Data Indexing and Selection

Data Selection in Series

As we saw previously, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

In [2]:
```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[2]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [3]:
```
data['b']
```

Out[3]:
```
0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

In [4]:
```
'a' in data
```

Out[4]:
```
True
```

In [5]:
```
data.keys()
```

Out[5]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [6]:
```
list(data.items()) #Not values()
```

Out[6]:
```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a `Series` by assigning to a new index value:

In [7]:
```
data['e'] = 1.25
data
```

Out[7]:
```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as one-dimensional array

A `Series` builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

In [8]:
```
# slicing by explicit index
data['a':'c']
```

Out[8]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [9]:
```
# slicing by implicit integer index
data[0:2]
```

Out[9]:
```
a    0.25
b    0.50
dtype: float64
```

In [10]:
```
# masking
data[(data > 0.3) & (data < 0.8)]
```

Out[10]:
```
b    0.50
c    0.75
dtype: float64
```

In [11]:
```
# fancy indexing
data[['a', 'e']]
```

Out[11]:
```
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., `data['a':'c']`), the final index is *included* in the slice, while when slicing with an implicit index (i.e., `data[0:2]`), the final index is *excluded* from the slice.

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

In [12]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

Out[12]:
```
1    a
3    b
5    c
dtype: object
```

In [13]:
```
# explicit index when indexing
data[1]
```

Out[13]:
```
'a'
```

In [14]:
```
# implicit index when slicing
data[1:3]
```

Out[14]:
```
3    b
5    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the `Series` .

First, the `loc` attribute allows indexing and slicing that always references the explicit index:

In [15]:
```
data.loc[1]
```

Out[15]:
```
'a'
```

In [16]:
```
data.loc[1:3]
```

Out[16]:
```
1    a
3    b
dtype: object
```

The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

In [17]:
```
data.iloc[1]
```

Out[17]:
```
'b'
```

In [18]:
```
data.iloc[1:3]
```

Out[18]:
```
3    b
5    c
dtype: object
```

A third indexing attribute, `ix`, is a hybrid of the two, and for `Series` objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of `DataFrame` objects, which we will discuss in a moment.

One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a `DataFrame` acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of `Series` structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a dictionary

The first analogy we will consider is the `DataFrame` as a dictionary of related `Series` objects. Let's return to our example of areas and populations of states:

In [19]:
```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[19]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

The individual `Series` that make up the columns of the `DataFrame` can be accessed via dictionary-style indexing of the column name:

In [20]:
```
data['area']
```

Out[20]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

In [21]:
```
data.area
```

Out[21]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

In [22]:
```
data.area is data['area']
```

Out[22]:
```
True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the `DataFrame` , this attribute-style access is not possible. For example, the `DataFrame` has a `pop()` method, so `data.pop` will point to this rather than the `"pop"` column:

In [23]:
```
data.pop is data['pop']
```

Out[23]:
```
False
```

In particular, you should avoid the temptation to try column assignment via attribute (i.e., use `data['pop'] = z` rather than `data.pop = z`).

Like with the `Series` objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

In [24]:
```
data['density'] = data['pop'] / data['area']
data
```

Out[24]:

	area	pop	density
California	423967	38332521	90.413926
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

This shows a preview of the straightforward syntax of element-by-element arithmetic between `Series` objects; we'll dig into this further in [Operating on Data in Pandas (03.03-Operations-in-Pandas.ipynb)](03.03-Operations-in-Pandas.ipynb).

DataFrame as two-dimensional array

As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

In [25]:
```
data.values
```

Out[25]:
```
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])
```

With this picture in mind, many familiar array-like observations can be done on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

In [26]:
```
data.T
```

Out[26]:

	California	Texas	New York	Florida	Illinois
area	4.239670e+05	6.956620e+05	1.412970e+05	1.703120e+05	1.499950e+05
pop	3.833252e+07	2.644819e+07	1.965113e+07	1.955286e+07	1.288214e+07
density	9.041393e+01	3.801874e+01	1.390767e+02	1.148061e+02	8.588376e+01

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

In [27]:
```
data.values[0]
```

Out[27]:
```
array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

and passing a single "index" to a `DataFrame` accesses a column:

In [28]:
```
data['area']
```

Out[28]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` indexers mentioned earlier. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

In [29]:
```
data.iloc[:3, :2]
```

Out[29]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127

Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

In [30]:
```
data.loc[:'Illinois', :'pop']
```

Out[30]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127
Florida	170312	19552860
Illinois	149995	12882135

The `ix` indexer allows a hybrid of these two approaches:

In [31]:
```
data.ix[:3, :'pop']
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated (http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated)
  """Entry point for launching an IPython kernel.
```

Out[31]:

	area	pop
California	423967	38332521
Texas	695662	26448193
New York	141297	19651127

Keep in mind that for integer indices, the `ix` indexer is subject to the same potential sources of confusion as discussed for integer-indexed `Series` objects.

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

In [32]:
```
data.loc[data.density > 100, ['pop', 'density']]
```

Out[32]:

	pop	density
New York	19651127	139.076746
Florida	19552860	114.806121

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

In [33]:

```
data.iloc[0, 2] = 90
data
```

Out[33]:

	area	pop	density
California	423967	38332521	90.000000
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

To build up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be very useful in practice. First, while *indexing* refers to columns, *slicing* refers to rows:

In [34]:

```
data['Florida':'Illinois']
```

Out[34]:

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Such slices can also refer to rows by number rather than by index:

In [35]:
```
data[1:3]
```

Out[35]:

	area	pop	density
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746

Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

In [36]:
```
data[data.density > 100]
```

Out[36]:

	area	pop	density
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121

HYBRID INDEXING

In [37]:

```
####ix
"""DataFrame.ix[ ] is both Label and Integer based slicing technique."""
x2 = data.ix[:'California', 1:2]
#print(x2)
x2
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated (http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated)
  This is separate from the ipykernel package so we can avoid doing imports until
```

Out[37]:

	pop
California	38332521

In [38]:

```
x2 = data.ix[:1, 1:2]
#print(x2)
x2
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated (http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated)
  """Entry point for launching an IPython kernel.
```

Out[38]:

	pop
California	38332521

ADD, APPEND & JOIN

In [39]:

```

# Importing pandas as pd
import pandas as pd

# Creating the first Dataframe using dictionary
df1 = df = pd.DataFrame({"a":[1, 2, 3, 4],
                         "b":[5, 6, 7, 8]})

# Creating the Second Dataframe using dictionary
df2 = pd.DataFrame({"a":[1, 2, 3],
                    "b":[5, 6, 7]})

# Print df1
print(df1, "\n")

# Print df2
df2
```

```
   a  b
0  1  5
1  2  6
2  3  7
3  4  8
```

Out[39]:

	a	b
0	1	5
1	2	6
2	3	7

In [40]:

```
# to append df2 at the end of df1 dataframe
res=df1.append(df2)
res
```

Out[40]:

	a	b
0	1	5
1	2	6
2	3	7
3	4	8
0	1	5
1	2	6
2	3	7

In [86]:

```python
###===================CONCAT====================###
##Concat series
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
print(pd.concat([ser1, ser2]))
##Concat Dataframes
# Creating the first Dataframe using dictionary
df1 = df = pd.DataFrame({"a":[1, 2, 3, 4],
                         "b":[5, 6, 7, 8]})

# Creating the Second Dataframe using dictionary
df2 = pd.DataFrame({"a":[1, 2, 3],
                    "b":[5, 6, 7]})
#df2 = pd.DataFrame({"a":[1, 2, 3], "b":[5, 6, 7]}, index=[4,5,6])
res=pd.concat([df1, df2])
res
```

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Out[86]:

	a	b
0	1	5
1	2	6
2	3	7
3	4	8
0	1	5
1	2	6
2	3	7

In [70]:

```
####JOIN
result_in = pd.concat([df1, df2], axis=1, join='inner')
print(result_in)
result_outer = pd.concat([df1, df2], axis=1, join='outer')
print(result_outer)
```

```
   a  b  a  b
0  1  5  1  5
1  2  6  2  6
2  3  7  3  7
   a  b    a    b
0  1  5  1.0  5.0
1  2  6  2.0  6.0
2  3  7  3.0  7.0
3  4  8  NaN  NaN
```

In [89]:

```
###MERGE
print(df1)
print(df2)
#res1 = pd.concat(df1, df2)
res2 = pd.merge(df1, df2)
#print(res1)
res2
```

```
   a  b
0  1  5
1  2  6
2  3  7
3  4  8
   a  b
0  1  5
1  2  6
2  3  7
```

Out[89]:

	a	b
0	1	5
1	2	6
2	3	7

In [73]:

```
df3 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df4 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df3)
print(df4)
res = pd.merge(df3, df4)
res
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

Out[73]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In [74]:
```
###Many to 1 merge
df5 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
res2=print(pd.merge(res, df5))
res2
```

```
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

In [91]:
```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

Out[91]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

In [101]:

```
1 ###Aggregation
2 grp=df.groupby('data')
3 print(grp)
4 df.groupby('key').sum()
5 
6 ####NOTE: In Line number3 groupby is giving you just address because groupby works with only with aggregate function
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000020BA406DE80>
```

Out[101]:

key	data
A	3
B	5
C	7

In [96]:

```
a = df.sort_values(by ='key', ascending = 0)
print("Sorting rows by key:\n \n", a)
```

```
Sorting rows by key:

  key  data
2   C     2
5   C     5
1   B     1
4   B     4
0   A     0
3   A     3
```

In [ ]:

```

```