Handling Missing Data with Pandas

pandas borrows all the capabilities from numpy selection + adds a number of convenient methods to handle missing values. Let's see one at a time:

Hands on!

In [1]:

```
import numpy as np
import pandas as pd
```

Pandas utility functions

Similarly to `numpy`, pandas also has a few utility functions to identify and detect null values:

In [2]:

```
pd.isnull(np.nan)
```

Out[2]:

```
True
```

In [3]:

```
pd.isnull(None)
```

Out[3]:

```
True
```

In [4]:

```
pd.isna(np.nan)
```

Out[4]:

```
True
```

In [5]:

```
pd.isna(None)
```

Out[5]:

```
True
```

The opposite ones also exist:

In [6]:

```
pd.notnull(None)
```

Out[6]:

```
False
```

In [7]:

```
pd.notnull(np.nan)
```

Out[7]:

```
False
```

In [9]:

```
pd.notna(np.nan)
```

Out[9]:

```
False
```

In [8]:

```
pd.notnull(3)
```

Out[8]:

```
True
```

These functions also work with Series and `DataFrame` s:

In [10]:

```
pd.isnull(pd.Series([1, np.nan, 7]))
```

Out[10]:

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

In [11]:

```
pd.notnull(pd.Series([1, np.nan, 7]))
```

Out[11]:

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

In [12]:

```
pd.isnull(pd.DataFrame({
    'Column A': [1, np.nan, 7],
    'Column B': [np.nan, 2, 3],
    'Column C': [np.nan, 2, np.nan]
}))
```

Out[12]:

	Column A	Column B	Column C
0	False	True	True
1	True	False	False
2	False	False	True

Pandas Operations with Missing Values

Pandas manages missing values more gracefully than numpy. `nan` s will no longer behave as "viruses", and operations will just ignore them completely:

In [ ]:

```
pd.Series([1, 2, np.nan]).count()
```

In [ ]:

```
pd.Series([1, 2, np.nan]).sum()
```

In [ ]:

```
pd.Series([2, 2, np.nan]).mean()
```

Filtering missing data

As we saw with numpy, we could combine boolean selection + `pd.isnull` to filter out those `nan` s and null values:

In [13]:

```
s = pd.Series([1, 2, 3, np.nan, np.nan, 4])
```

In [14]:

```
pd.notnull(s)
```

Out[14]:

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

In [18]:

```
pd.isnull(s)
```

Out[18]:

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

In [16]:

```
pd.notnull(s).sum() # how many non-null values we have
```

Out[16]:

```
4
```

In [17]:

```
pd.isnull(s).sum() # how many null values we have
```

Out[17]:

```
2
```

In [19]:

```
s[pd.notnull(s)]
```

Out[19]:

```
0    1.0
1    2.0
2    3.0
5    4.0
dtype: float64
```

But both `notnull` and `isnull` are also methods of `Series` and `DataFrame` s, so we could use it that way:

In [20]:

```
s.isnull()
```

Out[20]:

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

In [21]:

```
s.notnull()
```

Out[21]:

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

In [22]:

```
s[s.notnull()]
```

Out[22]:

```
0    1.0
1    2.0
2    3.0
5    4.0
dtype: float64
```

Dropping null values

Boolean selection + `notnull()` seems a little bit verbose and repetitive. And as we said before: any repetitive task will probably have a better, more DRY way. In this case, we can use the `dropna` method:

In [24]:

```
s
```

Out[24]:

```
0    1.0
1    2.0
2    3.0
3    NaN
4    NaN
5    4.0
dtype: float64
```

In [25]:

```
s.dropna()  # drop null values
```

Out[25]:

```
0    1.0
1    2.0
2    3.0
5    4.0
dtype: float64
```

Dropping null values on DataFrames

You saw how simple it is to drop `na` s with a Series. But with `DataFrame` s, there will be a few more things to consider, because you can't drop single values. You can only drop entire columns or rows. Let's start with a sample `DataFrame` :

In [26]:

```
df = pd.DataFrame({
    'Column A': [1, np.nan, 30, np.nan],
    'Column B': [2, 8, 31, np.nan],
    'Column C': [np.nan, 9, 32, 100],
    'Column D': [5, 8, 34, 110],
})
```

In [27]:

```
df
```

Out[27]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	NaN	5
1	NaN	8.0	9.0	8
2	30.0	31.0	32.0	34
3	NaN	NaN	100.0	110

In [31]:

```
df.shape
```

Out[31]:

```
(4, 4)
```

In [30]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Column A  2 non-null      float64
 1   Column B  3 non-null      float64
 2   Column C  3 non-null      float64
 3   Column D  4 non-null      int64
dtypes: float64(3), int64(1)
memory usage: 256.0 bytes
```

In [28]:

```
df.isnull()
```

Out[28]:

	Column A	Column B	Column C	Column D
0	False	False	True	False
1	True	False	False	False
2	False	False	False	False
3	True	True	False	False

In [29]:

```
df.isnull().sum()
```

Out[29]:

```
Column A    2
Column B    1
Column C    1
Column D    0
dtype: int64
```

The default `dropna` behavior will drop all the rows in which *any* null value is present:

In [32]:

```
df.dropna()
```

Out[32]:

	Column A	Column B	Column C	Column D
2	30.0	31.0	32.0	34

In this case we're dropping **rows.** Rows containing null values are dropped from the DF. You can also use the `axis` parameter to drop columns containing null values:

In [33]:

```
df.dropna(axis=1)  # axis='columns' also works
```

Out[33]:

	Column D
0	5
1	8
2	34
3	110

In this case, any row or column that contains **at least** one null value will be dropped. Which can be, depending on the case, too extreme. You can control this behavior with the `how` parameter. Can be either `'any'` or `'all'`:

In [ ]:

```
df2 = pd.DataFrame({
    'Column A': [1, np.nan, 30],
    'Column B': [2, np.nan, 31],
    'Column C': [np.nan, np.nan, 100]
})
```

In [ ]:

```
df2
```

In [34]:

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

Out[34]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	NaN	5
1	NaN	8.0	9.0	8
2	30.0	31.0	32.0	34
3	NaN	NaN	100.0	110

In [35]:

```
df.dropna(how='any')  # default behavior
```

Out[35]:

	Column A	Column B	Column C	Column D
2	30.0	31.0	32.0	34

You can also use the `thresh` parameter to indicate a *threshold* (a minimum number) of non-null values for the row/column to be kept:

In [ ]:

```
df
```

In [36]:

```
df.dropna(thresh=3)
```

Out[36]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	NaN	5
1	NaN	8.0	9.0	8
2	30.0	31.0	32.0	34

In [37]:

```
df.dropna(thresh=3, axis='columns')
```

Out[37]:

	Column B	Column C	Column D
0	2.0	NaN	5
1	8.0	9.0	8
2	31.0	32.0	34
3	NaN	100.0	110

Filling null values

Sometimes instead than dropping the null values, we might need to replace them with some other value. This highly depends on your context and the dataset you're currently working. Sometimes a `nan` can be replaced with a `0`, sometimes it can be replaced with the `mean` of the sample, and some other times you can take the closest value. Again, it depends on the context. We'll show you the different methods and mechanisms and you can then apply them to your own problem.

In [38]:

```
s
```

Out[38]:

```
0    1.0
1    2.0
2    3.0
3    NaN
4    NaN
5    4.0
dtype: float64
```

Filling nulls with a arbitrary value

In [39]:

```
s.fillna(0)
```

Out[39]:

```
0    1.0
1    2.0
2    3.0
3    0.0
4    0.0
5    4.0
dtype: float64
```

In [40]:

```
s.fillna(s.mean())
```

Out[40]:

```
0    1.0
1    2.0
2    3.0
3    2.5
4    2.5
5    4.0
dtype: float64
```

In [41]:

```
s
```

Out[41]:

```
0    1.0
1    2.0
2    3.0
3    NaN
4    NaN
5    4.0
dtype: float64
```

Filling nulls with contiguous (close) values

The `method` argument is used to fill null values with other values close to that null one:

In [42]:

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

Out[42]:

```
0    1.0
1    2.0
2    3.0
3    3.0
4    3.0
5    4.0
dtype: float64
```

In [43]:

```
s.fillna(method='bfill')
```

Out[43]:

```
0    1.0
1    2.0
2    3.0
3    4.0
4    4.0
5    4.0
dtype: float64
```

This can still leave null values at the extremes of the Series/DataFrame:

In [44]:

```
pd.Series([np.nan, 3, np.nan, 9]).fillna(method='ffill')
```

Out[44]:

```
0    NaN
1    3.0
2    3.0
3    9.0
dtype: float64
```

In [45]:

```
pd.Series([1, np.nan, 3, np.nan, np.nan]).fillna(method='bfill')
```

Out[45]:

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

Filling null values on DataFrames

The `fillna` method also works on `DataFrame` s, and it works similarly. The main differences are that you can specify the `axis` (as usual, rows or columns) to use to fill the values (specially for methods) and that you have more control on the values passed:

In [46]:

```
df
```

Out[46]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	NaN	5
1	NaN	8.0	9.0	8
2	30.0	31.0	32.0	34
3	NaN	NaN	100.0	110

In [47]:

```
df.fillna({'Column A': 0, 'Column B': 99, 'Column C': df['Column C'].mean()})
```

Out[47]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	47.0	5
1	0.0	8.0	9.0	8
2	30.0	31.0	32.0	34
3	0.0	99.0	100.0	110

In [48]:

```
df.fillna(method='ffill', axis=0)
```

Out[48]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	NaN	5
1	1.0	8.0	9.0	8
2	30.0	31.0	32.0	34
3	30.0	31.0	100.0	110

In [49]:

```
df.fillna(method='ffill', axis=1)
```

Out[49]:

	Column A	Column B	Column C	Column D
0	1.0	2.0	2.0	5.0
1	NaN	8.0	9.0	8.0
2	30.0	31.0	32.0	34.0
3	NaN	NaN	100.0	110.0

Checking if there are NAs

The question is: Does this `Series` or `DataFrame` contain any missing value? The answer should be yes or no: `True` or `False`. How can you verify it?

Example 1: Checking the length

If there are missing values, `s.dropna()` will have less elements than `s`:

In [ ]:

```
s.dropna().count()
```

In [ ]:

```
missing_values = len(s.dropna()) != len(s)
missing_values
```

There's also a `count` method, that excludes `nan` s from its result:

In [ ]:

```
len(s)
```

In [ ]:

```
s.count()
```

So we could just do:

In [ ]:

```
missing_values = s.count() != len(s)
missing_values
```

More Pythonic solution `any`

The methods `any` and `all` check if either there's `any` True value in a Series or `all` the values are `True`. They work in the same way as in Python:

In [50]:

```
pd.Series([True, False, False]).any()
```

Out[50]:

```
True
```

In [51]:

```
pd.Series([True, False, False]).all()
```

Out[51]:

```
False
```

In [52]:

```
pd.Series([True, True, True]).all()
```

Out[52]:

```
True
```

The `isnull()` method returned a Boolean `Series` with `True` values wherever there was a `nan` :

In [ ]:

```
s.isnull()
```

So we can just use the `any` method with the boolean array returned:

In [ ]:

```
pd.Series([1, np.nan]).isnull().any()
```

In [ ]:

```
pd.Series([1, 2]).isnull().any()
```

In [ ]:

```
s.isnull().any()
```

A more strict version would check only the `values` of the Series:

In [ ]:

```
s.isnull().values
```

In [ ]:

```
s.isnull().values.any()
```