Pandas for Data Cleaning and Preparation

Nicholas Shaw

October 21, 2019

Academy AI

Pandas for Cleaning

Pandas

Pandas is, probably, the most useful tool that you have in python for managing data. It helps you working with data keeping the benefits of numpy behind the hood. Pandas provide a number of features.

- Data structures **with labeled axes**.
- Date time handling.
- Element-wise operations.
- Comfortable handle of missing values.
- SQL-like operations. (Merging, ...)

DataFrame

DataFrames are a collection of series, forming a bidimensional matrix. Indices of the series represent the columns of the dataframe.

```
data ={
    'name': ['Maria', 'Carla', 'Juan', 'Ana', 'Sergio'],
    'age': [15, 33, 12, 21, 45],
    'gender': [True, True, False, True, False]
}

pd.DataFrame(data)
# name age gender
# 0 Maria 15 True
# 1 Carla 33 True
# 2 Juan 12 False
# 3 Ana 21 True
# 4 Sergio 45 False
```

The most common way to create a dataframe for is to use a dictionary of equally sized arrays in which every key represent a column.

Index Object

Every row in a Dataframe has its own index assigned, the Index of a DataFrame is formed by a list of object assigned to each row.

This comes specially useful when we have to filter the data or divide them as we will maintain a way to refer to each specific row.

To access the index of a DataFrame we can call the method `index`, this object is inmutable.

```
df =pd.DataFrame(data)
df.index # Int64Index([1, 2, 3, 4, 5], dtype='int64')
```

Changing the Index

You have the possibility of reindexing a dataframe by accessing the property `index` and passing an index.

```
df =pd.DataFrame(data)
df.index =['m','c','j','a','s']
df
# name age gender
# m Maria 15 True
# c Carla 33 True
# j Juan 12 False
# a Ana 21 True
# s Sergio 45 False
```

Index Types

There are different types of indexes

1. **Range Index** Accesses the dataframe by row position not taking into account the index.
2. **Item/Multi Index** Accesses the dataframe by index.
3. **Datetime Index** Accesses the dataframe by index.

```
#range index
range =pd.RangeIndex(0, 100)
df.set_index(range)

#item index
items =['m','c','j','a','s']
df.set_index(items)

#datetime index
dates =pd.date_range('2019-01-01', '2019-01-01', freq='1h')
df.set_index(dates)
```

Indexing and Slicing I

There are two ways to index a DataFrame.

1. **iloc** Accesses the dataframe by row position not taking into account the index.
2. **loc** Accesses the dataframe by index.

```
a =pd.DataFrame(data, index=['m','c','j','a','s'])

a.loc['m'] ==a.iloc[0] # True
```

Indexing and Slicing I

Slicing is performed in a similar fashion. Note that you can slice using labels and it will take all elements in between both labels.

```
a =pd.DataFrame(data, index=['m','c','j','a','s'])

a.loc['m':'c']
# name age gender
# m Maria 15 True
# c Carla 33 True
```

Dropping

Pandas lets us drop columns or rows. To do so we just need to call `drop` with the labels or columns and the axis, 0 for rows and 1 for columns. The method performs a copy instead of doing it inplace

```
df =pd.DataFrame(data, index=['m','c','j','a','s'])

df.drop('name', axis=1)
# age gender
# m 15 True
# c 33 True
# j 12 False
# a 21 True
# s 45 False
a.drop(['m', 'c'])
# name age gender
# j Juan 12 False
# a Ana 21 True
# s Sergio 45 False
```

Dropping duplicates

In same scenarios we will face duplicated samples to get rid of them we can rely on `drop_duplicates`. This method will remove at a row level the duplicates.

```
df =pd.DataFrame({'f1': [1, 2, 2, 1], 'f2': [0, 1, 1, 1]})
df.drop_duplicates()
# f1 f2
# 0 1 0
# 1 2 1
# 3 1 1
```

Filtering

Analogously to numpy pandas allows filtering by boolean indexing. In this sense we can perform check as the following ones.

```
df =pd.DataFrame(data, index=['m','c','j','a','s'])
df[df.gender ==False]
# name age gender
# j Juan 12 False
# s Sergio 45 False
```

Arithmetic Operations

In the same manner we did with numpy we can perform arithmetic operations over dataframes preerving the indexes.

```
df +df
df /df
df *df
```

We can also do these operations among dataframes and series.

Custom function application

Element wise operations can be performed throughout the dataframe easily be means of the `apply` method.

```
df['gender'] =df['gender'].apply(lambda x: not x)
```

The operation is not performed in place and for it to take place we need to assign it to the original dataframe.

Sorting

We can sort a Dataframe with the method `sort_index` or `sort_values`, by providing the column we will be able to create a sorted copy of the dataframe.

```
df.sort_index(by='age')
# name age gender
# j Juan 12 False
# m Maria 15 True
# a Ana 21 True
# c Carla 33 True
# s Sergio 45 False
```

Statistics

Computing statistics from a dataframe is very easy thanks to the ability of pandas to handle functions. It also provides some utility methods for convenience (`count`, `min`, `max`, `std`, `mean`, `median`). A statistical summary of the dataframe can be obtained by means of `describe`.

```
df.describe()
# age
# count 5.000000
# mean 25.200000
# std 13.682105
# min 12.000000
# 25% 15.000000
# 50% 21.000000
# 75% 33.000000
# max 45.000000
```

Value counts

In a pandas series we can count the number of appareances of the values by means of `value_counts`.

```
df['gender'].value_counts()
# True 3
# False 2
# Name: gender, dtype: int64
```

Filling missing values

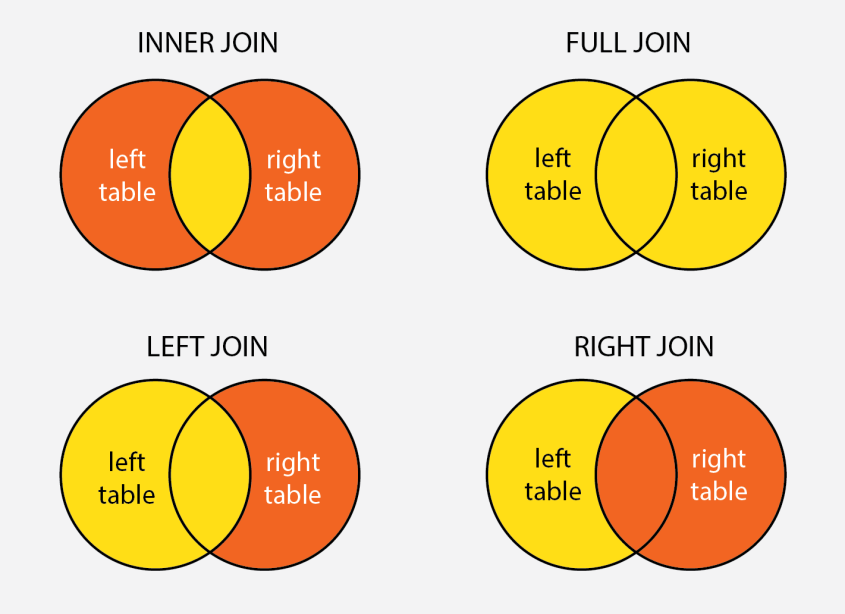
Sometimes we encounter missing values in our data, we have seen how to detect them, but to fill them with a imputation technique we can use the method `fillna`

Joining Methods

There are several ways to do SQL like joins in Pandas.

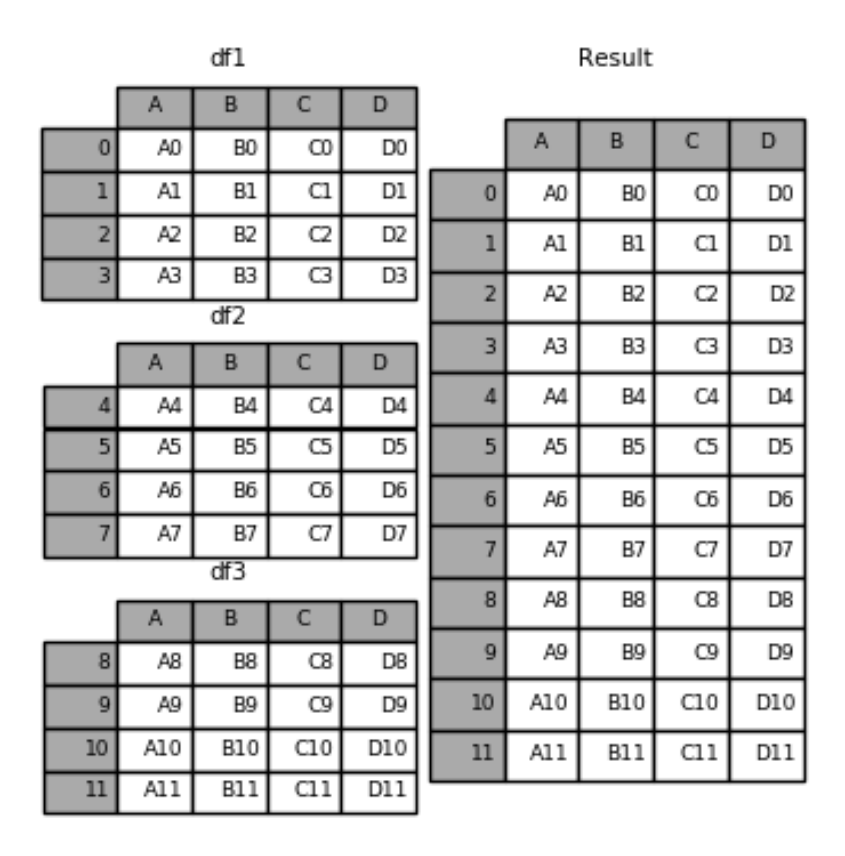
To join multiple dataframes at once we can use `pd.concat`

To join two dataframes by key or join type at we can use `pd.merge`



Joining vertically

When dealing with several files or dataframes we may have the data splitted in several ones. To join them we can use `pd.concat`.



	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Joining horixontally

In other scenarios we may have to deal with having the dataset splitted by columns in this cases we can use aswell `concat` but along the axis 1.

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

This kind of operations have a lot of variants I recommend taking a look to the documentation for more information. Documentation

Cleaning Methodology

Introduction

Data preprocessing is a data mining technique which is used to transform the raw data in a useful and efficient format.

The main steps involved in data preprocessing are:

- **Data cleaning:** data can have many irrelevant and missing parts. To handle this, data cleaning is done. It involves handling of missing data, noisy data etc.
- **Data transformation:** this step is taken in order to transform the data in appropriate forms suitable for mining process.
- **Data reduction:** while working with huge volume of data, analysis became harder. In order to overcome this issue, we uses data reduction techniques, which aim to increase the storage efficiency and reduce data storage and analysis costs.

Data Cleaning Steps

When first approaching a dataset our goal is to understand it and prepare it for use.

A basic methodology is:

1. Detect unexpected, incorrect, and inconsistent data.
2. Fix or remove the anomalies discovered.
3. After cleaning, the results are inspected to verify correctness.
4. A report about the changes made and the quality of the currently stored data is recorded.

Reference Article: Massive Overview of Data Cleaning

Data format

Data can come in different formats, for example:

- csv file
- txt file
- xls/xlsx file
- arff format
- json format
- web pages (html,xml,etc)
- SQL database
- etc.

There are methods among `sklearn` and `scipy` that can handle the different type of formats that you can encounter.

In some cases, some type of encoding/decoding might be necessary (e.g. UTF8, ASCII)

Missing data I

This situation arises when some data is missing in the dataset. Missing data is tricky because it can appear in different ways, for example:

- null values: `np.nan, null, ""`, etc.
- "placeholders": `"?", "999", "9999", "-"`, etc.

The second type is the most problematic because it is not always easy to identify, so it is important to read carefully the dataset description before using it, or if not available, to carefully analyze the dataset (type of values for each features, distribution of the data for numerical attributes, possible values for categorical attributes).

Missing data II

Missing can be handled in various ways. Some of them are:

- **Remove the entries that contain missing values:** this approach is suitable only when the dataset we have is quite large and multiple values are missing within a tuple.
- **Fill the missing values:** there are various ways to do this task. You can choose to fill the missing values manually, by attribute mean or the most probable value.

Data manipulation

Data aggregation

Aggregation is the process of combining similar pieces of data to gain insight and see the data from a broader perspective

We can aggreate with the `groupby()` function. This returns a groupby object that we can do operations on i.e.

```
group =data.groupby('year')
#get the average per year
group.mean()
```

Data aggregation II

We can also aggregate with custom specific or custom functions using the agg() function.

```
group =data.groupby('year')

group.agg([np.sum, np.mean, np.abs])
```

Data transformation

This step is taken in order to transform the data in appropriate forms suitable for mining process. This involves the following processes:

- **Normalization:** it is done in order to scale the data values in a specified range (-1.0 to 1.0 or 0.0 to 1.0)
- **Attribute selection:** in this strategy, new attributes are constructed from the given set of attributes to help the mining process.
- **Discretization:** this is done to replace the raw values of numeric attribute by interval levels or conceptual levels.
- **Concept Hierarchy Generation:** here attributes are converted from level to higher level in hierarchy. For Example-The attribute "city" can be converted to "country".

Data transformation in Python

- **Normalization:** `sklearn.preprocessing.MinMaxScaler`
- **Attribute selection:** `sklearn.preprocessing.OneHotEncoder`, `pandas.get_dummies`
- **Discretization (binning):** `sklearn.preprocessing.KBinsDiscretizer`, `pandas.cut`