Data Science with Python

Noble Xavier

Pandas is an open-source Python library providing efficient, easy-to-use data structures and data analysis tools. The name Pandas is derived from “Panel Data” – an Econometrics from Multidimensional data

Pandas is well suited for many different kinds of data:

- ❑ Tabular data with heterogeneously-typed columns.
- ❑ Ordered and unordered time series data.
- ❑ Arbitrary matrix data with row and column labels
- ❑ Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

Data Structures in Pandas

- Pandas provides three data structures – **Series**, **DataFrames**, and **Panels**; all of which are built on top of the NumPy array

Data Structure	Dimensions	Description
Series	1	Labeled, homogenous array of immutable size
DataFrames	2	Labeled, heterogeneously typed, size-mutable tabular data structures
Panels	3	Labeled, size-mutable array

- All the above data structures are **value-mutable**

Series

- A Series is a single-dimensional array structures that stores homogenous data i.e., data of a single type
- All the elements of a Series are **value-mutable** and **size-immutable**

Data	Can be of multiple data types such as ndarray, lists, constants, series, dict etc.
Index	Indexes must be unique, hashable and have the same length as data Defaults to np.arrange(n) if no index is passed
dtype	Data type of each column; If none is mentioned, it will be inferred; automatically
copy	Deep copies data, set to false as default

Creating a Series

Creating an empty series

```
import pandas
series = pandas.Series()
print(series)
```

The Series() function creates a new Series

```
Series([], dtype: float64)
```

Output

Creating a Series (Contd...)

Creating a series from an ndarray

```
import pandas, numpy

arr = numpy.array([10, 20, 30, 40, 50])
series = pandas.Series(arr)
print(series)
```

note that indexes are assigned automatically if not specified

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Output

Creating a series from a Python dict

```python
import pandas

data = {'a':10, 'b':20, 'c':30}
series = pandas.Series(data)
print(series)
```

Note that the keys of the dictionary are used to assign indexes during conversion

```
a    10
b    20
c    30
dtype: int64
```

Output

Accessing Data From a Series

Retrieving a part of the series using slicing

```
import pandas

s = pandas.Series([10, 20, 30, 40, 50])

print(s[1:4])
```

```
1    20
2    30
3    40
dtype: int64
```

Output

DataFrames

- A DataFrame is a 2D data structure in which data is aligned in a tabular fashion consisting of rows & columns
- A DataFrame can be created using the following constructor –

pandas.DataFrame(data, index, dtype, copy)

Data	Can be of multiple data types such as ndarray, lists, constants, series, dict etc.
Index	Row and column labels of the dataframe; defaults to np.arrange(n) if no index is passed
dtype	Data type of each column
copy	Creates a deep copy of the data, set to false as default

Creating a DataFrame

Converting a list into a DataFrame

```python
import pandas

listx = [10, 20, 30, 40]
table = pandas.DataFrame(listx)
print(table)
```

list 'listx' converted to a DataFrame

```
    0
0  10
1  20
2  30
3  40
```

Output

Creating a DataFrame (Contd...)

Creating a DataFrame from a list of dictionaries and accompanying row indices

```
import pandas

data = [{'a':1, 'b':2}, {'a':2, 'b':4, 'c':8}]
table = pandas.DataFrame(data, index=['first', 'second'])
print(table)
```

Dict keys become column labels

```
        a  b    c
first   1  2  NaN
second  2  4  8.0
```

Creating a DataFrame (Contd...)

Converting a dictionary of series into a DataFrame

```
import pandas

data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
        'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
table = pandas.DataFrame(data)
print(table)
```

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

the resultant index is the union of all the series indexes passed

A new column can be added to a DataFrame when the data is passed as a Series

```
import pandas

data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
        'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
table = pandas.DataFrame(data)
table['three'] = pandas.Series([10, 20, 30], index=['a', 'b', 'c'])
print(table)
```

```
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
```

Column 'three' is newly added

DataFrame – Column Deletion

DataFrame columns can be deleted using the del() function

```
print(table)
```

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Deleted column

```
del table['one']
print(table)
```

	two	three
a	1	10.0
b	2	20.0
c	3	30.0
d	4	NaN

DataFrame columns can be deleted using the pop() function

```
print(table)
```

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Deleted column

```
table.pop('one')
print(table)
```

	two	three
a	1	10.0
b	2	20.0
c	3	30.0
d	4	NaN

DataFrame rows can be selected by passing the row label to the loc() function

```
print(table)
```

```
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
```

Selected row

```
print(table.loc['c'])
```

```
one       3.0
two       3.0
three    30.0
Name: c, dtype: float64
```

DataFrame – Row Addition

The append() function can be used to add more rows to the DataFrame

```python
import pandas

data = {'one': pandas.Series([1, 2, 3], index=['a', 'b', 'c']),
        'two': pandas.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
table = pandas.DataFrame(data)
table['three'] = pandas.Series([10, 20, 30], index=['a', 'b', 'c'])
print(table)
row  = pandas.DataFrame([[11, 13], [17, 19]], columns=['two', 'three'])
table = table.append(row)

print(table)
```

	one	three	two
a	1.0	10.0	1
b	2.0	20.0	2
c	3.0	30.0	3
d	NaN	NaN	4
0	NaN	13.0	11
1	NaN	19.0	17

Appended rows

The drop() function can be used to drop rows whose labels are provided

```
row  = pandas.DataFrame([[11, 13], [17, 19]], columns=['two', 'three'])
table = table.append(row)
table = table.drop('a')
print(table)
```

```
   one  three  two
b  2.0   20.0    2
c  3.0   30.0    3
d  NaN    NaN    4
0  NaN   13.0   11
1  NaN   19.0   17
```

Loading CSV data into DataFrames

- Data can be loaded into DataFrames from input data stored in the CSV format using the **read_csv()** function

```
table = pandas.read_csv("/home/edupy/Datasets/USArrests.csv")
```

Path to file

Storing Data in CSV Files

- Data present in DataFrames can be written to a CSV file using the **to_csv()** function
- If the specified path doesn't exist, a file of the same name is automatically created

```
table.to_csv("/home/edupy/Datasets/USArrests2.csv")
```

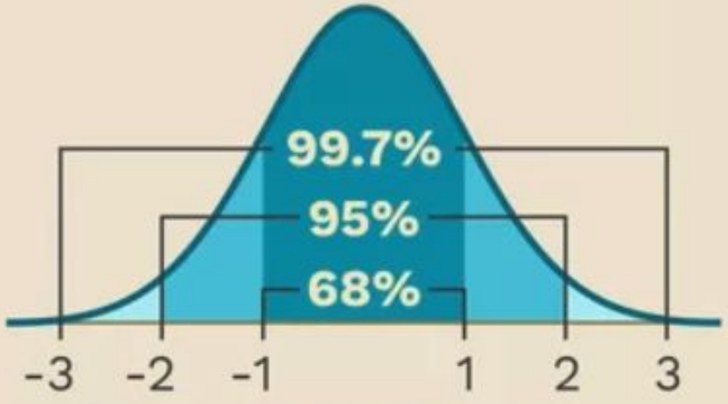
Calculating Standard Deviation

$$s_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

n = The number of data points

x_i = Each of the values of the data

$\bar{x}$ = The mean of x_i



Normal Distribution Curve

https://www.mathsisfun.com/data/percentiles.html

Percentiles

Percentile: the value below which a percentage of data falls.

Example: You are the fourth tallest person in a group of 20

80% of people are shorter than you:

That means you are at the **80th percentile**.

If your height is 1.85m then "1.85m" is the 80th percentile height in that group.

Grouped Data

When the data is grouped:

Add up all percentages **below** the score,
plus **half** the percentage **at** the score.

Example: You Score a B!

In the test 12% got D, 50% got C, 30% got B and 8% got A

You got a B, so add up

- all the 12% that got D,
- all the 50% that got C,
- half of the 30% that got B,

for a total percentile of 12% + 50% + 15% = **77%**

In other words you did "as well or better than 77% of the class"

(Why take half of B? Because you shouldn't imagine you got the "Best B", or the "Worst B", just an average B.)

Difference .loc and iloc

loc in Pandas	iloc in Pandas
Label-based data selector	Index-based data selector
Indices should be sorted in order, or loc[ ] will only select the mentioned indices when slicing	Indices need not be sorted in order when slicing
Indices should be numerical, else slicing cannot be done	Indices can be numerical or categorical
The end index is included during slicing	The end index is excluded during slicing
Accepts bool series or list in conditions	Only accepts bool list in conditions

Pandas Summary

Create Test Objects

pd.DataFrame(np.random.rand(20,5))	5 columns and 20 rows of random floats
pd.Series(my_list)	Create a series from an iterable my_list
df.index = pd.date_range('1900/1/30', periods=df.shape[0])	Add a date index

Viewing/Inspecting Data

df.head(n)	First n rows of the DataFrame
df.tail(n)	Last n rows of the DataFrame
df.shape	Number of rows and columns
df.info()	Index, Datatype and Memory information
df.describe()	Summary statistics for numerical columns
s.value_counts(dropna=False)	View unique values and counts
df.apply(pd.Series.value_counts)	Unique values and counts for all columns

Pandas Summary

Selection

df[col]	Returns column with label col as Series
df[[col1, col2]]	Returns columns as a new DataFrame
s.iloc[0]	Selection by position
s.loc['index_one']	Selection by index
df.iloc[0,:]	First row
df.iloc[0,0]	First element of first column

Pandas Summary

Data Cleaning

df.columns = ['a','b','c']	Rename columns
pd.isnull()	Checks for null Values, Returns Boolean Arrray
pd.notnull()	Opposite of pd.isnull()
df.dropna()	Drop all rows that contain null values
df.dropna(axis=1)	Drop all columns that contain null values
df.dropna(axis=1,thresh=n)	Drop all rows have have less than n non null values
df.fillna(x)	Replace all null values with x
s.fillna(s.mean())	Replace all null values with the mean
s.astype(float)	Convert the datatype of the series to float
s.replace(1,'one')	Replace all values equal to 1 with 'one'
s.replace([2,3],['two', 'three'])	Replace all 2 with 'two' and 3 with 'three'
df.rename(columns=lambda x: x + 1)	Mass renaming of columns
df.rename(columns={'old_name': 'new_ name'})	Selective renaming
df.set_index('column_one')	Change the index
df.rename(index=lambda x: x + 1)	Mass renaming of index

Pandas Summary

Filter, Sort, and Groupby

df[df[col] > 0.6]	Rows where the column col is greater than 0.6
df[(df[col] > 0.6) & (df[col] < 0.8)]	Rows where 0.8 > col > 0.6
df.sort_values(col1)	Sort values by col1 in ascending order
df.sort_values(col2,ascending=False)	Sort values by col2 in descending order.5
df.sort_values([col1,col2],ascending=[True,False])	Sort values by col1 in ascending order then col2 in descending order
df.groupby(col)	Returns a groupby object for values from one column
df.groupby([col1,col2])	Returns groupby object for values from multiple columns
df.groupby(col1)[col2]	Returns the mean of the values in col2, grouped by the values in col1
df.pivot_table(index=col1,values=[col2,col3],aggfunc=mean)	Create a pivot table that groups by col1 and calculates the mean of col2 and col3
df.groupby(col1).agg(np.mean)	Find the average across all columns for every unique col1 group
df.apply(np.mean)	Apply the function np.mean() across each column
nf.apply(np.max,axis=1)	Apply the function np.max() across each row

Join/Combine

df1.append(df2)	Add the rows in df1 to the end of df2 (columns should be identical)
pd.concat([df1, df2],axis=1)	Add the columns in df1 to the end of df2 (rows should be identical)
df1.join(df2,on=col1, how='inner')	SQL-style join the columns in df1 with the columns on df2 where the rows for col have identical values. The 'how' can be 'left', 'right', 'outer' or 'inner'

Pandas Summary

Statistics

df.describe()	Summary statistics for numerical columns
df.mean()	Returns the mean of all columns
df.corr()	Returns the correlation between columns in a DataFrame
df.count()	Returns the number of non-null values in each DataFrame column
df.max()	Returns the highest value in each column
df.min()	Returns the lowest value in each column
df.median()	Returns the median of each column
df.std()	Returns the standard deviation of each column

Importing Data

pd.read_csv(filename)	From a CSV file
pd.read_table(filename)	From a delimited text file (like TSV)
pd.read_excel(filename)	From an Excel file
pd.read_sql(query, connection_object)	Read from a SQL table/database
pd.read_json(json_string)	Read from a JSON formatted string, URL or file.
pd.read_html(url)	Parses an html URL, string or file and extracts tables to a list of dataframes
pd.read_clipboard()	Takes the contents of your clipboard and passes it to read_table()
pd.DataFrame(dict)	From a dict, keys for columns names, values for data as lists

Pandas Summary

Exporting Data

df.to_csv(filename)	Write to a CSV file
df.to_excel(filename)	Write to an Excel file
df.to_sql(table_name, connection_object)	Write to a SQL table
df.to_json(filename)	Write to a file in JSON format