

Selecting multiple columns in a pandas dataframe



I have data in different columns but I don't know how to extract it to save it in another variable.

778



index	a	b	c
1	2	3	4
2	3	4	5



How do I select 'a' , 'b' and save it in to df1?

225

I tried

```
df1 = df['a':'b']
df1 = df.ix[:, 'a':'b']
```

None seem to work.

python

pandas

dataframe

select

edited Dec 2 '18 at 6:06



cs95

153k

26

202

272

asked Jul 1 '12 at 21:03



user1234440

5,908

12

42

75

2 You never want to use `.ix` as it's ambiguous. Use `.iloc` or `.loc` if you must. – [A-B-B](#) Jul 12 '17 at 17:14

1 Is there a way it can be done without referring to the header names? like in R, I can do it like this: `> csvtable_imp_1 <- csvtable_imp[0:6]` and it selects the delta amount of the first columns between 0 and 6. All I had to do is to read the csv-table as delimited with the readr lib. – [MichaelR](#) Oct 19 '18 at 0:30

I've worked a bit more with it. Found something that worked as wanted. Default is to select numbers of char and not columns. `infile_1 = largefile_stay.ix[:,0:6]` – [MichaelR](#) Oct 19 '18 at 0:43

1 For those stumbling on this late, `ix` is now deprecated. Pandas recommends using either: `loc` (label-based indexing) or `iloc` (positional based indexing). – [ZaydH](#) Dec 4 '18 at 16:20

[Pandas: Replacement for .ix](#) – [Connor](#) Mar 24 at 20:18

17 Answers



The column names (which are strings) cannot be sliced in the manner you tried.

1292



Here you have a couple of options. If you know from context which variables you want to slice out, you can just return a view of only those columns by passing a list into the `__getitem__` syntax (the `[]`'s).



```
df1 = df[['a','b']]
```

Alternatively, if it matters to index them numerically and not by their name (say your code should automatically do this without knowing the names of the first two columns) then you can do this instead:

```
df1 = df.iloc[:,0:2] # Remember that Python does not slice inclusive of the
ending index.
```

Additionally, you should familiarize yourself with the idea of a view into a Pandas object vs. a copy of that object. The first of the above methods will return a new copy in memory of the desired sub-object (the desired slices).

Sometimes, however, there are indexing conventions in Pandas that don't do this and instead give you a new variable that just refers to the same chunk of memory as the sub-object or slice in the original object. This will happen with the second way of indexing, so you can modify it with the `copy()` function to get a regular copy. When this happens, changing what you think is the sliced object can sometimes alter the original object. Always good to be on the look out for this.

```
df1 = df.iloc[0,0:2].copy() # To avoid the case where changing df1 also changes
df
```

To use `iloc`, you need to know the column positions (or indices). As the column positions may change, instead of hard-coding indices, you can use `iloc` along with `get_loc` function of `columns` method of dataframe object to obtain column indices.

```
{df.columns.get_loc(c):c for idx, c in enumerate(df.columns)}
```

Now you can use this dictionary to access columns through names and using `iloc`.

edited Mar 29 at 8:21



Harshit

153 1 3

answered Jul 2 '12 at 2:43



ely

41.4k 24 104 171

144 Note: `df[['a', 'b']]` produces a copy – [Wes McKinney](#) Jul 8 '12 at 17:54

1 Yes this was implicit in my answer. The bit about the copy was only for use of `ix[]` if you *prefer* to use `ix[]` for any reason. – [ely](#) Jul 8 '12 at 18:09

1 `ix` indexes rows, not columns. I thought the OP wanted columns. – [hobs](#) Oct 31 '12 at 18:58

9 `ix` accepts slice arguments, so you can also get columns. For example, `df.ix[0:2, 0:2]` gets the upper left 2x2 sub-array just like it does for a NumPy matrix (depending on your column names of course). You can even use the slice syntax on string names of the columns, like `df.ix[0, 'Col1':'Col5']`. That gets all columns that happen to be ordered between `Col1` and `Col5` in the `df.columns` array. It is incorrect to say that `ix` indexes rows. That is just its most basic use. It also supports much more indexing than that. So, `ix` is perfectly general for this question. – [ely](#) Oct 31 '12 at 19:02 ✎

7 @AndrewCassidy Never use `.ix` again. If you want to slice with integers use `.iloc` which is exclusive of the last position just like Python lists. – [Ted Petrou](#) Jul 1 '17 at 13:55

Assuming your column names (`df.columns`) are `['index', 'a', 'b', 'c']`, then the data you want is in the 3rd & 4th columns. If you don't know their names when your script runs, you can do this

92

```
newdf = df[df.columns[2:4]] # Remember, Python is 0-offset! The "3rd" entry is
                             at slot 2.
```

As EMS points out in [his answer](#), `df.ix` slices columns a bit more concisely, but the `.columns` slicing interface might be more natural because it uses the vanilla 1-D python list indexing/slicing syntax.

WARN: `'index'` is a bad name for a `DataFrame` column. That same label is also used for the real `df.index` attribute, a `Index` array. So your column is returned by `df['index']` and the real `DataFrame` index is returned by `df.index`. An `Index` is a special kind of `Series` optimized for lookup of it's elements' values. For `df.index` it's for looking up rows by their label. That `df.columns` attribute is also a `pd.Index` array, for looking up columns by their labels.

edited May 23 '17 at 12:10



Community ♦

1 1

answered Oct 31 '12 at 18:57



hobs

11.2k 6 65 73

- 3 As I noted in my comment above, `.ix` is *not* just for rows. It is for general purpose slicing, and can be used for multidimensional slicing. It is basically just an interface to NumPy's usual `__getitem__` syntax. That said, you can easily convert a column-slicing problem into a row-slicing problem by just applying a transpose operation, `df.T`. Your example uses `columns[1:3]`, which is a little misleading. The result of `columns` is a `Series`; be careful not to just treat it like an array. Also, you should probably change it to be `columns[2:3]` to match up with your "3rd & 4th" comment. – [ely](#) Oct 31 '12 at 19:11 ✎

@Mr.F: My `[2:4]` is correct. Your `[2:3]` is wrong. And using standard python slicing notation to generate a sequence/Series is not misleading IMO. But I like your bypass of the `DataFrame` interface to access the underlying numpy array with `ix`. – [hobs](#) Feb 4 '16 at 17:26

You are correct in this case, but the point I was trying to make is that in general, [slicing with labels in Pandas is inclusive of the slice endpoint](#) (or at least this was the behavior in most previous Pandas versions). So if you retrieve `df.columns` and want to slice it *by label*, then you'd have different slice semantics than if you slice it *by integer index position*. I definitely did not explain it well in my previous comment though. – [ely](#) Feb 4 '16 at 18:05

Ahh, now I see your point. I forgot that `columns` is an immutable Series and the getter has been overridden to use labels as indices. Thanks for taking the time to clarify. – [hobs](#) Feb 5 '16 at 0:17

- 2 Note the Deprecation Warning: `.ix` is deprecated. Therefore this makes sense: `newdf = df[df.columns[2:4]]` – [Martien Lubberink](#) Jul 1 '17 at 23:57

As of version 0.11.0, columns *can be* sliced in the manner you tried using the `.loc` indexer:

89

```
df.loc[:, 'C':'E']
```

is equivalent of

```
df[['C', 'D', 'E']] # or df.loc[:, ['C', 'D', 'E']]
```

and returns columns C through E.

A demo on a randomly generated DataFrame:

```
import pandas as pd
import numpy as np
np.random.seed(5)
df = pd.DataFrame(np.random.randint(100, size=(100, 6)),
                  columns=list('ABCDEF'),
                  index=['R{}'.format(i) for i in range(100)])
df.head()
```

Out:

	A	B	C	D	E	F
R0	99	78	61	16	73	8
R1	62	27	30	80	7	76
R2	15	53	80	27	44	77
R3	75	65	47	30	84	86
R4	18	9	41	62	1	82

To get the columns from C to E (note that unlike integer slicing, 'E' is included in the columns):

```
df.loc[:, 'C':'E']
```

Out:

	C	D	E
R0	61	16	73
R1	30	80	7
R2	80	27	44
R3	47	30	84
R4	41	62	1
R5	5	58	0
...			

Same works for selecting rows based on labels. Get the rows 'R6' to 'R10' from those columns:

```
df.loc['R6':'R10', 'C':'E']
```

Out:

	C	D	E
R6	51	27	31
R7	83	19	18
R8	11	67	65
R9	78	27	29
R10	7	16	94

`.loc` also accepts a boolean array so you can select the columns whose corresponding entry in the array is `True`. For example, `df.columns.isin(list('BCD'))` returns `array([False, True, True, True, False, False], dtype=bool)` - `True` if the column name is in the list `['B', 'C', 'D']`; `False`, otherwise.

```
df.loc[:, df.columns.isin(list('BCD'))]
```

Out:

	B	C	D
R0	78	61	16
R1	27	30	80
R2	53	80	27
R3	65	47	30
R4	9	41	62
R5	78	5	58
...			

edited Jan 25 at 11:12

answered Apr 30 '16 at 12:39



ayhan

39.6k

6

81

117



57

**In** [39]: df**Out** [39]:

	index	a	b	c
0	1	2	3	4
1	2	3	4	5

In [40]: df1 = df[['b', 'c']]**In** [41]: df1**Out** [41]:

	b	c
0	3	4
1	4	5

answered Jul 8 '12 at 17:55



Wes McKinney

58.9k

21

116

94

- 1 What if I wanted to rename the column, for example something like: df[['b as foo', 'c as bar']] such that the output renames column b as foo and column c as bar? – [kuanb](#) Feb 14 '17 at 20:30
- 3 df[['b', 'c']].rename(columns = {'b' : 'foo', 'c' : 'bar'}) – [Greg](#) Aug 25 '17 at 22:48



47



I realize this question is quite old, but in the latest version of pandas there is an easy way to do exactly this. Column names (which are strings) **can** be sliced in whatever manner you like.

```
columns = ['b', 'c']
df1 = pd.DataFrame(df, columns=columns)
```

answered Feb 4 '16 at 14:05



zerovector

767

7

12

- 5 This can only be done on creation. The question is asking if you already have it in a dataframe. – [Banjocat](#) Nov 28 '17 at 7:05



20



You could provide a list of columns to be dropped and return back the DataFrame with only the columns needed using the `drop()` function on a Pandas DataFrame.

Just saying

```
colsToDrop = ['a']
df.drop(colsToDrop, axis=1)
```

would return a DataFrame with just the columns `b` and `c`.

The `drop` method is documented [here](#).

edited Nov 3 '14 at 22:16



Alex Riley

87.8k

31

173

172

answered Sep 3 '14 at 11:30



Muthu Chithambara

Jothi

351

2

2

I found this method to be very useful:

17

```
# iloc[row slicing, column slicing]
surveys_df.iloc [0:3, 1:4]
```

More details can be found [here](#)

edited Apr 2 '18 at 18:38



Sylhare

1,235

1

15

25

answered May 2 '17 at 9:41



Alvis

470

8

14

just use: it will select `b` and `c` column.

13

```
df1=pd.DataFrame()
df1=df[['b','c']]
```

then u can just call `df1`:

```
df1
```

answered Nov 10 '17 at 9:35



Akash Nayak

571

5

12

With pandas,

7

with column names

```
dataframe[['column1','column2']]
```

with `iloc`, column index can be used like

```
dataframe[:, [1,2]]
```

with loc column names can be used like

```
dataframe[:, ['column1', 'column2']]
```

hope it helps !

answered Nov 21 '18 at 15:32



Vivek Ananthan

1,402 2 18 31

If you want to get one element by row index and column name, you can do it just like `df['b'][0]` . It is as simple as you can image.

6

Or you can use `df.ix[0, 'b']` ,mixed usage of index and label.

Note: Since v0.20 `ix` has been deprecated in favour of `loc` / `iloc` .

edited Aug 9 '18 at 14:38



jpp

105k 21 77 122

answered Jan 3 '18 at 7:56



W.Perrin

631 7 10

Below is my code:

4

```
import pandas as pd
df = pd.read_excel("data.xlsx", sheet_name = 2)
print df
df1 = df[['emp_id', 'date']]
print df1
```

Output:

	emp_id	date	count
0	1001	11/1/2018	3
1	1002	11/1/2018	4
2		11/2/2018	2
3		11/3/2018	4

	emp_id	date
0	1001	11/1/2018
1	1002	11/1/2018
2		11/2/2018
3		11/3/2018

First dataframe is the master one. I just copied two columns into df1.

answered Dec 11 '18 at 11:46



Rishi Bansal

875 1 2 19

4

The different approaches discussed in above responses are based on the assumption that either the user knows column indices to drop or subset on, or the user wishes to subset a dataframe using a range of columns (for instance between 'C' : 'E'). **pandas.DataFrame.drop()** is certainly an option to subset data based on a list of columns defined by user (though you have to be cautious that you always use copy of dataframe and *inplace* parameters should not be set to *True*!!)

Another option is to use **pandas.columns.difference()**, which does a set difference on column names, and returns an index type of array containing desired columns. Following is the solution:

```
df = pd.DataFrame([[2,3,4],[3,4,5]],columns=['a','b','c'],index=[1,2])
columns_for_differencing = ['a']
df1 = df.copy()[df.columns.difference(columns_for_differencing)]
print(df1)
```

The output would be:

	b	c
1	3	4
2	4	5

answered Jul 21 '18 at 21:28



Harshit

153 1 3

-
- 1 The copy() is not necessary. i.e: `df1 = df[df.columns.difference(columns_for_differencing)]` will return a new/copied dataframe. You will be able to modify `df1` without altering `df`. Thank you, btw. This was exactly what I needed. – [Bazyli Debowski](#) Aug 8 '18 at 17:20
-

One different and easy approach : iterating rows

3 using iterrows

```
`df1= pd.DataFrame() #creating an empty dataframe
for index,i in df.iterrows():
    df1.loc[index,'A']=df.loc[index,'A']
    df1.loc[index,'B']=df.loc[index,'B']
df1.head()
```

edited May 18 at 20:37

answered Oct 15 '18 at 11:43



Ankita

144 2 14

[Please do not recommend the use of iterrows\(\)](#). It is a blatant enabler of the worst anti-pattern in the history of pandas. – [cs95](#) Jun 9 at 3:49

Could you please explain what do you mean by "worst anti-pattern" ? – [Ankita](#) Jun 9 at 19:41

en.wikipedia.org/wiki/Anti-pattern – [cs95](#) Jun 9 at 19:41

You can use pandas. I create the DataFrame:

2

```
import pandas as pd
df = pd.DataFrame([[1, 2, 5], [5, 4, 5], [7, 7, 8], [7, 6, 9]],
                  index=['Jane', 'Peter', 'Alex', 'Ann'],
                  columns=['Test_1', 'Test_2', 'Test_3'])
```

The DataFrame:

	Test_1	Test_2	Test_3
Jane	1	2	5
Peter	5	4	5
Alex	7	7	8
Ann	7	6	9

To select 1 or more columns by name:

```
df[['Test_1', 'Test_3']]
```

	Test_1	Test_3
Jane	1	5
Peter	5	5
Alex	7	8
Ann	7	9

You can also use:

```
df.Test_2
```

And you get column Test_2

Jane	2
Peter	4
Alex	7
Ann	6

You can also select columns and rows from these rows using `.loc()`. This is called **"slicing"**.

Notice that I take from column Test_1 to Test_3

```
df.loc[:, 'Test_1': 'Test_3']
```

The "Slice" is:

	Test_1	Test_2	Test_3
Jane	1	2	5
Peter	5	4	5
Alex	7	7	8
Ann	7	6	9

And if you just want Peter and Ann from columns Test_1 and Test_3 :

```
df.loc[['Peter', 'Ann'], ['Test_1', 'Test_3']]
```

You get:

	Test_1	Test_3
Peter	5	5
Ann	7	9

answered Feb 20 at 1:01



pink.slash

190 1 7

Starting in 0.21.0, using `.loc` or `[]` with a list with one or more missing labels, is deprecated, in favor of `.reindex`. So, the answer to your question is:

2

```
df1 = df.reindex(columns=['b','c'])
```

In prior versions, using `.loc[list-of-labels]` would work as long as at least 1 of the keys was found (otherwise it would raise a `KeyError`). This behavior is deprecated and now shows a warning message. The recommended alternative is to use `.reindex()`.

Read more at [Indexing and Selecting Data](#)

edited Jan 31 at 7:30



Nursnaaz

623 9 19

answered Aug 15 '18 at 18:13



tozCSS

1,313 12 18

you can also use [df.pop\(\)](#)

0

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=('name', 'class', 'max_speed'))
>>> df
```

```
   name  class  max_speed
0  falcon   bird    389.0
1  parrot   bird     24.0
2   lion  mammal     80.5
3  monkey  mammal
```

```
>>> df.pop('class')
0    bird
1    bird
2  mammal
3  mammal
Name: class, dtype: object
```

```
>>> df
   name  max_speed
0  falcon    389.0
1  parrot     24.0
2   lion     80.5
3  monkey      NaN
```

let me know if this helps so for you , please use `df.pop(c)`

edited Apr 7 at 22:27



cs95

153k

26

202

272

answered Jan 29 at 5:37



Puneet Sinha

415

4

17

I've seen several answers on that, but on remained unclear to me. How would you select those columns of interest? The answer to that is that if you have them gathered in a list, you can just reference the columns using the list.

Example

```
print(extracted_features.shape)
print(extracted_features)

(63,)
['f000004' 'f000005' 'f000006' 'f000014' 'f000039' 'f000040' 'f000043'
 'f000047' 'f000048' 'f000049' 'f000050' 'f000051' 'f000052' 'f000053'
 'f000054' 'f000055' 'f000056' 'f000057' 'f000058' 'f000059' 'f000060'
 'f000061' 'f000062' 'f000063' 'f000064' 'f000065' 'f000066' 'f000067'
 'f000068' 'f000069' 'f000070' 'f000071' 'f000072' 'f000073' 'f000074'
 'f000075' 'f000076' 'f000077' 'f000078' 'f000079' 'f000080' 'f000081'
 'f000082' 'f000083' 'f000084' 'f000085' 'f000086' 'f000087' 'f000088'
 'f000089' 'f000090' 'f000091' 'f000092' 'f000093' 'f000094' 'f000095'
 'f000096' 'f000097' 'f000098' 'f000099' 'f000100' 'f000101' 'f000103']
```

I have the following list/numpy array `extracted_features` , specifying 63 columns. The original dataset has 103 columns, and I would like to extract exactly those, then I would use

```
dataset[extracted_features]
```

And you will end up with this

	f000004	f000005	f000006	f000014	f000039	f000040	f000043	f000047	f000048	f0
0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.230860e-04	0.0
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	9.354540e-05	0.0
2	0.000000	0.000000	0.000000	0.026013	0.000000	0.000000	0.000000	0.000000	7.702980e-05	0.0
3	0.044790	0.026013	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.499770e-04	0.0
4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	7.352400e-05	0.0
5	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	7.352400e-05	0.0
6	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	7.352400e-05	0.0
7	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	7.352400e-05	0.0
8	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000e+00	0.0

This something you would use quite often in Machine Learning (more specifically, in feature selection). I would like to discuss other ways too, but I think that has already been covered by other stackoverflowers. Hope this've been helpful!

answered May 26 at 19:21



Kareem Jeiroudi

116 2 7

protected by [jezrael](#) Jan 3 '18 at 8:01

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?