) available. I am highlighting a couple of simple plots that I use the most.

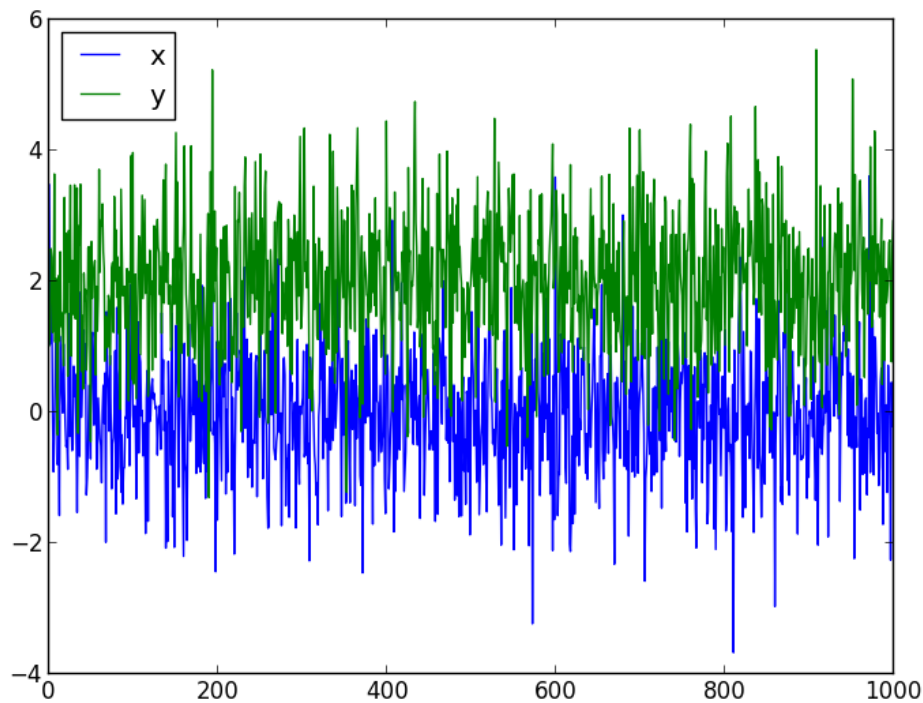
Let's start with a simple data frame to plot.

```
1 In [3]: plot_df = DataFrame(np.random.randn(1000,2),columns=['x','y'])
2
3 In [4]: plot_df['y'] = plot_df['y'].map(lambda x : x + 1)
```

Plot

A simple plot command goes a long way.

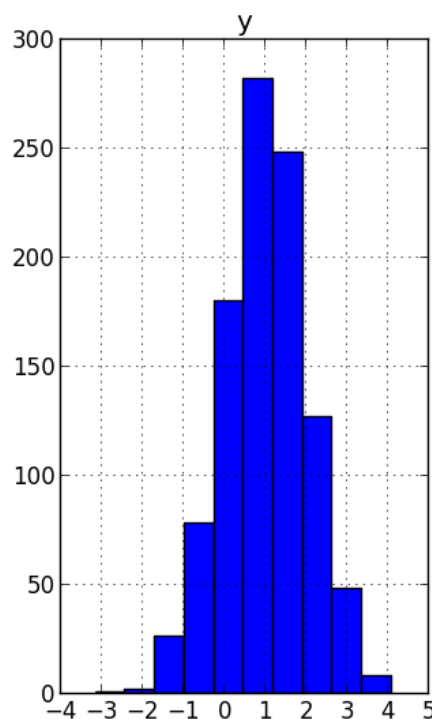
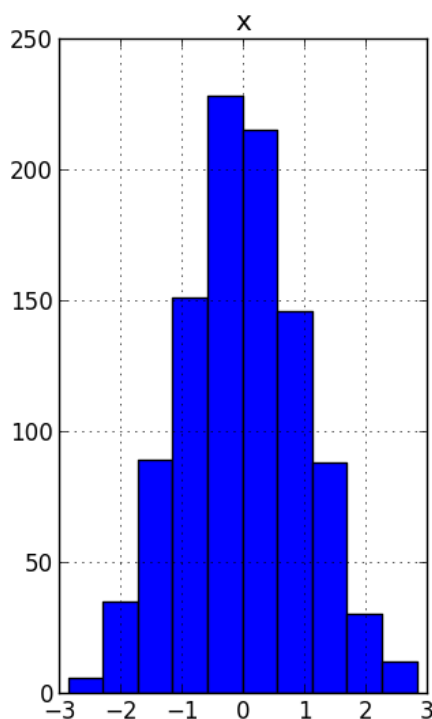
```
1 In [5]: plot_df.plot()
2 Out[5]: <matplotlib.axes.AxesSubplot at 0x10e6cad10>
```



Histograms

I really enjoy histograms to get a quick idea about the distribution of the data.

```
1 In [6]: plot_df.hist()
2 Out[6]: array([[Axes(0.125, 0.1; 0.336957x0.8), Axes(0.563043, 0.1; 0.336957x0.8)]], dtype=object)
```



Scikit-learn conversion

This took me a non-trivial amount of time to figure out and I hope others can avoid this mistake. According to the pandas [documentation](#), the `ndarray` object obtained via the `values` method has `object` dtype if values contain more than `float` and `integer` dtypes. Now even if you slice the str columns away, the resulting array will still consist of `object dtype` and might not play well with other libraries such as scikit-learn which are expecting a `float dtype`. Explicitly converting type works well in this scenario.

```

1 n [18]: df
2 Out[18]:
3      float_col  int_col  str_col
4 0          0.1         1        a
5 1          0.2         2        b
6 2          0.2         6       NaN
7 3         10.1         8        c
8 4          NaN        -1        a
9
10 In [25]: df.values[:, :-1]
11 Out[25]:
12 array([[0.1, 1],
13        [0.2, 2],
14        [0.2, 6],
15        [10.1, 8],
16        [nan, -1]], dtype=object)
17
18 In [26]: df.values[:, :-1].astype(float32)
19 Out[26]:
20 array([[ 0.1,      ,  1.      ],
21        [ 0.2,      ,  2.      ],
22        [ 0.2,      ,  6.      ],
23        [10.10000038,  8.      ],
24        [      nan, -1.      ]], dtype=float32)

```

EDIT HT: Wes Turner via comments. The [sklearn-pandas](#) library looks great for bridging pandas scikit-learn.

Summary

I hope these examples will help new users quickly extract a lot of value out of pandas and serve as a useful quick reference for the pandas pros.

Happy munging!

Posted by Manish Amde Mar 7th, 2013 1:43 pm [introduction](#), [machine learning](#), [pandas](#), [python](#), [tutorial](#)

Tweet

Like Share 27 people like this. Be the first of your friends.

[« Octopress Mocking Python With Kung Fu Panda »](#)

Comments

66 Comments Curiosity  Disqus' Privacy Policy

 Login

 Recommend 14  Tweet  Share

Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Rob Levy • 8 years ago

Awesome. People who write human-friendly tutorials like this deserve prizes.

here's a question I asked on [stackoverflow.com](#) about indexing on multiple columns. (Like doing SELECT * FROM tbl WHERE foo = 'bar' AND boo = 'far')

14   • Reply • Share




manish9ue Mod  Rob Levy • 8 years ago

I updated the article with your suggestion. Thanks!

  • Reply • Share



Rob Levy  manish9ue • 8 years ago

Well done. Good work!

  • Reply • Share



manish9ue Mod  Rob Levy • 8 years ago

Thanks Rob. I will update my post with the SO answers. Also, I am planning to write a similar article for intermediate and advanced users as well. I hope I get time to do that in the near future. :-)

  • Reply • Share



Jan Schulz • 8 years ago

Nice tutorial! Maybe you could transform some of it for the new "10 min intro" which will available in the next version of pandas: <https://github.com/pydata/p...>

Most of your new users can be reached by simply adding up columns or using summary functions on the complete column:

most of your map uses can be replaced by simply adding up columns or using numpy functions on the complete column:

```
df = pd.DataFrame(data={"A":[1,2], "B":[1.2,1.3]})
df["C"] = df["A"]+df["B"]
df["D"] = df["A"]*3
import numpy as np
df["E"] = np.sqrt(df["A"])
```

3 ^ | v • Reply • Share ›



manish9ue Mod → Jan Schulz • 8 years ago

Hi Jan,

Thanks! I will be happy to contribute to the "10 min intro" for the next version.

Also, thanks for highlighting the fact that map operation can be replaced by the simpler numpy functions for integers and floats. Maybe, it can be the #11 point in the list. :-) I was originally planning to use those with string columns/functions where the numpy functions would not work well.

^ | v • Reply • Share ›



Jan Schulz → manish9ue • 8 years ago

There are also new column.str accessors (with several functions) to manipulate the column values as a string:

```
df = pd.DataFrame(data={"A":[1,2], "B":[1.2,1.3], "Z":["a","b"]})
df["F"] = df.Z.str.upper()
df["G"] = df.Z.str.pad(12)
```

:-)

^ | v • Reply • Share ›



manish9ue Mod → Jan Schulz • 8 years ago

Added your suggestions (verbatim :-)) to the vectorized operations section. Better late than never! :-)

^ | v • Reply • Share ›



manish9ue Mod → Jan Schulz • 8 years ago

Thanks! Didn't know this. Let me update the post with your inputs.

It will be nice to get your twitter/github handle for the HT. Else a name str will be used. :-)

^ | v • Reply • Share ›



Jan Schulz → manish9ue • 8 years ago

@janschulz :-)

^ | v • Reply • Share ›



Mindey • 8 years ago

Very nice!! Btw., from the single example about apply with np.sqrt function it is not clear which axis it is applied to by default. Example with np.sum function would be more informative.

1 ^ | v • Reply • Share ›



manish9ue Mod → Mindey • 8 years ago

Hey Mindey, I updated the example to use np.sum. Thanks for your suggestion.

^ | v • Reply • Share ›



Mindey → manish9ue • 8 years ago

Great! It's a nice way to keep it concise. I like your approach.

^ | v • Reply • Share ›



manish9ue Mod → Mindey • 8 years ago

Thanks Mindey! It's a great point -- return type depends on whether the passed function aggregates or not. I will add np.sum as an example as well.

^ | v • Reply • Share ›



Kevin Davenport • 8 years ago

Nice writeup, it looks like pandas is closing the gap between python and R's core functionality. Now we just need to bring over the functionality of the thousands of R packages out there.

<http://kldavenport.com>

1 ^ | v • Reply • Share ›



manish9ue Mod → Kevin Davenport • 7 years ago

Thanks Kevin.

^ | v • Reply • Share ›



sujith chandra • 4 years ago



Awesome tutorial. It would have saved me days of work, if i had stumbled on it earlier.

^ | v • Reply • Share ›



manishg9ue Mod ➔ sujith chandra • 4 years ago

Thanks!

^ | v • Reply • Share ›



Semihsan Doken • 5 years ago

ok, wow, this post seems old but I still find it useful.

^ | v • Reply • Share ›



manishg9ue Mod ➔ Semihsan Doken • 5 years ago

Glad you find it useful!

^ | v • Reply • Share ›



Semihsan Doken • 5 years ago

In the apply example, I still do not understand the use of 'np.sqrt'. I think maybe it is a typo and should be np.sum ?

^ | v • Reply • Share ›



manishg9ue Mod ➔ Semihsan Doken • 5 years ago

Correct: I already clarified the typo in the post. I need to find time to update the example.

"(Thanks to Mindey in the comments below to use np.sum instead of np.sqrt in the example)"

^ | v • Reply • Share ›



Berto • 6 years ago

Hi...any advice on following please...col A = [0 1 2 3 4 0 0 0 5 6 7 9] would like it to appear as A = [0 1 1 1 0 0 0 5 5 5 5] that is find first non zero and pass it to all others #s till 0 again , repeat till end of column. How to do this via pandas? thank you

^ | v • Reply • Share ›



manishg9ue Mod ➔ Berto • 5 years ago

Hi Berto, this question might be better for stack overflow or the pandas group.

^ | v • Reply • Share ›



Sherry Parker • 6 years ago

Thank you so much!! I am just diving into Pandas and Numpy and your article helped me figure out what I had been missing for my assignment. Much appreciated.

^ | v • Reply • Share ›



manishg9ue Mod ➔ Sherry Parker • 6 years ago

Thanks Sherry!

^ | v • Reply • Share ›



Hans Zimmermann • 6 years ago

Extremely handy tutorial.

^ | v • Reply • Share ›



manishg9ue Mod ➔ Hans Zimmermann • 6 years ago

Thanks Hans.

^ | v • Reply • Share ›



harijay • 6 years ago • edited

Thank you ..Thank you ..Thank you...Please can accept writing documentation as your life mission..this was the most clear explanation of pandas "apply" I found after much searching

^ | v • Reply • Share ›



manishg9ue Mod ➔ harijay • 6 years ago

Thanks harijay. I am glad to know that the documentation was useful. I need to write a follow-up post for advanced users that I have been procrastinating on for two years! :-)

^ | v • Reply • Share ›



Amy • 6 years ago

The creation of a new column from existing columns is exactly what I was looking for and having trouble figuring out. Thank you so much for sharing!

^ | v • Reply • Share ›



manishg9ue Mod ➔ Amy • 6 years ago

Thanks Amy. I noted solutions to problems I was struggling with and that took me a long time to figure out. I am glad this post was helpful.

^ | v • Reply • Share ›



Jonathan Berthet • 7 years ago

Thanks so much! You need to put up more examples! Very well explained :)

^ | v • Reply • Share ›



manishguc Mod → Jonathan Berthet • 7 years ago

Thanks Jonathan. I plan to add more examples especially for intermediate and advanced users.

^ | v • Reply • Share ›



Isaac Xin Pei • 8 years ago

very nice post, thumbs up!

^ | v • Reply • Share ›



manishguc Mod → Isaac Xin Pei • 7 years ago

Thanks Isaac.

^ | v • Reply • Share ›



Archs • 8 years ago

Wow, Pandas is really powerful!

^ | v • Reply • Share ›



manishguc Mod → Archs • 7 years ago

Yes, it is. :-)

^ | v • Reply • Share ›



sbzzzz • 8 years ago

Hi,

I've just heard about panda and now try to figure out what the extra stuff it provides compared to numpy/scipy and matplotlib?

thanks

^ | v • Reply • Share ›



manishguc Mod → sbzzzz • 8 years ago

Quick answer: it provides data structures (DataFrame, Series) at a higher level of abstraction than numpy/scipy for easier data manipulation.

1 ^ | v • Reply • Share ›



Stanley Roache • 8 years ago • edited

Just going to add some praise to the pile. It's easy to go find some documentation and nut things out eventually with a cool new tool/package but after finding your post I've been coming back here on the regular for a concise explanation and example of these fundamental operations in pandas, super clear and it made getting up and running easy. Thank you.

^ | v • Reply • Share ›



manishguc Mod → Stanley Roache • 8 years ago

Thanks for the kind words Stanley!

^ | v • Reply • Share ›



Wes Turner • 8 years ago

Also, it's possible to (re-) style matplotlib plots and visualizations: <http://redd.it/1gsxcb#canksc8>

^ | v • Reply • Share ›



manishguc Mod → Wes Turner • 7 years ago

Thanks for the links Wes.

^ | v • Reply • Share ›



Guest • 8 years ago • edited

Nice post. Very clear.

I get often puzzled with pandas join, merge, concat and append, Never sure which one to use and always looking at the docs for the meaning of their parameters. Maybe you can give examples as you did for map, apply and applymap.

In the tutorial you only discuss merge and still it is not clear in the example the use of the parameters: for example, how=inner and how=right give the same result. Same with outer and left...

^ | v • Reply • Share ›



manishguc Mod → Guest • 8 years ago

Thanks Joaquin! I plan to update this post and also write a follow-up post (or two!) with advanced topics. I will use a better yet simple example to explain the merge parameters (inner, left, right, outer) and also try to write more about merge, concat and append.

^ | v • Reply • Share ›



Wes Turner • 8 years ago

Cool. In this instance, I'm not sure that "itertools.zip" would be much faster than "zip" ... vbench, IPython `%timeit?` and `%prun -r`

For Pandas integration with scikit-learn (sklearn), there is <https://github.com/paulgb/s...>

^ | v • Reply • Share ›



manish9ue Mod Wes Turner • 8 years ago

The sklearn-pandas library looks really useful. Thanks!

1 ^ | v • Reply • Share ›



manish9ue Mod Wes Turner • 8 years ago

Updated the post with your comments.

^ | v • Reply • Share ›



Andrew Cameron • 8 years ago • edited

Hi, I've just started using pandas, so thanks for the great post showing me some of the awesome stuff pandas can do! I just wanted to mention that I found your simple plot example a little confusing, for a couple reasons.

First, there's a much better way of incrementing the 'y' column by 1. Instead of doing:

```
df['y'] = df['y'].map(lambda x: x + 1)
```

(which is itself quite confusing because now 'x' is both a column in your data frame, AND a local variable in your lambda function)

you can do this, thanks to what pandas calls "broadcasting" - the 1 literal gets expanded to the size of the entire column!

```
df['y'] = df['y'] + 1
```

Second, the plot you show reveals that y has a mean of 2, not 1! (Maybe you accidentally ran your map command twice.)

I think these fixes would help make your great post even better. Thanks again! =)

^ | v • Reply • Share ›

[Load more comments](#)

[Subscribe](#) [Add Disqus to your site](#)[Add Disqus](#) [Do Not Sell My Data](#)

About Me

Manish Amde. [Linked in](#) profile

Recent Posts

- [theCUBE Interview at Spark Summit 2015](#)
- [Decision Trees, Random Forests and Boosting in Spark](#)
- [Spark Summit Talk: Scalable Distributed Decision Trees in Spark MLlib](#)
- [Mocking Python With Kung Fu Panda](#)
- [Pandas and Python: Top 10](#)

GitHub Repos

[@manishamde](#) on GitHub

Twitter

Tweets by [@manish9ue](#)

Manish Amde Retweeted



David Simon
[@AoDespair](#)

Have I seen The Wire?

Sep 17, 2019

Manish Amde Retweeted



Richard Marx
[@richardmarx](#)

Went to the dentist today. My teeth are fine. I just wanted to hear some of my songs.

Feb 5, 2019

Manish Amde Retweeted



Ryan Caldbeck
[@ryan_caldbeck](#)

1/ A picture is worth a thousand words. And to me, an x-y is worth a million. Here are some things I believe to be true based on my experience as a CEO and investor. Some or all may be completely wrong.

Apr 8, 2018

[Embed](#)

[View on Twitter](#)