Problem Set 3 (7.5 points)

Sheikh-Sedat Touray

DSP 562

Spring 2024

Data Manipulation in Python (continued)

Week 3 References: McKinney, Chapters 8, 10, (12)

Data Wrangling: Join, Combine, and Reshape - Chapter 8

In [106]:
```python
import pandas as pd
from pandas import Series, DataFrame
import numpy as np
```

Database-Style DataFrame Joins

1 Merge or join the following 2 datasets

In [107]:
```python
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
                    'data1': range(7)})
df1
```

Out[107]:

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

In [108]:
```python
df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
                    'data2': range(3)})
df2
```

Out[108]:

	key	data2
0	a	0
1	b	1
2	d	2

In [109]:
```python
#we are going to do the inner join first
inner = pd.merge(df1, df2)
inner
```

Out[109]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

In [110]:
```python
# Inner join on key
inner2 = pd.merge(df1, df2, on='key')
inner2
```

Out[110]:

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

pd.merge by default performs and innere join that is why the results from the first merge and the one above are the same. Although by default the key is the common intersection, it is also important to specify it because it could be useful if there are more arguments to be used.

In [111]:
```python
# outer join
outer = pd.merge(df1, df2, how = 'outer')
outer
```

Out[111]:

	key	data1	data2
0	b	0.0	1.0
1	b	1.0	1.0
2	b	6.0	1.0
3	a	2.0	0.0
4	a	4.0	0.0
5	a	5.0	0.0
6	c	3.0	NaN
7	d	NaN	2.0

The outer join takes the union of the keys, combining the effect of applying both left and right joins. In an outer join, rows from the left or right DataFrame objects that do not match on keys in the other DataFrame will appear with NA values in the other DataFrame's columns for the nonmatching rows as we can see in the above output.

In [112]:
```python
#performing the left join
left = pd.merge(df1, df2, on='key', how = 'left')
left
```

Out[112]:

	key	data1	data2
0	b	0	1.0
1	b	1	1.0
2	a	2	0.0
3	c	3	NaN
4	a	4	0.0
5	a	5	0.0
6	b	6	1.0

The left join uses all the key combinations found in the left table.

In [113]:
```python
#Performing the right join
right = pd.merge(df1, df2, on='key', how = 'right')
right
```

Out[113]:

	key	data1	data2
0	b	0.0	1
1	b	1.0	1
2	b	6.0	1
3	a	2.0	0
4	a	4.0	0
5	a	5.0	0
6	d	NaN	2

The right join uses all the key combinations found in the right table.

Concatenating Along an Axis

2 Concatenate, bind, or stack df2 to df1, ignoring df2's Index

In [114]:
```
df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
df1
```
Out[114]:

	a	b	c	d
0	1.754588	0.970266	0.703816	2.090140
1	0.651201	0.107759	-0.310928	-0.178697
2	0.485525	0.335399	-1.288644	0.782624

In [115]:
```
df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
df2
```
Out[115]:

	b	d	a
0	-1.575212	0.019180	-0.314276
1	-0.431994	-0.456637	0.687777

In [116]:
```
# concatenate the two data frames and inoring the df2's index
result = pd.concat([df1,df2],ignore_index = True)
result
```
Out[116]:

	a	b	c	d
0	1.754588	0.970266	0.703816	2.090140
1	0.651201	0.107759	-0.310928	-0.178697
2	0.485525	0.335399	-1.288644	0.782624
3	-0.314276	-1.575212	NaN	0.019180
4	0.687777	-0.431994	NaN	-0.456637

Combining Data with Overlap

3 Use the combine_first method to patch missing data in df1 from df2

In [117]:
```
df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
                    'b': [np.nan, 2., np.nan, 6.],
                    'c': range(2, 18, 4)})
df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
                    'b': [np.nan, 3., 4., 6., 8.]})
df1
```
Out[117]:

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

In [118]:
```
df2
```
Out[118]:

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

In [119]:
```
# Using combine_first to patch missing data in df1 from df2
result_df = df1.combine_first(df2)

print("\nResult after combining:")
print(result_df)
```
```
Result after combining:
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

From the the combine first we have observed that it is a union of the two dataframe column names and it patches missing data in df1 with values from df2. However, if there are no missing values in df1 then it retains original value of df1.

Pivoting "Long" to "Wide" Format

4 Pivot the dataset to a DataFrame containing one column per distinct item value indexed by timestamps in the date column:

In [120]:
```
data = pd.read_csv('macrodata.csv')
data.head()
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
                         name='date')
columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')
data = data.reindex(columns=columns)
data.index = periods.to_timestamp('D', 'end')
ldata = data.stack().reset_index().rename(columns={0: 'value'})
```
In [121]:
```
ldata[:10]
```
Out[121]:

	date	item	value
0	1959-03-31 23:59:59.999999999	realgdp	2710.349
1	1959-03-31 23:59:59.999999999	infl	0.000
2	1959-03-31 23:59:59.999999999	unemp	5.800
3	1959-06-30 23:59:59.999999999	realgdp	2778.801
4	1959-06-30 23:59:59.999999999	infl	2.340
5	1959-06-30 23:59:59.999999999	unemp	5.100
6	1959-09-30 23:59:59.999999999	realgdp	2775.488
7	1959-09-30 23:59:59.999999999	infl	2.740
8	1959-09-30 23:59:59.999999999	unemp	5.300
9	1959-12-31 23:59:59.999999999	realgdp	2785.204

In [122]:
```
pivoted = ldata.pivot(index="date", columns="item",values="value")

print("\nPivoted:")
pivoted
```
```
Pivoted:
```
Out[122]:

item	infl	realgdp	unemp
date			
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2
...	...	...	...
2008-09-30 23:59:59.999999999	-3.16	13324.600	6.0
2008-12-31 23:59:59.999999999	-8.79	13141.920	6.9
2009-03-31 23:59:59.999999999	0.94	12925.410	8.1
2009-06-30 23:59:59.999999999	3.37	12901.504	9.2
2009-09-30 23:59:59.999999999	3.56	12990.341	9.6

203 rows × 3 columns

The goal of this pivot is to accomplish a DataFrame containing one column per distinct item value indexed by timestamps in the date column. DataFrame's pivot method allows us to have data in a format that is easy to work with by creating the distinct item value in each column.

In [123]:
```
#Now pivoting with two values for better visualization and understanding
ldata["value2"] = np.random.standard_normal(len(ldata))
ldata[:10]
```

Out[123]:

	date	item	value	value2
0	1959-03-31 23:59:59.999999999	realgdp	2710.349	-0.471564
1	1959-03-31 23:59:59.999999999	infl	0.000	-2.125875
2	1959-03-31 23:59:59.999999999	unemp	5.800	0.795429
3	1959-06-30 23:59:59.999999999	realgdp	2778.801	0.676248
4	1959-06-30 23:59:59.999999999	infl	2.340	0.907748
5	1959-06-30 23:59:59.999999999	unemp	5.100	0.955681
6	1959-09-30 23:59:59.999999999	realgdp	2775.488	1.236429
7	1959-09-30 23:59:59.999999999	infl	2.740	-0.740987
8	1959-09-30 23:59:59.999999999	unemp	5.300	0.093231
9	1959-12-31 23:59:59.999999999	realgdp	2785.204	-0.884016

In [124]:
```
#Pivoting and printing the result
pivoted = ldata.pivot(index="date", columns="item")

print("\nPivoted with multiple Values:")
pivoted
```

```
Pivoted with multiple Values:
```

Out[124]:

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8	-2.125875	-0.471564	0.795429
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1	0.907748	0.676248	0.955681
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3	-0.740987	1.236429	0.093231
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6	-1.008876	-0.884016	1.093631
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2	0.069059	-1.357264	-1.198328
...	...	...	...	...	...	...
2008-09-30 23:59:59.999999999	-3.16	13324.600	6.0	0.170127	-0.292380	-0.237762
2008-12-31 23:59:59.999999999	-8.79	13141.920	6.9	-0.206679	0.868887	-0.214924
2009-03-31 23:59:59.999999999	0.94	12925.410	8.1	0.071009	-0.971166	0.621225
2009-06-30 23:59:59.999999999	3.37	12901.504	9.2	0.270799	0.586981	0.190624
2009-09-30 23:59:59.999999999	3.56	12990.341	9.6	1.687299	1.101753	-0.447138

203 rows × 6 columns

In [125]:
```
#hierarchical index using set_index gives the same result
unstacked = ldata.set_index(["date", "item"]).unstack(level = "item")
print("\nHierarchical using Set Index:")
unstacked.head()
```

```
Hierarchical using Set Index:
```

Out[125]:

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-03-31 23:59:59.999999999	0.00	2710.349	5.8	-2.125875	-0.471564	0.795429
1959-06-30 23:59:59.999999999	2.34	2778.801	5.1	0.907748	0.676248	0.955681
1959-09-30 23:59:59.999999999	2.74	2775.488	5.3	-0.740987	1.236429	0.093231
1959-12-31 23:59:59.999999999	0.27	2785.204	5.6	-1.008876	-0.884016	1.093631
1960-03-31 23:59:59.999999999	2.31	2847.699	5.2	0.069059	-1.357264	-1.198328

Note that pivot is equivalent to creating a hierarchical index using set_index fol- lowed by a call to unstack.

Pivoting "Wide" to "Long" Format

5 The inverse operation of pivot for DataFrames is pandas.melt. It merges multiple columns into one. Melt the DataFrame below using 'key' as the group indicator:

In [126]:
```
df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
                   'A': [1, 2, 3],
                   'B': [4, 5, 6],
                   'C': [7, 8, 9]})
df
```

Out[126]:

	key	A	B	C
0	foo	1	4	7
1	bar	2	5	8
2	baz	3	6	9

In [127]:
```
melted = pd.melt(df, id_vars="key")
print("\nPivoted from wide to long:")
melted
```

```
Pivoted from wide to long:
```

Out[127]:

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6
6	foo	C	7
7	bar	C	8
8	baz	C	9

Using pivot, reshape the data back into the original layout:

In [128]:
```
reshaped = melted.pivot(index="key", columns="variable", values="value")
reshaped
```

Out[128]:

variable	A	B	C
key			
bar	2	5	8
baz	3	6	9
foo	1	4	7

Get the index back by using the reset_index method:

In [129]:
```
reshaped = reshaped.reset_index()
reshaped
```

Out[129]:

variable	key	A	B	C
0	bar	2	5	8
1	baz	3	6	9
2	foo	1	4	7

We can also specify a subset of columns to use as value columns.

In [130]:
```
shapn = pd.melt(df, id_vars="key", value_vars=["A", "B"])
shapn
```

Out[130]:

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6

pandas.melt can be used without group identifiers, too.

In [131]:
```
shapv = pd.melt(df, value_vars=["A", "B", "C"])
shapv
```

Out[131]:

	variable	value
0	A	1
1	A	2
2	A	3
3	B	4
4	B	5
5	B	6
6	C	7
7	C	8
8	C	9

In [133]:
```
print("\nPivoted from wide to long with value Variables and key:")

shapvs = pd.melt(df, value_vars=["key", "A", "B"])

shapvs
```

```
Pivoted from wide to long with value Variables and key:
```

Out[133]:

	variable	value
0	key	foo
1	key	bar
2	key	baz
3	A	1
4	A	2
5	A	3
6	B	4
7	B	5
8	B	6

Data Aggregation and Group Operations - Chapter 10

GroupBy Mechanics

6 Introducing GroupBy

In [73]:
```
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randn(5),
                   'data2' : np.random.randn(5)})
df
```

Out[73]:

	key1	key2	data1	data2
0	a	one	-0.011462	-0.737893
1	a	two	0.752467	-1.687782
2	b	one	-0.457975	-1.620191
3	b	two	0.853417	0.051763
4	a	one	-0.812930	-0.312422

In [74]:
```
grouped = df['data1'].groupby(df['key1'])
```

Get the mean from the GroupBy object by calling the mean method:

In [76]:
```
gbyo = grouped.mean()
print("\nGroup by Object mean:")
gbyo
```

```
Group by Object mean:
```

Out[76]:
```
key1
a   -0.023975
b    0.197721
Name: data1, dtype: float64
```

The data (a Series) has been aggregated by splitting the data on the group key, producing a new Series that is now indexed by the unique values in the key1 column. The result index has the name "key1" because the DataFrame column df["key1"] did.

In [43]:
```
means = df["data1"].groupby([df["key1"], df["key2"]]).mean()
means
```

Out[43]:
```
key1  key2
a     one     0.981031
      two     1.284528
b     one     0.431800
      two     1.283593
Name: data1, dtype: float64
```

We have a different result here above because passed multiple arrays as a list. Depending on preference of viewing the results we can unstack the series to view the hierachical index consisting of the unique pairs of keys.

In [45]:
```
#unstacking the resulting series
means.unstack()
```

Out[45]:

key2	one	two
key1		
a	0.981031	1.284528
b	0.431800	1.283593

7

In [77]:
```
states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
years = np.array([2005, 2005, 2006, 2005, 2006])
```

Get the Means of states by years, accessed in 'data1':

In [80]:
```
smy = df["data1"].groupby([states, years]).mean()

print("\nMeans of States by Years:")
DataFrame(smy)
```

```
Means of States by Years:
```

Out[80]:

		data1
California	**2005**	0.752467
	2006	-0.457975
Ohio	**2005**	0.420978
	2006	-0.812930

Column-Wise and Multiple Function Application

8 Follow this example of using the GroupBy method

In [134]:
```
tips = pd.read_csv('tips.csv')
# Add tip percentage of total bill
tips['tip_pct'] = tips['tip'] / tips['total_bill']
tips[:6]
```

Out[134]:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808
5	25.29	4.71	No	Sun	Dinner	4	0.186240

In [135]:
```
#Group the tips by day and smoker
grouped = tips.groupby(['day', 'smoker'])
```

In [137]:
```
# Group the tip percentage by day and smoker status
grouped_pct = grouped['tip_pct']
```

Get the mean:

```
In [138]: #Getting the mean with agg() function
grouped_mean = grouped.agg('mean')
grouped_mean
```

Out[138]:

day	smoker	total_bill	tip	size	tip_pct
Fri	No	18.420000	2.812500	2.250000	0.151650
	Yes	16.813333	2.714000	2.066667	0.174783
Sat	No	19.661778	3.102889	2.555556	0.158048
	Yes	21.276667	2.875476	2.476190	0.147906
Sun	No	20.506667	3.167895	2.929825	0.160113
	Yes	24.120000	3.516842	2.578947	0.187250
Thur	No	17.113111	2.673778	2.488889	0.160298
	Yes	19.190588	3.030000	2.352941	0.163863

This result produces a dataframe because of the agg function used. The output would be different if the mean() function was used

```
In [94]: grouped_pct_mean = grouped_pct.agg('mean')
grouped_pct_mean
```

```
Out[94]: day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

Get the mean and standard deviation:

```
In [92]: grouped_pct_std = grouped_pct.std()
grouped_pct_std
```

```
Out[92]: day   smoker
Fri   No        0.028123
      Yes       0.051293
Sat   No        0.039767
      Yes       0.061375
Sun   No        0.042347
      Yes       0.154134
Thur  No        0.038774
      Yes       0.039389
Name: tip_pct, dtype: float64
```

```
In [98]: stdpct = grouped_pct.agg(["mean", "std"])
stdpct
```

Out[98]:

day	smoker	mean	std
Fri	No	0.151650	0.028123
	Yes	0.174783	0.051293
Sat	No	0.158048	0.039767
	Yes	0.147906	0.061375
Sun	No	0.160113	0.042347
	Yes	0.187250	0.154134
Thur	No	0.160298	0.038774
	Yes	0.163863	0.039389

I got a dataframe from the output above because I past a list of function names. The difference can be seen from the previous output when I just used the std() function.

Returning Aggregated Data Without Row Indexes

9 Disable the index composed from the unique group key combinations:

```
In [93]: grouped = tips.groupby(['day', 'smoker'], as_index=False)
grouped_pct_mean = grouped['tip_pct'].mean()
print(grouped_pct_mean)

    day smoker   tip_pct
0   Fri     No  0.151650
1   Fri    Yes  0.174783
2   Sat     No  0.158048
3   Sat    Yes  0.147906
4   Sun     No  0.160113
5   Sun    Yes  0.187250
6  Thur     No  0.160298
7  Thur    Yes  0.163863
```

reset_index also works on result but using as_index = false helps avoind unnecessary computations. Also, setting as_index=False ensures that the group labels are not used as an index in the resulting DataFrame. This can be useful if you prefer to have a flat DataFrame structure after performing groupby operations

Apply: General split-apply-combine

10

```
In [99]: def top(df, n=5, column='tip_pct'):
    return df.sort_values(by=column)[-n:]
top(tips, n=6)
```

Out[99]:

	total_bill	tip	smoker	day	time	size	tip_pct
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
232	11.61	3.39	No	Sat	Dinner	2	0.291990
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

```
In [100]: tips.groupby('smoker').apply(top)
```

Out[100]:

smoker		total_bill	tip	smoker	day	time	size	tip_pct
No	88	24.71	5.85	No	Thur	Lunch	2	0.236746
	185	20.69	5.00	No	Sun	Dinner	5	0.241663
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	232	11.61	3.39	No	Sat	Dinner	2	0.291990
Yes	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

Pass the function top:

```
In [101]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
```

Out[101]:

smoker	day		total_bill	tip	smoker	day	time	size	tip_pct
No	Fri	94	22.75	3.25	No	Fri	Dinner	2	0.142857
	Sat	212	48.33	9.00	No	Sat	Dinner	4	0.186220
	Sun	156	48.17	5.00	No	Sun	Dinner	6	0.103799
	Thur	142	41.19	5.00	No	Thur	Lunch	5	0.121389
Yes	Fri	95	40.17	4.73	Yes	Fri	Dinner	4	0.117750
	Sat	170	50.81	10.00	Yes	Sat	Dinner	3	0.196812
	Sun	182	45.35	3.50	Yes	Sun	Dinner	3	0.077178
	Thur	197	43.11	5.00	Yes	Thur	Lunch	4	0.115982

Get descriptive statistics on 'smoker':

In [139]:
```
result = tips.groupby("smoker")["tip_pct"].describe()
print("\nDescriptive Stats of Smoker grouped by tip_pct column:")

result
```

```
Descriptive Stats of Smoker grouped by tip_pct column:
```

Out[139]:

smoker	count	mean	std	min	25%	50%	75%	max
No	151.0	0.159328	0.039910	0.056797	0.136906	0.155625	0.185014	0.291990
Yes	93.0	0.163196	0.085119	0.035638	0.106771	0.153846	0.195059	0.710345

Unstack 'smoker':

In [140]:
```
ustr = result.unstack("smoker")
print("\nUnstacking smoker:")

ustr
```

```
Unstacking smoker:
```

Out[140]:
```
       smoker
count  No        151.000000
       Yes        93.000000
mean   No          0.159328
       Yes         0.163196
std    No          0.039910
       Yes         0.085119
min    No          0.056797
       Yes         0.035638
25%    No          0.136906
       Yes         0.106771
50%    No          0.155625
       Yes         0.153846
75%    No          0.185014
       Yes         0.195059
max    No          0.291990
       Yes         0.710345
dtype: float64
```

In [141]:
```
print("\nUnstacking smoker as a DataFrame:")

DataFrame(ustr)
```

```
Unstacking smoker as a DataFrame:
```

Out[141]:

	smoker	0
count	No	151.000000
	Yes	93.000000
mean	No	0.159328
	Yes	0.163196
std	No	0.039910
	Yes	0.085119
min	No	0.056797
	Yes	0.035638
25%	No	0.136906
	Yes	0.106771
50%	No	0.155625
	Yes	0.153846
75%	No	0.185014
	Yes	0.195059
max	No	0.291990
	Yes	0.710345

In [ ]: