Introduction to Pandas

October 30, 2019

1 Pandas

pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. In this notebook we will cover the following topics:

- Series
- DataFrame
- Dropping Entries
- Indexing, Selecting, Filtering
- Arithmetic and Data Alignment
- Function Application and Mapping
- Sorting
- Axis Indices with Duplicate Values
- Summarising and Computing Descriptive Statistics
- Cleaning Data
- Input and Output

For help please refer to The official documentation page.

1.1 Imports

```
In [1]: import pandas as pd
        import numpy as np
```

1.2 Series

A Series is a one-dimensional array-like object containing an array of data and an associated array of data labels. The data can be any NumPy data type and the labels are the Series' indices.

```
In [2]: # 1) Create a List
        myList = [1, 2, 3, -3, 0, 2, 1]
        # 2) Convert the list to a series
        Series1 = pd.Series(myList)
        Series1
```

```
Out[2]: 0    1
        1    2
```

```
2    3
3   -3
4    0
5    2
6    1
dtype: int64
```

Note that each element of the list now has an index when it's converted to a series.
Get the array representation of a Series:

```
In [3]: Series1.values

Out[3]: array([ 1,  2,  3, -3,  0,  2,  1])
```

Get the index of the Series:

```
In [4]: Series1.index

Out[4]: RangeIndex(start=0, stop=7, step=1)
```

Index objects are immutable and hold the axis labels and metadata such as names and axis names. Now let's create a series with a custom index:

```
In [5]: Series2 = pd.Series(myList, index=['a', 'John', 'c', 'd', '-1', 'f', 'g'])
        Series2

Out[5]: a       1
        John    2
        c       3
        d      -3
        -1      0
        f       2
        g       1
        dtype: int64
```

Get a value from a Series:

```
In [6]: Series2[2]

Out[6]: 3
```

Verify the index number agains the index name:

```
In [7]: Series2[2] == Series2['c']

Out[7]: True
```

Get a set of values from a Series by passing in a list od indices:

```
In [8]: Series2[['a', '-1', 'John']]
```

```
Out[8]: a       1
        -1      0
        John    2
        dtype: int64
```

Get values greater than 1:

```
In [9]: Series2[Series2 > 1]
```

```
Out[9]: John    2
        c       3
        f       2
        dtype: int64
```

Multiply by a scalar:

```
In [10]: Series2 * 5
```

```
Out[10]: a        5
         John    10
         c       15
         d      -15
         -1       0
         f       10
         g        5
         dtype: int64
```

Apply a function

```
In [11]: np.exp(Series2)
```

```
Out[11]: a        2.718282
         John     7.389056
         c       20.085537
         d        0.049787
         -1       1.000000
         f        7.389056
         g        2.718282
         dtype: float64
```

A Series is like a fixed-length, ordered dictionary. We can create a series from dictionaries:

```
In [12]: # 1) Create a dictionary with keys as: A, B, C, and values as: 1, 2, 100
         dict1 = {"A": 1, 'B': 2, 'C': 100}
         # 2) Create a Series from the dictionary
         Series3 = pd.Series(dict1)
         Series3
```

```
Out[12]: A      1
         B      2
         C    100
         dtype: int64
```

Note that the keys have become the indices in the Series.

We can also re-order a Series by passing in an index list (indices which are not found are considered as NaN) when creating from a dictionary:

```
In [13]: index_list = ['C', 'B', 'A', 'D']
         Series4 = pd.Series(dict1, index=index_list)
         Series4

Out[13]: C    100.0
         B      2.0
         A      1.0
         D      NaN
         dtype: float64
```

We can also check for NaNs:

```
In [14]: Series4.isnull()

Out[14]: C    False
         B    False
         A    False
         D     True
         dtype: bool
```

Or:

```
In [15]: pd.isnull(Series4)

Out[15]: C    False
         B    False
         A    False
         D     True
         dtype: bool
```

Series automatically aligns differently indexed data in arithmetic operations:

```
In [16]: Series3

Out[16]: A      1
         B      2
         C    100
         dtype: int64

In [17]: Series4

Out[17]: C    100.0
         B      2.0
         A      1.0
         D      NaN
         dtype: float64
```

```
In [18]: Series3 + Series4

Out[18]: A      2.0
         B      4.0
         C    200.0
         D      NaN
         dtype: float64
```

We can also name a Series and its index:

```
In [19]: Series4.name = 'mySeries'
         Series4.index.name = "myIndex"
         Series4

Out[19]: myIndex
         C    100.0
         B      2.0
         A      1.0
         D      NaN
         Name: mySeries, dtype: float64
```

We can rename a Series' index in place:

```
In [20]: Series4.index = ["CC", "BB", "AA", "DD"]
         Series4

Out[20]: CC    100.0
         BB      2.0
         AA      1.0
         DD      NaN
         Name: mySeries, dtype: float64
```

1.3 DataFrame

A DataFrame is a tabular data structure containing an ordered collection of columns. Each column can have a different type. DataFrames have both row and column indices. Row and column operations are treated roughly symmetrically. Columns returned when indexing a DataFrame are views of the underlying data, not a copy. To obtain a copy, use the copy() method.

Pandas can create DataFrames in different ways (e.g., reading in a file (txt, json, csv), or from a dictionary). Let's start by creating a DataFrame from a dictionary:

```
In [21]: # 1) Create a dictionary
         dict2 = {'City': ['London', 'London', 'London', 'New York', 'New York'],
                 'Year': [2015, 2016, 2017, 2016, 2017],
                 'Population': [8.60, 8.71, 8.79, 8.61, 8.62]}

         # 2) Create a DataFrame from the dictionary
         df1 = pd.DataFrame(dict2)
         df1
```

```
Out[21]:        City  Year  Population
         0    London  2015        8.60
         1    London  2016        8.71
         2    London  2017        8.79
         3  New York  2016        8.61
         4  New York  2017        8.62
```

Create a DataFrame specifying a sequence of columns:

```
In [22]: df2 = pd.DataFrame(df1, columns=['Year', 'City', 'Population'])
         df2
```

```
Out[22]:    Year      City  Population
         0  2015    London        8.60
         1  2016    London        8.71
         2  2017    London        8.79
         3  2016  New York        8.61
         4  2017  New York        8.62
```

Like Series, columns that are not present in the data are `NaN`:

```
In [23]: df3 = pd.DataFrame(df1, columns=['Year', 'City', 'Population', 'Unemployment'])
         df3
```

```
Out[23]:    Year      City  Population  Unemployment
         0  2015    London        8.60           NaN
         1  2016    London        8.71           NaN
         2  2017    London        8.79           NaN
         3  2016  New York        8.61           NaN
         4  2017  New York        8.62           NaN
```

We can retrieve a column by the column name, returning a Series:

```
In [24]: df3['City']
```

```
Out[24]: 0      London
         1      London
         2      London
         3    New York
         4    New York
         Name: City, dtype: object
```

We can retrieve a column by attribute, returning a Series:

```
In [25]: df3.Year
```

```
Out[25]: 0    2015
         1    2016
         2    2017
         3    2016
         4    2017
         Name: Year, dtype: int64
```

We can retrieve a row by position:

```
In [26]: df3.iloc[2]

Out[26]: Year                2017
         City              London
         Population          8.79
         Unemployment         NaN
         Name: 2, dtype: object
```

We can update a column by assignment:

```
In [27]: df3['Unemployment'] = np.arange(5)
         df3

Out[27]:    Year      City  Population  Unemployment
         0  2015    London        8.60             0
         1  2016    London        8.71             1
         2  2017    London        8.79             2
         3  2016  New York        8.61             3
         4  2017  New York        8.62             4
```

We can assign a Series to a column (note if assigning a list or array, the length must match the DataFrame, unlike a Series):

```
In [28]: unemployment = pd.Series([5.9, 6.0, 6.2], index=[2, 3, 4])
         df3['Unemployment'] = unemployment
         df3

Out[28]:    Year      City  Population  Unemployment
         0  2015    London        8.60           NaN
         1  2016    London        8.71           NaN
         2  2017    London        8.79           5.9
         3  2016  New York        8.61           6.0
         4  2017  New York        8.62           6.2
```

We can assign a new column that doesn't exist to any existing column to create a new column (a copy):

```
In [29]: df3['Misc'] = df3['City']
         df3

Out[29]:    Year      City  Population  Unemployment      Misc
         0  2015    London        8.60           NaN    London
         1  2016    London        8.71           NaN    London
         2  2017    London        8.79           5.9    London
         3  2016  New York        8.61           6.0  New York
         4  2017  New York        8.62           6.2  New York
```

We can also delete the column:

```
In [30]: del df3['Misc']
         df3
```

```
Out[30]:    Year      City  Population  Unemployment
         0  2015    London        8.60           NaN
         1  2016    London        8.71           NaN
         2  2017    London        8.79           5.9
         3  2016  New York        8.61           6.0
         4  2017  New York        8.62           6.2
```

We can create a DataFrame from a nested dictionary of dicts (the keys in the inner dicts are unioned and sorted to form the index in the result, unless an explicit index is specified):

```
In [31]: population = {'London': {2015:8.6, 2016:8.71, 2017:8.79},
                       'New York': {2016:8.61, 2017:8.62}
                       }
         df4 = pd.DataFrame(population)
         df4
```

```
Out[31]:       London  New York
         2015    8.60       NaN
         2016    8.71      8.61
         2017    8.79      8.62
```

We can transpose a DataFrame:

```
In [32]: df4.T
```

```
Out[32]:           2015  2016  2017
         London     8.6  8.71  8.79
         New York   NaN  8.61  8.62
```

We can set an index name for the DataFrame:

```
In [33]: df4.index.name = 'year'
         df4
```

```
Out[33]:       London  New York
         year
         2015    8.60       NaN
         2016    8.71      8.61
         2017    8.79      8.62
```

We can also set a name for the DataFrame columns

```
In [34]: df4.columns.name = 'City'
         df4
```

```
Out[34]: City  London  New York
         year
         2015    8.60       NaN
         2016    8.71      8.61
         2017    8.79      8.62
```

Return the data contained in a DataFrame as a 2D ndarray:

```
In [35]: df4.values
```

```
Out[35]: array([[8.6 ,  nan],
                [8.71, 8.61],
                [8.79, 8.62]])
```

1.4 Dropping Entries

```
In [36]: df3
```

```
Out[36]:    Year      City  Population  Unemployment
         0  2015    London        8.60           NaN
         1  2016    London        8.71           NaN
         2  2017    London        8.79           5.9
         3  2016  New York        8.61           6.0
         4  2017  New York        8.62           6.2
```

Drop rows from a Series or DataFrame:

```
In [37]: df5 = df3.drop([0])
         df5
```

```
Out[37]:    Year      City  Population  Unemployment
         1  2016    London        8.71           NaN
         2  2017    London        8.79           5.9
         3  2016  New York        8.61           6.0
         4  2017  New York        8.62           6.2
```

Drop columns from a DataFrame:

```
In [38]: df6 = df5.drop('Unemployment', axis=1)
         df6
```

```
Out[38]:    Year      City  Population
         1  2016    London        8.71
         2  2017    London        8.79
         3  2016  New York        8.61
         4  2017  New York        8.62
```

1.5 Indexing, Selecting, Filtering in DataFrames

In [39]: df3

Out[39]:

	Year	City	Population	Unemployment
0	2015	London	8.60	NaN
1	2016	London	8.71	NaN
2	2017	London	8.79	5.9
3	2016	New York	8.61	6.0
4	2017	New York	8.62	6.2

Select specified columns from a DataFrame:

```
In [40]: df3[['Population', 'City']]
```

Out[40]:

	Population	City
0	8.60	London
1	8.71	London
2	8.79	London
3	8.61	New York
4	8.62	New York

Select a slice from a DataFrame:

```
In [41]: df3[:3]
```

Out[41]:

	Year	City	Population	Unemployment
0	2015	London	8.60	NaN
1	2016	London	8.71	NaN
2	2017	London	8.79	5.9

or

```
In [42]: df3.iloc[0:3]
```

Out[42]:

	Year	City	Population	Unemployment
0	2015	London	8.60	NaN
1	2016	London	8.71	NaN
2	2017	London	8.79	5.9

Select from a DataFrame based on a filter:

```
In [43]: df3[df3['Population'] > 8.7]
```

Out[43]:

	Year	City	Population	Unemployment
1	2016	London	8.71	NaN
2	2017	London	8.79	5.9

or

```
In [44]: df3.loc[df3.Population > 8.7]
```

```
Out[44]:    Year    City  Population  Unemployment
         1  2016  London        8.71           NaN
         2  2017  London        8.79           5.9
```

Select a slice of rows from a specific column of a DataFrame:

```
In [45]: df3.loc[0:2, "Population"]
```

```
Out[45]: 0    8.60
         1    8.71
         2    8.79
         Name: Population, dtype: float64
```

1.6 Arithmetic and Data Alignment

Adding DataFrame objects results in the union of index pairs for rows and columns if the pairs are not the same, resulting in NaN for indices that do not overlap:

```
In [46]: np.random.seed(0)
         df7 = pd.DataFrame(np.random.rand(9).reshape((3, 3)), columns=['a', 'b', 'c'])
         df7
```

```
Out[46]:           a         b         c
         0  0.548814  0.715189  0.602763
         1  0.544883  0.423655  0.645894
         2  0.437587  0.891773  0.963663
```

```
In [47]: np.random.seed(1)
         df8 = pd.DataFrame(np.random.rand(9).reshape((3, 3)), columns=['b', 'c', 'd'])
         df8
```

```
Out[47]:           b         c         d
         0  0.417022  0.720324  0.000114
         1  0.302333  0.146756  0.092339
         2  0.186260  0.345561  0.396767
```

```
In [48]: df7 + df8
```

```
Out[48]:     a         b         c   d
         0 NaN  1.132211  1.323088 NaN
         1 NaN  0.725987  0.792650 NaN
         2 NaN  1.078033  1.309223 NaN
```

```
In [49]: df9 = df8.add(df7, fill_value=0)
         df9
```

```
Out[49]:           a         b         c         d
         0  0.548814  1.132211  1.323088  0.000114
         1  0.544883  0.725987  0.792650  0.092339
         2  0.437587  1.078033  1.309223  0.396767
```

1.7 Function Application and Mapping

```
In [50]: df10 = df8.sub(df9, fill_value=0)
         df10
```

```
Out[50]:           a         b         c    d
         0 -0.548814 -0.715189 -0.602763  0.0
         1 -0.544883 -0.423655 -0.645894  0.0
         2 -0.437587 -0.891773 -0.963663  0.0
```

```
In [51]: df10 = np.abs(df10)
         df10
```

```
Out[51]:           a         b         c    d
         0  0.548814  0.715189  0.602763  0.0
         1  0.544883  0.423655  0.645894  0.0
         2  0.437587  0.891773  0.963663  0.0
```

Apply a function on 1D arrays to each column:

```
In [52]: myFunc = lambda x: x.max() - x.min()
         df10.apply(myFunc)
```

```
Out[52]: a    0.111226
         b    0.468118
         c    0.360899
         d    0.000000
         dtype: float64
```

Apply a function on 1D arrays to each row:

```
In [53]: df10.apply(myFunc, axis=1)
```

```
Out[53]: 0    0.715189
         1    0.645894
         2    0.963663
         dtype: float64
```

1.8 Sorting

```
In [54]: df11 = pd.DataFrame(np.arange(12).reshape((3, 4)),
                             index=['three', 'one', 'two'],
                             columns=['c', 'a', 'b', 'd'])
         df11
```

```
Out[54]:        c  a   b   d
         three  0  1   2   3
         one    4  5   6   7
         two    8  9  10  11
```

Sort a DataFrame by its index:

```
In [55]: df11.sort_index()
```

```
Out[55]:        c  a   b   d
         one    4  5   6   7
         three  0  1   2   3
         two    8  9  10  11
```

Sort a DataFrame by columns in descending order:

```
In [56]: df11.sort_index(axis=1, ascending=False)
```

```
Out[56]:         d  c   b  a
         three   3  0   2  1
         one     7  4   6  5
         two    11  8  10  9
```

1.9 Axis Indices with Duplicate Values

Labels do not have to be unique in Pandas:

Select DataFrame elements:

```
In [57]: df12 = pd.DataFrame(np.random.randn(5, 4),
                             index=['foo', 'foo', 'bar', 'bar', 'baz'])
         df12
```

```
Out[57]:             0         1         2         3
         foo -2.363469  1.135345 -1.017014  0.637362
         foo -0.859907  1.772608 -1.110363  0.181214
         bar  0.564345 -0.566510  0.729976  0.372994
         bar  0.533811 -0.091973  1.913820  0.330797
         baz  1.141943 -1.129595 -0.850052  0.960820
```

```
In [58]: df12.loc['bar']
```

```
Out[58]:             0         1         2         3
         bar  0.564345 -0.566510  0.729976  0.372994
         bar  0.533811 -0.091973  1.913820  0.330797
```

1.10 Summarising and Computing Descriptive Statistics

Unlike NumPy arrays, Pandas descriptive statistics automatically exclude missing data. NaN values are excluded unless the entire row or column is NA.

```
In [59]: df3
```

```
Out[59]:    Year      City  Population  Unemployment
         0  2015    London        8.60           NaN
         1  2016    London        8.71           NaN
         2  2017    London        8.79           5.9
         3  2016  New York        8.61           6.0
         4  2017  New York        8.62           6.2
```

```
In [60]: df3.sum()
```

```
Out[60]: Year                                        10081
         City          LondonLondonLondonNew YorkNew York
         Population                                  43.33
         Unemployment                                 18.1
         dtype: object
```

Sum over the rows:

```
In [61]: df3.sum(axis=1)
```

```
Out[61]: 0    2023.60
         1    2024.71
         2    2031.69
         3    2030.61
         4    2031.82
         dtype: float64
```

Account for NaNs:

```
In [62]: df3.sum(axis=1, skipna=False)
```

```
Out[62]: 0        NaN
         1        NaN
         2    2031.69
         3    2030.61
         4    2031.82
         dtype: float64
```

1.11 Cleaning Data

- Replace
- Drop
- Concatenate

1.11.1 Replace

Replace all occurrences of a string with another string, in place (no copy):

```
In [63]: df1
```

```
Out[63]:        City  Year  Population
         0    London  2015        8.60
         1    London  2016        8.71
         2    London  2017        8.79
         3  New York  2016        8.61
         4  New York  2017        8.62
```

```
In [64]: df1.replace('London', 'Lon', inplace=True)
         df1
```

```
Out[64]:        City  Year  Population
         0       Lon  2015        8.60
         1       Lon  2016        8.71
         2       Lon  2017        8.79
         3  New York  2016        8.61
         4  New York  2017        8.62
```

In a specified column, replace all occurrences of a string with another string, in place (no copy):

```
In [65]: df1.replace({'City' : { 'Lon' : 'London' }}, inplace=True)
         df1
```

```
Out[65]:        City  Year  Population
         0    London  2015        8.60
         1    London  2016        8.71
         2    London  2017        8.79
         3  New York  2016        8.61
         4  New York  2017        8.62
```

1.11.2 Drop

Drop the 'Population' column and return a copy of the DataFrame:

```
In [66]: df13 = df1.drop('Population', axis=1)
         df13
```

```
Out[66]:        City  Year
         0    London  2015
         1    London  2016
         2    London  2017
         3  New York  2016
         4  New York  2017
```

1.11.3 Concatenate

Concatenate two DataFrames:

```
In [67]: dict3 = {'City': ['Manchester', 'Manchester', 'Manchester', 'Beijing', 'Beijing'],
                  'Year': [2015, 2016, 2017, 2016, 2017],
                  'Population': [2.72, 2.75, 2.81, 21.01, 20.50]}
         df14 = pd.DataFrame(dict3)
         df14
```

```
Out[67]:          City  Year  Population
         0  Manchester  2015        2.72
         1  Manchester  2016        2.75
         2  Manchester  2017        2.81
         3     Beijing  2016       21.01
         4     Beijing  2017       20.50
```

```
In [68]: df15 = pd.concat([df1, df14], axis=0, sort=False)
         df15
```

```
Out[68]:          City  Year  Population
         0      London  2015        8.60
         1      London  2016        8.71
         2      London  2017        8.79
         3    New York  2016        8.61
         4    New York  2017        8.62
         0  Manchester  2015        2.72
         1  Manchester  2016        2.75
         2  Manchester  2017        2.81
         3     Beijing  2016       21.01
         4     Beijing  2017       20.50
```

1.12 Input and Output

- Reading
- Writing

1.12.1 Reading

```
In [69]: data = pd.read_csv("oscar_age_female.csv")
```

```
In [70]: data.head()
```

```
Out[70]:    Index   "Year"                "Age"                             "Name"   \
         1   1928       22       "Janet Gaynor"                   "Seventh Heaven
         2   1929       37      "Mary Pickford"                        "Coquette"
         3   1930       28      "Norma Shearer"                 "The Divorcee"\t
         4   1931       63     "Marie Dressler"                    "Min and Bill"
         5   1932       32        "Helen Hayes"   "The Sin of Madelon Claudet"\t

                                                    "Movie"
         1   Street Angel and Sunrise: A Song of Two Humans"
         2                                              NaN
         3                                              NaN
         4                                              NaN
         5                                              NaN
```

```
In [71]: data.describe()
```

```
Out[71]:              Index     "Year"
         count    89.000000  89.000000
         mean   1972.000000  36.123596
         std      25.836021  11.745231
         min    1928.000000  21.000000
         25%    1950.000000  28.000000
         50%    1972.000000  33.000000
         75%    1994.000000  41.000000
         max    2016.000000  80.000000
```

1.12.2 Writing

```
In [72]: data.to_csv("new.csv")
```