How do I select rows from a DataFrame based on column values?

Asked 9 years, 5 months ago Modified 2 months ago Viewed 5.4m times

3207

How can I select rows from a DataFrame based on values in some column in Pandas?

In SQL, I would use:

```
SELECT *
FROM table
WHERE column_name = some_value
```

python pandas dataframe

Share Edit Follow

edited Sep 18 at 10:42
cottontail
4,565 18 21 38

asked Jun 12, 2013 at 17:42
szli
35k 9 30 39

16 Answers

Sorted by: Highest score (default)

5969

To select rows whose column value equals a scalar, `some_value` , use `==` :

```
df.loc[df['column_name'] == some_value]
```

To select rows whose column value is in an iterable, `some_values` , use `isin` :

```
df.loc[df['column_name'].isin(some_values)]
```

Combine multiple conditions with `&` :

```
df.loc[(df['column_name'] >= A) & (df['column_name'] <= B)]
```

Note the parentheses. Due to Python's operator precedence rules, `&` binds more tightly than `<=` and `>=` . Thus, the parentheses in the last example are necessary. Without the parentheses

```
df['column_name'] >= A & df['column_name'] <= B
```

is parsed as

```
df['column_name'] >= (A & df['column_name']) <= B
```

which results in a Truth value of a Series is ambiguous error.

To select rows whose column value *does not equal* `some_value` , use `!=` :

```
df.loc[df['column_name'] != some_value]
```

`isin` returns a boolean Series, so to select rows whose value is *not* in `some_values`, negate the boolean Series using `~`:

```
df.loc[~df['column_name'].isin(some_values)]
```

For example,

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'A': 'foo bar foo bar foo bar foo foo'.split(),
                   'B': 'one one two three two two one three'.split(),
                   'C': np.arange(8), 'D': np.arange(8) * 2})
print(df)
#      A      B  C   D
# 0  foo    one  0   0
# 1  bar    one  1   2
# 2  foo    two  2   4
# 3  bar  three  3   6
# 4  foo    two  4   8
# 5  bar    two  5  10
# 6  foo    one  6  12
# 7  foo  three  7  14

print(df.loc[df['A'] == 'foo'])
```

yields

```
     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

If you have multiple values you want to include, put them in a list (or more generally, any iterable) and use `isin`:

```
print(df.loc[df['B'].isin(['one','three'])])
```

yields

```
     A      B  C   D
0  foo    one  0   0
1  bar    one  1   2
3  bar  three  3   6
6  foo    one  6  12
7  foo  three  7  14
```

Note, however, that if you wish to do this many times, it is more efficient to make an index first, and then use `df.loc`:

```
df = df.set_index(['B'])
print(df.loc['one'])
```

yields

```
       A  C   D
B
one  foo  0   0
one  bar  1   2
one  foo  6  12
```

or, to include multiple values from the index use `df.index.isin` :

```
df.loc[df.index.isin(['one','two'])]
```

yields

```
       A  C   D
B
one  foo  0   0
one  bar  1   2
two  foo  2   4
two  foo  4   8
two  bar  5  10
one  foo  6  12
```

Share Edit Follow

edited Jan 18, 2019 at 2:47

answered Jun 12, 2013 at 17:44

unutbu
807k 174 1735
1633

728

+500

There are several ways to select rows from a Pandas dataframe:

1. **Boolean indexing (`df[df['col'] == value`])**
2. **Positional indexing (`df.iloc[...]`)**
3. **Label indexing (`df.xs(...)`)**
4. **`df.query(...)` API**

Below I show you examples of each, with advice when to use certain techniques. Assume our criterion is column `'A'` == `'foo'`

(Note on performance: For each base type, we can keep things simple by using the Pandas API or we can venture outside the API, usually into NumPy, and speed things up.)

Setup

The first thing we'll need is to identify a condition that will act as our criterion for selecting rows. We'll start with the OP's case `column_name == some_value` , and include some other common use cases.

Borrowing from @unutbu:

```
import pandas as pd, numpy as np

df = pd.DataFrame({'A': 'foo bar foo bar foo bar foo foo'.split(),
                   'B': 'one one two three two two one three'.split(),
                   'C': np.arange(8), 'D': np.arange(8) * 2})
```

1. Boolean indexing

... Boolean indexing requires finding the true value of each row's `'A'` column being equal to `'foo'`, then using those truth values to identify which rows to keep. Typically, we'd name this series, an array of truth values, `mask`. We'll do so here as well.

```
mask = df['A'] == 'foo'
```

We can then use this mask to slice or index the data frame

```
df[mask]

     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

This is one of the simplest ways to accomplish this task and if performance or intuitiveness isn't an issue, this should be your chosen method. However, if performance is a concern, then you might want to consider an alternative way of creating the `mask`.

2. Positional indexing

Positional indexing (`df.iloc[...]`) has its use cases, but this isn't one of them. In order to identify where to slice, we first need to perform the same boolean analysis we did above. This leaves us performing one extra step to accomplish the same task.

```
mask = df['A'] == 'foo'
pos = np.flatnonzero(mask)
df.iloc[pos]

     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

3. Label indexing

Label indexing can be very handy, but in this case, we are again doing more work for no benefit

```
df.set_index('A', append=True, drop=False).xs('foo', level=1)

     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

4. `df.query()` API

`pd.DataFrame.query` is a very elegant/intuitive way to perform this task, but is often slower. **However**, if you pay attention to the timings below, for large data, the query is very efficient. More so than the standard approach and of similar magnitude as my best suggestion.

```
df.query('A == "foo"')

     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

My preference is to use the `Boolean` `mask`

Actual improvements can be made by modifying how we create our `Boolean` `mask`.

`mask` alternative 1 *Use the underlying NumPy array and forgo the overhead of creating another `pd.Series`*

```
mask = df['A'].values == 'foo'
```

I'll show more complete time tests at the end, but just take a look at the performance gains we get using the sample data frame. First, we look at the difference in creating the `mask`

```
%timeit mask = df['A'].values == 'foo'
%timeit mask = df['A'] == 'foo'

5.84 µs ± 195 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
166 µs ± 4.45 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Evaluating the `mask` with the NumPy array is ~ 30 times faster. This is partly due to NumPy evaluation often being faster. It is also partly due to the lack of overhead necessary to build an index and a corresponding `pd.Series` object.

Next, we'll look at the timing for slicing with one `mask` versus the other.

```
mask = df['A'].values == 'foo'
%timeit df[mask]
mask = df['A'] == 'foo'
%timeit df[mask]

219 µs ± 12.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
239 µs ± 7.03 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The performance gains aren't as pronounced. We'll see if this holds up over more robust testing.

`mask` alternative 2 We could have reconstructed the data frame as well. There is a big caveat when reconstructing a dataframe—you must take care of the `dtypes` when doing so!

Instead of `df[mask]` we will do this

```
pd.DataFrame(df.values[mask], df.index[mask], df.columns).astype(df.dtypes)
```

If the data frame is of mixed type, which our example is, then when we get `df.values` the resulting array is of `dtype object` and consequently, all columns of the new data frame will be of `dtype object` . Thus requiring the `astype(df.dtypes)` and killing any potential performance gains.

```
%timeit df[m]
%timeit pd.DataFrame(df.values[mask], df.index[mask],
df.columns).astype(df.dtypes)

216 µs ± 10.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1.43 ms ± 39.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

However, if the data frame is not of mixed type, this is a very useful way to do it.

Given

```
np.random.seed([3,1415])
d1 = pd.DataFrame(np.random.randint(10, size=(10, 5)), columns=list('ABCDE'))

d1

   A  B  C  D  E
0  0  2  7  3  8
1  7  0  6  8  6
2  0  2  0  4  9
3  7  3  2  4  3
4  3  6  7  7  4
5  5  3  7  5  9
6  8  7  6  4  7
7  6  2  6  6  5
8  2  8  7  5  8
9  4  7  6  1  5
```

```
%%timeit
mask = d1['A'].values == 7
d1[mask]

179 µs ± 8.73 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Versus

```
%%timeit
mask = d1['A'].values == 7
pd.DataFrame(d1.values[mask], d1.index[mask], d1.columns)

87 µs ± 5.12 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

We cut the time in half.

`mask` alternative 3

@unutbu also shows us how to use `pd.Series.isin` to account for each element of `df['A']` being in a set of values. This evaluates to the same thing if our set of values is a set of one value, namely `'foo'` . But it also generalizes to include larger sets of values if needed. Turns out, this is still pretty fast even though it is a more general solution. The only real loss is in intuitiveness for those not familiar with the concept.

```
mask = df['A'].isin(['foo'])
df[mask]

     A      B  C   D
```

```
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

However, as before, we can utilize NumPy to improve performance while sacrificing virtually nothing. We'll use `np.in1d`

```
mask = np.in1d(df['A'].values, ['foo'])
df[mask]

     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

Timing

I'll include other concepts mentioned in other posts as well for reference.

Code Below

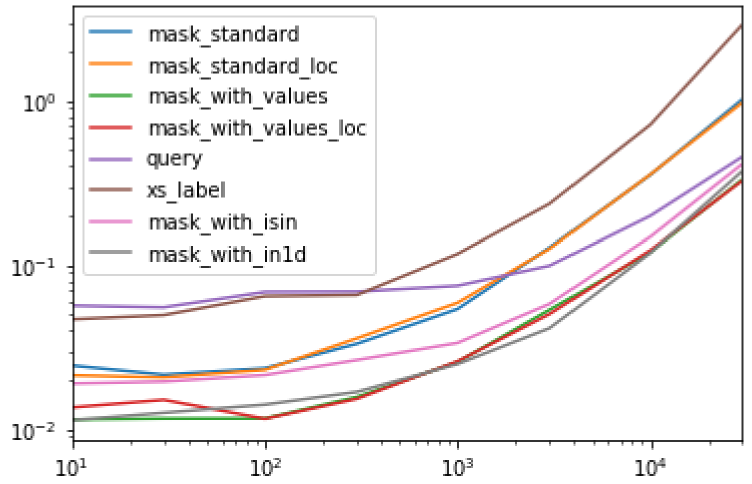
Each *column* in this table represents a different length data frame over which we test each function. Each column shows relative time taken, with the fastest function given a base index of `1.0`.

```
res.div(res.min())

                         10         30        100       300        1000      3000
10000     30000
mask_standard         2.156872  1.850663  2.034149  2.166312  2.164541
3.090372  2.981326  3.131151
mask_standard_loc     1.879035  1.782366  1.988823  2.338112  2.361391
3.036131  2.998112  2.990103
mask_with_values      1.010166  1.000000  1.005113  1.026363  1.028698
1.293741  1.007824  1.016919
mask_with_values_loc  1.196843  1.300228  1.000000  1.000000  1.038989
1.219233  1.037020  1.000000
query                 4.997304  4.765554  5.934096  4.500559  2.997924
2.397013  1.680447  1.398190
xs_label              4.124597  4.272363  5.596152  4.295331  4.676591
5.710680  6.032809  8.950255
mask_with_isin        1.674055  1.679935  1.847972  1.724183  1.345111
1.405231  1.253554  1.264760
mask_with_in1d        1.000000  1.083807  1.220493  1.101929  1.000000
1.000000  1.000000  1.144175
```

You'll notice that the fastest times seem to be shared between `mask_with_values` and `mask_with_in1d`.

```
res.T.plot(loglog=True)
```



Functions

```
def mask_standard(df):
    mask = df['A'] == 'foo'
    return df[mask]

def mask_standard_loc(df):
    mask = df['A'] == 'foo'
    return df.loc[mask]

def mask_with_values(df):
    mask = df['A'].values == 'foo'
    return df[mask]

def mask_with_values_loc(df):
    mask = df['A'].values == 'foo'
    return df.loc[mask]

def query(df):
    return df.query('A == "foo"')

def xs_label(df):
    return df.set_index('A', append=True, drop=False).xs('foo', level=-1)

def mask_with_isin(df):
    mask = df['A'].isin(['foo'])
    return df[mask]

def mask_with_in1d(df):
    mask = np.in1d(df['A'].values, ['foo'])
    return df[mask]
```

Testing

```
res = pd.DataFrame(
    index=[
        'mask_standard', 'mask_standard_loc', 'mask_with_values',
'mask_with_values_loc',
        'query', 'xs_label', 'mask_with_isin', 'mask_with_in1d'
    ],
    columns=[10, 30, 100, 300, 1000, 3000, 10000, 30000],
    dtype=float
)

for j in res.columns:
    d = pd.concat([df] * j, ignore_index=True)
    for i in res.index:a
        stmt = '{}(d)'.format(i)
        setp = 'from __main__ import d, {}'.format(i)
        res.at[i, j] = timeit(stmt, setp, number=50)
```

Special Timing

Looking at the special case when we have a single non-object `dtype` for the entire data frame.

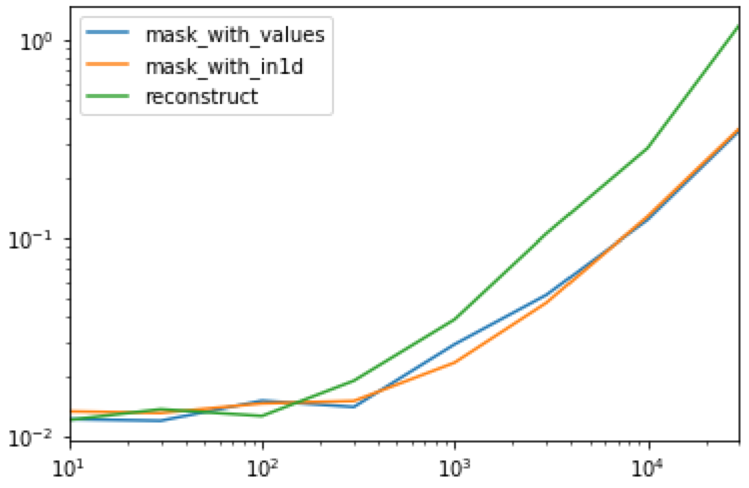
Code Below

```
spec.div(spec.min())
```

```
                        10        30       100       300      1000      3000
10000     30000
mask_with_values  1.009030  1.000000  1.194276  1.000000  1.236892  1.095343
1.000000  1.000000
mask_with_in1d    1.104638  1.094524  1.156930  1.072094  1.000000  1.000000
1.040043  1.027100
reconstruct       1.000000  1.142838  1.000000  1.355440  1.650270  2.222181
2.294913  3.406735
```

Turns out, reconstruction isn't worth it past a few hundred rows.

```
spec.T.plot(loglog=True)
```



Functions

```
np.random.seed([3,1415])
d1 = pd.DataFrame(np.random.randint(10, size=(10, 5)), columns=list('ABCDE'))

def mask_with_values(df):
    mask = df['A'].values == 'foo'
    return df[mask]

def mask_with_in1d(df):
    mask = np.in1d(df['A'].values, ['foo'])
    return df[mask]

def reconstruct(df):
    v = df.values
    mask = np.in1d(df['A'].values, ['foo'])
    return pd.DataFrame(v[mask], df.index[mask], df.columns)

spec = pd.DataFrame(
    index=['mask_with_values', 'mask_with_in1d', 'reconstruct'],
    columns=[10, 30, 100, 300, 1000, 3000, 10000, 30000],
        dtype=float
    )
```

Testing

```
for j in spec.columns:
    d = pd.concat([df] * j, ignore_index=True)
    for i in spec.index:
        stmt = '{}(d)'.format(i)
        setp = 'from __main__ import d, {}'.format(i)
        spec.at[i, j] = timeit(stmt, setp, number=50)
```

Share Edit Follow

edited Feb 4, 2021 at 16:52
Trenton McKinney
49.6k 31 120 134

answered Sep 11, 2017 at 22:14
piRSquared
275k 54 453 595

339

tl;dr

The Pandas equivalent to

```
select * from table where column_name = some_value
```

is

```
table[table.column_name == some_value]
```

Multiple conditions:

```
table[(table.column_name == some_value) | (table.column_name2 == some_value2)]
```

or

```
table.query('column_name == some_value | column_name2 == some_value2')
```

Code example

```
import pandas as pd

# Create data set
d = {'foo':[100, 111, 222],
     'bar':[333, 444, 555]}
df = pd.DataFrame(d)

# Full dataframe:
df

# Shows:
#    bar   foo
# 0  333   100
# 1  444   111
# 2  555   222

# Output only the row(s) in df where foo is 222:
df[df.foo == 222]
```

```
# Shows:
#    bar  foo
# 2  555  222
```

In the above code it is the line `df[df.foo == 222]` that gives the rows based on the column value, `222` in this case.

Multiple conditions are also possible:

```
df[(df.foo == 222) | (df.bar == 444)]
#    bar  foo
# 1  444  111
# 2  555  222
```

But at that point I would recommend using the query function, since it's less verbose and yields the same result:

```
df.query('foo == 222 | bar == 444')
```

Share Edit Follow

edited Oct 5, 2020 at 18:26
Peter Mortensen
30.6k 21 104 125

answered Jul 8, 2015 at 15:17
imolit
7,202 3 25 28

87

I find the syntax of the previous answers to be redundant and difficult to remember. Pandas introduced the `query()` method in v0.13 and I much prefer it. For your question, you could do `df.query('col == val')` .

Reproduced from *The query() Method (Experimental)*:

```
In [167]: n = 10

In [168]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [169]: df
Out[169]:
          a         b         c
0  0.687704  0.582314  0.281645
1  0.250846  0.610021  0.420121
2  0.624328  0.401816  0.932146
3  0.011763  0.022921  0.244186
4  0.590198  0.325680  0.890392
5  0.598892  0.296424  0.007312
6  0.634625  0.803069  0.123872
7  0.924168  0.325076  0.303746
8  0.116822  0.364564  0.454607
9  0.986142  0.751953  0.561512

# pure python
In [170]: df[(df.a < df.b) & (df.b < df.c)]
Out[170]:
          a         b         c
3  0.011763  0.022921  0.244186
8  0.116822  0.364564  0.454607

# query
In [171]: df.query('(a < b) & (b < c)')
Out[171]:
          a         b         c
3  0.011763  0.022921  0.244186
8  0.116822  0.364564  0.454607
```

You can also access variables in the environment by prepending an `@`.

```
exclude = ('red', 'orange')
df.query('color not in @exclude')
```

Share Edit Follow

edited Sep 16 at 21:55
Peter Mortensen
30.6k 21 104 125

answered Feb 9, 2016 at 1:36
fredcallaway
1,352 11 7

68

More flexibility using `.query` with pandas >= 0.25.0:

Since pandas >= 0.25.0 we can use the `query` method to filter dataframes with pandas methods and even column names which have spaces. Normally the spaces in column names would give an error, but now we can solve that using a backtick (`) - see GitHub:

```
# Example dataframe
df = pd.DataFrame({'Sender email':['ex@example.com', "reply@shop.com",
"buy@shop.com"]})

     Sender email
0  ex@example.com
1  reply@shop.com
2    buy@shop.com
```

Using `.query` with method `str.endswith`:

```
df.query('`Sender email`.str.endswith("@shop.com")')
```

Output

```
     Sender email
1  reply@shop.com
2    buy@shop.com
```

Also we can use local variables by prefixing it with an `@` in our query:

```
domain = 'shop.com'
df.query('`Sender email`.str.endswith(@domain)')
```

Output

```
     Sender email
1  reply@shop.com
2    buy@shop.com
```

Share Edit Follow

edited Mar 28 at 11:50

answered Aug 3, 2019 at 12:05
Erfan
39k 8 58 73

For selecting only specific columns out of multiple columns for a given value in Pandas:

39

```
select col_name1, col_name2 from table where column_name = some_value.
```

Options `loc` :

```
df.loc[df['column_name'] == some_value, [col_name1, col_name2]]
```

or `query` :

```
df.query('column_name == some_value')[[col_name1, col_name2]]
```

Share Edit Follow

edited Sep 5, 2021 at 8:26
Henry Ecker
33k 18 32 52

answered Dec 7, 2017 at 10:39
SP001
519 5 5

34

In newer versions of Pandas, inspired by the documentation (*Viewing data*):

```
df[df["colume_name"] == some_value] #Scalar, True/False..

df[df["colume_name"] == "some_value"] #String
```

Combine multiple conditions by putting the clause in parentheses, `()` , and combining them with `&` and `|` (and/or). Like this:

```
df[(df["colume_name"] == "some_value1") & (pd[pd["colume_name"] ==
"some_value2"])]
```

Other filters

```
pandas.notna(df["colume_name"]) == True # Not NaN
df['colume_name'].str.contains("text") # Search for "text"
df['colume_name'].str.lower().str.contains("text") # Search for "text", after
converting  to lowercase
```

Share Edit Follow

edited Feb 8, 2021 at 15:58
Peter Mortensen
30.6k 21 104 125

answered Jan 5, 2021 at 11:43
Punnerud
6,272 2 47 43

32

Faster results can be achieved using numpy.where.

For example, with unubtu's setup -

```
In [76]: df.iloc[np.where(df.A.values=='foo')]
Out[76]:
     A      B  C   D
0  foo    one  0   0
2  foo    two  2   4
4  foo    two  4   8
6  foo    one  6  12
7  foo  three  7  14
```

Timing comparisons:

```
In [68]: %timeit df.iloc[np.where(df.A.values=='foo')]  # fastest
1000 loops, best of 3: 380 µs per loop

In [69]: %timeit df.loc[df['A'] == 'foo']
1000 loops, best of 3: 745 µs per loop

In [71]: %timeit df.loc[df['A'].isin(['foo'])]
1000 loops, best of 3: 562 µs per loop

In [72]: %timeit df[df.A=='foo']
1000 loops, best of 3: 796 µs per loop

In [74]: %timeit df.query('(A=="foo")')  # slowest
1000 loops, best of 3: 1.71 ms per loop
```

Share Edit Follow

edited Oct 3, 2017 at 16:17
Brian Burns
19.1k 8 80 72

answered Jul 5, 2017 at 16:34
shivsn
7,270 24 33

29

Here is a simple example

```
from pandas import DataFrame

# Create data set
d = {'Revenue':[100,111,222],
     'Cost':[333,444,555]}
df = DataFrame(d)


# mask = Return True when the value in column "Revenue" is equal to 111
mask = df['Revenue'] == 111

print mask

# Result:
# 0    False
# 1     True
# 2    False
# Name: Revenue, dtype: bool


# Select * FROM df WHERE Revenue = 111
df[mask]

# Result:
#    Cost    Revenue
# 1  444     111
```

Share Edit Follow

answered Jun 13, 2013 at 11:49
DataByDavid
1,019 2 13 20

19

To add: You can also do `df.groupby('column_name').get_group('column_desired_value').reset_index()` to make a new data frame with specified column having a particular value. E.g.,

```
import pandas as pd
df = pd.DataFrame({'A': 'foo bar foo bar foo bar foo foo'.split(),
                   'B': 'one one two three two two one three'.split()})
print("Original dataframe:")
print(df)
```

```
b_is_two_dataframe =
pd.DataFrame(df.groupby('B').get_group('two').reset_index()).drop('index', axis
= 1)
#NOTE: the final drop is to remove the extra index column returned by groupby
object
print('Sub dataframe where B is two:')
print(b_is_two_dataframe)
```

Running this gives:

```
Original dataframe:
     A      B
0  foo    one
1  bar    one
2  foo    two
3  bar  three
4  foo    two
5  bar    two
6  foo    one
7  foo  three
Sub dataframe where B is two:
     A    B
0  foo  two
1  foo  two
2  bar  two
```

Share Edit Follow

edited Sep 16 at 22:03
Peter Mortensen
30.6k 21 104 125

answered Nov 18, 2016 at 12:10
TuanDT
1,655 12 26

11

You can also use .apply:

```
df.apply(lambda row: row[df['B'].isin(['one','three'])])
```

It actually works row-wise (i.e., applies the function to each row).

The output is

```
   A      B  C   D
0  foo    one  0   0
1  bar    one  1   2
3  bar  three  3   6
6  foo    one  6  12
7  foo  three  7  14
```

The results is the same as using as mentioned by @unutbu

```
df[[df['B'].isin(['one','three'])]]
```

Share Edit Follow

answered Dec 7, 2018 at 17:38
vahid
325 5 13

If you want to make query to your dataframe repeatedly and speed is important to you, the best thing is to convert your dataframe to dictionary and then

5

by doing this you can make query thousands of times faster.

```
my_df = df.set_index(column_name)
my_dict = my_df.to_dict('index')
```

After make my_dict dictionary you can go through:

```
if some_value in my_dict.keys():
   my_result = my_dict[some_value]
```

If you have duplicated values in column_name you can't make a dictionary. but you can use:

```
my_result = my_df.loc[some_value]
```

Share Edit Follow

edited Nov 27, 2021 at 17:09
marc_s
717k 172 1319 1436

answered Nov 26, 2021 at 6:41
lord-hasan
83 1 6

3

SQL statements on DataFrames to select rows using DuckDB

With DuckDB we can query pandas DataFrames with SQL statements, in a highly performant way.

Since the question is *How do I select rows from a DataFrame based on column values?*, and the example in the question is a SQL query, this answer looks logical in this topic.

Example:

```
In [1]: import duckdb

In [2]: import pandas as pd

In [3]: con = duckdb.connect()

In [4]: df = pd.DataFrame({"A": range(11), "B": range(11, 22)})

In [5]: df
Out[5]:
     A   B
0    0  11
1    1  12
2    2  13
3    3  14
4    4  15
5    5  16
6    6  17
7    7  18
8    8  19
9    9  20
10  10  21

In [6]: results = con.execute("SELECT * FROM df where A > 2").df()

In [7]: results
Out[7]:
    A   B
0   3  14
1   4  15
```

```
2   5  16
3   6  17
4   7  18
5   8  19
6   9  20
7  10  21
```

Share Edit Follow

edited Sep 16 at 22:02
Peter Mortensen
30.6k 21 104 125

answered Apr 21 at 9:42
Erfan
39k 8 58 73

2

You can use `loc` (square brackets) with a function:

```
# Series
s = pd.Series([1, 2, 3, 4])
s.loc[lambda x: x > 1]
# s[lambda x: x > 1]
```

Output:

```
1    2
2    3
3    4
dtype: int64
```

or

```
# DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [10, 20, 30]})
df.loc[lambda x: x['A'] > 1]
# df[lambda x: x['A'] > 1]
```

Output:

```
   A   B
1  2  20
2  3  30
```

The advantage of this method is that you can chain selection with previous operations. For example:

```
df.mul(2).loc[lambda x: x['A'] > 3, 'B']
# (df * 2).loc[lambda x: x['A'] > 3, 'B']
```

vs

```
df_temp = df * 2
df_temp.loc[df_temp['A'] > 3, 'B']
```

Output:

```
1    40
2    60
Name: B, dtype: int64
```

Share Edit Follow

edited Aug 2 at 20:59

answered Aug 2 at 20:42

Mykola Zotko
13.6k 3 49 62

2

1. Install `numexpr` to speed up `query()` calls

The pandas documentation recommends installing numexpr to speed up numeric calculation when using `query()`. Use `pip install numexpr` (or `conda`, `sudo` etc. depending on your environment) to install it.

For larger dataframes (where performance actually matters), `df.query()` with `numexpr` engine performs much faster than `df[mask]`. In particular, it performs better for the following cases.

Logical and/or comparison operators on columns of strings

If a column of strings are compared to some other string(s) and matching rows are to be selected, even for a single comparison operation, `query()` performs faster than `df[mask]`. For example, for a dataframe with 80k rows, it's 30% faster[1] and for a dataframe with 800k rows, it's 60% faster.[2]

```
df[df.A == 'foo']
df.query("A == 'foo'")  # <--- performs 30%-60% faster
```

This gap increases as the number of operations increases (if 4 comparisons are chained `df.query()` is 2-2.3 times faster than `df[mask]`)[1,2] and/or the dataframe length increases.[2]

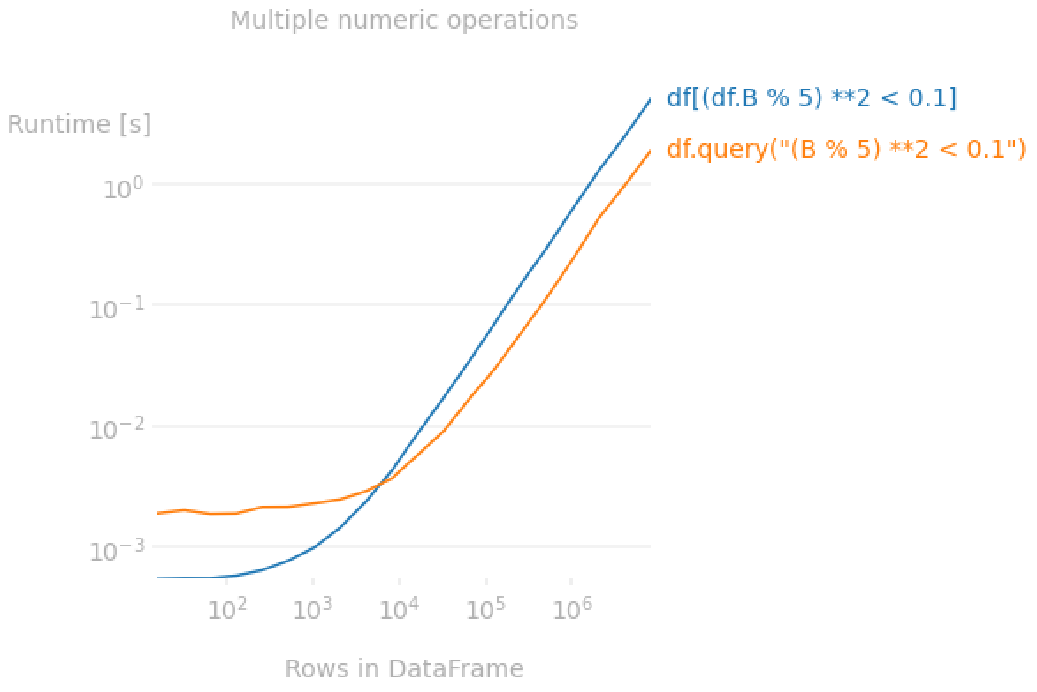
Multiple operations on numeric columns

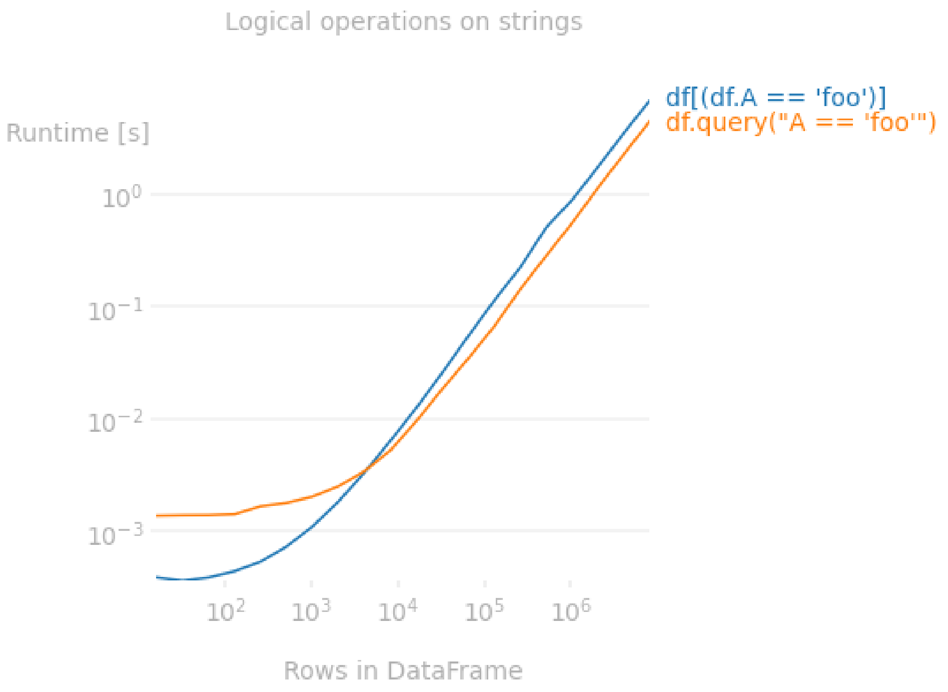
If multiple arithmetic, logical or comparison operations need to be computed to create a boolean mask to filter `df`, `query()` performs faster. For example, for a frame with 80k rows, it's 20% faster[1] and for a frame with 800k rows, it's 2 times faster.[2]

```
df[(df.B % 5) **2 < 0.1]
df.query("(B % 5) **2 < 0.1")  # <--- performs 20%-100% faster.
```

This gap in performance increases as the number of operations increases and/or the dataframe length increases.[2]

The following plot shows how the methods perform as the dataframe length increases.[3]





2. Access `.values` to call pandas methods inside `query()`

`Numexpr` [currently supports](#) only logical (`&` , `|` , `~`), comparison (`==` , `>` , `<` , `>=` , `<=` , `!=`) and basic arithmetic operators (`+` , `-` , `*` , `/` , `**` , `%`).

For example, it doesn't support integer division (`//`). However, calling the equivalent pandas method (`floordiv()`) and accessing the `values` attribute on the resulting Series makes `numexpr` evaluate its underlying numpy array and `query()` works. Or setting `engine` parameter to `'python'` also works.

```
df.query('B.floordiv(2).values <= 3')  # or
df.query('B.floordiv(2).le(3).values') # or
df.query('B.floordiv(2).le(3)', engine='python')
```

The same applies for Erfan's suggested method calls as well. The code in their answer spits TypeError as is (as of Pandas 1.3.4) for `numexpr` engine but accessing `.values` attribute makes it work.

```
df.query('`Sender email`.str.endswith("@shop.com")')         # <--- TypeError
df.query('`Sender email`.str.endswith("@shop.com").values')  # OK
```

[1]: Benchmark code using a frame with 80k rows

```
import numpy as np
df = pd.DataFrame({'A': 'foo bar foo baz foo bar foo foo'.split()*10000,
                   'B': np.random.rand(80000)})

%timeit df[df.A == 'foo']
# 8.5 ms ± 104.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
%timeit df.query("A == 'foo'")
# 6.36 ms ± 95.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

%timeit df[((df.A == 'foo') & (df.A != 'bar')) | ((df.A != 'baz') & (df.A !=
'buz'))]
# 29 ms ± 554 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
%timeit df.query("A == 'foo' & A != 'bar' | A != 'baz' & A != 'buz'")
# 16 ms ± 339 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)

%timeit df[(df.B % 5) **2 < 0.1]
# 5.35 ms ± 37.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit df.query("(B % 5) **2 < 0.1")
# 4.37 ms ± 46.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[2]: Benchmark code using a frame with 800k rows

```
df = pd.DataFrame({'A': 'foo bar foo baz foo bar foo foo'.split()*100000,
                   'B': np.random.rand(800000)})

%timeit df[df.A == 'foo']
# 87.9 ms ± 873 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
%timeit df.query("A == 'foo'")
# 54.4 ms ± 726 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)

%timeit df[((df.A == 'foo') & (df.A != 'bar')) | ((df.A != 'baz') & (df.A !=
'buz'))]
# 310 ms ± 3.4 ms per loop (mean ± std. dev. of 10 runs, 100 loops each)
%timeit df.query("A == 'foo' & A != 'bar' | A != 'baz' & A != 'buz'")
# 132 ms ± 2.43 ms per loop (mean ± std. dev. of 10 runs, 100 loops each)

%timeit df[(df.B % 5) **2 < 0.1]
# 54 ms ± 488 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
%timeit df.query("(B % 5) **2 < 0.1")
# 26.3 ms ± 320 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
```

[3]: Code used to produce the performance graphs of the two methods for strings and numbers.

```
from perfplot import plot
constructor = lambda n: pd.DataFrame({'A': 'foo bar foo baz foo bar foo
foo'.split()*n, 'B': np.random.rand(8*n)})
plot(
    setup=constructor,
    kernels=[lambda df: df[(df.B%5)**2<0.1], lambda df: df.query("
(B%5)**2<0.1")],
    labels= ['df[(df.B % 5) **2 < 0.1]', 'df.query("(B % 5) **2 < 0.1")'],
    n_range=[2**k for k in range(4, 24)],
    xlabel='Rows in DataFrame',
    title='Multiple mathematical operations on numbers',
    equality_check=pd.DataFrame.equals);
plot(
    setup=constructor,
    kernels=[lambda df: df[df.A == 'foo'], lambda df: df.query("A == 'foo'")],
    labels= ["df[df.A == 'foo']", """df.query("A == 'foo'")"""],
    n_range=[2**k for k in range(4, 24)],
    xlabel='Rows in DataFrame',
    title='Comparison operation on strings',
    equality_check=pd.DataFrame.equals);
```

Share Edit Follow

answered Sep 18 at 10:36
cottontail
4,565 18 21 38

1

Great answers. Only, when the **size of the dataframe approaches million rows**, many of the methods tend to take ages when using `df[df['col']==val]`. I wanted to have all possible values of "another_column" that correspond to specific values in "some_column" (in this case in a dictionary). This worked and fast.

```
s=datetime.datetime.now()

my_dict={}

for i, my_key in enumerate(df['some_column'].values):
    if i%100==0:
        print(i)  # to see the progress
    if my_key not in my_dict.keys():
        my_dict[my_key]={}
        my_dict[my_key]['values']=[df.iloc[i]['another_column']]
    else:
        my_dict[my_key]['values'].append(df.iloc[i]['another_column'])

e=datetime.datetime.now()

print('operation took '+str(e-s)+' seconds')```
```

Share Edit Follow

answered Feb 16 at 21:13

L. Astola

37 4