pandas

$y_it = \beta' x_{it} + \mu_i + \epsilon_{it}$

An Intro for Data Scientists

Presented by @jessecdaniel
Denver Data Science Day 2017

What Is Pandas?

Pandas is a library for Python that specifically provides tools for data manipulation and analysis. The shortcut commonly used for pandas is pd.

```
import pandas as pd
```

Its two main structures are Series and DataFrame. These are used so much, that we can actually import them from pandas so that we don't have to use pd.Series and pd.DataFrame when calling them.

```
from pandas import Series, DataFrame
```

Creating A Series

A pandas Series is a one-dimensional (1D) array-like object containing an array of data (of any numpy type) and an associated array of data labels called its index.

```
mySales = Series([32, 73, -1, 20])
mySales
```

```
0    32
1    73
2    -1
3    20
dtype: int64
```

You can use the .values method to view just its data without the index.

```
mySales.values
```

```
array([32, 73, -1, 20])
```

Indexing A Series

You can use the .index method to see the index (or labels) assigned to the rows or to assign the index.

```
mySales = Series([32, 73, -1, 20])
mySales.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

We can assign a new index with this syntax:

```
mySales.index = ['F','L','Fa','D']
```

```
Index([u'F', u'L', u'Fa', u'D'], dtype='object')
```

```
mySales
```

```
F      32
L      73
Fa     -1
D      20
dtype: int64
```

Indexing A Series

You can also assign the **`.index`** when you create the **`Series`**.

```
mySales2 = Series([32, 73, -1, 20], index=['Frisco',
           'Leadville', 'Fairplay', 'Dillon'])
mySales2
```

```
Frisco       32
Leadville    73
Fairplay     -1
Dillon       20
dtype: int64
```

```
mySales2.index
```

```
Index([u'Frisco', u'Leadville', u'Fairplay',
                u'Dillon'], dtype='object')
```

Accessing Values

Finally, you can use the **`.name`** method to label the column.

```
mySales2.name = "Stores"
```

```
Frisco       32
Leadville    73
Fairplay     -1
Dillon       20
Name: Stores, dtype: int64
```

You can reference values in the Series by using [] to indicate the specific indexes you want.

```
mySales2['Frisco']
```

```
32
```

To have multiple conditions, you need double braces (one for the index and one for the list of items).

```
mySales2[['Frisco', 'Dillon']]
```

```
Frisco    32
Dillon    20
Name: Stores, dtype: int64
```

Operating on Values

Let's assume the original data was in thousands, so to get the actual values we can multiply by 1000

```
mySales3 = mySales2 * 1000
mySales3
```

```
Frisco       32000
Leadville    73000
Fairplay     -1000
Dillon       20000
Name: Stores, dtype: int64
```

You can also use standard comparison (boolean) operators for selection.

```
mySales3[mySales3 > 10000]
```

```
Frisco       32000
Leadville    73000
Dillon       20000
Name: Stores, dtype: int64
```

Note you have to repeat mySales3 to make the complete condition

Creating A Series From A Dictionary

If we have data already in a Python dictionary, we can convert it to a pandas **`Series`** by using the **`Series`** function.

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000,
         'Utah': 5000}
obj1 = Series(sdata)
obj1
```

```
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

We can also use the index option. Note the interesting results.

```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj2 = Series(sdata, index=states)
obj2
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Detecting Missing Values

We can use the **`isnull`** and **`notnull`** to check for missing values. This will be useful later when cleaning up datasets.

```
obj2
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
pd.isnull(obj2)
```

```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
pd.notnull(obj2)
```

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

What is a DataFrame?

A DataFrame is a specialized two-dimensional (2D) container in pandas. We will use this as a way to store data that contains row and column information. By default, it will sort the columns alphabetically.

Note: A *column* in a DataFrame is really a Series object – we will use *column* and *Series* interchangably

Syntax to define the DataFrame:
DataFrame(data, columns = list, index = list)

DataFrame Operations

Methods
`columns`: lists/changes the column names
`index`: defines the row names/index

`.ix['indexname']`: returns the row data for that index
`.drop`: remove a row (axis=0 default) or column (axis=1)

Creating A DataFrame

The dictionary syntax is also used to populate a DataFrame. You can do this in one or two steps.

```
data = {'store': ['Parker', 'Frisco', 'Dillon',
                  'Leadville', 'Fairplay'],
       'year': [2010, 2011, 2012, 2013, 2014],
       'sales': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
frame
```

```
   sales      store  year
0    1.5     Parker  2010
1    1.7     Frisco  2011
2    3.6     Dillon  2012
3    2.4  Leadville  2013
4    2.9   Fairplay  2014
```

Sorting The Columns

If you want to change the order of the columns, you can use the **`columns`** option.

```
data = {'store': ['Parker', 'Frisco', 'Dillon',
                  'Leadville', 'Fairplay'],
       'year': [2010, 2011, 2012, 2013, 2014],
       'sales': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame2 = DataFrame(data,
                         columns=['store', 'year',
'sales'])
```

```
       store  year  sales
0     Parker  2010    1.5
1     Frisco  2011    1.7
2     Dillon  2012    3.6
3  Leadville  2013    2.4
4   Fairplay  2014    2.9
```

Indexing The DataFrame

We can use the **`index`** option to create the row names.

```
data = {'store': ['Parker', 'Frisco', 'Dillon',
                  'Leadville', 'Fairplay'],
       'year': [2010, 2011, 2012, 2013, 2014],
       'sales': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame2 = DataFrame(data,  columns=['store', 'year',
                                  'sales', 'rate'],
                  index=['Parker', 'Frisco',
                  'Dillon', 'Leadville', 'Fairplay'])
frame2
```

```
               store  year  sales rate
Parker        Parker  2010    1.5  NaN
Frisco        Frisco  2011    1.7  NaN
Dillon        Dillon  2012    3.6  NaN
Leadville  Leadville  2013    2.4  NaN
Fairplay    Fairplay  2014    2.9  NaN
```

We snuck in rate!

Renaming The Columns

We can use the **`columns`** option to change the column names.

```
           store  year  sales rate
Parker        Parker  2010    1.5  NaN
Frisco        Frisco  2011    1.7  NaN
Dillon        Dillon  2012    3.6  NaN
Leadville  Leadville  2013    2.4  NaN
Fairplay    Fairplay  2014    2.9  NaN
```

```
frame2.columns = ['Store','Year','Sales','Rate']
frame2
```

```
                Store  Year  Sales Rate
Parker         Parker  2010    1.5  NaN
Frisco         Frisco  2011    1.7  NaN
Dillon         Dillon  2012    3.6  NaN
Leadville   Leadville  2013    2.4  NaN
Fairplay     Fairplay  2014    2.9  NaN
```

Filling/Updating Values

We can put 3.5 in as the rates for all stores

```
                 Store   Year   Sales Rate
Parker          Parker   2010     1.5  NaN
Frisco         Frisco   2011     1.7  NaN
Dillon          Dillon   2012     3.6  NaN
Leadville    Leadville   2013     2.4  NaN
Fairplay      Fairplay   2014     2.9  NaN
```

```
frame2['Rate'] = 3.5
frame2
```

```
                 Store   Year   Sales   Rate
Parker          Parker   2010     1.5    3.5
Frisco         Frisco   2011     1.7    3.5
Dillon          Dillon   2012     3.6    3.5
Leadville    Leadville   2013     2.4    3.5
Fairplay      Fairplay   2014     2.9    3.5
```

Filling/Updating Values

Or we could use **arange** to put in a series of values.

	Store	Year	Sales	Rate
Parker	Parker	2010	1.5	3.5
Frisco	Frisco	2011	1.7	3.5
Dillon	Dillon	2012	3.6	3.5
Leadville	Leadville	2013	2.4	3.5
Fairplay	Fairplay	2014	2.9	3.5

```
frame2['Rate'] = np.arange(.5, 3, step =.5)
frame2
```

	Store	Year	Sales	Rate
Parker	Parker	2010	1.5	0.5
Frisco	Frisco	2011	1.7	1.0
Dillon	Dillon	2012	3.6	1.5
Leadville	Leadville	2013	2.4	2.0
Fairplay	Fairplay	2014	2.9	2.5

Filling/Updating Values

Or we could set all values that meet a criteria to a certain value.

	Store	Year	Sales	Rate
Parker	Parker	2010	1.5	0.5
Frisco	Frisco	2011	1.7	1.0
Dillon	Dillon	2012	3.6	1.5
Leadville	Leadville	2013	2.4	2.0
Fairplay	Fairplay	2014	2.9	2.5

```
frame2.Rate[frame2.Rate < 1.5] = 0
frame2
```

	Store	Year	Sales	Rate
Parker	Parker	2010	1.5	0.0
Frisco	Frisco	2011	1.7	0.0
Dillon	Dillon	2012	3.6	1.5
Leadville	Leadville	2013	2.4	2.0
Fairplay	Fairplay	2014	2.9	2.5

warning: A value is trying to be set on a copy of a slice from a DataFrame

Dropping A Row

If we need to remove a row you can use **drop**.

	Store	Year	Sales	Rate
Parker	Parker	2010	1.5	0.0
Frisco	Frisco	2011	1.7	0.0
Dillon	Dillon	2012	3.6	1.5
Leadville	Leadville	2013	2.4	2.0
Fairplay	Fairplay	2014	2.9	2.5

```
frame3 = frame2
frame3.drop('Dillon')
frame3
```

To drop multiple use []

```
frame3.drop(['Frisco','Fairplay'])
```

	Store	Sales	Rate
Parker	Parker	1.5	0.0
Frisco	Frisco	1.7	0.0
Leadville	Leadville	2.4	2.0
Fairplay	Fairplay	2.9	2.5

Dropping A Column

If we need to remove a column you can use **drop** with axis=1 option.

	Store	Year	Sales	Rate
Parker	Parker	2010	1.5	0.0
Frisco	Frisco	2011	1.7	0.0
Dillon	Dillon	2012	3.6	1.5
Leadville	Leadville	2013	2.4	2.0
Fairplay	Fairplay	2014	2.9	2.5

```
frame3 = frame2
frame3.drop('Store',axis=1)
frame3
```

	Year	Sales	Rate
Parker	2010	1.5	0.0
Frisco	2011	1.7	0.0
Dillon	2012	3.6	1.5
Leadville	2013	2.4	2.0
Fairplay	2014	2.9	2.5

Accessing A Column

If you want to just reference a single column, you can use either of these versions:

```
                 Store  Sales  Rate
Parker          Parker    1.5   0.0
Frisco          Frisco    1.7   0.0
Dillon          Dillon    3.6   1.5
Leadville    Leadville    2.4   2.0
Fairplay      Fairplay    2.9   2.5
```

```
frame3['Sales']
frame3.Sales
```

```
Parker       1.5
Frisco       1.7
Dillon       3.6
Leadville    2.4
Fairplay     2.9
Name: Sales, dtype: float64
```

Accessing A Subset of Columns

If you want to view multiple columns you can use:

```
frame3[['Sales','Rate']]
```

```
            Sales  Rate
Parker        1.5   0.5
Frisco        1.7   1.0
Dillon        3.6   1.5
Leadville     2.4   2.0
Fairplay      2.9   2.5
```

```
               Store  Sales  Rate
Parker        Parker    1.5   0.0
Frisco        Frisco    1.7   0.0
Dillon        Dillon    3.6   1.5
Leadville  Leadville    2.4   2.0
Fairplay    Fairplay    2.9   2.5
```

If you want to select a row, you use the index with **`.ix`**.

```
frame2.ix['Frisco']
```

```
Store    Frisco
Sales       1.7
Rate          1
Name: Frisco, dtype: object
```

Accessing A Subset of Rows & Columns

Finally, we will expand the use of the **`.ix`** method to view rows based on more conditions.

```
frame3.ix['Frisco', ['Sales', 'Rate']]
```

```
Sales    1.7
Rate       0
Name: Frisco, dtype: object
```

```
frame3.ix[:'Dillon', 'Sales']
```

```
Parker    1.5
Frisco    1.7
Dillon    3.6
Name: Sales, dtype: float64
```

```
frame3.ix[frame3.Sales > 2, 1:2]
```

```
 Sales
Dillon       3.6
Leadville    2.4
Fairplay     2.9
```

```
                 Store  Sales  Rate
Parker          Parker    1.5   0.0
Frisco          Frisco    1.7   0.0
Dillon          Dillon    3.6   1.5
Leadville    Leadville    2.4   2.0
Fairplay      Fairplay    2.9   2.5
```

Creating a DataFrame from a File

First, you want to set your path to be where the file is located. Save ex1.csv to a folder on your hard drive.

Next, change the working directory with the command **`os.chdir().`**

Then we can use the pandas command pd.read_csv() to open the csv file.

```
os.chdir('Users/Jesse/ … /Lesson 2') #Mac
# os.chdir('C:\\Users\\Jesse\\ … \\Lesson 2') #Windows
import pandas as pd
df = pd.read_csv('ex1.csv')
df
```

```
   a   b   c   d      Store
0  1   2   3   4     Parker
1  5   6   7   8         DU
2  9  10  11  12  Littleton
```

Creating a DataFrame from a File

There is also an alternate command **`pd.read_table()`**. This command requires you to specifically list the separator. This also works for files that have different delimiters like a semicolon, tab or space.

```
df2 = pd.read_table('ex1.csv', sep=',')
df2
```

```
   a   b   c   d      Store
0  1   2   3   4     Parker
1  5   6   7   8         DU
2  9  10  11  12  Littleton
```

Creating a DataFrame from a File

There is an option **`header=None`** if the file has no header row.

```
df3 = pd.read_csv('ex2.csv', header=None)
df3
```

```
   0   1   2   3          4
0  1   2   3   4     Parker
1  5   6   7   8         DU
2  9  10  11  12  Littleton
```

So by default it assumes there IS a header row of column names

Creating a DataFrame from a File

If the file has no headers, you can specify them when you read them in with the **names** option.

```
df4 = pd.read_csv('ex2.csv',
                  names=['a', 'b', 'c', 'd', 'Store'])
df4
```

```
   a   b   c   d      Store
0  1   2   3   4     Parker
1  5   6   7   8         DU
2  9  10  11  12  Littleton
```

Creating a DataFrame from a File

You can define one of the columns to be the index or row names. For ease of reading the code, you can store the names in a list.

```
names = ['a', 'b', 'c', 'd', 'Store']
df5 = pd.read_csv('ex2.csv', names=names,
                                             index_col='Store')
df5
```

```
            a   b   c   d
Store
Parker      1   2   3   4
DU          5   6   7   8
Littleton   9  10  11  12
```

Creating a DataFrame from a File

If you have space delimited files, you can use the separator symbol
`\s+` (this is known as a regular expression)

```
df6 = pd.read_table('ex3.txt', sep='\s+')
df6
```

```
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

ex3.txt

```
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Creating a DataFrame from a File

If your file has an irregular form you can still read in the data.

ex4.csv

```
# hey!
a,b,c,d,Store
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,Parker
5,6,7,8,DU
9,10,11,12,Littleton
```

Header
Lines to ignore

```
df7 = pd.read_csv('ex4.csv', skiprows=[0, 2, 3])
df7
```

	a	b	c	d	Store
0	1	2	3	4	Parker
1	5	6	7	8	DU
2	9	10	11	12	Littleton

Creating a DataFrame from a File

It will also handle missing values. If you have blank or NA in your text file, it will convert it to the numpy/pandas missing NaN.

ex5.csv

```
a,b,c,d,Store
1,2,3,4,NA
5,6,,8,DU
9,10,11,12,Littleton
```

```python
df8 = pd.read_csv('ex5.csv')
df8
```

	a	b	c	d	Store
0	1	2	3.0	4	NaN
1	5	6	NaN	8	DU
2	9	10	11.0	12	Littleton

Creating a File from a DataFrame

The to_csv function will take a pandas dataframe and output the contents to a text file that is comma delimited.

```
   a   b   c   d      Store
0  1   2   3   4     Parker
1  5   6   7   8         DU
2  9  10  11  12  Littleton
```

```
df4.to_csv('outdf4.csv')
```

outdf4.csv

```
,a,b,c,d,Store
0,1,2,3,4,Parker
1,5,6,7,8,DU
2,9,10,11,12,Littleton
```

	A	B	C	D	E	F
1		a	b	c	d	Store
2	0	1	2	3	4	Parker
3	1	5	6	7	8	DU
4	2	9	10	11	12	Littleton

Creating a File from a DataFrame

If the data has an index column of row names, then that column will become a column in the output text file.

```
           a   b   c   d
Store
Parker     1   2   3   4
DU         5   6   7   8
Littleton  9  10  11  12
```

```
df5.to_csv('outdf5.csv')
```

outdf5.csv

```
Store,a,b,c,d
Parker,1,2,3,4
DU,5,6,7,8
Littleton,9,10,11,12
```

	A	B	C	D	E
1	Store	a	b	c	d
2	Parker	1	2	3	4
3	DU	5	6	7	8
4	Littleton	9	10	11	12

Creating a File from a DataFrame

We can have the **`to_csv`** output to the console so that we can check what we are sending before we send it, and then change it back to the .csv file name to write to the file.

To output to the console, we use **`sys.stdout`** from the **`sys`** package.

```
import sys
df5.to_csv(sys.stdout)
```

```
Store,a,b,c,d
Parker,1,2,3,4
DU,5,6,7,8
Littleton,9,10,11,12
```

Creating a File from a DataFrame

We can specify the delimiter with the **`sep`** option.

```
df5.to_csv(sys.stdout, sep='|')
```

```
Store|a|b|c|d
Parker|1|2|3|4
DU|5|6|7|8
Littleton|9|10|11|12
```

```
df5.to_csv(sys.stdout, sep=' ')
```

```
Store a b c d
Parker 1 2 3 4
DU 5 6 7 8
Littleton 9 10 11 12
```

To write to a file, just replace sys.stdout with 'xxxx.csv'

```
df5.to_csv('df5space.csv', sep=' ')
```

Creating a File from a DataFrame

We can control what the missing values will be in the text file. Here we replace NaN with NULL.

```
   a   b     c   d      Store
0  1   2   3.0   4        NaN
1  5   6   NaN   8         DU
2  9  10  11.0  12  Littleton
```

```
df8.to_csv(sys.stdout, na_rep='NULL')
```

```
,a,b,c,d,Store
0,1,2,3.0,4,NULL
1,5,6,NULL,8,DU
2,9,10,11.0,12,Littleton
```

```
df8.to_csv('df8NULL.csv', na_rep='NULL')
```

Creating a File from a DataFrame

We can remove the **`header`** row or the **`index`** column by setting those options to False.

```
           a   b   c   d
Store
Parker     1   2   3   4
DU         5   6   7   8
Littleton  9  10  11  12
```

```
df5.to_csv(sys.stdout, index=False, header=False)
```

```
1,2,3,4
5,6,7,8
9,10,11,12
```

We can also select the columns to output with the columns option.

```
df5.to_csv(sys.stdout, index=False,
                                columns=['a', 'b', 'c'])
```

```
a,b,c
1,2,3
5,6,7
9,10,11
```

Handling Missing Data

Pandas provides some built in methods for dealing with missing data. It is helpful that when you apply descriptive statistics on pandas objects (DataFrames and Series) that the missing values are skipped.

`np.nan` (not a number) stands for the missing value from numpy. When you view output it will be NaN.

You will sometimes see

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

used so that you can simplify the expression **`np.nan`** to just **`NA`**.

`None` is the built in Python **`None`** value to represent missing.

`isnull()` method returns True/False if values are missing or not.

Handling Missing Data

```
import pandas as pd
import numpy as np
string_data = pd.Series(
          ['aardvark', 'artichoke', np.nan, None])
string_data
```

Note the syntax – don't just put NaN!!!

```
0      aardvark
1     artichoke
2           NaN
3          None
dtype: object
```

```
string_data.isnull()
```

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

Handling Missing Data

Let's create a series with a missing value. We can do it three different ways: with **`None`**, **`np.nan`**, or **`NA`** after using the **`from numpy import nan as NA`**.

```
data = pd.Series([1,4,None, 7, 9])
data = pd.Series([1,4,np.nan, 7, 9])
from numpy import nan as NA
data = pd.Series([1,4,NA, 7, 9])
data
```

```
0    1.0
1    4.0
2    NaN
3    7.0
4    9.0
dtype: float64
```

Handling Missing Data

Some calculations will, by default, ignore missing values. These summaries from **numpy** will ignore missing values

```
np.mean(data) or data.mean()
np.max(data) or data.max()
np.min(data) or data.min()
np.sum(data) or data.sum()
np.std(data) or data.std()
np.var(data) or data.var()
data.median()
```

```
5.25
9.0
1.0
21.0
3.031088913245535
9.1875
5.5
```

But some, like np.median, won't work so there is a **nanmedian** version.

```
print(np.median(data)) #nan
print(np.nanmedian(data))
```

```
nan
5.5
```

Handling Missing Data

The **`dropna`** method will return only the non-null data and index values.

- When using this with a DataFrame, you can determine if you want to drop rows or columns that are all NA or just those containing any NAs.
- By default, **`dropna`** will drop any row or column (if you use axis = 1) that has at least 1 NA.
- Using the option **`how='all'`** will drop rows or columns that are all NA.

Handling Missing Data

```
0      1.0
1      4.0
2      NaN
3      7.0
4      9.0
```

We can view the data removing the missing values.

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

```
0      1.0
1      4.0
3      7.0
4      9.0
dtype:  float64
```

Or we can remove the values from the DataFrame with **`dropna`**.

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

```
0      1.0
1      4.0
3      7.0
4      9.0
dtype:  float64
```

Note if we want it permanent reassign with data =

Handling Missing Data

Let's compare the output with different **`dropna`** configurations.

```
data2 = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
                      [NA, NA, NA], [NA, 6.5, 3.]])
data2
```

```
     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
```

Both of these are the same (**`dropna`** default axis is 0 (rows))

```
data2.dropna()
data2.dropna(axis=0)
```

```
     0    1    2
0  1.0  6.5  3.0
```

and if we use the axis = 1 for columns, we get an empty DataFrame!

```
data2.dropna(axis=1)
```

```
Empty DataFrame
Columns: []
```

Handling Missing 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

The how='all' will drop if all are NA.

```
data2.dropna(how='all')
```

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

```
data2.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
```

```
data2.dropna(axis=0, how='all')
```

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

SAME

Handling Missing Data

Let's create a larger DataFrame to work with.

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

```
          0         1         2
0 -0.906506       NaN       NaN
1 -0.834261       NaN       NaN
2 -1.082828       NaN       NaN
3  1.347348       NaN -0.711371
4  0.602594       NaN -2.454121
5 -0.971363  1.635675 -1.620692
6 -0.132770 -2.398985 -0.425243
```

Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the thresh (threshold) argument.

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

```
          0         1         2
3  1.347348       NaN -0.711371
4  0.602594       NaN -2.454121
5 -0.971363  1.635675 -1.620692
6 -0.132770 -2.398985 -0.425243
```

Handling 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. 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 -0.906506  0.000000  0.000000
1 -0.834261  0.000000  0.000000
2 -1.082828  0.000000  0.000000
3  1.347348  0.000000 -0.711371
4  0.602594  0.000000 -2.454121
5 -0.971363  1.635675 -1.620692
6 -0.132770 -2.398985 -0.425243
```

Handling Missing Data

Calling **`fillna`** with a dictionary you can use a different fill value for each column.

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

```
          0         1         2
0 -0.906506  0.500000 -1.000000
1 -0.834261  0.500000 -1.000000
2 -1.082828  0.500000 -1.000000
3  1.347348  0.500000 -0.711371
4  0.602594  0.500000 -2.454121
5 -0.971363  1.635675 -1.620692
6 -0.132770 -2.398985 -0.425243
```

Handling Missing Data

The result of **`fillna`** will be a new object, but you can modify the existing object in place.

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

```
          0         1         2
0 -0.906506  0.500000 -1.000000
1 -0.834261  0.500000 -1.000000
2 -1.082828  0.500000 -1.000000
3  1.347348  0.500000 -0.711371
4  0.602594  0.500000 -2.454121
5 -0.971363  1.635675 -1.620692
6 -0.132770 -2.398985 -0.425243
```

Handling Missing Data

Let's create a new DataFrame with a different missing pattern.

```
df2 = pd.DataFrame(np.random.randn(6, 3))
df2.ix[2:, 1] = NA
df2.ix[4:, 2] = NA
df2
```

```
          0         1         2
0  1.986148 -2.198400 -1.393433
1 -0.364097 -1.747339  2.310672
2 -0.436066       NaN  0.450623
3 -1.528445       NaN  0.972657
4  0.914882       NaN       NaN
5  0.472371       NaN       NaN
```

Handling Missing Data

You can use interpolation methods with **`fillna`** like **`ffill`** for "forward fill".

```
df2.fillna(method='ffill')
```

```
          0         1         2
0  1.986148 -2.198400 -1.393433
1 -0.364097 -1.747339  2.310672
2 -0.436066 -1.747339  0.450623
3 -1.528445 -1.747339  0.972657
4  0.914882 -1.747339  0.972657
5  0.472371 -1.747339  0.972657
```

There is also a **`limit`** option, where **`limit = 2`** means it will only fill up to 2 missing values.

```
df2.fillna(method='ffill', limit=2)
```

```
          0         1         2
0  1.986148 -2.198400 -1.393433
1 -0.364097 -1.747339  2.310672
2 -0.436066 -1.747339  0.450623
3 -1.528445 -1.747339  0.972657
4  0.914882       NaN  0.972657
5  0.472371       NaN  0.972657
```

Handling Missing Data

Finally, you can also do other things like pass the mean or median value of a Series instead of a constant value.

```
data
```

```
0    1.0
1    4.0
2    NaN
3    7.0
4    9.0
dtype: float64
```

```
data.mean()
```

```
5.25
```

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

```
0    1.00
1    4.00
2    5.25
3    7.00
4    9.00
dtype: float64
```

Handling Missing Data

Finally, you can do this for a DataFrame as well and using a Dictionary you can determine different replacement values for each column.

```
          a         b         c
0  1.986148 -2.198400 -1.393433
1 -0.364097 -1.747339  2.310672
2 -0.436066       NaN  0.450623
3 -1.528445       NaN  0.972657
4  0.914882       NaN       NaN
5  0.472371       NaN       NaN
```

```
df2.b.mean()
```

```
-1.9728697913734363
```

```
df2.c.median()
```

```
0.711639629194302
```

```
df2.fillna({'b': df2.b.mean(), 'c': df2.c.median()})
df2
```

```
          a         b         c
0  1.986148 -2.198400 -1.393433
1 -0.364097 -1.747339  2.310672
2 -0.436066 -1.972870  0.450623
3 -1.528445 -1.972870  0.972657
4  0.914882 -1.972870  0.711640
5  0.472371 -1.972870  0.711640
```

More Information

Pandas: http://pandas.pydata.org

Thank You!

Please connect with me on LinkedIn @ linkedin.com/in/jcdaniel91

Or follow me on Twitter @jessecdaniel