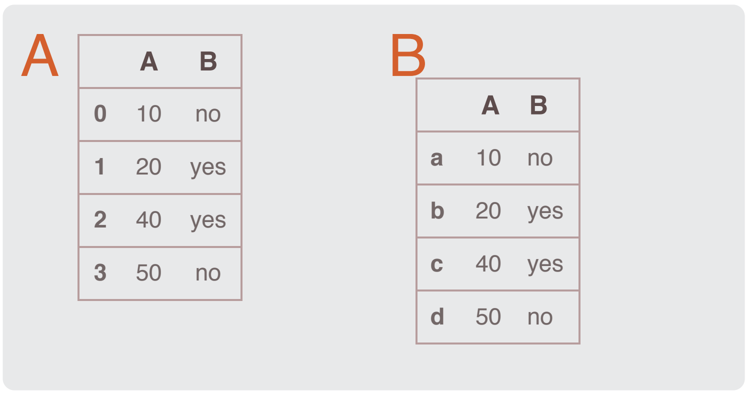
What's the difference between loc, iloc, and ix?

Jul 10, 2016

pandas

Part of the regular trade of conducting data analysis is slicing. Slicing means taking a part of your data set for further inspection. `pandas` offers at least three methods for slicing data: `.loc[]`, `.iloc[]`, and `.ix[]`. It is really easy to take one for the other. Here is a quick reference to help you tell them apart.

These three methods belong to index selection methods. Index is the identifier `pandas` uses for each row of the data set. A key thing to have into account is that indexing can take specific labels. These labels can either be integers or any other specified value by the user (e.g. dates, names). Check the following example:



A

	A	B
0	10	no
1	20	yes
2	40	yes
3	50	no

B

	A	B
a	10	no
b	20	yes
c	40	yes
d	50	no

The data set is exactly the same. The only things that changes is the index. In one case is integer based, i.e. data set A. In the other, it uses a set of strings to identify the rows. This distinction is important to take into account when using selection methods.

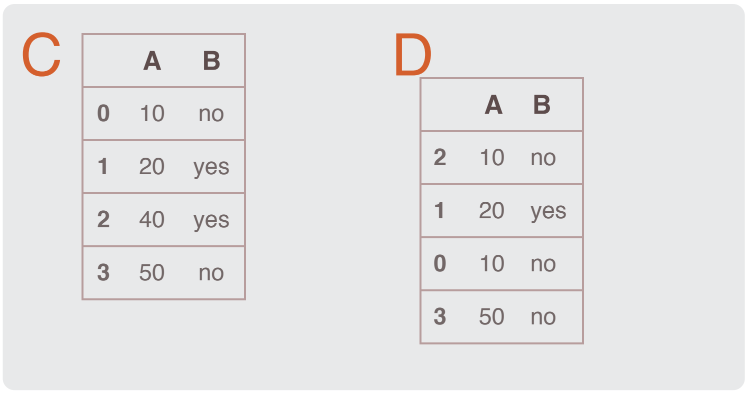
`.loc[]` is a **label based** method. This means that it will take into account the *names* or labels of the index when taking slices. For our data set B, `df_B.loc["b"]` will result in all the second row being selected.

On the other hand, `.iloc[]` takes slices based on index's position. For the ones familiar with Python, it behaves like regular slicing. You just indicate the positional index number, and you get the appropriate slice. For example `df_A.iloc[0]` and `df_B.iloc[0]`, both will give you the first row of the data set.

Note that, as in Python, `.iloc` is zero positional based, i.e. it starts at 0. `.ix[]` is the more flexible option. It accepts label based or index positional arguments.

Things get a bit tricky

Things start getting a bit tricky when you have an integer based index. In this case, you can access to slices of the data set with `.loc[]` and `.iloc[]`. Note, however, that the results vary depending on which method you are using. Is always better to know what you get with each method to avoid possible mistakes. Here is a simple way in which you can mess it up. Suppose you have the following data sets:



C

	A	B
0	10	no
1	20	yes
2	40	yes
3	50	no

D

	A	B
2	10	no
1	20	yes
0	10	no
3	50	no

Let's concentrate in data set C for a moment. Suppose you want to get the first three elements of the data set. Well, simple, you can use `.loc[]` or `.iloc[]`. Let's do that:

```
df_C.loc[0:3]
+----+-----+-----+
|    |   A | B   |
|----+-----+-----|
|  0 |  10 | no  |
|  1 |  20 | yes |
|  2 |  40 | yes |
|  3 |  50 | no  |
+----+-----+-----+
```

```
df_C.iloc[0:3]
+----+-----+-----+
|    |   A | B   |
|----+-----+-----|
|  0 |  10 | no  |
|  1 |  20 | yes |
|  2 |  40 | yes |
+----+-----+-----+
```

Both `.loc[]` and `.iloc[]` work here, because the integer index can also be taken as a label. We get a different result depending on the method we are using. In the case of `.loc[]`, the selection includes the last term. With `.iloc[]`, normal Python index selection rules apply. Thus, the last term is excluded.

Now, let's see data set D. Suppose you want to obtain a slice from observation 0 to 3. As you want a slice based in its position, this calls for `.iloc[]` See what happens if you mix both methods up:

```
df_d.loc[0:3]                    df_d.iloc[0:3]
+----+-----+-----+               +----+-----+-----+
|    |   A | B   |               |    |   A | B   |
|----+-----+-----|               |----+-----+-----|
|  0 |  40 | yes |               |  2 |  10 | no  |
|  3 |  50 | no  |               |  1 |  20 | yes |
+----+-----+-----+               |  0 |  40 | yes |
                                 +----+-----+-----+
```

With `.iloc[]` you are getting what was expected. With `.loc[]` you are not. Once again, this is because `.loc[]` is *label based*. The method will look at the "name" and not the position of the index. Therefore, it will find the index with the label 0 and gives you a slide that *includes* the index with the label 3.

Be careful with .ix

As `.ix[]` is the option that gives more flexibility, one might be tempted to use it. After all, for data set B, you could have accessed the first two elements either specifying `df_b.ix["a":"b"]` or `df_b.ix[0:2]`. Both options will yield the same result. However, it is important to be careful when one tries to access to slice integer based index. As seen before, when you have an integer based index, confusion may arise between location and label based methods. `.ix[]` solves this confusion by falling to label based access (i.e. like `.loc[]`), which might not be what you are looking for.

Suppose, again, that you want to have a slice of the three first observations in D and you use `.ix[]`. Here is what you will get:

```
df_d.ix[0:3]
+----+-----+-----+
|    |   A | B   |
|----+-----+-----|
|  0 |  40 | yes |
|  3 |  50 | no  |
+----+-----+-----+
```

In general, as pointed in the documentation, whenever you have a integer based index, is better to be explicit about the way you are taking slices. This means, one must avoid using `.ix` and prefer using `.loc` or `.iloc` depending on what is needed.

So, this is what you need to remember:

- `.loc` takes slices based on labels. `.iloc` uses observations' position. `.ix` uses both labels and positions.

- `.loc` includes last element, when using an interval slice (e.g, [“a”:”c”]). `.iloc` does not include last element.
- When using a integer based index be clear with what you want and avoid confusion by appropriately selecting `.loc` or `iloc` for the task. Avoid using `.ix`.

Happy slicing!