Getting Started with pandas

Part 3

Introduction to pandas Data Structures

Part 3

Index Objects

- pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names).
- Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [55]: obj = pd.Series(range(3), index=['a', 'b', 'c'])
         index = obj.index
         index

Out[55]: Index(['a', 'b', 'c'], dtype='object')

In [56]: index[1:]

Out[56]: Index(['b', 'c'], dtype='object')
```

- Index objects are immutable and thus can’t be modified by the user:

```
In [57]: index[1] = 'd'  # TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-d11f5623d88a> in <module>
----> 1 index[1] = 'd'  # TypeError

~/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py in __setitem__(self, key, value)
   3936
   3937     def __setitem__(self, key, value):
-> 3938         raise TypeError("Index does not support mutable operations")
   3939
   3940     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

- Immutability makes it safer to share Index objects among data structures:

```
In [58]: labels = pd.Index(np.arange(3))
         labels

Out[58]: Int64Index([0, 1, 2], dtype='int64')

In [59]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)
         obj2

Out[59]: 0    1.5
         1   -2.5
         2    0.0
         dtype: float64

In [60]: obj2.index is labels

Out[60]: True
```

- In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [61]: frame3
```

Out[61]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [62]: frame3.columns
Out[62]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [63]: 'Ohio' in frame3.columns
Out[63]: True

In [64]: 2003 in frame3.index
Out[64]: False
```

- Unlike Python sets, a pandas Index can contain duplicate labels:

```
In [65]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
         dup_labels

Out[65]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

- Selections with duplicate labels will select all occurrences of that label.

- Some Index methods and properties

Method	Description
`append`	Concatenate with additional Index objects, producing a new Index
`difference`	Compute set difference as an Index
`intersection`	Compute set intersection
`union`	Compute set union
`isin`	Compute boolean array indicating whether each value is contained in the passed collection

`delete`	Compute new Index with element at index `i` deleted
`drop`	Compute new Index by deleting passed values
`insert`	Compute new Index by inserting element at index `i`
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique`	Compute the array of unique values in the Index

Essential Functionality

Part 1

Reindexing

- An important method on pandas objects is `reindex`, which means to create a new object with the data *conformed* to a new index.
- Consider an example:

```
In [66]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
         obj

Out[66]: d    4.5
         b    7.2
         a   -5.3
         c    3.6
         dtype: float64
```

- Calling `reindex` on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [67]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
         obj2

Out[67]: a   -5.3
         b    7.2
         c    3.6
         d    4.5
         e    NaN
         dtype: float64
```

- For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing.
- The `method` option allows us to do this, using a method such as `ffill`, which forward-fills the values:

```
In [68]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
         obj3

Out[68]: 0      blue
         2    purple
         4    yellow
         dtype: object

In [69]: obj3.reindex(range(6), method='ffill')

Out[69]: 0      blue
         1      blue
         2    purple
         3    purple
         4    yellow
         5    yellow
         dtype: object
```

- With DataFrame, `reindex` can alter either the (row) index, columns, or both.
- When passed only a sequence, it reindexes the rows in the result:

```
In [70]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                              index=['a', 'c', 'd'],
                              columns=['Ohio', 'Texas', 'California'])
         frame
```

Out[70]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
In [71]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
         frame2
```

Out[71]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

- The columns can be reindexed with the `columns` keyword:

```
In [72]: frame
```

Out[72]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
In [73]: states = ['Texas', 'Utah', 'California']
         frame.reindex(columns=states)
```

Out[73]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

- As we'll explore in more detail, you can reindex more succinctly by label-indexing with `loc`, and many users prefer to use it exclusively:

```
In [74]: frame
```

Out[74]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
In [75]: states
Out[75]: ['Texas', 'Utah', 'California']
```

```
In [76]: frame.loc[['a', 'b', 'c', 'd'], states]
/home/joshua/anaconda3/lib/python3.7/site-packages/pandas/core/indexing.py:1494: FutureWarning:
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-listlike
  return self._getitem_tuple(key)
```

Out[76]:

	Texas	Utah	California
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

Dropping Entries from an Axis

- Dropping one or more entries from an axis is easy if you already have an index array or list without those entries.
- As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis.

In [77]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[77]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [78]:
```
new_obj = obj.drop('c')
new_obj
```

Out[78]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [79]:
```
obj.drop(['d', 'c'])
```

Out[79]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

- With DataFrame, index values can be deleted from either axis.
- To illustrate this, we first create an example DataFrame:

```
In [80]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                             index=['Ohio', 'Colorado', 'Utah', 'New York'],
                             columns=['one', 'two', 'three', 'four'])
         data
```

Out[80]:

	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

- Calling `drop` with a sequence of labels will drop values from the row labels (axis 0):

```
In [81]: data.drop(['Colorado', 'Ohio'])
```

Out[81]:

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

- You can drop values from the columns by passing `axis=1` or `axis='columns'`:

```
In [82]: data.drop('two', axis=1)
```

Out[82]:

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

```
In [83]: data.drop(['two', 'four'], axis='columns')
```

Out[83]:

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

- Many functions, like `drop`, which modify the size or shape of a Series or DataFrame, can manipulate an object *in-place* without returning a new object:

```
In [84]: obj.drop('c', inplace=True)
         obj

Out[84]: a    0.0
         b    1.0
         d    3.0
         e    4.0
         dtype: float64
```

- Be careful with the `inplace`, as it destroys any data that is dropped.