10 Python Pandas tricks that make your work more efficient

Some commands you may know already but may not know they can be used this way

Shiu-Tang Li Follow
Mar 13 · 5 min read

Photo from https://unsplash.com/

Pandas is a widely used Python package for structured data. There're many nice tutorials of it, but here I'd still like to introduce a few cool tricks the readers may not know before and I believe they're useful.

read_csv

Everyone knows this command. But the data you're trying to read is large, try adding this argument: **nrows = 5** to only read in a tiny portion of the table before actually loading the whole table. Then you could avoid the mistake by choosing wrong delimiter (it may not always be comma separated).

(Or, you can use 'head' command in linux to check out the first 5 rows (say) in any text file: **head -n 5 data.txt** (Thanks **Ilya Levinson** for pointing out a typo here))

Then, you can extract the column list by using `df.columns.tolist()` to extract all columns, and then add **usecols = [‘c1’, ‘c2’, …]** argument to load the columns you need. Also, if you know the data types of a few specific columns, you can add the argument **dtype = {‘c1’: str, ‘c2’: int, …}** so it would load faster. Another advantage of this argument that if you have a column which contains both strings and numbers, it’s a good practice to declare its type to be string, so you won’t get errors while trying to merge tables using this column as a key.

select_dtypes

If data preprocessing has to be done in Python, then this command would save you some time. After reading in a table, the default data types for each column could be bool, int64, float64, object, category, timedelta64, or datetime64. You can first check the distribution by

```
df.dtypes.value_counts()
```

to know all possible data types of your dataframe, then do

```
df.select_dtypes(include=['float64', 'int64'])
```

to select a sub-dataframe with only numerical features.

copy

This is an important command if you haven’t heard of it already. If you do the following commands:

```
import pandas as pd
df1 = pd.DataFrame({ 'a':[0,0,0], 'b': [1,1,1]})
df2 = df1
df2['a'] = df2['a'] + 1
df1.head()
```

You’ll find that df1 is changed. This is because df2 = df1 is not making a copy of df1 and assign it to df2, but setting up a pointer pointing to df1. So any changes in df2 would result in changes in df1. To fix this, you can do either

```
df2 = df1.copy()
```

or

```
from copy import deepcopy
df2 = deepcopy(df1)
```

map

This is a cool command to do easy data transformations. You first define a dictionary with 'keys' being the old values and 'values' being the new values.

```
level_map = {1: 'high', 2: 'medium', 3: 'low'}
df['c_level'] = df['c'].map(level_map)
```

Some examples: True, False to 1, 0 (for modeling); defining levels; user defined lexical encodings.

apply or not apply?

If we'd like to create a new column with a few other columns as inputs, apply function would be quite useful sometimes.

```
def rule(x, y):
    if x == 'high' and y > 10:
         return 1
    else:
         return 0
```

```
df = pd.DataFrame({ 'c1':[ 'high' ,'high', 'low', 'low'],
'c2': [0, 23, 17, 4]})
df['new'] = df.apply(lambda x: rule(x['c1'], x['c2']), axis
=  1)
df.head()
```

In the codes above, we define a function with two input variables, and use the apply function to apply it to columns 'c1' and 'c2'.

but **the problem of 'apply' is that it's sometimes too slow**. Say if you'd like to calculate the maximum of two columns 'c1' and 'c2', of course you can do

```
df['maximum'] = df.apply(lambda x: max(x['c1'], x['c2']), axis = 1)
```

but you'll find it much slower than this command:

```
df['maximum'] = df[['c1','c2']].max(axis =1)
```

Takeaway: Don't use apply if you can get the same work done with other built-in functions (they're often faster). For example, if you want to round column 'c' to integers, do **round(df['c'], 0)** or **df['c'].round(0)** instead of using the apply function: `df.apply(lambda x: round(x['c'], 0), axis = 1)` .

value counts

This is a command to check value distributions. For example, if you'd like to check what are the possible values and the frequency for each individual value in column 'c' you can do

```
df['c'].value_counts()
```

There're some useful tricks / arguments of it:
A. **normalize = True**: if you want to check the frequency instead of counts.
B. **dropna = False**: if you also want to include missing values in the stats.
C. `df['c'].value_counts().reset_index()` : if you want to convert the stats table into a pandas dataframe and manipulate it
D. `df['c'].value_counts().reset_index().sort_values(by='index')` : show the stats sorted by distinct values in column 'c' instead of counts.

(Update 2019.4.18—for D. above, **Hao Yang** points out a simpler way without .reset_index(): `df['c'].value_counts().sort_index()`)

number of missing values

When building models, you might want to exclude the row with too many missing values / the rows with all missing values. You can use .isnull() and .sum() to count the number of missing values within the specified columns.

```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame({ 'id': [1,2,3], 'c1':[0,0,np.nan], 'c2':
[np.nan,1,1]})
df = df[['id', 'c1', 'c2']]
df['num_nulls'] = df[['c1', 'c2']].isnull().sum(axis=1)
df.head()
```

select rows with specific IDs

In SQL we can do this using SELECT * FROM … WHERE ID in ('A001', 'C022', …) to get records with specific IDs. If you want to do the same thing with pandas, you can do

```
df_filter = df['ID'].isin(['A001','C022',...])
df[df_filter]
```

Percentile groups

You have a numerical column, and would like to classify the values in that column into groups, say top 5% into group 1, 5–20% into group 2, 20%-50% into group 3, bottom 50% into group 4. Of course, you can do it with pandas.cut, but I'd like to provide another option here:

```
import numpy as np
cut_points = [np.percentile(df['c'], i) for i in [50, 80,
95]]
df['group'] = 1
for i in range(3):
    df['group'] = df['group'] + (df['c'] < cut_points[i])
# or <= cut_points[i]
```

which is fast to run (no apply function used).

to_csv

Again this is a command that everyone would use. I'd like to point out two tricks here. The first one is

```
print(df[:5].to_csv())
```

You can use this command to print out the first five rows of what are going to be written into the file exactly.

Another trick is dealing with integers and missing values mixed together. If a column contains both missing values and integers, the data type would still be float instead of int. When you export the table, you can add **float_format='%.0f'** to round all the floats to integers. Use this trick if you only want integer outputs for all columns—you'll get rid of all annoying '.0's.