Data Cleaning and Preparation with Python

Contents

- Handling Missing Data
- Data Transformation

Overview

- During the course of data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging.
- Such tasks are often reported to take up 80% or more of an analyst’s time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task.
- Fortunately, pandas, along with the built-in Python language features, provides us with a high-level, flexible, and fast set of tools to enable one to manipulate data into the right form.
- Here we will discuss tools for missing data, duplicate data, and some other analytical data transformations.

Handling Missing Data

- Missing data occurs commonly in many data analysis applications.
- One of the goals of pandas is to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data by default.

```
string_data = pd.Series(['Vishal', 'Pranay', np.nan, 'Viraj'])
string_data
```

```
0    Vishal
1    Pranay
2       NaN
3     Viraj
dtype: object
```

- The way that missing data is represented in pandas objects is somewhat imperfect, but it is functional for a lot of users. For numeric data, pandas uses the floating-point value **NaN** (Not a Number) to represent missing data.

Handling Missing Data

- When cleaning up data for analysis, it is often important to do analysis on the missing data itself to identify data collection problems or potential biases in the data caused by missing data.

```
string_data.isnull()
```

```
0    False
1    False
2     True
3    False
dtype: bool
```

```
string_data[0] = None
string_data.isnull()
```

```
0     True
1    False
2     True
3    False
dtype: bool
```

- The built-in Python **None** value is also treated as NA in object arrays.

Filtering Out Missing Data

- There are a few ways to filter out missing data. While one always have the option to do it by hand using **pandas.isnull()** and boolean indexing, the **dropna()** can be helpful.

- On a Series, it returns the Series with only the non-null data and index values.

- In pandas, we will adopt a convention used in the R programming language by referring to missing data as **NA**, which stands for **not available**.

- In statistics applications, **NA** data may either be data that does not exist or that exists but was not observed.

```
from numpy import nan as NA
```

```
data = pd.Series([1, NA, 3.5, NA, 7])
data
```

```
0     1.0
1     NaN
2     3.5
3     NaN
4     7.0
dtype: float64
```

```
data.dropna()
```

```
0     1.0
2     3.5
4     7.0
dtype: float64
```

```
data[data.notnull()]
```

```
0     1.0
2     3.5
4     7.0
dtype: float64
```

Filtering Out Missing Data

- With DataFrame objects, things are a bit more complex. One may want to drop rows or columns that are all NA or only those containing any NAs.
- **dropna()** by default drops any row containing a missing value.

```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                     [NA, NA, NA], [NA, 6.5, 3.]])
data
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
cleaned = data.dropna()
cleaned
```

	0	1	2
0	1.0	6.5	3.0

Filtering Out Missing Data

- Passing **how='all'** will only drop rows that are all NA.
- To drop columns in the same way, pass **axis=1.**

```
data.dropna(how='all')
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

```
data[4] = NA
data
```

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
data.dropna(axis=1, how='all')
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

Filtering Out Missing Data

- Suppose one want to keep only rows containing a certain number of observations. then it can be done with the **thresh** argument.

```
df = pd.DataFrame(np.random.randn(7, 3))
df.iloc[:4, 1] = NA
df.iloc[:2, 2] = NA
df
```

	0	1	2
0	-0.555960	NaN	NaN
1	-0.431926	NaN	NaN
2	-1.298832	NaN	0.084469
3	-0.223855	NaN	-1.244506
4	1.310749	-1.271158	-0.107044
5	-2.036641	1.559647	-0.743544
6	0.454430	-0.086190	0.339858

```
df.dropna()
```

	0	1	2
4	1.310749	-1.271158	-0.107044
5	-2.036641	1.559647	-0.743544
6	0.454430	-0.086190	0.339858

```
df.dropna(thresh=2)
```

	0	1	2
2	-1.298832	NaN	0.084469
3	-0.223855	NaN	-1.244506
4	1.310749	-1.271158	-0.107044
5	-2.036641	1.559647	-0.743544
6	0.454430	-0.086190	0.339858

Filling In Missing Data

- Rather than filtering out missing data , one may want to fill in the “holes” in any number of ways. For most purposes, the **fillna()** method is the workhorse function to use.

- Calling **fillna()** with a constant replaces missing values with that value.

```
df.fillna(0)
```

	0	1	2
0	-0.056266	0.000000	0.000000
1	1.002331	0.000000	0.000000
2	-0.583575	0.000000	-0.464536
3	-0.659612	0.000000	1.134826
4	0.726211	-2.071342	0.403916
5	-0.267061	1.571610	0.044672
6	-0.477977	-1.166620	0.654115

Filling In Missing Data

- Calling **fillna()** with a **dict**, one can use a different fill value for each column.

```
df.fillna({1: 0.5, 2: 0})
```

	0	1	2
0	-0.056266	0.500000	0.000000
1	1.002331	0.500000	0.000000
2	-0.583575	0.500000	-0.464536
3	-0.659612	0.500000	1.134826
4	0.726211	-2.071342	0.403916
5	-0.267061	1.571610	0.044672
6	-0.477977	-1.166620	0.654115

- **fillna** returns a new object, but you can modify the existing object **in-place.**

```
df.fillna(0, inplace=True)
df
```

	0	1	2
0	-0.056266	0.000000	0.000000
1	1.002331	0.000000	0.000000
2	-0.583575	0.000000	-0.464536
3	-0.659612	0.000000	1.134826
4	0.726211	-2.071342	0.403916
5	-0.267061	1.571610	0.044672
6	-0.477977	-1.166620	0.654115

Filling In Missing Data

```
df = pd.DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA
df.iloc[4:, 2] = NA
df
```

	0	1	2
0	0.151661	-0.101958	0.299952
1	-0.269132	-0.857208	0.233967
2	0.211398	NaN	-1.491571
3	0.291306	NaN	-1.308640
4	-1.224203	NaN	NaN
5	0.858421	NaN	NaN

- The same interpolation methods available for **reindexing** can be used with **fillna().**

```
df.fillna(method='ffill')
```

	0	1	2
0	0.151661	-0.101958	0.299952
1	-0.269132	-0.857208	0.233967
2	0.211398	-0.857208	-1.491571
3	0.291306	-0.857208	-1.308640
4	-1.224203	-0.857208	-1.308640
5	0.858421	-0.857208	-1.308640

```
df.fillna(method='ffill', limit=2)
```

	0	1	2
0	0.151661	-0.101958	0.299952
1	-0.269132	-0.857208	0.233967
2	0.211398	-0.857208	-1.491571
3	0.291306	-0.857208	-1.308640
4	-1.224203	NaN	-1.308640
5	0.858421	NaN	-1.308640

Filling In Missing Data

- With **fillna()** one can do lots of other things like, one might pass the mean or median value of a Series for missing values.

```
data = pd.Series([1., NA, 3.5, NA, 7])
data
```

```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

```
data.fillna(data.mean())
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

Data Transformation

- Along with rearranging data. Filtering, cleaning, and other transformations are another class of important operations.

- *Removing Duplicates* - Duplicate rows may be found in a DataFrame for any number of reasons. The DataFrame method **duplicated()** returns a boolean Series indicating whether each row is a duplicate or not i.e. has been observed in a previous row.

```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                     'k2': [1, 1, 2, 3, 3, 4, 4]})
data
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

```
data.duplicated()
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Data Transformation

- Similarly, **drop_duplicates()** returns a DataFrame where the duplicated array is False.
- Both of these methods by default consider all of the columns; alternatively, one can specify any subset of them to detect duplicates.
- **Duplicated()** and **drop_duplicates()** by default keep the first observed value combination. Passing **keep='last'** will return the last one.

```
data.drop_duplicates()
```

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

```
data.drop_duplicates(['k1'])
```

	k1	k2
0	one	1
1	two	1

```
data.drop_duplicates(['k1'], keep='last')
```

	k1	k2
4	one	3
6	two	4

Replacing Values & Renaming Index

- Filling in missing data with the **fillna()** method is a special case of more general value replacement. **replace()** provides a simpler and more flexible way to modify a subset of values in an object.

- To replace outeline values with NA values that pandas understands, we can use **replace()**, producing a new Series (unless we pass **inplace=True**).

```
data = pd.Series([1, -999, 2, -999, -1000, 3])
data
```

```
0        1
1     -999
2        2
3     -999
4    -1000
5        3
dtype: int64
```

```
data.replace(-999, np.nan)
```

```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

Replacing Values & Renaming Index

- If you want to **replace()** multiple values at once, one instead pass a list and then the substitute value.

```
data.replace([-999, -1000], np.nan)
```

```
0     1.0
1     NaN
2     2.0
3     NaN
4     NaN
5     3.0
dtype: float64
```

```
data.replace([-999, -1000], [np.nan, 0])
```

```
0     1.0
1     NaN
2     2.0
3     NaN
4     0.0
5     3.0
dtype: float64
```

Replacing Values & Renaming Index

- To use a different replacement for each value, pass a list of substitutes or dict can be used.

```
data.replace({-999: np.nan, -1000: 0})
```

```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Replacing Values & Renaming Index

- Like values in a Series, axis labels can be similarly transformed using **rename()** function. **rename()** can be used in conjunction with a dict-like object providing new values for a subset of the axis labels.

```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
                    index=['Nagpur', 'Raipur', 'Hyderabad'],
                    columns=['one', 'two', 'three', 'four'])
data
```

	one	two	three	four
Nagpur	0	1	2	3
Raipur	4	5	6	7
Hyderabad	8	9	10	11

```
data.rename(index=str.lower, columns=str.upper)
```

	ONE	TWO	THREE	FOUR
nagpur	0	1	2	3
raipur	4	5	6	7
hyderabad	8	9	10	11

Replacing Values & Renaming Index

```
data.rename(index={'Nagpur': 'NGP'},
            columns={'three': 'five'})
```

	one	two	five	four
NGP	0	1	2	3
Raipur	4	5	6	7
Hyderabad	8	9	10	11

- To modify a dataset in-place, pass ***inplace=True***.

```
data.rename(index={'Nagpur': 'NGP'}, inplace=True)
data
```

	one	two	three	four
NGP	0	1	2	3
Raipur	4	5	6	7
Hyderabad	8	9	10	11

Computing Dummy Variables

- Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a "*dummy*" or "*indicator*" matrix.
- If a column in a DataFrame has k distinct values, we would derive a matrix or Data-Frame with k columns containing all 1s and 0s.
- pandas has a **get_dummies()** function for doing same.

```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                   'data1': range(6)})
df
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
pd.get_dummies(df['key'])
```

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

```
pd.get_dummies(df)
```

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

Computing Dummy Variables

- In some cases, we may want to add a prefix to the columns in the indicator Data-Frame, which can then be merged with the other data.
- **get_dummies()** has a prefix argument for doing this also.

```
pd.get_dummies(df, prefix='k')
```

	data1	k_a	k_b	k_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

```
pd.get_dummies(df, prefix='k',drop_first=True)
```

	data1	k_b	k_c
0	0	1	0
1	1	1	0
2	2	0	0
3	3	0	1
4	4	0	0
5	5	1	0