13 Most Important Pandas Functions for Data Science

BEGINNER DATA EXPLORATION LIBRARIES PANDAS PYTHON STRUCTURED DATA TECHNIQUE

This article was published as a part of the Data Science Blogathon.

Introduction

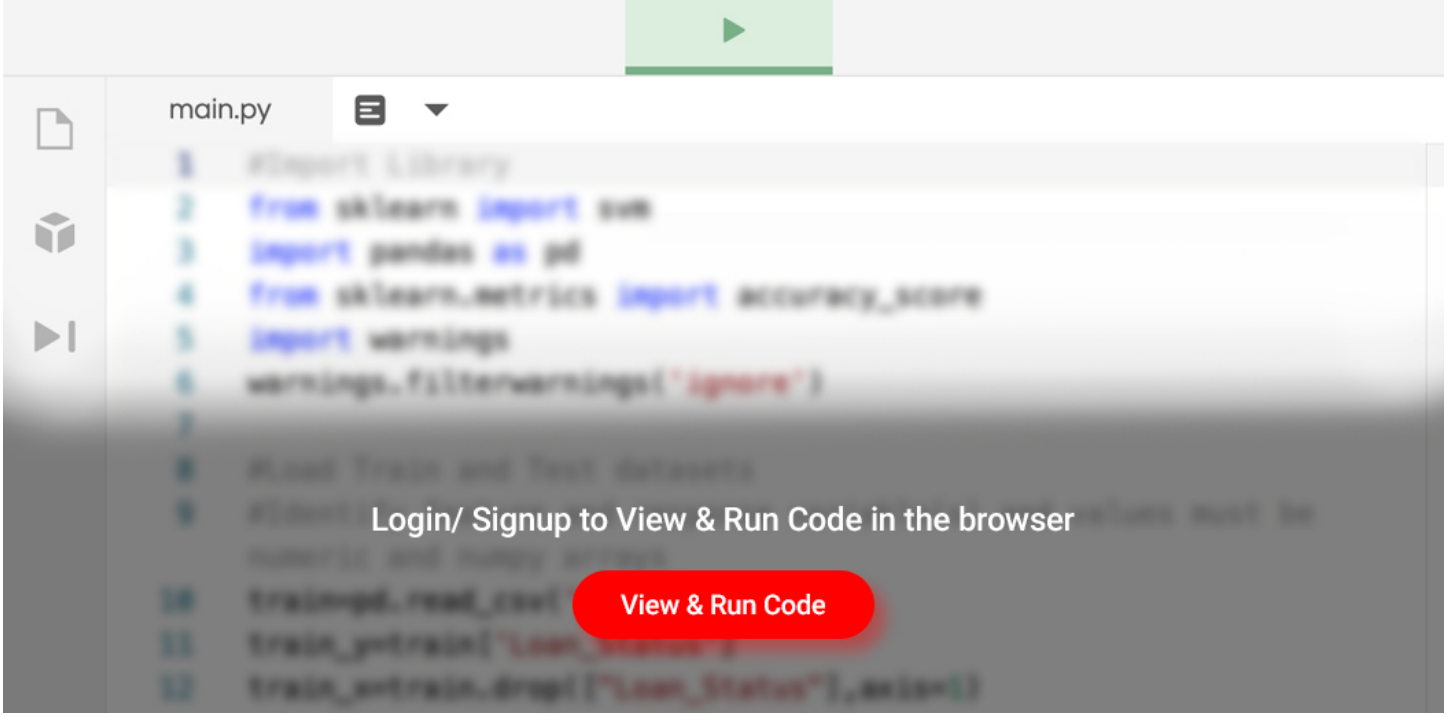
Python is one of the most widely used language for Data Analysis and Data Science. Python is easy to learn, has a great online community of learners and instructors, and has some really powerful data-centric libraries. **Pandas** is one of the most important libraries in Python for Data Analysis, and Data Science.

In this article, we will look at the 13 most important and basic Pandas functions in Python and methods that are essential for every Data Analyst and Data Scientist to know.

1. read_csv()

This is one of the most crucial pandas methods in Python. read_csv() function helps read a comma-separated values (csv) file into a Pandas DataFrame. All you need to do is mention the path of the file you want it to read. It can also read files separated by delimiters other than comma, like | or tab. More details here.

Python Code:



The data has been read from the data source into the Pandas DataFrame. You will have to change the path of the file you want to read. You can download the dataset used in the blog.

to_csv() function works exactly opposite of read_csv(). It helps to write data contained in a Pandas DataFrame or Series to a csv file. You can read more about to_csv() here. read_csv() and to_csv() are one of the most used functions in Pandas because they are used while reading data from a data source, and are very important to know.

2. head()

head(n) is used to return the first n rows of a dataset. By default, df.head() will return the first 5 rows of the DataFrame. If you want more/less number of rows, you can specify n as an integer.

```
data_1.head(6)
```

Output:

	Name	Age	City	State	DOB	Gender	City temp	Salary
0	Alam	29	Indore	Madhya Pradesh	20-11-1991	Male	35.5	50000
1	Rohit	23	New Delhi	Delhi	19-09-1997	Male	39.0	85000
2	Bimla	35	Rohtak	Haryana	09-01-1985	Female	39.7	20000
3	Rahul	25	Kolkata	West Bengal	19-09-1995	Male	36.5	40000
4	Chaman	32	Chennai	Tamil Nadu	12-03-1988	Male	41.1	65000
5	Vivek	38	Gurugram	Haryana	22-06-1982	Male	38.9	35000

The first 6 rows (indexed 0 to 5) are returned as output as per expectation.

tail() is similar to head(), and returns the bottom n rows of a dataset. head() and tail() help you get a quick glance at your dataset, and check if data has been read into the DataFrame properly.

3. describe()

describe() is used to generate descriptive statistics of the data in a Pandas DataFrame or Series. It summarizes central tendency and dispersion of the dataset. describe() helps in getting a quick overview of

the dataset. More details about describe() can be found here.

```
data_1.describe()
```

Output:

	Age	City temp	Salary
count	9.000000	8.000000	9.000000
mean	32.000000	38.575000	44444.444444
std	5.894913	1.771803	21360.659582
min	23.000000	35.500000	18000.000000
25%	29.000000	38.300000	35000.000000
50%	32.000000	38.950000	40000.000000
75%	38.000000	39.175000	52000.000000
max	39.000000	41.100000	85000.000000

describe() lists out different descriptive statistical measures for all numerical columns in our dataset. By assigning the include attribute the value 'all', we can get the description to include all columns, including those containing categorical information.

4. memory_usage()

memory_usage() returns a Pandas Series having the memory usage of each column (in bytes) in a Pandas DataFrame. By specifying the deep attribute as True, we can get to know the actual space being taken by each column. More details on memory_usage() can be found here.

```
data_1.memory_usage(deep=True)
```

Output:

```
Index 80 Name 559 Age 72 City 578 State 584 DOB 603 Gender 553 City temp 72 Salary 72 dtype: int64
```

The memory usage of each column has been given as output in a Pandas Series. It is important to know the memory usage of a DataFrame, so that you can tackle errors like MemoryError in Python.

5. astype()

astype() is used to cast a Python object to a particular data type. It can be a very helpful function in case your data is not stored in the correct format (data type). For instance, if floating point numbers have somehow been misinterpreted by Python as strings, you can convert them back to floating point numbers with astype(). Or if you want to convert an object datatype to category, you can use astype().

```
data_1['Gender'] = data_1.Gender.astype('category')
```

You can verify the change in data type by looking at the data types of all columns in the dataset using the dtypes attribute. For looking at the documentation for astype(), click here.

6. loc[:]

loc[:] helps to access a group of rows and columns in a dataset, a slice of the dataset, as per our requirement. For instance, if we only want the last 2 rows and the first 3 columns of a dataset, we can

access them with the help of loc[:]. We can also access rows and columns based on labels instead of row and column number.

```
data_1.loc[0:4, ['Name', 'Age', 'State']]
```

Output:

	Name	Age	State
0	Alam	29	Madhya Pradesh
1	Rohit	23	Delhi
2	Bimla	35	Haryana
3	Rahul	25	West Bengal
4	Chaman	32	Tamil Nadu

The above code will return the “Name”, “Age”, and “State” columns for the first 5 customer records. Keep in mind that index starts from 0 in Python, and that loc[:] is **inclusive** on both values mentioned. So 0:4 will mean indices 0 to 4, both included.

loc[:] is one of the most powerful functions in Pandas, and is a must-know for all Data Analysts and Data Scientists. You can find the documentation for loc[:] here.

iloc[:] works in a similar manner, just that iloc[:] is **not inclusive** on both values. So iloc[0:4] would return rows with index 0, 1, 2, and 3, while loc[0:4] would return rows with index 0, 1, 2, 3, and 4. The documentation for iloc[:] can be found here.

7. to_datetime()

to_datetime() converts a Python object to datetime format. It can take an integer, floating point number, list, Pandas Series, or Pandas DataFrame as argument. to_datetime() is very powerful when the dataset has time series values or dates.

```
data_1['DOB'] = pd.to_datetime(data_1['DOB'])
```

The DOB column has now been changed to Pandas datatime format. All datetime functions can now be applied on this column. You can read more about to_datetime() here.

8. value_counts()

value_counts() returns a Pandas Series containing the counts of unique values. Consider a dataset that contains customer information about 5,000 customers of a company. value_counts() will help us in identifying the number of occurrences of each unique value in a Series. It can be applied to columns containing data like State, Industry of employment, or age of customers.

```
data_1['State'].value_counts()
```

Output:

```
Haryana 3 Delhi 2 West Bengal 1 Tamil Nadu 1 Bihar 1 Madhya Pradesh 1 Name: State, dtype: int64
```

The number of occurrences of each state in our dataset has been returned in the output, as expected. value_counts() can also be used to plot bar graphs of categorical and ordinal data.

```
data_1['State'].value_counts(normalize=True).plot(kind='bar', title='State')
```

The documentation for value_counts() can be found here.

9. drop_duplicates()

drop_duplicates() returns a Pandas DataFrame with duplicate rows removed. Even among duplicates, there is an option to keep the first occurrence (record) of the duplicate or the last. You can also specify the inplace and ignore_index attribute.

```
data_1.drop_duplicates(inplace=True)
```

inplace=True makes sure the changes are applied to the original dataset. You can verify the changes by looking at the shape of the original dataset, and the modified dataset (after dropping duplicates). You will notice the number of rows have reduced from 9 to 8 (because 1 duplicate has been dropped).

10. groupby()

groupby() is used to group a Pandas DataFrame by 1 or more columns, and perform some mathematical operation on it. groupby() can be used to summarize data in a simple manner.

```
data_1.groupby(by='State').Salary.mean()
```

Output:

```
State Bihar 18000 Delhi 68500 Haryana 27500 Madhya Pradesh 50000 Tamil Nadu 65000 West Bengal 40000 Name: Salary, dtype: int64
```

The above code will group the dataset by “State” column, and will return the mean age across states. You can click here to know more about groupby().

11. merge()

merge() is used to merge 2 Pandas DataFrame objects or a DataFrame and a Series object on a common column (field). If you are familiar with the concept of JOIN in SQL, merge function similar to that. It returns the merged DataFrame.

```
data_1.merge(data_2, on='Name', how='left')
```

To know more about attributes like on (including left_on and right_on), how, and suffixes, refer to the documentation.

12. sort_values()

sort_values() is used to sort column in a Pandas DataFrame (or a Pandas Series) by values in ascending or descending order. By specifying the inplace attribute as True, you can make a change directly in the original DataFrame.

```
data_1.sort_values(by='Name', inplace=True)
```

Output:

	Name	Age	City	State	DOB	Gender	City temp	Salary
0	Alam	29	Indore	Madhya Pradesh	1991-11-20	Male	35.5	50000
2	Bimla	35	Rohtak	Haryana	1985-09-01	Female	39.7	20000
4	Chaman	32	Chennai	Tamil Nadu	1988-12-03	Male	41.1	65000
6	Charu	29	New Delhi	Delhi	1992-03-18	Female	39.0	52000
7	Ganesh	39	Patna	Bihar	1981-07-12	Male	NaN	18000
3	Rahul	25	Kolkata	West Bengal	1995-09-19	Male	36.5	40000
1	Rohit	23	New Delhi	Delhi	1997-09-19	Male	39.0	85000
5	Vivek	38	Gurugram	Haryana	1982-06-22	Male	38.9	35000

You can see that the ordering of records has changed now. Records are now listed in alphabetical order of Names. sort_values() has many other attributes which can be specified. You can read about it here.

Similar to sort_values() is sort_index(). It is used to sort the DataFrame by index instead of a column value.

13. fillna()

Typically in a large dataset, you will find several entries labelled NaN by Python. NaN stands for “not a number”, and represents entries that were not populated in the original data source. While populating the values in the DataFrame, Pandas makes sure that these entries can be identified separately by the user.

fillna() helps to replace all NaN values in a DataFrame or Series by imputing these missing values with more appropriate values.

```
data_1['City temp'].fillna(38.5, inplace=True)
```

The above code will replace all blank “City temp” entries with 38.5. The missing values could be imputed with the mean, median, mode, or some other value. We have chosen mean for our case.

EndNotes

In this article, we had a look at the 13 most important functions and methods in Pandas that are important for Data Analytics and Data Science. This article was written by Vishesh Arora (LinkedIn).

The media shown in this article are not owned by Analytics Vidhya and is used at the Author’s discretion.

Article Url - https://www.analyticsvidhya.com/blog/2021/05/pandas-functions-13-most-important/

Vishesh Arora