How to iterate over rows in a DataFrame in Pandas?

Ask Question

1137

317

I have a `DataFrame` from pandas:

```
import pandas as pd
inp = [{'c1':10, 'c2':100}, {'c1':11,'c2':110}, {'c1':12,'c2':120}]
df = pd.DataFrame(inp)
print df
```

Output:

```
   c1   c2
0  10  100
1  11  110
2  12  120
```

Now I want to iterate over the rows of this frame. For every row I want to be able to access its elements (values in cells) by the name of the columns. For example:

```
for row in df.rows:
   print row['c1'], row['c2']
```

Is it possible to do that in pandas?

I found this similar question. But it does not give me the answer I need. For example, it is suggested there to use:

```
for date, row in df.T.iteritems():
```

or

```
for row in df.iterrows():
```

But I do not understand what the `row` object is and how I can work with it.

python pandas rows dataframe

edited Aug 24 '18 at 19:20

petezurich
3,840 8 19 36

28.9k 125 285 380

3 The df.iteritems() iterates over columns and not rows. Thus, to make it iterate over rows, you have to transpose (the "T"), which means you change rows and columns into each other (reflect over diagonal). As a result, you effectively iterate the original dataframe over its rows when you use df.T.iteritems() – Stefan Gruenwald Dec 14 '17 at 23:41

DON'T use `iterrows()` !. Depending on what you're trying to do, there are possibly much better alternatives.. `iter*` functions should be used in very rare circumstances. Also related. – coldspeed yesterday

18 Answers

1629

DataFrame.iterrows is a generator which yield both index and row

```
for index, row in df.iterrows():
    print(row['c1'], row['c2'])

Output:
   10 100
   11 110
   12 120
```

edited Dec 19 '18 at 19:00
gcamargo
911 1 10 22

answered May 10 '13 at 7:07
waitingkuo
38.4k 17 87 100

112 Note: "Because iterrows returns a Series for each row, it **does not** preserve dtypes across the rows." Also, "You **should never modify** something you are iterating over." According to pandas 0.19.1 docs – viddik13 Dec 7 '16 at 16:24

3 @viddik13 that's a great note thanks. Because of that I ran into a case where numerical values like `431341610650` where read as `4.31E+11` . Is there a way around preserving the dtypes? – Aziz Alto Sep 5 '17 at 16:30

9 @AzizAlto use `itertuples` , as explained below. See also pandas.pydata.org/pandas-docs/stable/generated/… – Axel Sep 7 '17 at 11:45

2 if you don't need to preserve the datatype, iterrows is fine. @waitingkuo's tip to separate the index makes it much easier to parse. – beep_check May 3 '18 at 16:55

242

To iterate through DataFrame's row in pandas one can use:

- DataFrame.iterrows()

```
for index, row in df.iterrows(
    print row["c1"], row["c2"]
```

- DataFrame.itertuples()

```
for row in df.itertuples(index
    print getattr(row, "c1"),
```

`itertuples()` is supposed to be faster than `iterrows()`

But be aware, according to the docs (pandas 0.21.1 at the moment):

- iterrows: `dtype` might not match from row to row

  > Because iterrows returns a Series for each row, it **does not preserve** dtypes across the rows (dtypes are preserved across columns for DataFrames).

- iterrows: Do not modify rows

  > You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

  Use DataFrame.apply() instead:

  ```
  new_df = df.apply(lambda x: x
  ```

- itertuples:

> identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

edited Dec 17 '17 at 3:54

answered Dec 7 '16 at 16:41
viddik13
2,663 1 7 15

2 Just a small question from someone reading this thread so long after its completion: how df.apply() compares to itertuples in terms of efficiency? – Raul Guarini Jan 26 '18 at 13:16

3 Note: you can also say something like `for row in df[['c1','c2']].itertuples(index=True, name=None):` to include only certain columns in the row iterator. – Brian Burns Jun 29 '18 at 7:29

6 Instead of `getattr(row, "c1")`, you can use just `row.c1`. – viraptor Aug 13 '18 at 6:20

I am about 90% sure that if you use `getattr(row, "c1")` instead of `row.c1`, you lose any performance advantage of `itertuples`, and if you actually need to get to the property via a string, you should use iterrows instead. – Noctiphobia Aug 24 '18 at 10:34

When I tried this it only printed the column values but not the headers. Are the column headers excluded from the row attributes? – Marlo Dec 6 '18 at 5:39

136

While `iterrows()` is a good option, sometimes `itertuples()` can be much faster:

```
df = pd.DataFrame({'a': randn(1000
(1000)), 'x': 'x'})

%timeit [row.a * 2 for idx, row in
# => 10 loops, best of 3: 50.3 ms

%timeit [row[1] * 2 for row in df.
# => 1000 loops, best of 3: 541 µs
```

edited Jun 1 '16 at 9:00

3 Much of the time difference in your two examples seems like it is due to the fact that you appear to be using label-based indexing for the .iterrows() command and integer-based indexing for the .itertuples() command. – Alex Sep 20 '15 at 17:00

2 For a finance data based dataframe(timestamp, and 4x float), itertuples is 19,57 times faster then iterrows on my machine. Only `for a,b,c in izip(df["a"],df["b"],df["c"]:` is almost equally fast. – harbun Oct 19 '15 at 13:03

6 Can you explain why it's faster? – Abe Miessler Jan 10 '17 at 22:05

3 @AbeMiessler `iterrows()` boxes each row of data into a Series, whereas `itertuples()` does not. – miradulo Feb 13 '17 at 17:30

3 Note that the order of the columns is actually indeterminate, because `df` is created from a dictionary, so `row[1]` could refer to any of the columns. As it turns out though the times are roughly the same for the integer vs the float columns. – Brian Burns Nov 5 '17 at 17:29

73

You can also use `df.apply()` to iterate over rows and access multiple columns for a function.

docs: DataFrame.apply()

```
def valuation_formula(x, y):
    return x * y * 0.5

df['price'] = df.apply(lambda row:
axis=1)
```

answered Jun 1 '15 at 6:24

cheekybastard
3,811 1 17 24

Is the df['price'] refers to a column name in the data frame? I am trying to create a dictionary with unique values from several columns in a csv file. I used your logic to create a dictionary with unique keys and values and got an error stating **TypeError: ("'Series' objects are mutable, thus they cannot be hashed", u'occurred at index 0')** – SRS Jul 1 '15 at 17:55

Code: df['Workclass'] = df.apply(lambda row: dic_update(row), axis=1) **end of line** id = 0 **end of line** def dic_update(row): if row not in dic: dic[row] = id id = id + 1 – SRS Jul 1 '15 df[workclass].apply(same thing) – SRS Jul 1 '15 at 19:06

2 Having the axis default to 0 is the worst – zthomas.nc Nov 29 '17 at 23:58

4 Notice that `apply` doesn't "iteratite" over rows, rather it applies a function row-wise. The above code wouldn't work if you really *do* need iterations and indeces, for instance when comparing values across different rows (in that case you can do nothing but iterating). – gented Apr 4 '18 at 13:44

68

You can use the df.iloc function as follows:

```
for i in range(0, len(df)):
    print df.iloc[i]['c1'], df.ilo
```

edited Nov 7 '16 at 9:09

answered Sep 7 '16 at 12:56

PJay
952 6 11

17 Using `0` in `range` is pointless, you can omit it. – Pedro Lobito Apr 6 '17 at 8:51

1 I know that one should avoid this in favor of iterrows or itertuples, but it would be interesting to know why. Any thoughts? – rocarvaj Oct 5 '17 at 14:50

6 This is the only valid technique I know of if you want to preserve the data types, and also refer to columns by name. `itertuples` preserves data types, but gets rid of any name it doesn't like. `iterrows` does the opposite. – Ken Williams Jan 18 '18 at 19:22

3 Spent hours trying to wade through the idiosyncrasies of pandas data structures to do something simple AND expressive. This results in readable code. – Sean Anderson Sep 19 '18 at 12:13

While `for i in range(df.shape[0])` might speed this approach up a bit, it's still about 3.5x slower than the iterrows() approach above for my application. – Kim Miller Dec 14 '18 at 18:18

```
for i, row in df.iterrows():
    for j, column in row.iteritems
        print(column)
```

answered Jan 17 '18 at 9:41
Lucas B
778 1 9 18

16

Use *itertuples()*. It is faster than *iterrows()*:

```
for row in df.itertuples():
    print "c1 :",row.c1,"c2 :",row
```

answered Jul 27 '17 at 16:32
Nurul Akter Towhid
1,500 17 26

8 I don't see how this answer adds anything that was not in the previous answers – chrisfs Jun 25 '18 at 0:49

14

You can write your own iterator that implements `namedtuple`

```
from collections import namedtuple

def myiter(d, cols=None):
    if cols is None:
        v = d.values.tolist()
        cols = d.columns.values.to
    else:
        j = [d.columns.get_loc(c)
        v = d.values[:, j].tolist(

    n = namedtuple('MyTuple', cols

    for line in iter(v):
        yield n(*line)
```

This is directly comparable to `pd.DataFrame.itertuples` . I'm aiming at performing the same task with more efficiency.

For the given dataframe with my function:

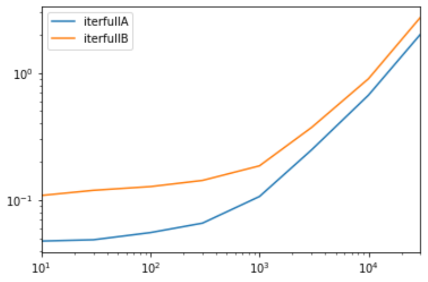
```
list(myiter(df))

[MyTuple(c1=10, c2=100), MyTuple(c
```

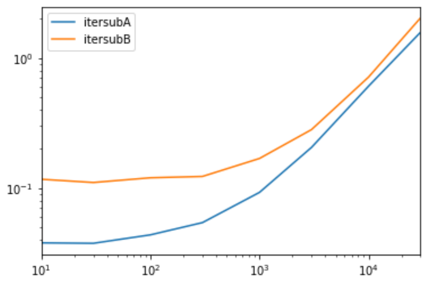
Or with `pd.DataFrame.itertuples` :

```
list(df.itertuples(index=False))

[Pandas(c1=10, c2=100), Pandas(c1=
```

A comprehensive test
We test making all columns available and subsetting the columns.

```
def iterfullA(d):
    return list(myiter(d))

def iterfullB(d):
    return list(d.itertuples(index

def itersubA(d):
    return list(myiter(d, ['col3',

def itersubB(d):
    return list(d[['col3', 'col4',
'col7']].itertuples(index=False))

res = pd.DataFrame(
    index=[10, 30, 100, 300, 1000,
    columns='iterfullA iterfullB i
    dtype=float
)

for i in res.index:
    d = pd.DataFrame(np.random.ran
    for j in res.columns:
        stmt = '{}(d)'.format(j)
        setp = 'from __main__ impo
        res.at[i, j] = timeit(stmt

res.groupby(res.columns.str[4:-1],
```





edited Nov 7 '17 at 4:29

answered Nov 7 '17 at 4:15
piRSquared
160k 24 159 303

1 For people who don't want to read the code: blue line is `intertuples` , orange line is a list of an iterator thru a yield block. `interrows` is not compared. – James L. Dec 1 '17 at 16:06

12

To loop all rows in a `dataframe` you can use:

```
for x in range(len(date_example.in
    print date_example['Date'].ilo
```

edited Apr 4 '17 at 20:46

answered Mar 11 '17 at 22:44
Pedro Lobito
50.6k 16 138 172

10

IMHO, the simplest decision

```
for ind in df.index:
    print df['c1'][ind], df['c2']
```

answered Nov 2 '17 at 10:33
Grag2015
178 3 9

how is the performance of this option when used on a large dataframe (millions of rows for example)? – Bazyli Debowski Sep 10 '18 at 12:41

Honestly, I don't know exactly, I think that in comparison with the best answer, the elapsed time will be about the same, because both cases use "for"-construction. But the memory may be different in some cases. – Grag2015 Oct 25 '18 at 13:52

7

To loop all rows in a `dataframe` and **use** values of each row **conveniently**, `namedtuples` can be converted to `ndarray` s. For example:

```
df = pd.DataFrame({'col1': [1, 2],
```

Iterating over the rows:

```
for row in df.itertuples(index=Fal
    print np.asarray(row)
```

results in:

```
[ 1.   0.1]
[ 2.   0.2]
```

Please note that if `index=True` , **the** edited Apr 24 '18 at 8:48

answered Apr 23 '18 at 14:53
KutalmisB
794 2 8 20

5

Adding to the answers above, sometimes a useful pattern is:

```
# Borrowing @KutalmisB df example
df = pd.DataFrame({'col1': [1, 2],
# The to_dict call results in a li
# where each row_dict is a diction
row
for row_dict in df.to_dict(orient=
    print(row_dict)
```

Which results in:

```
{'col1':1.0, 'col2':0.1}
{'col1':2.0, 'col2':0.2}
```

answered Jun 27 '18 at 18:48
Zach
926 5 10

2

Why complicate things?

Simple.

```
import pandas as pd
import numpy as np

# Here is an example dataframe
df_existing = pd.DataFrame(np.rand
columns=list('ABCD'))

for idx,row in df_existing.iterrow
    print row['A'],row['B'],row['C
```

answered Jul 10 '18 at 15:05
Justin Malinchak
153 7

9 How is this different than the accepted answer?? – moi Jul 30 '18 at 7:39

I guess I prefer when coder can quickly just snip the entire code block run it, and it parses fine. Accepted answer requires piecing together blocks. Timesaver – Justin Malinchak Nov 2 '18 at 18:21

Don't!

Iteration in pandas is an anti-pattern, and is something you should only want to do when you have exhausted every other option possible. You should not consider using any function with " `iter` " in its name for anything more than a few thousand rows or you will have to get used to a **lot** of waiting.

Do you want to print a DataFrame? Use `DataFrame.to_string()` .

Do you want to compute something? In that case, search for methods in this order (list modified from here):

1. vectorization
2. cython routines
3. pure python list comprehension (for loop)
4. `apply`
   a. reductions that can be performed in cython
   b. iteration in python space
5. `itertuples` , `iteritems`
6. `iterrows`

`iterrows` and `itertuples` (both receiving many votes in answers to this question) should be used in very rare circumstances, such as generating row objects/nametuples for sequential processing, which these functions are good at.

Appeal to Authority
The docs page on iteration has a huge red warning box that says:

> Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed [...].

Next Best Thing: List Comprehensions

If you are iterating because there is no vectorized solution available, use a list comprehension. To iterate over rows using a single column, use

```
result = [f(x) for x in df['col']]
```

```
# two column format
result = [f(x, y) for x, y in zip(

# many column format
result = [f(row[0], ..., row[n]) f
```

If you need an integer row index while iterating, use `enumerate` :

```
result = [f(...) for i, row in enu
```

(where `df.index[i]` gets you the index label.)

If you can turn it into a function, you can use list comprehension. You can make arbitrarily complex things work through the simplicity and speed of raw python.

edited yesterday

answered yesterday
coldspeed
141k 25 157 242

1 I agree with @coldspeed. Don't unless you absolutely must. – Scott Boston yesterday

1

You can also do `numpy` indexing for even greater speed ups. It's not really iterating but works much better than iteration for certain applications.

```
subset = row['c1'][0:5]
all = row['c1'][:]
```

You may also want to cast it to an array. These indexes/selections are supposed to act like Numpy arrays already but I ran into issues and needed to cast

```
np.asarray(all)
imgs[:] = cv2.resize(imgs[:], (224
```

edited Dec 1 '17 at 18:22

answered Dec 1 '17 at 17:49
James L.
3,218 1 17 33

1

One very simple and intuitive way is :

```
df=pd.DataFrame({'A':[1,2,3], 'B':
print(df)
for i in range(df.shape[0]):
    # For printing the second colu
    print(df.iloc[i,1])
    # For printing more than one c
    print(df.iloc[i,[0,2]])
```

answered Jan 19 at 6:53

shubham ranjan
107 4

1

This example uses iloc to isolate each digit in the data frame.

```
import pandas as pd

 a = [1, 2, 3, 4]
 b = [5, 6, 7, 8]

 mjr = pd.DataFrame({'a':a, 'b':b}

 size = mjr.shape

 for i in range(size[0]):
     for j in range(size[1]):
         print(mjr.iloc[i, j])
```

answered Mar 16 at 22:33

mjr2000
44 4

0

For both viewing and modifying values, I would use `iterrows()` . In a for loop and by using tuple unpacking (see the example: `i, row`), I use the `row` for only viewing the value and use `i` with the `loc` method when I want to modify values. As stated in previous answers, here you should not modify something you are iterating over.

```
for i, row in df.iterrows():
    if row['A'] == 'Old_Value':
        df.loc[i,'A'] = 'New_value
```

Here the `row` in the loop is a copy of that row, and not a view of it. Therefore, you should NOT write something like `row['A'] = 'New_Value'` , it will not modify the DataFrame. However, you can use `i` and `loc` and specify the DataFrame to do the work.

answered Feb 27 at 0:29

HKRC
95 7

protected by Serenity Feb 12 at 6:05

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?