intro_to_pandas

September 26, 2019

1 Intro to pandas

Learning Objectives: * Gain an introduction to the `DataFrame` and `Series` data structures of the *pandas* library * Access and manipulate data within a `DataFrame` and `Series` * Import CSV data into a *pandas* `DataFrame` * Reindex a `DataFrame` to shuffle data

pandas is a column-oriented data analysis API. It's a great tool for handling and analyzing input data, and many ML frameworks support *pandas* data structures as inputs. Although a comprehensive introduction to the *pandas* API would span many pages, the core concepts are fairly straightforward, and we'll present them below. For a more complete reference, the *pandas* docs site contains extensive documentation and many tutorials.

1.1 Basic Concepts

The following line imports the *pandas* API and prints the API version:

[1]:
```
from __future__ import print_function

import pandas as pd
pd.__version__
```

[1]:
```
'0.25.1'
```

The primary data structures in *pandas* are implemented as two classes:

- **`DataFrame`**, which you can imagine as a relational data table, with rows and named columns.
- **`Series`**, which is a single column. A `DataFrame` contains one or more `Series` and a name for each `Series`.

The data frame is a commonly used abstraction for data manipulation. Similar implementations exist in Spark and R.

One way to create a `Series` is to construct a `Series` object. For example:

[2]:
```
pd.Series(['San Francisco', 'San Jose', 'Sacramento'])
```

[2]:
```
0    San Francisco
1         San Jose
2       Sacramento
```

```
dtype: object
```

DataFrame objects can be created by passing a dict mapping string column names to their respective Series. If the Series don't match in length, missing values are filled with special NA/NaN values. Example:

[3]:
```
city_names = pd.Series(['San Francisco', 'San Jose', 'Sacramento'])
population = pd.Series([852469, 1015785, 485199])

pd.DataFrame({ 'City name': city_names, 'Population': population })
```

[3]:
```
       City name  Population
0  San Francisco      852469
1       San Jose     1015785
2     Sacramento      485199
```

But most of the time, you load an entire file into a DataFrame. The following example loads a file with California housing data. Run the following cell to load the data and create feature definitions:

[4]:
```
california_housing_dataframe = pd.read_csv("https://download.mlcc.google.com/
 ↪mledu-datasets/california_housing_train.csv", sep=",")
california_housing_dataframe.describe()
```

[4]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  17000.000000  17000.000000        17000.000000  17000.000000
mean    -119.562108     35.625225           28.589353   2643.664412
std        2.005166      2.137340           12.586937   2179.947071
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.790000     33.930000           18.000000   1462.000000
50%     -118.490000     34.250000           29.000000   2127.000000
75%     -118.000000     37.720000           37.000000   3151.250000
max     -114.310000     41.950000           52.000000  37937.000000

       total_bedrooms    population    households  median_income  \
count    17000.000000  17000.000000  17000.000000   17000.000000
mean       539.410824   1429.573941    501.221941       3.883578
std        421.499452   1147.852959    384.520841       1.908157
min          1.000000      3.000000      1.000000       0.499900
25%        297.000000    790.000000    282.000000       2.566375
50%        434.000000   1167.000000    409.000000       3.544600
75%        648.250000   1721.000000    605.250000       4.767000
max       6445.000000  35682.000000   6082.000000      15.000100

       median_house_value
count        17000.000000
mean        207300.912353
std         115983.764387
min          14999.000000
```

```
25%          119400.000000
50%          180400.000000
75%          265000.000000
max          500001.000000
```

The example above used `DataFrame.describe` to show interesting statistics about a `DataFrame`. Another useful function is `DataFrame.head`, which displays the first few records of a `DataFrame`:

[5]:
```
california_housing_dataframe.head()
```

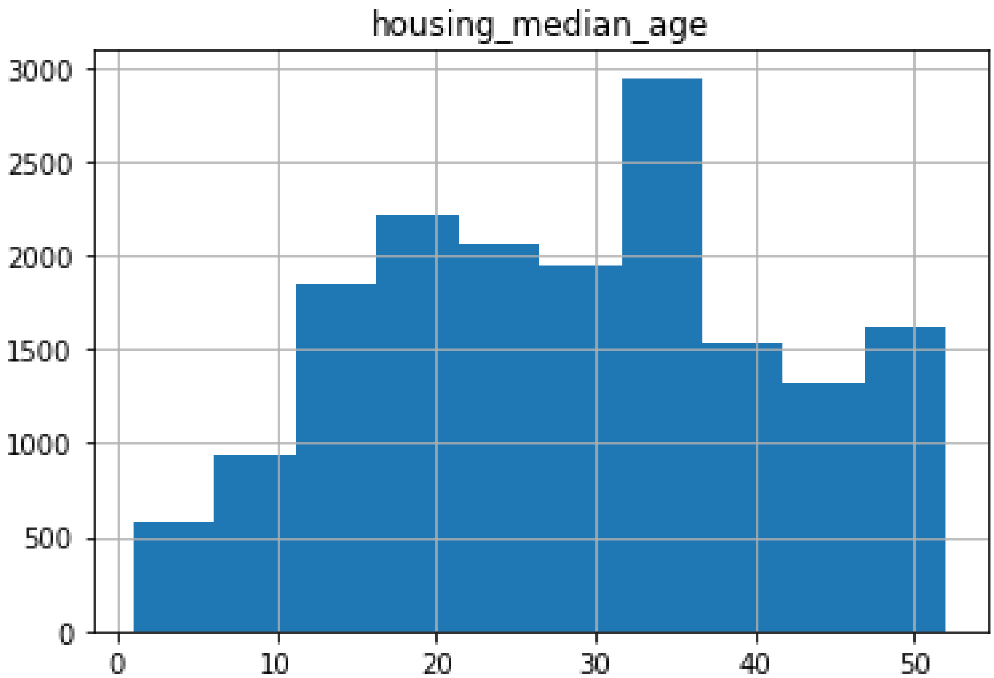
[5]:
```
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -114.31     34.19                15.0       5612.0          1283.0
1    -114.47     34.40                19.0       7650.0          1901.0
2    -114.56     33.69                17.0        720.0           174.0
3    -114.57     33.64                14.0       1501.0           337.0
4    -114.57     33.57                20.0       1454.0           326.0

   population  households  median_income  median_house_value
0      1015.0       472.0         1.4936             66900.0
1      1129.0       463.0         1.8200             80100.0
2       333.0       117.0         1.6509             85700.0
3       515.0       226.0         3.1917             73400.0
4       624.0       262.0         1.9250             65500.0
```

Another powerful feature of *pandas* is graphing. For example, `DataFrame.hist` lets you quickly study the distribution of values in a column:

[7]:
```
california_housing_dataframe.hist('housing_median_age')
```

[7]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x11816e390>]],
      dtype=object)
```



1.2 Accessing Data

You can access `DataFrame` data using familiar Python dict/list operations:

[11]:
```
cities = pd.DataFrame({ 'City name': city_names, 'Population': population })
print(type(cities['City name']))
cities['City name']
```

```
<class 'pandas.core.series.Series'>
```

[11]:
```
0    San Francisco
1         San Jose
2       Sacramento
Name: City name, dtype: object
```

[9]:
```
print(type(cities['City name'][1]))
cities['City name'][1]
```

```
<class 'str'>
```

[9]:
```
'San Jose'
```

[10]:
```
print(type(cities[0:2]))
cities[0:2]
```

```
<class 'pandas.core.frame.DataFrame'>
```

[10]:

```
        City name  Population
0  San Francisco      852469
1        San Jose     1015785
```

In addition, *pandas* provides an extremely rich API for advanced indexing and selection that is too extensive to be covered here.

1.3 Manipulating Data

You may apply Python's basic arithmetic operations to `Series`. For example:

[11]:

```python
population / 1000.
```

[11]:

```
0     852.469
1    1015.785
2     485.199
dtype: float64
```

NumPy is a popular toolkit for scientific computing. *pandas* `Series` can be used as arguments to most NumPy functions:

[12]:

```python
import numpy as np

np.log(population)
```

[12]:

```
0    13.655892
1    13.831172
2    13.092314
dtype: float64
```

For more complex single-column transformations, you can use `Series.apply`. Like the Python map function, `Series.apply` accepts as an argument a lambda function, which is applied to each value.

The example below creates a new `Series` that indicates whether `population` is over one million:

[13]:

```python
population.apply(lambda val: val > 1000000)
```

[13]:

```
0    False
1     True
2    False
dtype: bool
```

Modifying `DataFrames` is also straightforward. For example, the following code adds two `Series` to an existing `DataFrame`:

[14]:
```
cities['Area square miles'] = pd.Series([46.87, 176.53, 97.92])
cities['Population density'] = cities['Population'] / cities['Area square␣
 ↪miles']
cities
```

[14]:

	City name	Population	Area square miles	Population density
0	San Francisco	852469	46.87	18187.945381
1	San Jose	1015785	176.53	5754.177760
2	Sacramento	485199	97.92	4955.055147

2 Exercise #1

Modify the `cities` table by adding a new boolean column that is True if and only if *both* of the following are True:

- The city is named after a saint.
- The city has an area greater than 50 square miles.

Note: Boolean `Series` are combined using the bitwise, rather than the traditional boolean, operators. For example, when performing *logical and*, use `&` instead of `and`.

Hint: "San" in Spanish means "saint."

[0]:
```
# Your code here
```

2.1 Indexes

Both `Series` and `DataFrame` objects also define an `index` property that assigns an identifier value to each `Series` item or `DataFrame` row.

By default, at construction, *pandas* assigns index values that reflect the ordering of the source data. Once created, the index values are stable; that is, they do not change when data is reordered.

[17]:
```
city_names.index
```

[17]:
```
RangeIndex(start=0, stop=3, step=1)
```

[18]:
```
cities.index
```

[18]:
```
RangeIndex(start=0, stop=3, step=1)
```

Call `DataFrame.reindex` to manually reorder the rows. For example, the following has the same effect as sorting by city name:

[19]:
```
cities.reindex([2, 0, 1])
```

[19]:
```
        City name  Population  …  Population density  Is wide and has saint
name
2      Sacramento      485199  …         4955.055147
False
0   San Francisco      852469  …        18187.945381
False
1        San Jose     1015785  …         5754.177760
True

[3 rows x 5 columns]
```

Reindexing is a great way to shuffle (randomize) a `DataFrame`. In the example below, we take the index, which is array-like, and pass it to NumPy's `random.permutation` function, which shuffles its values in place. Calling `reindex` with this shuffled array causes the `DataFrame` rows to be shuffled in the same way. Try running the following cell multiple times!

[20]:
```
cities.reindex(np.random.permutation(cities.index))
```

[20]:
```
        City name  Population  …  Population density  Is wide and has saint
name
0   San Francisco      852469  …        18187.945381
False
1        San Jose     1015785  …         5754.177760
True
2      Sacramento      485199  …         4955.055147
False

[3 rows x 5 columns]
```

For more information, see the Index documentation.