## Copyright 2017 Google LLC.



# 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.

## Basic Concepts

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

```
from __future__ import print_function

import pandas as pd
pd.__version__
```

```
u'0.24.2'
```

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:

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

```
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:

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

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

| | 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:

```
california_housing_dataframe = pd.read_csv("https://download.mlcc.google.com/mledu-dataset
california_housing_dataframe.describe()
```

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms |
|---|---|---|---|---|---|
| **count** | 17000.000000 | 17000.000000 | 17000.000000 | 17000.000000 | 17000.000000 |
| **mean** | -119.562108 | 35.625225 | 28.589353 | 2643.664412 | 539.410824 |
| **std** | 2.005166 | 2.137340 | 12.586937 | 2179.947071 | 421.499452 |
| **min** | -124.350000 | 32.540000 | 1.000000 | 2.000000 | 1.000000 |
| **25%** | -121.790000 | 33.930000 | 18.000000 | 1462.000000 | 297.000000 |
| **50%** | -118.490000 | 34.250000 | 29.000000 | 2127.000000 | 434.000000 |
| **75%** | -118.000000 | 37.720000 | 37.000000 | 3151.250000 | 648.250000 |
| **max** | -114.310000 | 41.950000 | 52.000000 | 37937.000000 | 6445.000000 |

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`:

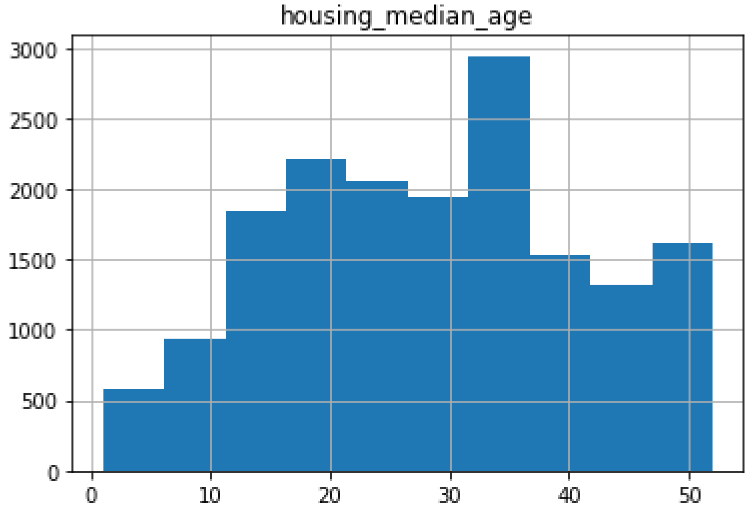
```
california_housing_dataframe.head()
```

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population |
|---|---|---|---|---|---|---|
| **0** | -114.31 | 34.19 | 15.0 | 5612.0 | 1283.0 | 1015.0 |
| **1** | -114.47 | 34.40 | 19.0 | 7650.0 | 1901.0 | 1129.0 |
| **2** | -114.56 | 33.69 | 17.0 | 720.0 | 174.0 | 333.0 |

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

```
california_housing_dataframe.hist('housing_median_age')
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f5535fb39d0>]],
      dtype=object)
```




## Accessing Data

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

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

```
<class 'pandas.core.series.Series'>
0    San Francisco
1         San Jose
2       Sacramento
Name: City name, dtype: object
```

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

```
<type 'str'>
'San Jose'
```

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

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

| | City name | Population |
|---|---|---|
| **0** | San Francisco | 852469 |

In addition, *pandas* provides an extremely rich API for advanced [indexing and selection](http://pandas.pydata.org/pandas-docs/stable/indexing.html) that is too extensive to be covered here.

## Manipulating Data

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

```
population / 1000.
```

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

[NumPy](http://www.numpy.org/) is a popular toolkit for scientific computing. *pandas* `Series` can be used as arguments to most NumPy functions:

```
import numpy as np

np.log(population)
```

```
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](https://docs.python.org/2/library/functions.html#map), `Series.apply` accepts as an argument a [lambda function](https://docs.python.org/2/tutorial/controlflow.html#lambda-expressions), which is applied to each value.

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

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

```
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`:

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

| | City name | Population | Area square miles | Population density |
|---|---|---|---|---|
| 0 | San Francisco | 852469 | 46.87 | 18187.945381 |
| 1 | San Jose | 1015785 | 176.53 | 5754.177760 |

## 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."

```
# Your code here
cities['area grater than 50'] = cities['Area square miles']>50
cities['named after saint'] = cities['City name'].apply(lambda val: val.startswith('San'))
cities
```

| | City name | Population | Area square miles | Population density | area grater than 50 | named after saint |
|---|---|---|---|---|---|---|
| 0 | San Francisco | 852469 | 46.87 | 18187.945381 | False | True |
| 1 | San Jose | 1015785 | 176.53 | 5754.177760 | True | True |

### Solution

Click below for a solution.

↳ *1 cells hidden*

## 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.

```
city_names.index
```

```
RangeIndex(start=0, stop=3, step=1)
```

```
cities.index
```

```
RangeIndex(start=0, stop=3, step=1)
```

Call `DataFrame.reindex` to manually reorder the rows. For example, the following has the same effect

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

| | City name | Population | Area square miles | Population density | area grater than 50 | named after saint |
|---|---|---|---|---|---|---|
| **2** | Sacramento | 485199 | 97.92 | 4955.055147 | True | False |
| **0** | San Francisco | 852469 | 46.87 | 18187.945381 | False | True |

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!

```
cities.reindex(np.random.permutation(cities.index))
```

| | City name | Population | Area square miles | Population density | area grater than 50 | named after saint |
|---|---|---|---|---|---|---|
| **2** | Sacramento | 485199 | 97.92 | 4955.055147 | True | False |
| **1** | San Jose | 1015785 | 176.53 | 5754.177760 | True | True |
| | San | | | | | |

For more information, see the [Index documentation](#).

## Exercise #2

The `reindex` method allows index values that are not in the original `DataFrame`'s index values. Try it and see what happens if you use such values! Why do you think this is allowed?

```
# Your code here
cities.reindex([2,1,3])
```

| | City name | Population | Area square miles | Population density | area grater than 50 | named after saint |
|---|---|---|---|---|---|---|
| **2** | Sacramento | 485199.0 | 97.92 | 4955.055147 | True | False |
| **1** | San Jose | 1015785.0 | 176.53 | 5754.177760 | True | True |
| **3** | NaN | NaN | NaN | NaN | NaN | NaN |

### Solution

Click below for the solution.

If your `reindex` input array includes values not in the original `DataFrame` index values, `reindex` will add new rows for these "missing" indices and populate all corresponding columns with `NaN` values:

```
cities.reindex([0, 4, 5, 2])
```

This behavior is desirable because indexes are often strings pulled from the actual data (see the *[pandas reindex documentation]()* for an example in which the index values are browser names).

In this case, allowing "missing" indices makes it easy to reindex using an external list, as you don't have to worry about sanitizing the input.