Getting Started with pandas

Part 7

Summarizing and Computing Descriptive Statistics

Part 1

- pandas objects are equipped with a set of common mathematical and statistical methods.
- Most of these fall into the category of *reductions* or *summary* statistics, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame.
- Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data.

In [201]:
```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
                   [np.nan, np.nan], [0.75, -1.3]],
                  index=['a', 'b', 'c', 'd'],
                  columns=['one', 'two'])
df
```

Out[201]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

In [202]:
```
df.sum()
```

Out[202]:
```
one    9.25
two   -5.80
dtype: float64
```

In [203]:
```
df.sum(axis='columns')
```

Out[203]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

- NA values are excluded unless the entire slice (row or column in this case) is NA.
- This can be disabled with the `skipna` option:

```
In [205]: df
```

Out[205]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
In [206]: df.mean(axis='columns', skipna=False)
```

```
Out[206]: a      NaN
          b    1.300
          c      NaN
          d   -0.275
          dtype: float64
```

- Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [207]: df
Out[207]:
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
In [208]: df.idxmax()
Out[208]: one    b
          two    d
          dtype: object
```

- Other methods are accumulations:

```
In [209]: df
```

Out[209]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
In [210]: df.cumsum()
```

Out[210]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

- Another type of method is neither a reduction nor an accumulation.
- `describe` is one such example, producing multiple summary statistics in one shot:

```
In [211]: df
```

Out[211]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
In [212]: df.describe()
```

Out[212]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

- On non-numeric data, `describe` produces alternative summary statistics:

```
In [213]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [214]: obj

Out[214]: 0     a
          1     a
          2     b
          3     c
          4     a
          5     a
          6     b
          7     c
          8     a
          9     a
          10    b
          11    c
          12    a
          13    a
          14    b
          15    c
          dtype: object

In [215]: obj.describe()

Out[215]: count     16
          unique     3
          top        a
          freq       8
          dtype: object
```

Correlation and Covariance

- Some summary statistics, like correlation and covariance, are computed from pairs of arguments.
- Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance using the add-on `pandas-datareader` package.
- If you don't have it installed already, it can be obtained via `conda` or `pip`:

```
(base) joshua@joshua-Virtual-Machine:~$ conda install pandas-datareader
Collecting package metadata: done
Solving environment: done

## Package Plan ##

  environment location: /home/joshua/anaconda3

  added / updated specs:
    - pandas-datareader
```

```
(base) joshua@joshua-Virtual-Machine:~$ pip install fix_yahoo_finance --upgrade --no-cache-dir
Collecting fix_yahoo_finance
  Downloading https://files.pythonhosted.org/packages/0c/b3/55468a37787c0a32d0e26155bdd2c712469d95071a125884fc1ff149b75c/fix-yahoo-finance-0.1.33.tar.gz
Requirement already satisfied, skipping upgrade: pandas>=0.24 in ./anaconda3/lib/python3.7/site-packages (from fix_yahoo_finance) (0.24.2)
Requirement already satisfied, skipping upgrade: numpy>1.15 in ./anaconda3/lib/python3.7/site-packages (from fix_yahoo_finance) (1.16.2)
Collecting requests<2.20 (from fix_yahoo_finance)
  Downloading https://files.pythonhosted.org/packages/65/47/7e02164a2a3db50ed6d8a6ab1d6d60b69c4c3fdf57a284257925dfc12bda/requests-2.19.1-py2.py3-none-any.whl (91kB)
    100% |                                        | 92kB 297kB/s
Collecting multitasking>=0.0.7 (from fix_yahoo_finance)
  Downloading https://files.pythonhosted.org/packages/ac/1a/0750416c5e3683d170757e423f097fdf78ceb9ccdc65658b24341664e53e/multitasking-0.0.7.tar.gz
Requirement already satisfied, skipping upgrade: python-dateutil>=2.5.0 in ./anaconda3/lib/python3.7/site-packages (from pandas>=0.24->fix_yahoo_finance) (2.8.0)
```

- The `pandas_datareader` module (together with `fix-yahoo-finance`) can be used to download some data for a few stock tickers:

```
In [217]: import pandas_datareader.data as web
          import fix_yahoo_finance as yf
          yf.pdr_override()
          all_data = {ticker: web.get_data_yahoo(ticker)
                      for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}

          price = pd.DataFrame({ticker: data['Adj Close']
                                for ticker, data in all_data.items()})
          volume = pd.DataFrame({ticker: data['Volume']
                                 for ticker, data in all_data.items()})
```

```
[**********************100%**********************]  1 of 1 downloaded
[**********************100%**********************]  1 of 1 downloaded
[**********************100%**********************]  1 of 1 downloaded
[**********************100%**********************]  1 of 1 downloaded
```

- We can now compute percent changes of the prices.

```
In [218]: returns = price.pct_change()
          returns.tail()
```

Out[218]:

Date	AAPL	IBM	MSFT	GOOG
2019-04-29	0.001517	-0.002797	-0.000924	0.012105
2019-04-30	-0.019256	0.008774	0.006396	-0.076966
2019-05-01	0.049086	0.002067	-0.020827	-0.017165
2019-05-02	-0.006508	-0.006901	-0.013059	-0.004683
2019-05-03	0.012431	0.004728	0.021314	0.019602

- The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series.
- Relatedly, `cov` computes the covariance:

```
In [219]: returns['MSFT'].corr(returns['IBM'])
Out[219]: 0.4177864219010938

In [220]: returns['MSFT'].cov(returns['IBM'])
Out[220]: 0.00016455420082779368
```

- Since `MSFT` is a valid Python attribute, we can also select these columns using more concise syntax:

```
In [221]: returns.MSFT.corr(returns.IBM)
Out[221]: 0.4177864219010938
```

- DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [222]: returns.corr()
```

Out[222]:

	AAPL	IBM	MSFT	GOOG
AAPL	1.000000	0.156030	0.294955	0.464469
IBM	0.156030	1.000000	0.417786	0.393765
MSFT	0.294955	0.417786	1.000000	0.479980
GOOG	0.464469	0.393765	0.479980	1.000000

```
In [223]: returns.cov()
```

Out[223]:

	AAPL	IBM	MSFT	GOOG
AAPL	0.004226	0.000170	0.000231	0.000184
IBM	0.000170	0.000257	0.000165	0.000101
MSFT	0.000231	0.000165	0.000520	0.000150
GOOG	0.000184	0.000101	0.000150	0.000363

- Using DataFrame's `corrwith` method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame.
- Passing a Series returns a Series with the correlation value computed for each column:

```
In [224]: returns.corrwith(returns.IBM)

Out[224]: AAPL    0.156030
          IBM     1.000000
          MSFT    0.417786
          GOOG    0.393765
          dtype: float64
```

- Passing a DataFrame computes the correlations of matching column names.
- Here I compute correlations of percent changes with volume:

```
In [225]: returns.corrwith(volume)

Out[225]: AAPL   -0.019328
          IBM    -0.002021
          MSFT   -0.004415
          GOOG    0.046090
          dtype: float64
```

- Passing `axis='columns'` does things row-by-row instead.
- In all cases, the data points are aligned by label before the correlation is computed.

Unique Values, Value Counts, and Membership

- Another class of related methods extracts information about the values contained in a one-dimensional Series.
- To illustrate these, consider this example:

```
In [226]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

- The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [227]: uniques = obj.unique()
          uniques
Out[227]: array(['c', 'a', 'd', 'b'], dtype=object)
```

- The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`).

- `value_counts` computes a Series containing value frequencies:

```
In [228]: obj.value_counts()

Out[228]: c    3
          a    3
          b    2
          d    1
          dtype: int64
```

- `value_counts` is also available as a top-level pandas method that can be used with any array or sequence:

```
In [229]: pd.value_counts(obj.values, sort=False)

Out[229]: b    2
          a    3
          c    3
          d    1
          dtype: int64
```

- `isin` performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

```
In [230]: obj
Out[230]: 0    c
          1    a
          2    d
          3    a
          4    a
          5    b
          6    b
          7    c
          8    c
          dtype: object

In [231]: mask = obj.isin(['b', 'c'])
          mask
Out[231]: 0     True
          1    False
          2    False
          3    False
          4    False
          5     True
          6     True
          7     True
          8     True
          dtype: bool

In [232]: obj[mask]
Out[232]: 0    c
          5    b
          6    b
          7    c
          8    c
          dtype: object
```

- Related to `isin` is the `Index.get_indexer` method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

```
In [233]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
          unique_vals = pd.Series(['c', 'b', 'a'])
          pd.Index(unique_vals).get_indexer(to_match)

Out[233]: array([0, 2, 1, 1, 0, 2])
```

- In some cases, you may want to compute a histogram on multiple related columns in a DataFrame.

```
In [234]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
                               'Qu2': [2, 3, 1, 2, 3],
                               'Qu3': [1, 5, 2, 4, 4]})
          data
```

Out[234]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

```
In [235]: result = data.apply(pd.value_counts).fillna(0)
          result
```

Out[235]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0