Getting Started with pandas

Part 4

Essential Functionality

Part 2

Indexing, Selection, and Filtering

- Series indexing (`obj[...]`) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers.

```
In [85]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
         obj
Out[85]: a    0.0
         b    1.0
         c    2.0
         d    3.0
         dtype: float64

In [86]: obj['b']
Out[86]: 1.0

In [87]: obj[1]
Out[87]: 1.0

In [88]: obj[2:4]
Out[88]: c    2.0
         d    3.0
         dtype: float64
```

```
In [89]: obj[['b', 'a', 'd']]

Out[89]: b    1.0
         a    0.0
         d    3.0
         dtype: float64

In [90]: obj[[1, 3]]

Out[90]: b    1.0
         d    3.0
         dtype: float64

In [91]: obj[obj < 2]

Out[91]: a    0.0
         b    1.0
         dtype: float64
```

- Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```
In [92]: obj['b':'c']

Out[92]: b    1.0
         c    2.0
         dtype: float64
```

- *Setting* using these methods modifies the corresponding section of the Series:

```
In [93]: obj
Out[93]: a    0.0
         b    1.0
         c    2.0
         d    3.0
         dtype: float64

In [94]: obj['b':'c'] = 5
         obj
Out[94]: a    0.0
         b    5.0
         c    5.0
         d    3.0
         dtype: float64
```

- Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

In [95]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[95]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [96]:
```
data['two']
```

Out[96]:
```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

In [97]:
```
data[['three', 'one']]
```

Out[97]:

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

- Indexing like this has a few special cases.
- First, slicing or selecting data with a boolean array:

```
In [98]: data
```

Out[98]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [99]: data[:2]
```

Out[99]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
In [100]: data[data['three'] > 5]
```

Out[100]:

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

- Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [101]: data < 5
```

Out[101]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
In [102]: data[data < 5] = 0
```

```
In [103]: data
```

Out[103]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Selection with loc and iloc

- For DataFrame label-indexing on the rows, I introduce the special indexing operators `loc` and `iloc`.
- They enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

- As a preliminary example, let’s select a single row and multiple columns by label:

```
In [104]: data.loc['Colorado', ['two', 'three']]

Out[104]: two      5
          three    6
          Name: Colorado, dtype: int64
```

- We'll then perform some similar selections with integers using `iloc`:

```
In [105]: data.iloc[2, [3, 0, 1]]
Out[105]: four    11
          one      8
          two      9
          Name: Utah, dtype: int64

In [106]: data.iloc[2]
Out[106]: one       8
          two       9
          three    10
          four     11
          Name: Utah, dtype: int64

In [107]: data.iloc[[1, 2], [3, 0, 1]]
Out[107]:
```

	four	one	two
Colorado	7	0	5
Utah	11	8	9

- Both indexing functions work with slices in addition to single labels or lists of labels:

```
In [108]: data
```

Out[108]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [109]: data.loc[:'Utah', 'two']
Out[109]: Ohio        0
          Colorado    5
          Utah        9
          Name: two, dtype: int64

In [110]: data.iloc[:, :3][data.three > 5]
```

Out[110]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

Integer Indexes

- Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples.

- For example, you might not expect the following code to generate an error:

```
In [111]: ser = pd.Series(np.arange(3.))

In [112]: ser
Out[112]: 0    0.0
          1    1.0
          2    2.0
          dtype: float64

In [113]: ser[-1]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-113-44969a759c20> in <module>
----> 1 ser[-1]

~/anaconda3/lib/python3.7/site-packages/pandas/core/series.py in __getitem__(self, key)
    866         key = com.apply_if_callable(key, self)
    867         try:
--> 868             result = self.index.get_value(self, key)
    869
    870             if not is_scalar(result):

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_value(self, series, key)
   4373         try:
   4374             return self._engine.get_value(s, k,
-> 4375                                           tz=getattr(series.dtype, 'tz', None))
   4376         except KeyError as e1:
   4377             if len(self) > 0 and (self.holds_integer() or self.is_boolean()):

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item()
```

- In this case, pandas could “fall back” on integer indexing, but it’s difficult to do this in general without introducing subtle bugs.
- Here we have an index containing `0`, `1`, `2`, but inferring what the user wants (label-based indexing or position-based) is difficult:

```
In [114]: ser

Out[114]: 0    0.0
          1    1.0
          2    2.0
          dtype: float64
```

- On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [115]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [116]: ser2
Out[116]: a    0.0
          b    1.0
          c    2.0
          dtype: float64

In [117]: ser2[-1]
Out[117]: 2.0
```

- To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented.
- For more precise handling, use `loc` (for labels) or `iloc` (for integers):

```
In [118]: ser
Out[118]: 0    0.0
          1    1.0
          2    2.0
          dtype: float64

In [119]: ser[:1]
Out[119]: 0    0.0
          dtype: float64

In [120]: ser.loc[:1]
Out[120]: 0    0.0
          1    1.0
          dtype: float64

In [121]: ser.iloc[:1]
Out[121]: 0    0.0
          dtype: float64
```

Arithmetic and Data Alignment

- An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes.
- When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.
- For users with database experience, this is similar to an automatic outer join on the index labels.

In [122]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
               index=['a', 'c', 'e', 'f', 'g'])
```

In [123]:
```
s1
```

Out[123]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [124]:
```
s2
```

Out[124]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [125]:
```
s1 + s2
```

Out[125]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

- In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [126]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                             index=['Ohio', 'Texas', 'Colorado'])
          df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                             index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [127]: df1
```

Out[127]:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

```
In [128]: df2
```

Out[128]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [129]: df1 + df2
```

Out[129]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

- If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

```
In [130]: df1 = pd.DataFrame({'A': [1, 2]})
          df2 = pd.DataFrame({'B': [3, 4]})

In [131]: df1
```

Out[131]:

	A
0	1
1	2

```
In [132]: df2
```

Out[132]:

	B
0	3
1	4

```
In [133]: df1 + df2
```

Out[133]:

	A	B
0	NaN	NaN
1	NaN	NaN