How are iloc and loc different?

Asked 7 years, 4 months ago Modified 6 months ago Viewed 704k times

876

Can someone explain how these two methods of slicing are different?
I've seen the docs, and I've seen these answers, but I still find myself unable to understand how the three are different. To me, they seem interchangeable in large part, because they are at the lower levels of slicing.

For example, say we want to get the first five rows of a `DataFrame`. How is it that these two work?

```
df.loc[:5]
df.iloc[:5]
```

Can someone present three cases where the distinction in uses are clearer?

Once upon a time, I also wanted to know how these two functions differ from `df.ix[:5]` but `ix` has been removed from pandas 1.0, so I don't care anymore.

python pandas dataframe indexing pandas-loc

Share Edit Follow

edited Jan 8 at 6:59
Henry Ecker
33.1k 18 33 53

asked Jul 23, 2015 at 16:34
AZhao
13.1k 7 29 52

11 very important to mention the SettingWithCopyWarning scenarios: stackoverflow.com/questions/20625582/... and stackoverflow.com/questions/23688307/... – Paul May 20, 2016 at 13:08

11 Note that ix is now planned for deprecation: github.com/pandas-dev/pandas/issues/14218 – JohnE Dec 20, 2016 at 17:57

6 Answers

Sorted by: Highest score (default)

1417

+150

Label *vs.* Location

The main distinction between the two methods is:

- `loc` gets rows (and/or columns) with particular **labels**.
- `iloc` gets rows (and/or columns) at integer **locations**.

To demonstrate, consider a series `s` of characters with a non-monotonic integer index:

```
>>> s = pd.Series(list("abcdef"), index=[49, 48, 47, 0, 1, 2])
49    a
48    b
47    c
0     d
1     e
2     f

>>> s.loc[0]    # value at index label 0
```

```
'd'

>>> s.iloc[0]   # value at index location 0
'a'

>>> s.loc[0:1]  # rows at index labels between 0 and 1 (inclusive)
0    d
1    e

>>> s.iloc[0:1] # rows at index location between 0 and 1 (exclusive)
49    a
```

Here are some of the differences/similarities between `s.loc` and `s.iloc` when passed various objects:

<object>	**description**	**`s.loc[<object>]`**	**`s.iloc[<object>]`**
`0`	single item	Value at index *label* `0` (the string `'d'`)	Value at index *location* 0 (the string `'a'`)
`0:1`	slice	**Two** rows (labels `0` and `1`)	**One** row (first row at location 0)
`1:47`	slice with out-of-bounds end	**Zero** rows (empty Series)	**Five** rows (location 1 onwards)
`1:47:-1`	slice with negative step	**three** rows (labels `1` back to `47`)	**Zero** rows (empty Series)
`[2, 0]`	integer list	**Two** rows with given labels	**Two** rows with given locations
`s > 'e'`	Bool series (indicating which values have the property)	**One** row (containing `'f'`)	`NotImplementedError`
`(s>'e').values`	Bool array	**One** row (containing `'f'`)	Same as `loc`
`999`	int object not in index	`KeyError`	`IndexError` (out of bounds)
`-1`	int object not in index	`KeyError`	Returns last value in `s`
`lambda x: x.index[3]`	callable applied to series (here returning 3rd item in index)	`s.loc[s.index[3]]`	`s.iloc[s.index[3]]`

`loc`'s label-querying capabilities extend well-beyond integer indexes and it's worth highlighting a couple of additional examples.

Here's a Series where the index contains string objects:

```
>>> s2 = pd.Series(s.index, index=s.values)
>>> s2
a    49
b    48
c    47
d     0
e     1
f     2
```

Since `loc` is label-based, it can fetch the first value in the Series using `s2.loc['a']`. It can also slice with non-integer objects:

```
>>> s2.loc['c':'e']  # all rows lying between 'c' and 'e' (inclusive)
c    47
d     0
e     1
```

For DateTime indexes, we don't need to pass the exact date/time to fetch by label. For example:

```
>>> s3 = pd.Series(list('abcde'), pd.date_range('now', periods=5, freq='M'))
>>> s3
2021-01-31 16:41:31.879768    a
2021-02-28 16:41:31.879768    b
2021-03-31 16:41:31.879768    c
2021-04-30 16:41:31.879768    d
2021-05-31 16:41:31.879768    e
```

Then to fetch the row(s) for March/April 2021 we only need:

```
>>> s3.loc['2021-03':'2021-04']
2021-03-31 17:04:30.742316    c
2021-04-30 17:04:30.742316    d
```

Rows and Columns

`loc` and `iloc` work the same way with DataFrames as they do with Series. It's useful to note that both methods can address columns and rows together.

When given a tuple, the first element is used to index the rows and, if it exists, the second element is used to index the columns.

Consider the DataFrame defined below:

```
>>> import numpy as np
>>> df = pd.DataFrame(np.arange(25).reshape(5, 5),
                      index=list('abcde'),
                      columns=['x','y','z', 8, 9])
>>> df
    x   y   z   8   9
a   0   1   2   3   4
b   5   6   7   8   9
c  10  11  12  13  14
d  15  16  17  18  19
e  20  21  22  23  24
```

Then for example:

```
>>> df.loc['c': , :'z']  # rows 'c' and onwards AND columns up to 'z'
    x   y   z
c  10  11  12
d  15  16  17
e  20  21  22

>>> df.iloc[:, 3]        # all rows, but only the column at index location 3
a     3
b     8
c    13
d    18
e    23
```

Sometimes we want to mix label and positional indexing methods for the rows and columns, somehow combining the capabilities of `loc` and `iloc`.

For example, consider the following DataFrame. How best to slice the rows up to and including 'c' *and* take the first four columns?

```
>>> import numpy as np
>>> df = pd.DataFrame(np.arange(25).reshape(5, 5),
                      index=list('abcde'),
                      columns=['x','y','z', 8, 9])
```

```
>>> df
    x   y   z   8   9
a   0   1   2   3   4
b   5   6   7   8   9
c  10  11  12  13  14
d  15  16  17  18  19
e  20  21  22  23  24
```

We can achieve this result using `iloc` and the help of another method:

```
>>> df.iloc[:df.index.get_loc('c') + 1, :4]
    x   y   z   8
a   0   1   2   3
b   5   6   7   8
c  10  11  12  13
```

`get_loc()` is an index method meaning "get the position of the label in this index". Note that since slicing with `iloc` is exclusive of its endpoint, we must add 1 to this value if we want row 'c' as well.

Share Edit Follow

edited Jul 12, 2021 at 17:23
Parham
2,847 4 29 44

answered Jul 23, 2015 at 16:59
Alex Riley
161k 44 253 234

14 Great explanation! One related question I've always had is what relation, if any, loc, iloc and ix have with SettingWithCopy warnings? There is some documentation but to be honest I'm still a little confused pandas.pydata.org/pandas-docs/stable/… – measureallthethings Jul 23, 2015 at 18:36

5 @measureallthethings: `loc`, `iloc` and `ix` might still trigger the warning if they are chained together. Using the example DataFrame in the linked docs `dfmi.loc[:, 'one'].loc[:, 'second']` triggers the warning just like `dfmi['one']['second']` because a copy of data (rather than a view) might be returned by the first indexing operation. – Alex Riley Jul 23, 2015 at 18:56

1 What do you use if you want to lookup a DateIndex with a Date, or something like `df.ix[date, 'Cash']` ? – cjm2671 Apr 29, 2016 at 8:51

1 @cjm2671: both `loc` or `ix` should work in that case. For example, `df.loc['2016-04-29', 'Cash']` will return all row indexes with that particular date from the 'Cash' column. (You can be as specific as you like when retrieving indexes with strings, e.g. `'2016-01'` will select all datetimes falling in January 2016, `'2016-01-02 11' will select datetimes on January 2 2016 with time 11:??:??.) – Alex Riley Apr 29, 2016 at 9:18

1 In case you want to update this answer at some point, there are suggestions here for how to use loc/iloc instead of ix github.com/pandas-dev/pandas/issues/14218 – JohnE Dec 20, 2016 at 18:00

167

`iloc` works based on integer positioning. So no matter what your row labels are, you can always, e.g., get the first row by doing

```
df.iloc[0]
```

or the last five rows by doing

```
df.iloc[-5:]
```

You can also use it on the columns. This retrieves the 3rd column:

```
df.iloc[:, 2]    # the : in the first position indicates all rows
```

You can combine them to get intersections of rows and columns:

```
df.iloc[:3, :3] # The upper-left 3 X 3 entries (assuming df has 3+ rows and
columns)
```

On the other hand, `.loc` use named indices. Let's set up a data frame with strings as row and column labels:

```
df = pd.DataFrame(index=['a', 'b', 'c'], columns=['time', 'date', 'name'])
```

Then we can get the first row by

```
df.loc['a']     # equivalent to df.iloc[0]
```

and the second two rows of the `'date'` column by

```
df.loc['b':, 'date']   # equivalent to df.iloc[1:, 1]
```

and so on. Now, it's probably worth pointing out that the default row and column indices for a `DataFrame` are integers from 0 and in this case `iloc` and `loc` would work in the same way. This is why your three examples are equivalent. **If you had a non-numeric index such as strings or datetimes,** `df.loc[:5]` **would raise an error.**

Also, you can do column retrieval just by using the data frame's `__getitem__`:

```
df['time']    # equivalent to df.loc[:, 'time']
```

Now suppose you want to mix position and named indexing, that is, indexing using names on rows and positions on columns (to clarify, I mean select from our data frame, rather than creating a data frame with strings in the row index and integers in the column index). This is where `.ix` comes in:

```
df.ix[:2, 'time']    # the first two rows of the 'time' column
```

I think it's also worth mentioning that you can pass boolean vectors to the `loc` method as well. For example:

```
 b = [True, False, True]
 df.loc[b]
```

Will return the 1st and 3rd rows of `df`. This is equivalent to `df[b]` for selection, but it can also be used for assigning via boolean vectors:

```
df.loc[b, 'name'] = 'Mary', 'John'
```

Share Edit Follow

edited Dec 7, 2018 at 10:29
nbro
14.5k 27 105 191

answered Jul 23, 2015 at 17:17
JoeCondron
8,196 2 25 27

1 Is df.iloc[:, :] equivalent to all rows and columns? – Ali May 3, 2017 at 10:03

2 It is, as would be `df.loc[:, :]`. It can be used to re-assign the values of the entire `DataFrame` or create a view of it. – JoeCondron May 3, 2017 at 20:45

1 hi, do you know why loc and iloc take parameters in between the square parenthesis [ ] and not as a normal method in between classical parenthesis ( ) ? – Marine Galantin Jun 10, 2020 at 17:27

@MarineGalantin because they are indicating *indexing* and slicing operations, not standard methods. You are selecting subsets of data. – eric Mar 14 at 15:59

159

In my opinion, the accepted answer is confusing, since it uses a DataFrame with only missing values. I also do not like the term **position-based** for `.iloc` and instead, prefer **integer location** as it is much more descriptive and exactly what `.iloc` stands for. The key word is INTEGER - `.iloc` needs INTEGERS.

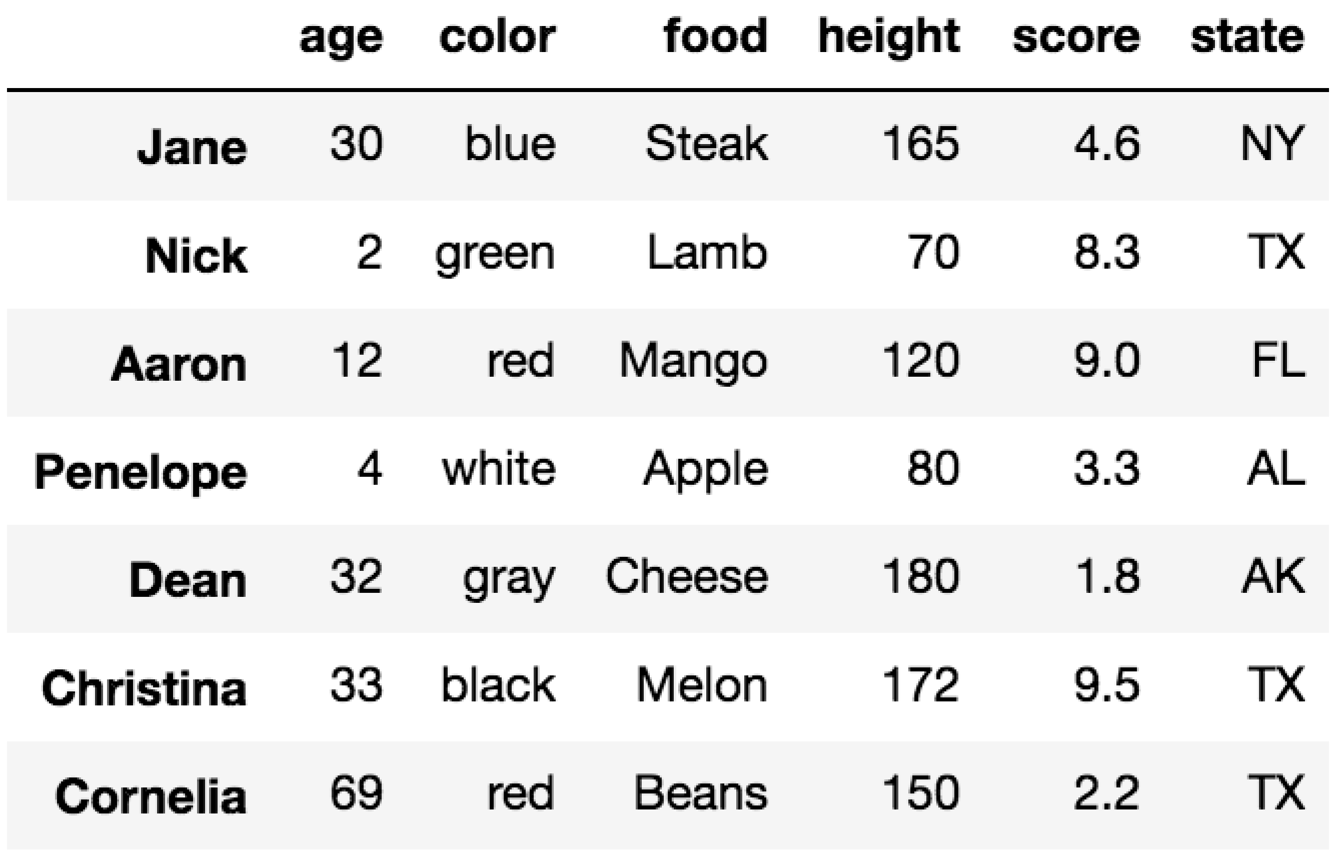
See my extremely detailed blog series on subset selection for more

.ix is deprecated and ambiguous and should never be used

Because `.ix` is deprecated we will only focus on the differences between `.loc` and `.iloc`.

Before we talk about the differences, it is important to understand that DataFrames have labels that help identify each column and each index. Let's take a look at a sample DataFrame:

```
df = pd.DataFrame({'age':[30, 2, 12, 4, 32, 33, 69],
                   'color':['blue', 'green', 'red', 'white', 'gray', 'black',
'red'],
                   'food':['Steak', 'Lamb', 'Mango', 'Apple', 'Cheese',
'Melon', 'Beans'],
                   'height':[165, 70, 120, 80, 180, 172, 150],
                   'score':[4.6, 8.3, 9.0, 3.3, 1.8, 9.5, 2.2],
                   'state':['NY', 'TX', 'FL', 'AL', 'AK', 'TX', 'TX']
                   },
                  index=['Jane', 'Nick', 'Aaron', 'Penelope', 'Dean',
'Christina', 'Cornelia'])
```



	age	color	food	height	score	state
Jane	30	blue	Steak	165	4.6	NY
Nick	2	green	Lamb	70	8.3	TX
Aaron	12	red	Mango	120	9.0	FL
Penelope	4	white	Apple	80	3.3	AL
Dean	32	gray	Cheese	180	1.8	AK
Christina	33	black	Melon	172	9.5	TX
Cornelia	69	red	Beans	150	2.2	TX

All the words in **bold** are the labels. The labels, `age`, `color`, `food`, `height`, `score` and `state` are used for the **columns**. The other labels, `Jane`, `Nick`, `Aaron`, `Penelope`, `Dean`, `Christina`, `Cornelia` are used for the **index**.

The primary ways to select particular rows in a DataFrame are with the `.loc` and `.iloc` indexers. Each of these indexers can also be used to simultaneously select columns but it is easier to just focus on rows for now. Also, each of the indexers use a set of brackets that immediately follow their name to make their selections.

.loc selects data only by labels

We will first talk about the `.loc` indexer which only selects data by the index or column labels. In our sample DataFrame, we have provided meaningful names as values for the index. Many DataFrames will not have any meaningful names and will instead, default to just the integers from 0 to n-1, where n is the length of the DataFrame.

There are three different inputs you can use for `.loc`

- A string
- A list of strings
- Slice notation using strings as the start and stop values

Selecting a single row with .loc with a string

To select a single row of data, place the index label inside of the brackets following `.loc` .

```
df.loc['Penelope']
```

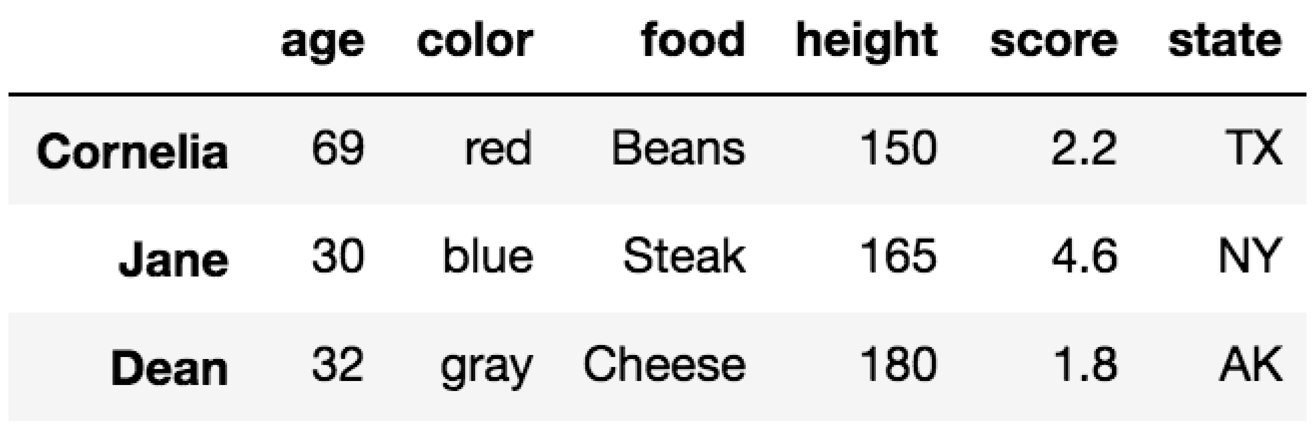
This returns the row of data as a Series

```
age           4
color     white
food      Apple
height       80
score       3.3
state        AL
Name: Penelope, dtype: object
```

Selecting multiple rows with .loc with a list of strings

```
df.loc[['Cornelia', 'Jane', 'Dean']]
```

This returns a DataFrame with the rows in the order specified in the list:



	age	color	food	height	score	state
Cornelia	69	red	Beans	150	2.2	TX
Jane	30	blue	Steak	165	4.6	NY
Dean	32	gray	Cheese	180	1.8	AK

Selecting multiple rows with .loc with slice notation

Slice notation is defined by a start, stop and step values. When slicing by label, pandas includes the stop value in the return. The following slices from Aaron to Dean, inclusive. Its step size is not explicitly defined but defaulted to 1.

```
df.loc['Aaron':'Dean']
```

	age	color	food	height	score	state
Aaron	12	red	Mango	120	9.0	FL
Penelope	4	white	Apple	80	3.3	AL
Dean	32	gray	Cheese	180	1.8	AK

Complex slices can be taken in the same manner as Python lists.

.iloc selects data only by integer location

Let's now turn to `.iloc`. Every row and column of data in a DataFrame has an integer location that defines it. *This is in addition to the label that is visually displayed in the output*. The integer location is simply the number of rows/columns from the top/left beginning at 0.

There are three different inputs you can use for `.iloc`

- An integer
- A list of integers
- Slice notation using integers as the start and stop values

Selecting a single row with .iloc with an integer

```
df.iloc[4]
```

This returns the 5th row (integer location 4) as a Series

```
age           32
color       gray
food      Cheese
height       180
score        1.8
state         AK
Name: Dean, dtype: object
```

Selecting multiple rows with .iloc with a list of integers

```
df.iloc[[2, -2]]
```

This returns a DataFrame of the third and second to last rows:

	age	color	food	height	score	state
Aaron	12	red	Mango	120	9.0	FL
Christina	33	black	Melon	172	9.5	TX

Selecting multiple rows with .iloc with slice notation

```
df.iloc[:5:3]
```

	age	color	food	height	score	state
Jane	30	blue	Steak	165	4.6	NY
Penelope	4	white	Apple	80	3.3	AL

Simultaneous selection of rows and columns with .loc and .iloc

One excellent ability of both `.loc/.iloc` is their ability to select both rows and columns simultaneously. In the examples above, all the columns were returned from each selection. We can choose columns with the same types of inputs as we do for rows. We simply need to separate the row and column selection with a **comma**.

For example, we can select rows Jane, and Dean with just the columns height, score and state like this:

```
df.loc[['Jane', 'Dean'], 'height':]
```

	height	score	state
Jane	165	4.6	NY
Dean	180	1.8	AK

This uses a list of labels for the rows and slice notation for the columns

We can naturally do similar operations with `.iloc` using only integers.

```
df.iloc[[1,4], 2]
Nick      Lamb
Dean    Cheese
Name: food, dtype: object
```

Simultaneous selection with labels and integer location

`.ix` was used to make selections simultaneously with labels and integer location which was useful but confusing and ambiguous at times and thankfully it has been deprecated. In the event that you need to make a selection with a mix of labels and integer locations, you will have to make both your selections labels or integer locations.

For instance, if we want to select rows `Nick` and `Cornelia` along with columns 2 and 4, we could use `.loc` by converting the integers to labels with the following:

```
col_names = df.columns[[2, 4]]
df.loc[['Nick', 'Cornelia'], col_names]
```

Or alternatively, convert the index labels to integers with the `get_loc` index method.

```
labels = ['Nick', 'Cornelia']
index_ints = [df.index.get_loc(label) for label in labels]
df.iloc[index_ints, [2, 4]]
```

Boolean Selection

The .loc indexer can also do boolean selection. For instance, if we are interested in finding all the rows wher age is above 30 and return just the `food` and `score` columns we can do the following:

```
df.loc[df['age'] > 30, ['food', 'score']]
```

You can replicate this with `.iloc` but you cannot pass it a boolean series. You must convert the boolean Series into a numpy array like this:

```
df.iloc[(df['age'] > 30).values, [2, 4]]
```

Selecting all rows

It is possible to use `.loc/.iloc` for just column selection. You can select all the rows by using a colon like this:

```
df.loc[:, 'color':'score':2]
```

	color	height
Jane	blue	165
Nick	green	70
Aaron	red	120
Penelope	white	80
Dean	gray	180
Christina	black	172
Cornelia	red	150

The indexing operator, `[]`, can select rows and columns too but not simultaneously.

Most people are familiar with the primary purpose of the DataFrame indexing operator, which is to select columns. A string selects a single column as a Series and a list of strings selects multiple columns as a DataFrame.

```
df['food']
```

```
Jane          Steak
Nick           Lamb
Aaron         Mango
Penelope      Apple
Dean         Cheese
Christina     Melon
Cornelia      Beans
Name: food, dtype: object
```

Using a list selects multiple columns

```
df[['food', 'score']]
```

	food	score
Jane	Steak	4.6
Nick	Lamb	8.3
Aaron	Mango	9.0
Penelope	Apple	3.3
Dean	Cheese	1.8
Christina	Melon	9.5
Cornelia	Beans	2.2

What people are less familiar with, is that, when slice notation is used, then selection happens by row labels or by integer location. This is very confusing and something that I almost never use but it does work.

```
df['Penelope':'Christina'] # slice rows by label
```

	age	color	food	height	score	state
Penelope	4	white	Apple	80	3.3	AL
Dean	32	gray	Cheese	180	1.8	AK
Christina	33	black	Melon	172	9.5	TX

```
df[2:6:2] # slice rows by integer location
```

	age	color	food	height	score	state
Aaron	12	red	Mango	120	9.0	FL
Dean	32	gray	Cheese	180	1.8	AK

The explicitness of `.loc`/`.iloc` for selecting rows is highly preferred. The indexing operator alone is unable to select rows and columns simultaneously.

```
df[3:5, 'color']
TypeError: unhashable type: 'slice'
```

Share Edit Follow

edited Oct 27, 2018 at 1:53

answered Oct 24, 2017 at 16:39

Ted Petrou
56k 19 124 129

11 Wow, this was one of the very well articulated and lucid explanations that i have ever come across of a programming topic, What you explained in the last about normal indexing which works either on row or columns is one of the reason we have loc and iloc method. I came across that caveat in the datacamp course. a.) What do df.columns and df.index return? Is it a list of strings? If it is a list, is it allowed to access two elements like this df.columns[[2,4]] in a list? b.) Can i call get_loc() on df.columns? c.) Why do we need to call df['age']>30.values in case of iloc. – pragun May 30, 2019 at 22:34

1 This is a really good answer, I liked that it doesn't get much into ix, which is deprecated and pointless to dive deep. Thanks. – omabena May 16, 2020 at 22:17

Made more sense than the top and currently accepted answer. – student010101 Apr 7, 2021 at 17:32

Why did they use `loc` instead of `label` ? It seems like the nomenclature is basically a confusion generator. – eric Sep 21, 2021 at 13:56

3

`.loc` and `.iloc` are used for indexing, i.e., to pull out portions of data. In essence, the difference is that `.loc` allows label-based indexing, while `.iloc` allows position-based indexing.

If you get confused by `.loc` and `.iloc`, keep in mind that `.iloc` is based on the index (starting with *i*) position, while `.loc` is based on the label (starting with *l*).

`.loc`

`.loc` is supposed to be based on the index labels and not the positions, so it is analogous to Python dictionary-based indexing. However, it can accept boolean arrays, slices, and a list of labels (none of which work with a Python dictionary).

`iloc`

`.iloc` does the lookup based on index position, i.e., `pandas` behaves similarly to a Python list. `pandas` will raise an `IndexError` if there is no index at that location.

Examples

The following examples are presented to illustrate the differences between `.iloc` and `.loc`. Let's consider the following series:

```
>>> s = pd.Series([11, 9], index=["1990", "1993"], name="Magic Numbers")
>>> s
1990    11
1993     9
Name: Magic Numbers , dtype: int64
```

`.iloc` Examples

```
>>> s.iloc[0]
11
>>> s.iloc[-1]
9
>>> s.iloc[4]
Traceback (most recent call last):
    ...
IndexError: single positional indexer is out-of-bounds
```

```
>>> s.iloc[0:3] # slice
1990 11
1993  9
Name: Magic Numbers , dtype: int64
>>> s.iloc[[0,1]] # list
1990 11
1993  9
Name: Magic Numbers , dtype: int64
```

`.loc` Examples

```
>>> s.loc['1990']
11
>>> s.loc['1970']
Traceback (most recent call last):
    ...
KeyError: ’the label [1970] is not in the [index]’
>>> mask = s > 9
>>> s.loc[mask]
1990 11
Name: Magic Numbers , dtype: int64
>>> s.loc['1990':] # slice
1990    11
1993     9
Name: Magic Numbers, dtype: int64
```

Because `s` has string index values, `.loc` will fail when indexing with an integer:

```
>>> s.loc[0]
Traceback (most recent call last):
    ...
KeyError: 0
```

Share Edit Follow

edited Dec 28, 2020 at 22:39

answered Dec 27, 2020 at 0:56
lmiguelvargasf
57.6k 44 213 216

1

- `DataFrame.loc()` : Select rows by index value
- `DataFrame.iloc()` : Select rows by rows number

Example:

Select first 5 rows of a table, `df1` is your dataframe

```
df1.iloc[:5]
```

Select first A, B rows of a table, `df1` is your dataframe

```
df1.loc['A','B']
```

Share Edit Follow

edited May 25, 2021 at 13:01
buhtz
9,464 14 66 132

answered Dec 10, 2020 at 5:21
Nandini Ashok Tuptewar
216 2 5

1

This example will illustrate the difference:

```
df = pd.DataFrame({'col1': [1,2,3,4,5], 'col2': ["foo", "bar", "baz", "foobar",
"foobaz"]})
  col1  col2
0   1   foo
1   2   bar
2   3   baz
3   4   foobar
4   5   foobaz

df = df.sort_values('col1', ascending = False)
      col1  col2
    4   5   foobaz
    3   4   foobar
    2   3   baz
    1   2   bar
    0   1   foo
```

Index based access:

```
df.iloc[0, 0:2]
col1         5
col2    foobaz
Name: 4, dtype: object
```

We get the first row of the sorted dataframe. (This is not the row with index 0, but with index 4).

Position based access:

```
df.loc[0, 'col1':'col2']
col1      1
col2    foo
Name: 0, dtype: object
```

We get the row with index 0, even when the df is sorted.

Share Edit Follow

answered May 27 at 12:01

TanuAD
615 6 12