Lets discuss the tools for missing data, duplicate data, string manipulation, and some other analytical data transformations.

Handling Missing Data

1. Import Libraries Start by importing the necessary libraries.

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
string_data = pd.Series(["aardvark", np.nan, None, "avocado"])
string_data
string_data.notnull()
```

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

In [3]:
```

float_data = pd.Series([1, 2, None], dtype='float64')
float_data
float_data.isna()
```

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

Argument	Description
dropna	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
fillna	Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.
isnull	Return boolean values indicating which values are missing/NA.
notnull	Negation of isnull.

Filtering Out Missing Data

In [4]:
```
from numpy import nan as NA
```

In [5]:
```
data = pd.Series([1, np.nan, 3.5, np.nan, 7])
data
#data.isnull()
```

Out[5]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [6]:
```
data.dropna()
#OR
#data[data.notnull()]
```

Out[6]:
```
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 [7]:
```python
data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],
                     [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])
data
```

Out[7]:

	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 [8]:
```python
cleaned=data.dropna()
cleaned
```

Out[8]:

	0	1	2
0	1.0	6.5	3.0

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

In [9]:
```python
data.dropna(how="all")
```

Out[9]:

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

In [10]:
```python
data[4] = np.nan
data
```

Out[10]:

	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

Dropping Column wise

In [11]:
```python
data.dropna(axis="columns", how="all")
#data.dropna(axis=1, how="all")
```

Out[11]:

	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.

In [12]:

```python
#creates a NumPy array with shape (7, 3) filled with random numbers
df = pd.DataFrame(np.random.standard_normal((7, 3)))
df
df.iloc[:4, 1] = np.nan
df.iloc[:2, 2] = np.nan
df
```

Out[12]:

	0	1	2
0	-0.505635	NaN	NaN
1	-1.530843	NaN	NaN
2	-1.685005	NaN	0.283322
3	0.541645	NaN	2.174634
4	0.508546	0.505258	-1.038756
5	-2.010293	-0.440277	-0.309846
6	1.108961	-0.684461	-1.121969

In [13]:

```python
df.dropna()
```

Out[13]:

	0	1	2
4	0.508546	0.505258	-1.038756
5	-2.010293	-0.440277	-0.309846
6	1.108961	-0.684461	-1.121969

In [14]:

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

Out[14]:

	0	1	2
2	-1.685005	NaN	0.283322
3	0.541645	NaN	2.174634
4	0.508546	0.505258	-1.038756
5	-2.010293	-0.440277	-0.309846
6	1.108961	-0.684461	-1.121969

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 [15]:

```python
#The argument passed to fillna is a dictionary where keys represent column labels,
#and values represent the values to use for filling missing values in the corresponding columns

#In this case:
#For column 1 (1 in the dictionary), missing values will be filled with 0.5.
#For column 2 (2 in the dictionary), missing values will be filled with 0.

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

Out[15]:

	0	1	2
0	-0.505635	0.500000	0.000000
1	-1.530843	0.500000	0.000000
2	-1.685005	0.500000	0.283322
3	0.541645	0.500000	2.174634
4	0.508546	0.505258	-1.038756
5	-2.010293	-0.440277	-0.309846
6	1.108961	-0.684461	-1.121969

'fillna' returns a new object. This operation doesn't modify the original DataFrame in-place by default.

In [16]:
```
df
```

Out[16]:

	0	1	2
0	-0.505635	NaN	NaN
1	-1.530843	NaN	NaN
2	-1.685005	NaN	0.283322
3	0.541645	NaN	2.174634
4	0.508546	0.505258	-1.038756
5	-2.010293	-0.440277	-0.309846
6	1.108961	-0.684461	-1.121969

If you want to modify the DataFrame in-place, you can use the "inplace=True" parameter:

In [17]:
```
df.fillna({1: 0.5, 2: 0}, inplace=True)
df
```

Out[17]:

	0	1	2
0	-0.505635	0.500000	0.000000
1	-1.530843	0.500000	0.000000
2	-1.685005	0.500000	0.283322
3	0.541645	0.500000	2.174634
4	0.508546	0.505258	-1.038756
5	-2.010293	-0.440277	-0.309846
6	1.108961	-0.684461	-1.121969

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

In [18]:
```
#creates a NumPy array with shape (6, 3) filled with random numbers
df = pd.DataFrame(np.random.randn(6, 3))
df
```

Out[18]:

	0	1	2
0	0.582494	-0.989675	0.103343
1	0.080548	-0.657917	1.314994
2	-0.433641	0.475108	-0.890910
3	-0.369772	-0.042355	-0.147070
4	-0.748587	-0.826320	0.276759
5	1.207600	1.244975	0.638699

In [19]:
```
df.iloc[2:, 1] = np.nan # It sets all these values to NaN, targets the values in the column
                        # at index 1 (second column)  of the DataFrame starting from the third r
df.iloc[4:, 2] = np.nan
df
```

Out[19]:

	0	1	2
0	0.582494	-0.989675	0.103343
1	0.080548	-0.657917	1.314994
2	-0.433641	NaN	-0.890910
3	-0.369772	NaN	-0.147070
4	-0.748587	NaN	NaN
5	1.207600	NaN	NaN

In [20]:
```python
# "ffill" stands for forward fill,
# which means it fills missing values with the last observed non-null value along each column.
df.fillna(method="ffill")
```

Out[20]:

	0	1	2
0	0.582494	-0.989675	0.103343
1	0.080548	-0.657917	1.314994
2	-0.433641	-0.657917	-0.890910
3	-0.369772	-0.657917	-0.147070
4	-0.748587	-0.657917	-0.147070
5	1.207600	-0.657917	-0.147070

In [21]:
```python
df.fillna(method="ffill", limit=2) # it limits the forward fill to at most 2 consecutive NaN va
```

Out[21]:

	0	1	2
0	0.582494	-0.989675	0.103343
1	0.080548	-0.657917	1.314994
2	-0.433641	-0.657917	-0.890910
3	-0.369772	-0.657917	-0.147070
4	-0.748587	NaN	-0.147070
5	1.207600	NaN	-0.147070

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 [22]:
```python
data = pd.Series([1., np.nan, 3.5, np.nan, 7])
data
```

Out[22]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [23]:
```python
data.mean()
```

Out[23]:
```
3.8333333333333335
```

In [24]:
```python
data.fillna(data.mean())
```

Out[24]:
```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

#Fillna function arguments

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation; by default 'ffill' if function called with no other arguments
axis	Axis to fill on; default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

```
# END OF PART 1 of Data Wrangling
By: Prashant Bikram Shah
```