Lecture 7: Pandas DataFrames

Goals for Today:

Pandas:

- Dataframes
- Initializing
- Modifying
- Operators
- Methods
- Example

Pandas DataFrames vs Series

Pandas **dataframes** are similar to **series** except that they are **two dimensional** and hold a **header/label** for each **column.**

In a practical sense, dataframes are natural for holding tabular data, where you might have a bunch of columns with different headers aligned on some variable or property.

At the implementation level, you can think of dataframes as combining **multiple series** into a single object with a **shared index**.

According to the pandas docs: "you can think of dataframes as a **dictionary of series**" . Thus, dataframes share many of the attributes and methods of series, except that they have specialized arguments for negotiating which columns to apply things to.

<u>Dataframes are the main pandas class</u>

Pandas Overview

“DataFrame” Example:

- Special **Headers** for each column
- Special **Index** for each row
- Note that **dataframes** can store distinct types of data in each column, accommodate **missing values**, and are ordered objects.
- You can think of **dataframes** as a “dictionary of series” where each key is the column name and each value is the column (i.e., 1D **series** object).

	year	plant	yield(tons)	costs(100k)
0	2015.0	lafayette	61.801602	70.658986
1	2016.0	lafayette	61.624070	70.690350
2	2017.0	lafayette	63.627158	NaN
3	2018.0	NaN	63.282196	68.685577
4	2019.0	lafayette	73.332652	74.523847
5	2020.0	lafayette	75.885230	66.967705
6	NaN	NaN	NaN	NaN
7	NaN	NaN	72.140582	74.971921
8	2017.0	houston	59.096354	70.174623
9	2018.0	NaN	NaN	69.036116
10	2019.0	houston	65.636263	NaN
11	2020.0	houston	71.006384	65.451957
12	NaN	baton_rouge	73.646556	NaN
13	2016.0	baton_rouge	63.460700	70.217273
14	2017.0	baton_rouge	59.453030	70.054766
15	2018.0	baton_rouge	63.330415	65.652063
16	2019.0	baton_rouge	63.639207	69.082489
17	2020.0	baton_rouge	69.226561	NaN

Initializing Dataframes – with Lists

Initializing a pandas dataframe is as simple passing a list to the `pd.DataFrame()` constructor:

```
a = pd.DataFrame(range(5))
print(a)
```

```
   0
0  0
1  1
2  2
3  3
4  4
```

Similar to series, you can see that the dataframes are initialized with an integer **index**.

Dataframes also have an additional 0 over the top of the column. This is the **column label** (sometimes we will call it a **header** to distinguish between column labels and row labels). The headers can be customized:

```
b = pd.DataFrame(range(5), columns = [ "c1" ])
print(b)
```

```
  c1
0  0
1  1
2  2
3  3
4  4
```

Initializing Dataframes – with Lists and Arrays

When we pass a list to the constructor it creates a column, not a row:

```
a = pd.DataFrame(range(5))
print(a)
```

```
   0
0  0
1  1
2  2
3  3
4  4
```

Whereas a list of lists is interpreted as a list of rows:

```
c = pd.DataFrame([range(5),range(5)])
print(c)
```

```
   0  1  2  3  4
0  0  1  2  3  4
1  0  1  2  3  4
```

If you were just trying to initialize a large chunk of numbers, using a numpy array would be more natural:

```
d = pd.DataFrame(np.zeros([2,6]))
print(d)
```

```
     0    1    2    3    4    5
0  0.0  0.0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0  0.0  0.0
```

Initializing Dataframes – with Dictionaries

You can pass multiple columns and their headers at once using dictionaries:

```
e = {"c0":[0,3],"c1":[1,4],"c2":[2,5]}
e = pd.DataFrame(e,index=["r0","r1"])
print(e)
```

```
    c0  c1  c2
r0   0   1   2
r1   3   4   5
```

The keys are interpreted as column labels, and each value (list) is interpreted as column values. The lists must all be the same length.

Renaming Labels: The index and columns are both attributes of the dataframe (`dataframe.index` and `dataframe.columns`, respectively) that can be reassigned in place:

```
e.index = ["trial_1","trial_2"]
e.columns = ["ease","use","quality"]
print(e)
```

```
         ease  use  quality
trial_1     0    1        2
trial_2     3    4        5
```

(The formulation in trial_2 is clearly superior!)

Initializing Dataframes – from File

Pandas supports importing data from a large variety of file types, including excel, delimited text files, and JSON, among many others. Initializing dataframes from files is more common than the direct initializations described above. The associated `pd.` functions are:

- `read_csv(file,index_col=None,sep=',',header=0)` can be used to initialize a dataframe from a comma delimited file. The `index_col` argument can be optionally assigned to use one of the columns as an index for the dataframe. The `sep` argument can be used to read space-delimited or arbitrarily delimited files. The `header` argument can be used to select the line used for defining the header names

- `read_excel(file,sheetname=0,header=0,index_col=None)` can be used to initialize a dataframe from an excel spreadsheet. The `sheetname` argument can be used to read a particular sheet from a multisheet spreadsheet (0-indexed, or you can supply the name directly). If you supply a list to `sheetname` then a dictionary is returned, with keys corresponding to the sheet names and values corresponding to the individual dataframes. The `header` and `index_col` arguments have similar behavior to `pd.read_csv`.

Initializing Dataframes – from File

Example of typical usage:

```
f = pd.read_csv('impurities.txt',sep=r"\s+")
print(f)
```

```
          Temp(C)  c_A(M)  c_B(M)  Purity(%)
Sample
0            10.0     0.1     0.5   0.542962
1            10.0     0.1     0.6   0.461638
2            10.0     0.1     0.7   0.428392
3            10.0     0.1     0.8   0.450456
4            10.0     0.1     0.9   0.506773
...           ...     ...     ...        ...
120          50.0     0.5     0.5   0.942649
121          50.0     0.5     0.6   0.881588
122          50.0     0.5     0.7   0.817216
123          50.0     0.5     0.8   0.886751
124          50.0     0.5     0.9   0.911524
[125 rows x 4 columns]
```

Note: `sep=" "` only uses a single white-space as a delimiter, whereas the `sep=r"\s+"` is recommended usage (raw-string interpretation) for using contiguous spaces as a delimiter.

Top of `impurities.txt`:

```
Sample       Temp(C)      c_A(M)       c_B(M)       Purity(%)
000000       10.000000    0.100000     0.500000     0.542962
000001       10.000000    0.100000     0.600000     0.461638
```

Dataframe – Upcasting Behavior

Dataframes exhibit similar up-casting behavior to series, except that up-casting is applied on a **column by column** basis:

```
g = pd.DataFrame([[1,2+5J],[100.0,"100"]])
print(g)
print("\ntypes:\n{}".format(g.dtypes))
```

```
       0       1
0    1.0  (2+5j)
1  100.0     100

types:
0 float64
1 object
dtype: object
```

You can use the `dtypes` argument to attempt to cast a specific type.

You can use the `dataframe.astype()` method to convert types by column or the whole dataframe.

Dataframe – Slicing

If you use `[]` notation, label-based slicing is interpreted in "row-label" fashion.

If you use integers it is interpreted in "row-position" fashion.

If you utilize `[]` notation with a single column label then the corresponding column is returned as series.

Confused? We'll briefly review these here for completeness, but the subsequent `iloc` and `loc` methods are more common for slicing dataframes:

```
h = pd.DataFrame({"c0":[0,3,6],"c1":[1,4,7],"c2":[2,5,8]},
index=["r0","r1","r2"])

# Example 1: slice rows by position
print("h[0:2]:\n{}\n".format(h[0:2]))

# Example 2: return every other row
print("h[::2]:\n{}\n".format(h[::2]))

# Example 3: return two rows
print("h['r1':'r2']:\n{}\n".format(h['r1':'r2']))

# Example 4: return a column as a series
print("h['c1']:\n{}\n".format(h['c1']))

# Example 5: return part of a column
print("h['c1'][:1]:\n{}\n".format(h['c1'][1:]))
```

```
h[0:2]:
   c0  c1  c2
r0  0   1   2
r1  3   4   5

h[::2]:
   c0  c1  c2
r0  0   1   2
r2  6   7   8

h['r1':'r2']:
   c0  c1  c2
r1  3   4   5
r2  6   7   8
```

Dataframe – Slicing

If you use `[]` notation, label-based slicing is interpreted in "row-label" fashion.

If you use integers it is interpreted in "row-position" fashion.

If you utilize `[]` notation with a single column label then the corresponding column is returned as series.

Confused? We'll briefly review these here for completeness, but the subsequent `iloc` and `loc` methods are more common for slicing dataframes:

```
h = pd.DataFrame({"c0":[0,3,6],"c1":[1,4,7],"c2":[2,5,8]},
index=["r0","r1","r2"])

# Example 1: slice rows by position
print("h[0:2]:\n{}\n".format(h[0:2]))

# Example 2: return every other row
print("h[::2]:\n{}\n".format(h[::2]))

# Example 3: return two rows
print("h['r1':'r2']:\n{}\n".format(h['r1':'r2']))

# Example 4: return a column as a series
print("h['c1']:\n{}\n".format(h['c1']))

# Example 5: return part of a column
print("h['c1'][:1]:\n{}\n".format(h['c1'][1:]))
```

```
h['c1']:
r0  1
r1  4
r2  7
Name: c1, dtype: int64

h['c1'][:1]:
r1  4
r2  7
Name: c1, dtype: int64
```

Dataframe – Slicing

`iloc` and `loc` Methods: For dataframes it is more common to utilize `iloc[row_slice,column_slice]` and `loc[row_slice,column_slice]` for position-based and label-based slicing, respectively.

`.iloc` behaves similarly to numpy array slicing (end exclusive), while `.loc` provides access to label based slicing (end inclusive):

```
h = pd.DataFrame({"c0":[0,3,6],"c1":[1,4,7],"c2":[2,5,8]},index=["r0","r1","r2"])

# Example 1: location based slicing (row,col)
print("h.iloc[0:2,0:2]:\n{}\n".format(h.iloc[1:,0:2]))

# Example 2: label based slicing (row,col)
print("h.loc['r1':'r2']['c2']:\n{}\n".format(h.loc['r1':'r2']['c2']))
```

Example 1: `iloc[row_slice,col_slice]` behaves similar to slicing with numpy arrays.

Example 2: `loc[row_slice,col_slice]` is end-inclusive.

```
h.iloc[0:2,0:2]:
  c0 c1
r1 3   4
r2 6   7

h.loc['r1':'r2']['c2'
]: r1 5
r2 8
Name: c2, dtype:
int64
```

Dataframe – Slicing

You can also supply lists of labels or positions to `.loc` and `.iloc`, respectively to return specific rows/columns in variable order:

```
# Example 3: list-based iloc
print("h.iloc[[1,0],[2,0]]:\n{}\n".format(h.iloc[[1,0],[2,0]]))

# Example 4: list-based loc
print("h.loc[['r2','r0','r1'],['c1','c0']]:\n{}\n".\
       format(h.loc[['r2','r0','r1'],['c1','c0']]))
```

```
h.iloc[[1,0],[2,0]]:
    c2  c0
r1   5   3
r0   2   0

h.loc[['r2','r0','r1'],['c1','c0']]:
    c1  c0
r2   7   6
r0   1   0
r1   4   3
```

Dataframe – Conditional Slicing

Conditional slicing on dataframes results in a returned series that has all `False` values masked as `NaN`. You can chain conditions as with series:

```
h = pd.DataFrame({"c0":[0,3,6],"c1":[1,4,7],"c2":[2,5,8]},index=["r0","r1","r2"])

# Example 1: conditional slicing on dataframe
print("h[h>4]:\n{}".format(h[h>4]))

# Example 2: chained conditional slicing
print("\nh[h>4][h<7]:\n{}".format(h[h>4][h<7]))
```

```
h[h>4]:
    c0  c1  c2
r0 NaN NaN NaN
r1 NaN NaN 5.0
r2 6.0 7.0 8.0

h[h>4][h<7]:
    c0  c1  c2
r0 NaN NaN NaN
r1 NaN NaN 5.0
r2 6.0 NaN NaN
```

Dataframe – Operator Behavior

By default, operators align on both column and row labels:

```
i = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r1","r2"],\
                 columns=["c0","c1","c2"])
j = pd.DataFrame(np.arange(9).reshape([3,3])[::-1][:,::-1],index=["r2","r1","r0"],\
                 columns=["c2","c1","c0"])
k = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r1","r2"],\
                 columns=["c0","c0","c0"])
l = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r0","r0"],\
                 columns=["c0","c1","c2"])
m = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r1","r1","r1"],\
                 columns=["c0","c0","c0"])

# Example 1: label matched operator application
print("i+i:\n{}".format(i+i))

# Example 2: labels match on both rows and columns
print("i+j:\n{}".format(i+j))
```

Example 1:

```
i+i:
    c0  c1  c2
r0   0   2   4
r1   6   8  10
r2  12  14  16
```

Example 2:

```
i+j:
    c0  c1  c2
r0   0   2   4
r1   6   8  10
r2  12  14  16
```

Dataframe – Operator Behavior

By default, operators align on both column and row labels:

```
i = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r1","r2"],\
                 columns=["c0","c1","c2"])
j = pd.DataFrame(np.arange(9).reshape([3,3])[::-1][:,::-1],index=["r2","r1","r0"],\
                 columns=["c2","c1","c0"])
k = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r1","r2"],\
                 columns=["c0","c0","c0"])
l = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r0","r0"],\
                 columns=["c0","c1","c2"])
m = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r1","r1","r1"],\
                 columns=["c0","c0","c0"])

# Example 3: repeated columns (c0 from i is added to all c0 from k)
print("\ni+k:\n{}".format(i+k))

# Example 4: repeated rows (r0 from i is added to all r0 from l)
print("\ni+l:\n{}".format(i+l))
```

Example 3:

```
i+k:
    c0  c0  c0   c1   c2
r0   0   1   2  NaN  NaN
r1   6   7   8  NaN  NaN
r2  12  13  14  NaN  NaN
```

Example 4:

```
i+l:
     c0   c1    c2
r0  0.0  2.0   4.0
r0  3.0  5.0   7.0
r0  6.0  8.0  10.0
r1  NaN  NaN   NaN
r2  NaN  NaN   NaN
```

Dataframe – Operator Behavior

By default, operators align on both column and row labels:

```
i = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r1","r2"],\
                 columns=["c0","c1","c2"])
j = pd.DataFrame(np.arange(9).reshape([3,3])[::-1][:,::-1],index=["r2","r1","r0"],\
                 columns=["c2","c1","c0"])
k = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r1","r2"],\
                 columns=["c0","c0","c0"])
l = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r0","r0","r0"],\
                 columns=["c0","c1","c2"])
m = pd.DataFrame(np.arange(9).reshape([3,3]),index=["r1","r1","r1"],\
                 columns=["c0","c0","c0"])

# Example 5: repeated rows and columns (only r1,c1 from i is added)
print("\ni+m:\n{}".format(i+m))

# Example 6: adding dataframes with both repeated labels
print("\nm+m:\n{}".format(m+m))
```

Example 5:

```
i+m:
     c0    c0    c0   c1   c2
r0  NaN   NaN   NaN  NaN  NaN
r1  3.0   4.0   5.0  NaN  NaN
r1  6.0   7.0   8.0  NaN  NaN
r1  9.0  10.0  11.0  NaN  NaN
r2  NaN   NaN  NaN   NaN  NaN
```

Example 6:

```
m+m:
    c0  c0  c0
r1   0   2   4
r1   6   8  10
r1  12  14  16
```

Dataframe – Operator Behavior

All numerical operators act according to the same logic as the preceding examples.

Logical operators require identical row/column labeling (repeated labels are allowed, but must be identical).

Lastly, when you apply operators to dataframes and individual objects (scalars and strings) it is applied to every element of the dataframe:

```
o = pd.DataFrame(np.arange(6).reshape([2,3]))

# Example 1: a scalar applied to a dataframe
print(o+10)

# Example 2: a row applied to a dataframe
print(o+np.array([2,3,4]))
```

```
    0   1   2
0  10  11  12
1  13  14  15

   0  1  2
0  2  4  6
1  5  7  9
```

If an array with length equal to the number of columns is given, then it is applied row-by-row

Dataframe – Adding Columns

You can add additional columns to dataframes using dictionary-like notation:

```
dataframe[header] = column
```

In this case, the new column is added to the end:

```
p = pd.DataFrame({"Temperature_K":\
[300,310,320],\
"Yield":[56.2,64.3,60.9]})
print("p:\n{}".format(p))

# Example 1: dictionary-like method
p["Facility"] = \
["Lexington","Lafayette","Lafayette"]
print("\nafter add:\n{}".format(p))
```

```
p:
   Temperature_K Yield
0            300  56.2
1            310  64.3
2            320  60.9

after add:
   Temperature_K Yield  Facility
0            300  56.2 Lexington
1            310  64.3 Lafayette
2            320  60.9 Lafayette
```

If the header-label already exists, then the new column will overwrite the old one

Dataframe – Adding Columns

The `dataframe.insert(pos,col_label,col_values)` method behaves similarly, but you can specify the position:

```
# Example 2: insert method
p.insert(1,"Pressure",[1.0,10.0,100.0])
print("\nafter p.insert():\n{}".format(p))
```

```
after p.insert():
   Temperature_K Pressure Yield  Facility
0            300      1.0  56.2 Lexington
1            310     10.0  64.3 Lafayette
2            320    100.0  60.9 Lafayette
```

The `dataframe.assign(label1=values1,…)` method allows you to insert multiple columns at once:

```
# Example 3: assign method
p = p.assign(Batch=[45332,45332,45332],Temperature_C=p["Temperature_K"]-273.15)
print("\nafter p.assign():\n{}".format(p))
```

```
after p.assign():
   Temperature_K Pressure Yield  Facility Batch Temperature_C
0            300      1.0  56.2 Lexington 45332         26.85
1            310     10.0  64.3 Lafayette 45332         36.85
2            320    100.0  60.9 Lafayette 45332         46.85
```

Dataframe – Adding Rows

Similar to series you can add rows to dataframes using the `dataframe.append()` method. As usual with Pandas objects, it is best to do this all at once to be memory efficient.

```
p = pd.DataFrame({"Temperature_K":[300,310,320],"Yield":[56.2,64.3,60.9]})
q = pd.DataFrame({"Yield":[56.2,64.3,60.9],"Temperature_K":[300,310,320]})
r = pd.DataFrame({"Temperature_K":[300,310,320],"Conc":[0.1,0.1,0.1]})
print("Example 1:\n{}".format(p.append(q,sort=True)))
print("\nExample 2:\n{}".format(p.append(r,sort=True)))
```

Note: append aligns columns based on shared labels.

Note: the optional `sort=True/False` argument controls whether the columns are sorted after the append. Currently `sort=True` is the default, but in the future `sort=False` will be the default.

```
Example 1:
   Temperature_K Yield
0            300  56.2
1            310  64.3
2            320  60.9
0            300  56.2
1            310  64.3
2            320  60.9

Example 2:
   Conc  Temperature_K Yield
0   NaN            300  56.2
1   NaN            310  64.3
2   NaN            320  60.9
0   0.1            300   NaN
1   0.1            310   NaN
2   0.1            320   NaN
```

Dataframe – Combining Dataframes

If you want to combine multiple dataframes, similar to append, the `concat([list_of_df],axis=0)` function can be used. This aligns on labels and stiches the dataframes together. The optional argument `axis` puts the dataframes side by side when it is 1 and stacks them when it is 0:

```
p = pd.DataFrame({"Temperature_K":[300,310,320],"Yield":[56.2,64.3,60.9]})
q = pd.DataFrame({"Yield":[56.2,64.3,60.9],"Temperature_K":[300,310,320]})
r = pd.DataFrame({"Temperature_K":[300,310,320],"Conc":[0.1,0.1,0.1]})
print("Example 1:\n{}".format(pd.concat([p,q],axis=0,sort=False)))
print("\nExample 2:\n{}".format(pd.concat([p,q],axis=1,sort=False)))
print("\nExample 3:\n{}".format(pd.concat([p,q,r],axis=0,sort=False)))
```

```
Example 1:
   Temperature_K  Yield
0            300   56.2
1            310   64.3
2            320   60.9
0            300   56.2
1            310   64.3
2            320   60.9
```

```
Example 2:
   Temperature_K  Yield  Yield  Temperature_K
0            300   56.2   56.2            300
1            310   64.3   64.3            310
2            320   60.9   60.9            320
```

```
Example 3:
   Temperature_K  Yield  Conc
0            300   56.2   NaN
1            310   64.3   NaN
2            320   60.9   NaN
0            300   56.2   NaN
1            310   64.3   NaN
2            320   60.9   NaN
0            300    NaN   0.1
1            310    NaN   0.1
2            320    NaN   0.1
```

Dataframe – Combining Dataframes

In a case like this where there are common headers with partially overlapping labels it is more common that you would want to *merge* the dataframes on their common lables. The `pd.merge(df1,df2)` function supplies this functionality:

```
p = pd.DataFrame({"Temperature_K":[300,310,320],"Yield":[56.2,64.3,60.9]})
q = pd.DataFrame({"Yield":[56.2,64.3,100],"Temperature_K":[300,310,320]})
r = pd.DataFrame({"Temperature_K":[300,310,320],"Conc":[0.1,0.1,0.1]})
merged = pd.merge(p,q)
merged = pd.merge(merged,r)
print("Example 1 (merged):\n{}".format(merged))
```

```
Example 1 (merged):
   Temperature_K Yield Conc
0            300  56.2   0.1
1            310  64.3   0.1
```

Here the result combines on the common labels in both the rows and columns. During the first merge, a value difference is observed in `"Yield"` and this row is dropped.

This behavior can be modified with `left_on` and `right_on` arguments. Merge has many additional arguments and capabilities, but if you are consistent with your labels, you can get a lot of milage out of the basic usage.

Dataframe – Methods

Dataframes share many of the same methods as Series, but since they are two dimensional they typical offer control over whether you apply methods to row or columns of a dataframe.

A common method argument that you will see is `axis` which by default is `0`. When present, this argument controls whether the method acts across rows `axis=0` or columns `axis=1`. For example:

```
df = pd.DataFrame({"c1":[1,2],"c2":[3,4],"c3":[5,6]},index=["r1","r2"])
print("df:\n{}".format(df))
print("\ndf.mean(axis=0):\n{}".format(df.mean(axis=0)))
print("\ndf.mean(axis=1):\n{}".format(df.mean(axis=1)))
```

Dataframe

```
df:
   c1  c2  c3
r1   1   3   5
r2   2   4   6
```

Across Rows

```
df.mean(axis=0):
c1  1.5
c2  3.5
c3  5.5
dtype: float64
```

Across Cols

```
df.mean(axis=1):
r1  3.0
r2  4.0
dtype: float64
```

The `axis` argument behaves similarly for most statistical methods that we have already seen for series.

Dataframe – Filtering Methods

Finding and analyzing subsets of dataframes is an extremely common activity generally called **filtering**. This is an area where Pandas shines.

At the center of this functionality is the `groupby` method, the `df[column].str.` methods and conditional slicing, which we covered above but will revisit here:

```
df = pd.read_csv("plant_data.txt",index_col=0)

print("data:\n{}".format(df))

# Example 1: group by column
groups = df.groupby("plant")
print("\nplant means:\n{}".\
      format(groups.mean()))

# Example 2: iterating over groups
for name,group in groups:
  print("\nplant: {}\n{}".\
        format(name,group))

# Example 3: grab an individual group
print("\nhouston data:\n{}".\
      format(groups.get_group("houston")))
```

The dataframe we are working with has data for specific plants over several years.

```
data:
   year        plant  yield(tons)  costs(100k)
0  2015    lafayette    62.444480    71.834629
1  2016    lafayette    77.258531    72.127020
2  2017    lafayette    78.540064    68.702508
3  2018    lafayette    80.372239    70.611962
4  2019    lafayette    69.956103    70.030832
5  2020    lafayette    67.745829    65.137684
0  2015      houston    81.391068    73.691274
1  2016      houston    76.853975    69.361734
2  2017      houston    66.592911    73.021476
3  2018      houston    76.928565    66.437668
4  2019      houston    62.090535    72.042610
5  2020      houston    70.050213    72.045813
0  2015  baton_rouge    58.135027    68.296684
1  2016  baton_rouge    77.746295    70.029668
2  2017  baton_rouge    71.313903    66.118943
3  2018  baton_rouge    74.759363    71.071937
4  2019  baton_rouge    56.464573    70.659446
5  2020  baton_rouge    68.071532    65.067641
```

Dataframe – Filtering Methods

Finding and analyzing subsets of dataframes is an extremely common activity generally called **filtering**. This is an area where Pandas shines.

At the center of this functionality is the `groupby` method, the `df[column].str.` methods and conditional slicing, which we covered above but will revisit here:

```
df = pd.read_csv("plant_data.txt",index_col=0)

print("data:\n{}".format(df))

# Example 1: group by column
groups = df.groupby("plant")
print("\nplant means:\n{}".\
      format(groups.mean()))

# Example 2: iterating over groups
for name,group in groups:
  print("\nplant: {}\n{}".\
        format(name,group))

# Example 3: grab an individual group
print("\nhouston data:\n{}".\
      format(groups.get_group("houston")))
```

```
plant means:
               year  yield(tons)  costs(100k)
plant
baton_rouge  2017.5    67.748449    68.540720
houston      2017.5    72.317878    71.100096
lafayette    2017.5    72.719541    69.740773
```

The dataframe we are working with has data for specific plants over several years.

Dataframe – Filtering Methods

Finding and analyzing subsets of dataframes is an extremely common activity generally called **filtering**. This is an area where Pandas shines.

At the center of this functionality is the `groupby` method, the `df[column].str.` methods and conditional slicing, which we covered above but will revisit here:

```
df = pd.read_csv("plant_data.txt",index_col=0)

print("data:\n{}".format(df))

# Example 1: group by column
groups = df.groupby("plant")
print("\nplant means:\n{}".\
      format(groups.mean()))

# Example 2: iterating over groups
for name,group in groups:
  print("\nplant: {}\n{}".\
        format(name,group))

# Example 3: grab an individual group
print("\nhouston data:\n{}".\
      format(groups.get_group("houston")))
```

The dataframe we are working with has data for specific plants over several years.

```
plant: baton_rouge
   year        plant  yield(tons)  costs(100k)
0  2015  baton_rouge    58.135027    68.296684
1  2016  baton_rouge    77.746295    70.029668
2  2017  baton_rouge    71.313903    66.118943
3  2018  baton_rouge    74.759363    71.071937
4  2019  baton_rouge    56.464573    70.659446
5  2020  baton_rouge    68.071532    65.067641

plant: houston
   year     plant  yield(tons)  costs(100k)
0  2015   houston    81.391068    73.691274
1  2016   houston    76.853975    69.361734
2  2017   houston    66.592911    73.021476
3  2018   houston    76.928565    66.437668
4  2019   houston    62.090535    72.042610
5  2020   houston    70.050213    72.045813

plant: lafayette
.........
```

Dataframe – Filtering Methods

Finding and analyzing subsets of dataframes is an extremely common activity generally called **filtering**. This is an area where Pandas shines.

At the center of this functionality is the `groupby` method, the `df[column].str.` methods and conditional slicing, which we covered above but will revisit here:

```
df = pd.read_csv("plant_data.txt",index_col=0)

print("data:\n{}".format(df))

# Example 1: group by column
groups = df.groupby("plant")
print("\nplant means:\n{}".\
      format(groups.mean()))

# Example 2: iterating over groups
for name,group in groups:
  print("\nplant: {}\n{}".\
        format(name,group))

# Example 3: grab an individual group
print("\nhouston data:\n{}".\
      format(groups.get_group("houston")))
```

The dataframe we are working with has data for specific plants over several years.

```
houston data:
   year    plant  yield(tons)  costs(100k)
0  2015  houston    81.391068    73.691274
1  2016  houston    76.853975    69.361734
2  2017  houston    66.592911    73.021476
3  2018  houston    76.928565    66.437668
4  2019  houston    62.090535    72.042610
5  2020  houston    70.050213    72.045813
```

Dataframe – Filtering Methods

For columns with string-values the `df[column].str` methods `str.startswith()`, `str.endswith()`, and `str.contains()` allow you to filter the dataframe by these conditions:

```
df = pd.read_csv("plant_data.txt",index_col=0)
print("\ndf[df['plant'].str.startswith('l')]:\n{}".format(df[df['plant'].str.startswith('l')]))
print("\ndf[df['plant'].str.endswith('e')]:\n{}".format(df[df['plant'].str.endswith('e')]))
print("\ndf[df['plant'].str.contains('ton')]:\n{}".format(df[df['plant'].str.contains('ton')]))
```

`str.startswith()` example:

```
df[df['plant'].str.startswith('l')]:
   year      plant  yield(tons)  costs(100k)
0  2015  lafayette    62.444480    71.834629
1  2016  lafayette    77.258531    72.127020
2  2017  lafayette    78.540064    68.702508
3  2018  lafayette    80.372239    70.611962
4  2019  lafayette    69.956103    70.030832
5  2020  lafayette    67.745829    65.137684
```

`str.endswith()` example:

```
df[df['plant'].str.endswith('e')]:
   year        plant  yield(tons)  costs(100k)
0  2015    lafayette    62.444480    71.834629
1  2016    lafayette    77.258531    72.127020
2  2017    lafayette    78.540064    68.702508
3  2018    lafayette    80.372239    70.611962
4  2019    lafayette    69.956103    70.030832
5  2020    lafayette    67.745829    65.137684
0  2015  baton_rouge    58.135027    68.296684
1  2016  baton_rouge    77.746295    70.029668
2  2017  baton_rouge    71.313903    66.118943
3  2018  baton_rouge    74.759363    71.071937
4  2019  baton_rouge    56.464573    70.659446
5  2020  baton_rouge    68.071532    65.067641
```

Dataframe – Filtering Methods

For columns with string-values the `df[column].str` methods `str.startswith()`, `str.endswith()`, and `str.contains()` allow you to filter the dataframe by these conditions:

```
df = pd.read_csv("plant_data.txt",index_col=0)
print("\ndf[df['plant'].str.startswith('l')]:\n{}".format(df[df['plant'].str.startswith('l')]))
print("\ndf[df['plant'].str.endswith('e')]:\n{}".format(df[df['plant'].str.endswith('e')]))
print("\ndf[df['plant'].str.contains('ton')]:\n{}".format(df[df['plant'].str.contains('ton')]))
```

`str.contains()` example:

```
df[df['plant'].str.contains('ton')]:
   year        plant  yield(tons)  costs(100k)
0  2015      houston    81.391068    73.691274
1  2016      houston    76.853975    69.361734
2  2017      houston    66.592911    73.021476
3  2018      houston    76.928565    66.437668
4  2019      houston    62.090535    72.042610
5  2020      houston    70.050213    72.045813
0  2015  baton_rouge    58.135027    68.296684
1  2016  baton_rouge    77.746295    70.029668
2  2017  baton_rouge    71.313903    66.118943
3  2018  baton_rouge    74.759363    71.071937
4  2019  baton_rouge    56.464573    70.659446
5  2020  baton_rouge    68.071532    65.067641
```

Dataframe – Filtering Methods

Conditional slicing is useful for finding values/rows that meet a specified condition:

```
df = pd.read_csv("plant_data.txt",index_col=0)

# Example 1: find the years that the houston plant kept
#            costs below 7m
print("Example 1:\n")
print(df[(df["plant"] == "houston") & (df["costs(100k)"] <70)])

# Example 2: find the years that the lafayette plant kept
#            costs below 7m and production above 60 tons
print("\nExample 2:\n")
print(df[(df["plant"] == "lafayette") & (df["costs(100k)"] < 70) & (df["yield(tons)"] > 60)])
```

```
Example 1:
   year   plant yield(tons)  costs(100k)
1  2016 houston   76.853975    69.361734
3  2018 houston   76.928565    66.437668

Example 2:
   year     plant yield(tons)  costs(100k)
2  2017 lafayette   78.540064    68.702508
5  2020 lafayette   67.745829    65.137684
```

In these cases we have utilized the `&` operator to join the results of multiple logical operations. (Note: We use `&` instead of `and` for element-by-element behavior on arrays and dataframes; the pipe symbol, `|`, is used for `or`.)

Dataframe – Sorting Methods

After merging and combining dataframes you might want to sort the dataframes based on the values of a column or columns.

The relevant method is `dataframe.sort_values(column_label)`, where column_label is a single column label:

```
df = pd.read_csv("plant_data.txt",index_col=0)

# Example 1: Sort based on year
print("Example 1 (sort based on year):\n")
print(df.sort_values('year'))
```

```
Example 1 (sort based on year):

   year        plant  yield(tons)  costs(100k)
0  2015    lafayette    62.444480    71.834629
0  2015  baton_rouge    58.135027    68.296684
0  2015      houston    81.391068    73.691274
1  2016    lafayette    77.258531    72.127020
1  2016  baton_rouge    77.746295    70.029668
1  2016      houston    76.853975    69.361734
2  2017  baton_rouge    71.313903    66.118943
2  2017      houston    66.592911    73.021476
2  2017    lafayette    78.540064    68.702508
3  2018      houston    76.928565    66.437668
3  2018    lafayette    80.372239    70.611962
3  2018  baton_rouge    74.759363    71.071937
...
```

Dataframe – Sorting Methods

After merging and combining dataframes you might want to sort the dataframes based on the values of a column or columns.

The relevant method is `dataframe.sort_values(column_label)`, where column_label is a single column label:

```
# Example 2: Sort based on year then yield
print("\nExample 2 (sort based on year and yield):\n")
print(df.sort_values(['year','yield(tons)'],ascending=[True,False]))
```

```
Example 2 (sort based on year and yield):

   year        plant  yield(tons)  costs(100k)
0  2015      houston    81.391068    73.691274
0  2015    lafayette    62.444480    71.834629
0  2015  baton_rouge    58.135027    68.296684
1  2016  baton_rouge    77.746295    70.029668
1  2016    lafayette    77.258531    72.127020
1  2016      houston    76.853975    69.361734
2  2017    lafayette    78.540064    68.702508
2  2017  baton_rouge    71.313903    66.118943
2  2017      houston    66.592911    73.021476
3  2018    lafayette    80.372239    70.611962
3  2018      houston    76.928565    66.437668
3  2018  baton_rouge    74.759363    71.071937
...
```

Lists can be supplied to break ties and sort on multiple properties.

Dataframe – Missing Values

Often times you will work with datasets that are incomplete, have corrupted values, or have values removed because you've identified them as outliers.

Dataframes come with several built-in methods that are relevant for identifying these values (`.isna()`, `.notna()`) and dropping rows involving them (`.dropna(how='any')`).

We'll use the following dataframe for demonstrate using these methods:

```
 year       plant  yield(tons)  costs(100k)
0  2015.0    lafayette    62.444480    71.834629
1  2016.0    lafayette    77.258531    72.127020
2  2017.0    lafayette    78.540064          NaN
3  2018.0          NaN    80.372239    70.611962
4  2019.0    lafayette    69.956103    70.030832
5  2020.0    lafayette    67.745829    65.137684
0     NaN          NaN          NaN          NaN
1     NaN          NaN    76.853975    69.361734
2  2017.0      houston    66.592911    73.021476
3  2018.0          NaN          NaN    66.437668
4  2019.0      houston    62.090535          NaN
5  2020.0      houston    70.050213    72.045813
0     NaN  baton_rouge    58.135027          NaN
1  2016.0  baton_rouge    77.746295    70.029668
2  2017.0  baton_rouge    71.313903    66.118943
3  2018.0  baton_rouge    74.759363    71.071937
4  2019.0  baton_rouge    56.464573    70.659446
5  2020.0  baton_rouge    68.071532          NaN
```

Dataframe – Missing Values

`dataframe.isna()` example:

```
# Example 1: Find missing rows in a column
print("\nisna() result:\n{}".format(df["year"].isna()))
print("\ndf[df['year'].isna()]):\n{}".format(df[df["year"].isna()]))
```

```
isna() result:
0     False
1     False
2     False
3     False
4     False
5     False
0      True
1      True
2     False
3     False
4     False
5     False
0      True
1     False
2     False
3     False
4     False
5     False
Name: year, dtype: bool
```

```
df[df['year'].isna()]):
   year        plant  yield(tons)  costs(100k)
0   NaN          NaN          NaN          NaN
1   NaN          NaN    76.853975    69.361734
0   NaN  baton_rouge    58.135027          NaN
```

The method returns a series of booleans that can be used to slice the dataframe.

Dataframe – Missing Values

`dataframe.notna()` example:

```
# Example 2: Find valid rows in a column
print("\ndf[df['year'].notna()]):\n{}".format(df[df["year"].notna()]))
```

```
df[df['year'].notna()]):
     year        plant  yield(tons)  costs(100k)
0  2015.0    lafayette    62.444480    71.834629
1  2016.0    lafayette    77.258531    72.127020
2  2017.0    lafayette    78.540064          NaN
3  2018.0          NaN    80.372239    70.611962
4  2019.0    lafayette    69.956103    70.030832
5  2020.0    lafayette    67.745829    65.137684
2  2017.0      houston    66.592911    73.021476
3  2018.0          NaN          NaN    66.437668
4  2019.0      houston    62.090535          NaN
5  2020.0      houston    70.050213    72.045813
1  2016.0  baton_rouge    77.746295    70.029668
2  2017.0  baton_rouge    71.313903    66.118943
3  2018.0  baton_rouge    74.759363    71.071937
4  2019.0  baton_rouge    56.464573    70.659446
5  2020.0  baton_rouge    68.071532          NaN
```

`.notna()` acts as the complement to `isna()`. Only those rows with date information are returned.

Dataframe – Missing Values

`dataframe.dropna()` examples:

```
# Example 3: drop rows with at least one NaN
print("\ndf.dropna():\n{}".format(df.dropna()))
```

```
df.dropna():
     year        plant  yield(tons)  costs(100k)
0  2015.0    lafayette    62.444480    71.834629
1  2016.0    lafayette    77.258531    72.127020
4  2019.0    lafayette    69.956103    70.030832
5  2020.0    lafayette    67.745829    65.137684
2  2017.0      houston    66.592911    73.021476
5  2020.0      houston    70.050213    72.045813
1  2016.0  baton_rouge    77.746295    70.029668
2  2017.0  baton_rouge    71.313903    66.118943
3  2018.0  baton_rouge    74.759363    71.071937
4  2019.0  baton_rouge    56.464573    70.659446
```

Default behavior removes rows that contain any NaN values.

Dataframe – Missing Values

`dataframe.dropna()` examples:

```
# Example 4: Drop rows where all values are missing
print("\ndf.dropna(how='all'):\n{}".format(df.dropna(how='all')))
```

```
df.dropna(how='all'):
     year         plant  yield(tons)  costs(100k)
0  2015.0     lafayette    62.444480    71.834629
1  2016.0     lafayette    77.258531    72.127020
2  2017.0     lafayette    78.540064          NaN
3  2018.0           NaN    80.372239    70.611962
4  2019.0     lafayette    69.956103    70.030832
5  2020.0     lafayette    67.745829    65.137684
1     NaN           NaN    76.853975    69.361734
2  2017.0       houston    66.592911    73.021476
3  2018.0           NaN          NaN    66.437668
4  2019.0       houston    62.090535          NaN
5  2020.0       houston    70.050213    72.045813
0     NaN   baton_rouge    58.135027          NaN
1  2016.0   baton_rouge    77.746295    70.029668
2  2017.0   baton_rouge    71.313903    66.118943
3  2018.0   baton_rouge    74.759363    71.071937
4  2019.0   baton_rouge    56.464573    70.659446
5  2020.0   baton_rouge    68.071532          NaN
```

The `how` argument can be used to modify whether `any` NaN value or `all` NaN values will trigger removal. Here only one row has NaN for all values and has been removed.

Dataframe – Missing Values

In the case that we want to replace missing values the relevant methods are `.fillna()`, `.replace()`, and `.interpolate()`:

```
# Fill missing values
print("\ndf.fillna(0.0):{}\n".format(df.fillna(0.0)))

# Replace missing values
print("\ndf.replace(np.NaN,value=0.0):\n{}".format(df.replace(np.NaN,value=0
.0)))

# Interpolate missing values (not sensible in this example)
print("\ndf.interpolate():\n{}".format(df.interpolate()))
```

```
df.fillna(0.0):
     year       plant  yield(tons)  costs(100k)
0  2015.0   lafayette    62.444480    71.834629
1  2016.0   lafayette    77.258531    72.127020
2  2017.0   lafayette    78.540064     0.000000
3  2018.0           0    80.372239    70.611962
4  2019.0   lafayette    69.956103    70.030832
5  2020.0   lafayette    67.745829    65.137684
0     0.0           0     0.000000     0.000000
1     0.0           0    76.853975    69.361734
...
```

Dataframe – Missing Values

In the case that we want to replace missing values the relevant methods are `.fillna()`, `.replace()`, and `.interpolate()`:

```
# Fill missing values
print("\ndf.fillna(0.0):{}\n".format(df.fillna(0.0)))

# Replace missing values
print("\ndf.replace(np.NaN,value=0.0):\n{}".format(df.replace(np.NaN,value=0
.0)))

# Interpolate missing values (not sensible in this example)
print("\ndf.interpolate():\n{}".format(df.interpolate()))
```

```
df.replace(np.NaN,value=0.0):
     year        plant  yield(tons)  costs(100k)
0  2015.0    lafayette    62.444480    71.834629
1  2016.0    lafayette    77.258531    72.127020
2  2017.0    lafayette    78.540064     0.000000
3  2018.0            0    80.372239    70.611962
4  2019.0    lafayette    69.956103    70.030832
5  2020.0    lafayette    67.745829    65.137684
0     0.0            0     0.000000     0.000000
1     0.0            0    76.853975    69.361734
...
```

`.replace()` can be used on any value, including NaN.

Dataframe – Missing Values

In the case that we want to replace missing values the relevant methods are `.fillna()`, `.replace()`, and `.interpolate()`:

```
# Fill missing values
print("\ndf.fillna(0.0):{}\n".format(df.fillna(0.0)))

# Replace missing values
print("\ndf.replace(np.NaN,value=0.0):\n{}".format(df.replace(np.NaN,value=0
.0)))

# Interpolate missing values (not sensible in this example)
print("\ndf.interpolate():\n{}".format(df.interpolate()))
```

```
df.interpolate():
     year       plant  yield(tons)  costs(100k)
0  2015.0   lafayette    62.444480    71.834629
1  2016.0   lafayette    77.258531    72.127020
2  2017.0   lafayette    78.540064    71.369491
3  2018.0         NaN    80.372239    70.611962
4  2019.0   lafayette    69.956103    70.030832
5  2020.0   lafayette    67.745829    65.137684
0  2019.0         NaN    72.299902    67.249709
1  2018.0         NaN    76.853975    69.361734
```

`.interpolate()` can be used on numeric values with multiple arguments.

Note: <u>this is only sensible for ordered data, this example has no basis for interpolation</u>

Dataframe – Missing Values

When calculating statistical quantities, pandas automatically disregards NaN values:

```
# Describe on dataframe with NaN values
print(df['yield(tons)'].describe())

# Describe on dataframe with dropped NaN values
print(df['yield(tons)'].dropna().describe())
```

Without dropping NaN

```
count    14.000000
mean     71.003571
std       7.107142
min      58.135027
25%      66.881141
50%      70.635003
75%      77.176040
max      80.372239
Name: yield(tons), dtype: float64
```

Dropping NaN

```
count    14.000000
mean     71.003571
std       7.107142
min      58.135027
25%      66.881141
50%      70.635003
75%      77.176040
max      80.372239
Name: yield(tons), dtype: float64
```

Example

As a working example, I have provided a notebook demonstrating some data ingestion, visualization, and curation steps using Pandas, Numpy/Scipy, and Matplotlib.

The data I have provided are some IR spectra for a few different compounds (`ethylene.csv`, `furan.csv`, `diethyl_ether.csv`, and `1-butanol.csv`):

ethylene.csv example:

```
cm^-1,Intensity
454.005,1.014
458.698378,1.014
463.391756,1.018
468.085134,1.021
472.778512,1.029
477.47189,1.034
482.165268,1.042
486.858646,1.05
491.552024,1.055
496.245402,1.061
500.93878,1.058
505.632158,1.059
510.325536,1.058
515.018914,1.059
519.712292,1.062
524.40567,1.062
529.099048,1.059
...
```

Example – Ingesting Data

As a working example, I have provided a notebook demonstrating some data ingestion, visualization, and curation steps using Pandas, Numpy/Scipy, and Matplotlib.

The data I have provided are some IR spectra for a few different compounds (`ethylene.csv`, `furan.csv`, `diethyl_ether.csv`, and `1-butanol.csv`):

.csv files can be parsed using the pandas function `pd.read_csv()`:

```
# Read in data files to pandas dataframe
data = {}
for i in ["1-butanol.csv","diethyl_ether.csv","ethylene.csv","furan.csv"]:
    data[i] = pd.read_csv(i,header=0)
for i in data.keys():
    print("{}:\n{}\n".format(i,data[i]))
```

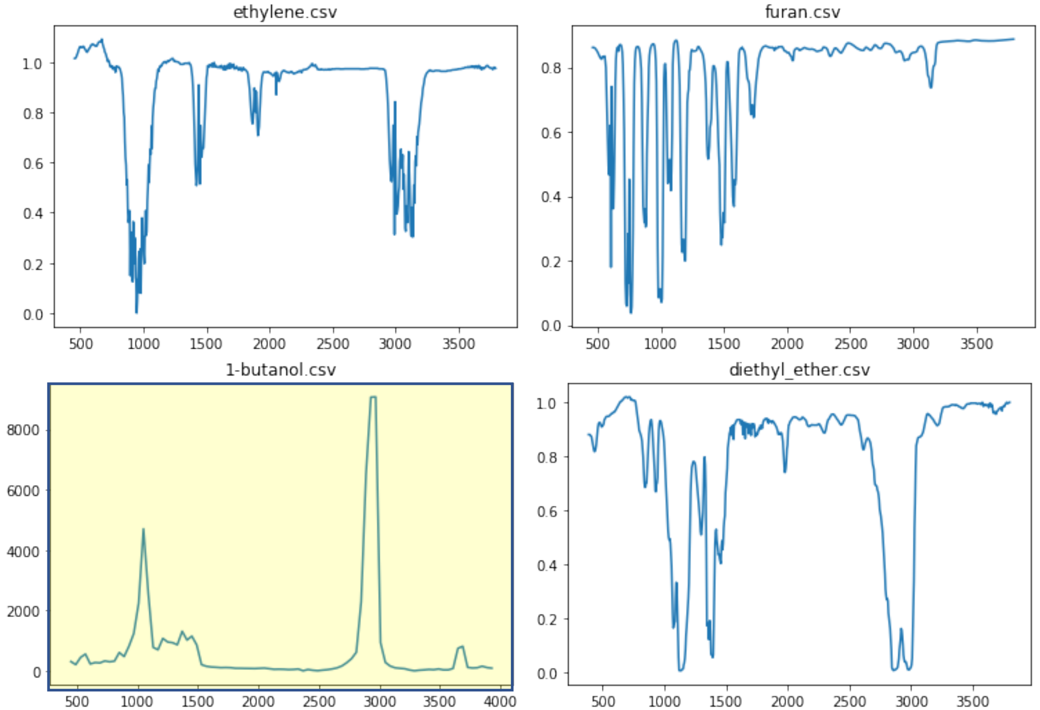
```
1-butanol.csv:
     cm^-1  Intensity
0    450.0      305.0
1    490.0      204.0
2    530.0      443.0
3    570.0      554.0
4    610.0      227.0
..     ...        ...
```

Example – Visualizing Data

The first rule of data analysis is to look at your data as you work with it. This will help you avoid common problems (e.g., unit inconsistencies, missing data, normalization issues, obvious outliers, etc.):

```
import matplotlib.pyplot as plt
for i in data.keys(): # Loop over the distinct keys in the dictionary
    plt.figure() # Initialize matplotlib figure
    plt.plot(data[i]["cm^-1"],data[i]["Intensity"]) # Make the current plot
    plt.title(i) # Add title based on the key
plt.show()
```

As a domain expert, you can tell that one of these is not like the others. Specifically, the data for 1-butanol is supplied as an absorbance rather than a transmittance.

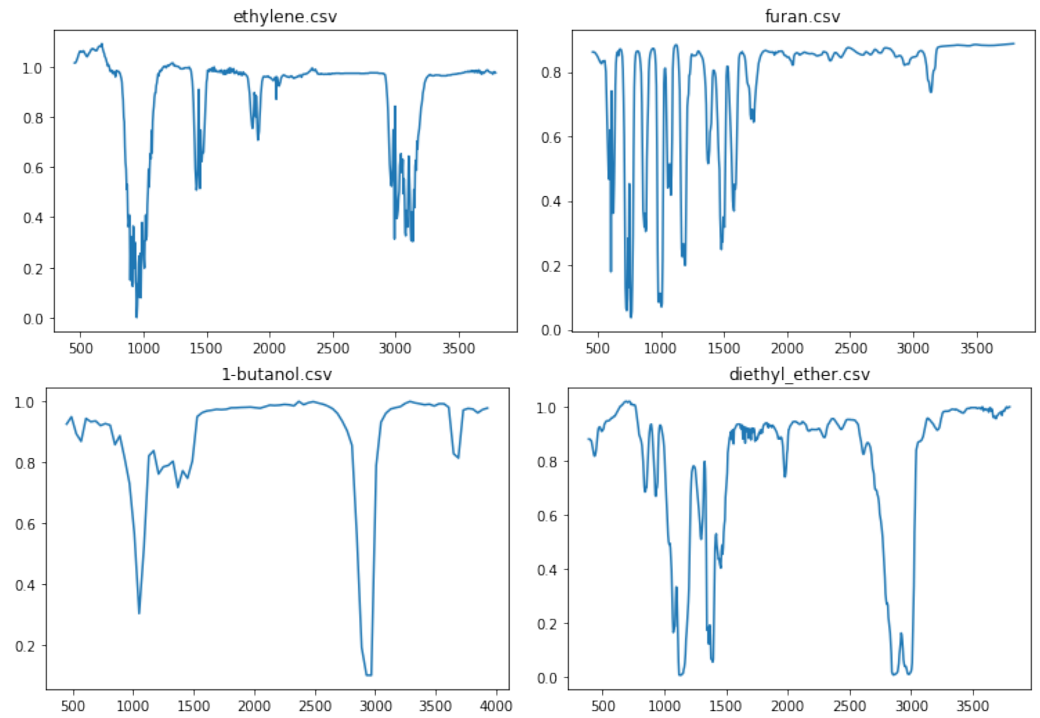


Example – Visualizing Data

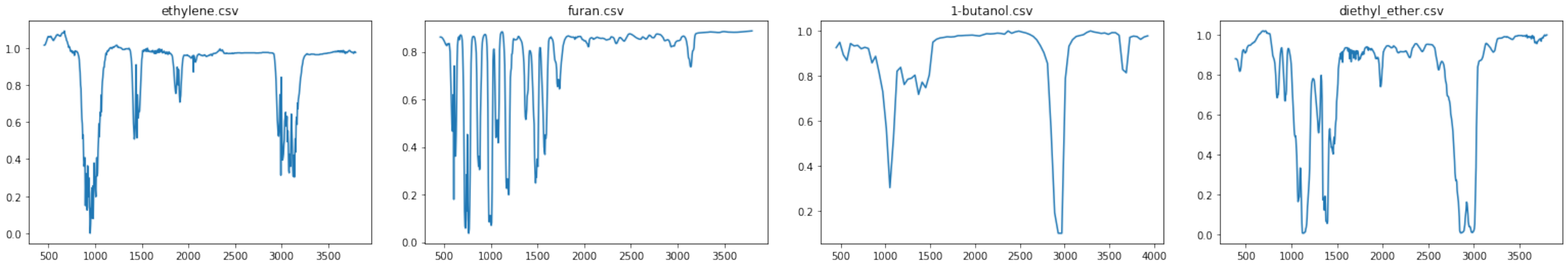
Let's convert the data for 1-butanol to transmittance and replot the results to check the conversion:

```
data['1-butanol.csv']["Intensity"] = \
10.0**(-(data['1-butanol.csv']["Intensity"]/data['1-butanol.csv']["Intensity"].max()))
for i in data.keys():
    plt.figure()
    plt.plot(data[i]["cm^-1"],data[i]["Intensity"])
    plt.title(i)
plt.show()
```

The conversion looks good, but there are still some inconsistencies across the spectra that we might want to deal before passing the data to an analysis or machine learning workflow.



Example – Curating Data



First, the spectra are reported in distinct wavenumber increments:

```
# Check the number of values in each dataframe
for i in data.keys():
    print("{}: {}".format(i,len(data[i]["cm^-1"])))
```

```
1-butanol.csv: 88
diethyl_ether.csv: 727
ethylene.csv: 713
furan.csv: 668
```

Second, the spectra are reported over distinct wavenumber ranges:

```
# Check the wavenumber range in each dataframe
for i in data.keys():
    print("{}: {}-{} cm^-1".format(i,data[i]["cm^-1"].min(),data[i]["cm^-1"].max()))
```

```
1-butanol.csv: 450.0-3930.0 cm^-1
diethyl_ether.csv: 386.531-3797.658597 cm^-1
ethylene.csv: 454.005-3795.690119 cm^-1
furan.csv: 460.0-3795.0 cm^-1
```

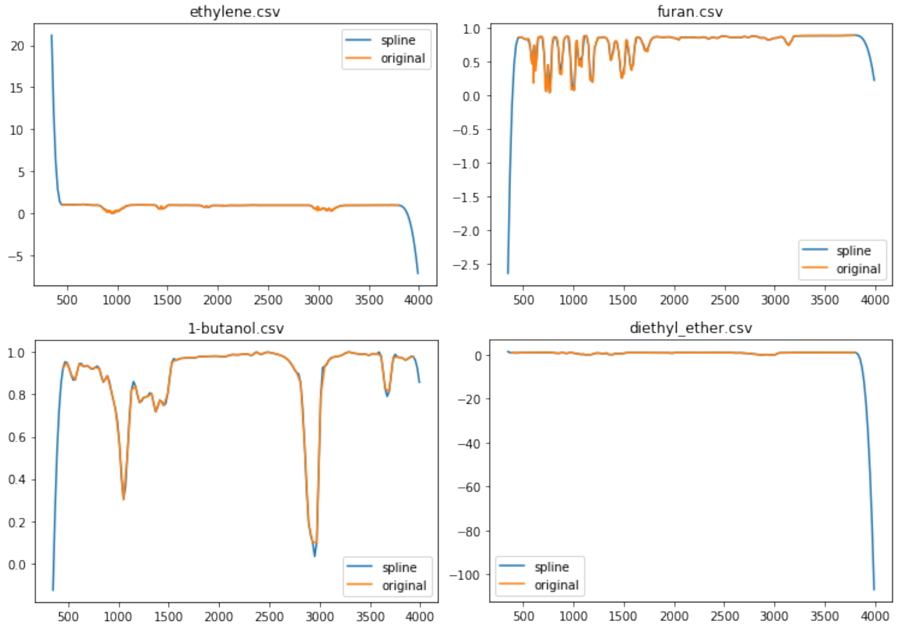
Example – Curating Data

This kind of misalignment is common and it isn't necessarily a problem. However, sometimes we need our data aligned and with a fixed length. Let's process our data so that all of the spectra are fixed length and at regular wavenumber intervals:

```
from scipy.interpolate import CubicSpline
new_x = np.arange(350.,4001.,20)
for i in data.keys():
    spline = CubicSpline(data[i]["cm^-1"],data[i]["Intensity"])
    plt.figure()
    plt.plot(new_x,spline(new_x),label="spline")
    plt.plot(data[i]["cm^-1"],data[i]["Intensity"],label="original")
    plt.title(i)
    plt.legend()
plt.show()
```

Where there is overlap, the spline seems to be working.

However, there are clearly problems in the regions where extrapolation occurs.



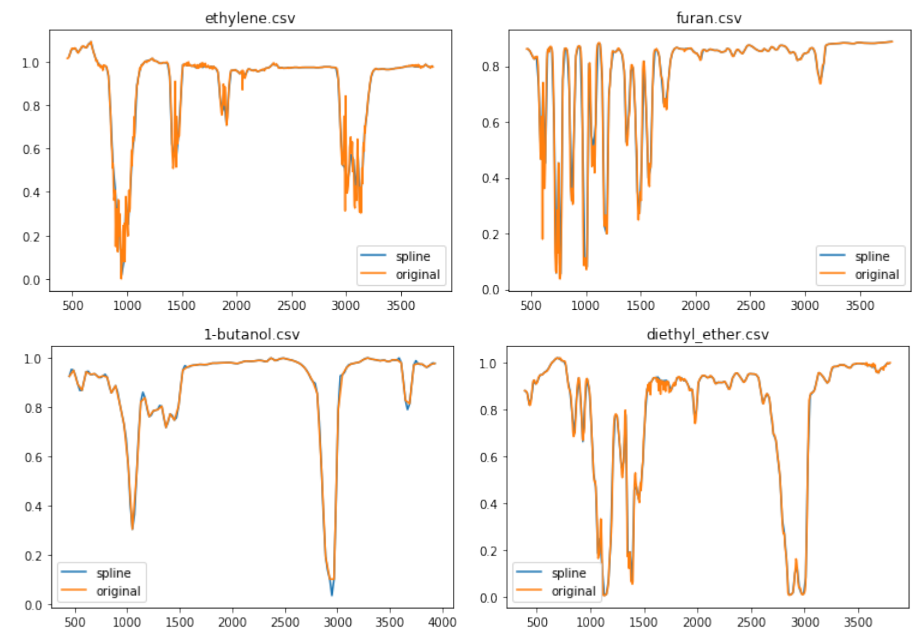
Example – Curating Data

A simple patch is to only use the spline for interpolation, and where the wavenumber range extends beyond the data to use a null value:

```
new_x = np.arange(350.,4001.,20)
for i in data.keys():
    spline = CubicSpline(data[i]["cm^-1"],data[i]["Intensity"])
    new_y = spline(new_x) # Get the new y-values.
    new_y[np.where((new_x<data[i]["cm^-1"].min()) | (new_x>data[i]["cm^-1"].max()))] = np.nan
    plt.figure()
    plt.plot(new_x,new_y,label="spline")
    plt.plot(data[i]["cm^-1"],data[i]["Intensity"],label="original")
    plt.title(i)
    plt.legend()
plt.show()
```

This is looking much more reasonable.

We can see some loss in fidelity near some of the peaks, but whether this is a problem depends on the application and is beyond the scope of the current example.



Example – Curating Data

Now that our preprocessing loop is working, let's combine the interpolated data into a shared dataframe:

```
new_x = np.arange(350.,4001.,20)
d_new = {}
for i in data.keys():
    spline = CubicSpline(data[i]["cm^-1"],data[i]["Intensity"])
    new_y = spline(new_x)
    new_y[np.where((new_x<data[i]["cm^-1"].min()) | (new_x>data[i]["cm^-1"].max()))] = np.nan
    d_new[i.split('.')[0]] = new_y
df = pd.DataFrame(d_new,index=new_x)
df
```

	1-butanol	diethyl_ether	ethylene	furan
350.0	NaN	NaN	NaN	NaN
370.0	NaN	NaN	NaN	NaN
390.0	NaN	0.878786	NaN	NaN
410.0	NaN	0.870040	NaN	NaN
430.0	NaN	0.819916	NaN	NaN
...	...	...	...	...
3910.0	0.979775	NaN	NaN	NaN
3930.0	0.978160	NaN	NaN	NaN
3950.0	NaN	NaN	NaN	NaN
3970.0	NaN	NaN	NaN	NaN
3990.0	NaN	NaN	NaN	NaN

A dataframe like this is often the product of a pre-processing workflow. For example, you might add a few more lines to automatically ingest data from many files, apply some cleaning operations, and then combine them into a dataframe.

Example – Curating Data

For completeness, let's plot the results and confirm that everything is working together. To do this, we can use the dataframe's plot method to automatically make a lineplot based on the columns:

```
new_x = np.arange(350.,4001.,20)
d_new = {}
for i in data.keys():
    spline = CubicSpline(data[i]["cm^-1"],data[i]["Intensity"])
    new_y = spline(new_x)
    new_y[np.where((new_x<data[i]["cm^-1"].min()) | (new_x>data[i]["cm^-1"].max()))] = np.nan
    d_new[i.split('.')[0]] = new_y
df = pd.DataFrame(d_new,index=new_x)
df.plot()
```