Data Cleaning and Preparation

Part 1

- During the course of doing 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.
- Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, like Python, Perl, R, or Java, or Unix text-processing tools like sed or awk.
- Fortunately, pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

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.

- 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.
- We call this a *sentinel value* that can be easily detected:

```
In [2]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [3]: string_data

Out[3]: 0     aardvark
        1    artichoke
        2          NaN
        3      avocado
        dtype: object

In [4]: string_data.isnull()

Out[4]: 0    False
        1    False
        2     True
        3    False
        dtype: bool
```

- In pandas, we’ve adopted 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 (through problems with data collection, for example).
- 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.

- The built-in Python `None` value is also treated as NA in object arrays:

```
In [5]: string_data[0] = None

In [6]: string_data.isnull()
Out[6]: 0     True
        1    False
        2     True
        3    False
        dtype: bool
```

Filtering Out Missing Data

- There are a few ways to filter out missing data.
- While you 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 [7]: from numpy import nan as NA

In [8]: data = pd.Series([1, NA, 3.5, NA, 7])

In [9]: data.dropna()
Out[9]: 0    1.0
        2    3.5
        4    7.0
        dtype: float64
```

- This is equivalent to:

```
In [10]: data[data.notnull()]

Out[10]: 0    1.0
         2    3.5
         4    7.0
         dtype: float64
```

- With DataFrame objects, things are a bit more complex.
- You 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:

```
In [11]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                              [NA, NA, NA], [NA, 6.5, 3.]])

In [12]: cleaned = data.dropna()

In [13]: data
Out[13]:
```

	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

```
In [14]: cleaned
Out[14]:
```

	0	1	2
0	1.0	6.5	3.0

- Passing `how='all'` will only drop rows that are all NA:

```
In [15]: data
Out[15]:
```

	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

```
In [16]: data.dropna(how='all')
Out[16]:
```

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

- To drop columns in the same way, pass `axis=1`:

```
In [17]: data[4] = NA

In [18]: data
Out[18]:
```

	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

```
In [19]: data.dropna(axis=1, how='all')
Out[19]:
```

	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

- A related way to filter out DataFrame rows tends to concern time series data.
- Suppose you want to keep only rows containing a certain number of observations.
- You can indicate this with the `thresh` argument:

```
In [20]: df = pd.DataFrame(np.random.randn(7, 3))

In [21]: df.iloc[:4, 1] = NA

In [22]: df.iloc[:2, 2] = NA
```

In [23]:
```
df
```

Out[23]:

	0	1	2
0	-0.204708	NaN	NaN
1	-0.555730	NaN	NaN
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

In [24]:
```
df.dropna()
```

Out[24]:

	0	1	2
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

In [25]:
```
df.dropna(thresh=2)
```

Out[25]:

	0	1	2
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Filling In Missing Data

- Rather than filtering out missing data (and potentially discarding other data along with it), you 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.

```
In [26]: df
```

Out[26]:

	0	1	2
0	-0.204708	NaN	NaN
1	-0.555730	NaN	NaN
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [27]: df.fillna(0)
```

Out[27]:

	0	1	2
0	-0.204708	0.000000	0.000000
1	-0.555730	0.000000	0.000000
2	0.092908	0.000000	0.769023
3	1.246435	0.000000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

- Calling `fillna` with a dict, you can use a different fill value for each column:

```
In [28]: df
Out[28]:
```

	0	1	2
0	-0.204708	NaN	NaN
1	-0.555730	NaN	NaN
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [29]: df.fillna({1: 0.5, 2: 0})
Out[29]:
```

	0	1	2
0	-0.204708	0.500000	0.000000
1	-0.555730	0.500000	0.000000
2	0.092908	0.500000	0.769023
3	1.246435	0.500000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

- `fillna` returns a new object, but you can modify the existing object in-place:

```
In [30]: _ = df.fillna(0, inplace=True)

In [31]: df
Out[31]:
```

	0	1	2
0	-0.204708	0.000000	0.000000
1	-0.555730	0.000000	0.000000
2	0.092908	0.000000	0.769023
3	1.246435	0.000000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

- The same interpolation methods available for reindexing can be used with `fillna`:

```
In [32]: df = pd.DataFrame(np.random.randn(6, 3))

In [33]: df.iloc[2:, 1] = NA

In [34]: df.iloc[4:, 2] = NA

In [35]: df
Out[35]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	NaN	1.343810
3	-0.713544	NaN	-2.370232
4	-1.860761	NaN	NaN
5	-1.265934	NaN	NaN

```
In [36]: df.fillna(method='ffill')
Out[36]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	0.124121	-2.370232
5	-1.265934	0.124121	-2.370232

In [37]:
```
df.fillna(method='ffill', limit=2)
```
Out[37]:

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	NaN	-2.370232
5	-1.265934	NaN	-2.370232

- With `fillna` you can do lots of other things with a little creativity.
- For example, you might pass the mean or median value of a Series:

```
In [38]: data = pd.Series([1., NA, 3.5, NA, 7])

In [39]: data.fillna(data.mean())
Out[39]: 0    1.000000
         1    3.833333
         2    3.500000
         3    3.833333
         4    7.000000
         dtype: float64
```